A Coarse-granular Approach to Software Development allowing Non-Programmers to Build and Deploy Reliable, Web-based Applications

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Universität Dortmund

am Fachbereich Informatik

von

Dipl. Inform. Volker Braun

Dortmund

2001

Tag der mündlichen Prüfung: 05.02.2002

Dekan: Prof. Dr. Thomas Herrmann

Gutachter: Prof. Dr. Bernhard Steffen

Prof. Dr. Heiko Krumm

Table of Contents

List of Tables

List of Figures

List of Examples

Preface

Many software tools have been developed both in academia and industry, covering different application domains and profiles. Unfortunately, understanding a tool's profile to the point of deciding whether it can be used for a specific application problem is very hard. In fact, looking for an adequate tool, one is typically confronted with a pool of alternatives, none of which matching exactly the expectations, and it is almost impossible to predict the necessary modifications, let alone estimate their cost. Thus in the course of:

1. searching for candidate tools,
2. installing the tools and getting acquainted with them, and
3. comparing the installed tools in the light of the own application profile and intended use,

people far too often decide to write their own tool, as this gives them the reassuring feeling of full control. Consequently, the wheel is developed over and over again, not necessarily with increasing quality. The main reason for this unsatisfactory situation is the lack of adequate decision support. In fact, none of the steps above is currently systematized:

1. Surfing in the Web may sound like a good solution for the first step, but also the use of search engines usually delivers an accidental collection of results rather than a comprehensive list of the relevant items.
2. The acquisition and installation effort depends very much on the specific situation, but, due to the plague of unpredictable problems, it is usually substantial and often becomes much higher than first expected.
3. A fair comparison is hardly possible because of strongly differing tool profiles, hardware/software constellation, etc.

Even if these problems do not strike, finding ready-to-use solutions for many practical problems would still be out of reach. Often the *cooperation of different* software tools is required to solve the faced problem.

The *Electronic Tool Integration (ETI) Project* addresses all these concerns by providing a Web-based, open platform for the interactive experimentation with and coordination of heterogeneous, off-the-shelf software tools. Via this platform, whose core is the *ToolZone software*, *tool providers* can publish their tools on the Internet, supply case studies and benchmarks defining the tool's profile, and get valuable feedback from the users. *End users* can experiment online with the available tools or combinations of them, evaluate them, and give feedback on their experiences.

A Web site which uses the ETI platform to offer this functionality is called an *ETI Site*. The first ETI site which has been built is the electronic component of the Springer International Journal on *Software Tools for Technology Transfer* (STTT) [STTT]. Here, software analysis and verification tools published in this journal are made available via the Internet. This Web site is intended to develop into a collaborative, independent tool presentation and evaluation site: users are invited to report on their experience with the available tools in the context of the site as a

- directory for possible tools and algorithms satisfying totally or partially their needs.
- (vendor- and producer-) independent test site for trying and comparing alternative products and solutions without any installation overhead.
- quality assessment site for the published tools, which are refereed according to requirements like originality, usability, stability, performance, and design.
- independent benchmarking site for performance on a growing basis of problems and case studies.

In addition to the platform, the ETI project offers an infrastructure

1. which organizes the distributed effort to enhance and maintain the platform,
2. which provides tools and processes helping people to make new tools accessible via the platform, and
3. which supports people who (want to) host an ETI site.

This infrastructure is accessible via the *ETI Community Online Service* at www.eti-service.org.

Acknowledgments

First of all, I would like to thank my supervisor Prof. Bernhard Steffen. After he introduced me to the theoretical foundations of computer science in Aachen in 1991, I attended him on his way from Aachen to Passau and finally to Dortmund. His visionary ideas were always a driving force for my work, in particular in the ETI project. His motivation gave me the energy to bring forward the project even in hard days.

I am grateful to Dr. Tiziana Margaria who, from the very beginning, took care of the users' perspective in the ETI project. This concerned usability aspects of the system itself as well as the forming of the ETI-community and the organization of the annual ETI-days.

A project like the one presented in this thesis could never been successfully realized without a good and motivated team. I would like to thank

- Dr. Andreas Claßen, Achim Dannecker, Carsten Friedrich, Andreas Holzmann, Dirk Koschützki, Falk Schreiber, and Matthias Seul who started the development of the METAFrame environment in 1993. Special thanks to Andreas Holzmann, who is still a member of the METAFrame team, for lots of valuable technical discussions.
- Haiseung Yoo who was responsible for the synthesis component.
- Jürgen Kreileder who is one of the members of the Blackdown team porting the Java platform to Linux. His deep knowledge of the Java platform and his contacts to the Java devel-

oper community helped me to provide fixes and sometimes workarounds for a lot of Java problems I faced during the development of the ToolZone software.

- Neda Zandi-Esser for designing the Web pages of the ETI Community Online Service and the ETI Sites.

I also benefited from many discussions with people of the Chair of Programming Systems and Compiler Construction at the University of Dortmund and with people working at METAFrame Technologies: Claudia Gsottberger, Dr. Andreas Hagerer, Dr. Jens Knoop, Ben Lindner, and Oliver Niese. Dr. Hardi Hungar deserves special mention for critically reading drafts of this thesis and giving valuable comments.

Last but not least I want to thank my family for their ongoing support during my time in Aachen, Passau and Dortmund, as well as my friends for giving me a good time when I visited my home town Düren.

How to read this Document

This thesis comprises four parts:

- I. Motivation and Background,
- II. The ToolZone Software,
- III. Building reliable Web Applications, and
- VI. Conclusion and Perspectives.

Whereas the first and fourth part of this document provide a high level overview for a general audience, Part II and Part III are more focussed. They address:

1. people, who are interested in *Internet-based integration and coordination techniques*. These people should read Section 2.1 of Part I and Part II of the thesis which introduce and go into the details of the ToolZone software.
2. developers, who look for an environment which supports the *component-based development of reliable Web applications*. The information they are interested in is presented in Section 2.2 of Part I and Part III of the document.

Part I presents the goals of the ETI project in Chapter 1 and the technology which is used to put them into practice (Chapter 2). For this, the second chapter provides two sections which introduce

1. the concepts of ToolZone software (Section 2.1) and
2. the fundamentals of the Web-development environment (Section 2.2).

Chapter 3 then presents related work. It discusses the similarities and differences of the applications offered by the ETI project with respect to already available projects aiming at (Web-based) tool access, tool integration and tool coordination as well as approaches to maintain a user community via the Internet.

Part II goes into the details of the design of the *ToolZone software* and the process which is used to make new tool features and data type accessible as activities and types. After a short introduction in Chapter 4 we look at the ToolZone software from a role-based perspective: End Users (Chapter 5), Platform Developers (Chapter 6) and Tool Integrators (Chapter 7). Since the tasks performed by the Site Managers are very technical and depend on the environment in which the ToolZone software is deployed, this role is not covered within this thesis. Specific information can be found within the technical documentation of the project which is available at the ETI Community Online Service.

Part III covers the *Service Definition Environment* and the software development process which is used to build and enhance the *ETI Community Online Service*. First, Chapter 8 gives some fundamental information. After that, Chapter 9 presents the proposed software development process. It is organized along the typical software-development phases: analysis (Chapter 10), modeling (Chapter 11), design (Chapter 12), implementation (Chapter 13) and integration test (Chapter 14).

Part IV presents conclusion and future work. Beside planned enhancements of the ETI platform (see Chapter 16), it introduces first ideas on extensions of the Web development environment supporting

- automated functional testing of Web applications (Chapter 17) and
- self-adapting Web applications (Chapter 18).

Appendix A and Appendix B present detailed information on the design and integration of graphs systems within the ETI platform, and a transformation algorithm used in Chapter 12.

At the end of the document, the glossary and two bibliographies, one providing printed references and one providing online references, are located.

I. Motivation and Background

Chapter 1. Introduction

1.1. The Problem

Modern software engineering is more and more dependent on automation and good tool support. However, faced with a problem, it is hard to identify the appropriate tools. In particular, since generic tools are often not adequate, (different) tools having a specific focus are needed to tackle the task.

Of course, the Internet is a good resource for information. But the advantage of the available information-variety is extenuated by the fact that the right software tool is difficult to find. Though, there is generic support for each step of the tool evaluation process, i.e.

1. searching the Web,
2. reading the available documentation,
3. installing the software tool and finally
4. experimenting with the tool,

tool-evaluation specific support and overlapping assistance is still missing.

Searching the Web:

> The search support currently ranges from generic search engines to Web portals focusing on a certain application domains.
>
> In the course of tool evaluation, the use of search engines often fails, since they deliver masses of results many of which are unqualified. Additionally, most search engines only search with respect to syntactic criteria, like substrings contained in Web pages. But often users are searching for tools with respect to properties which cannot be expressed within the query language.
>
> Web portals are moderated sites offering information on specific issues. In most cases they provide more structured access than general search-engines.

Reading the available documentation:

> In general, the documentation of the tool can be accessed by reading the corresponding Web pages or downloading documents, video and audio files. Due to some well-known standards like the Portable Document Format (PDF), MPEG-2 (Moving Picture Experts Group [MPEG]) covering video formats and MPEG audio layer 3 (MP3) covering audio formats in addition to appropriate software support, this step is very easy.

Installing the software tool:

In contrast to previous step, the software-installation procedure is much more complex. Here, the following (often painful and time-consuming) steps have to be undertaken:

1. The user must (eventually) apply for an evaluation license of the chosen software.
2. The source code or a binary version of the software has to be obtained, e.g. via Web-download or on a CD.
3. An appropriate environment (in terms of hardware, operating system, etc.) which is required by the chosen software has to be made available.
4. The software has to be (compiled and) made available on a local machine.

There are a few sites (see e.g. [HyTech, SMV]) which offer the remote use of a software tool, but they mostly accept and present data on a textual basis and focus of a single tool only (see also Chapter 3).

Experimenting with the tool:

Once the software is installed, the end user finally uses it to perform the evaluation process. To speed-up and improve this process, it is often vital to get in contact with other end users or even with the tool provider to discuss about experiences with respect to the software under evaluation. But most tool providers do not offer the infrastructure that allows tool-related communication. In this case, the end user must again search for appropriate Web sites or within the USENET News Groups.

The tools are evaluated on the basis of case studies sometimes offered by the tool providers. Since the tool providers want to present their tools in a good way, the case studies are often customized for the tool's profile. A fair comparison of two tools covering the same application-domain is only possible if they are run on the same data. For this, the case studies must be available in the tool specific format.

1.2. The Goal

The goal of this thesis is to establish an electronic medium intended to bring software-tool providers and potential end users together that supports all four steps of the tool evaluation process:

Searching the Web:

End users are able to search within a tool database using

- syntactic criteria, like substrings contained in a tool's name and description, as well as
- abstract properties specifying the profile of the tool they are interested in.

Reading the available documentation:

As mentioned in Section 1.1, this step is already supported in an adequate manner.

Installing the software tool:

The installation procedure is facilitated in a way that

- the software installation is performed by a central organization, not by the end user.
- single tools as well as combinations of features coming from different tools are executable.

Experimenting with the tool:

An infrastructure is offered which allows the communication between end users, and end users and tool providers.

For each tool covering a certain application domain, either the same set of case studies is offered, or there are data transformers available which are able to perform the appropriate format conversion, if conceptually possible.

To provide this service to a wide range of people in a reliable way, all this is available within an environment which is

- Internet-based,
- secure,
- performant and
- failsafe.

Besides a rigid organization, a project like this depends on an infrastructure which supports the distributed, collaborative effort to put the project's goals into practice.

1.3. The ETI Approach

The Electronic Tool Integration (ETI) Project is intended to offer an electronic communication medium for tool providers and end users looking for tools helping to solve their problems. This is done by providing a network of moderated Internet sites, called *ETI sites*.

1.3.1. ETI Sites

Each ETI site hosts a *Tool Repository* which contains a collection of functional entities called *ETI Activities*, each of them representing a certain functionality of an individual software tool. Additionally, the tool repository comprises the data types the activities work on. The activities and the associated data types are classified for ease of retrieval according to behavioral and interfacing criteria. In general single functionalities of a tool are identified and integrated as activities into the tool repository, rather than making the tool available as monolithic block. Note, that within this document the term *tool* is used to denote a piece of software which provides certain functionality. This may range from a complex software system to the implementation of a single algorithm.

The ETI sites are not software-distribution sites! End users cannot download the source code of the software nor a binary version. Instead of this, the tools are installed on server machines at the ETI site, which makes their features available via the Internet using the *ToolZone Software*. This client-server application provides Internet-based access to the activities contained in the tool repository offered by an ETI site. Using the *ToolZone Client*, the end user can get information on each activity contained in the tool repository, beside others, in form of a link to the underlying tool's home page or contact address. Whereas links to tools having a specific focus are also available via other Web sites like the Petri Nets Tool Database [PNTD] or the Formal Methods Europe [FME] database, the key feature of the ToolZone software is its unique Internet-based experimentation facility. This means, that via the ToolZone client end users can

- execute single activities as well as
- combine activities and finally execute the resulting programs via the Internet.

To combine (coordinate) the activities contained in the tool repository, experienced users can make use of ETI's procedural *coordination language HLL* (High-Level Language) [Hol-a] to manually implement the intended coordination task on the basis of the available activities. Unexperienced users are supported by means of *ETI's Synthesis Component* [SMF93] which generates sequences of activities out of abstract specifications.

Single activities or combinations of them can be run via the Internet on libraries of examples, case studies, and benchmarks which are also available in the tool repository. Additionally, the user can experiment with own sets of data, to be deployed in user-specific, protected home areas.

The technology which is used to provide these services is called the *ETI Platform* (see also [SMB97, SMB98]). This Web-based, open communication platform was built to offer facilities for the interactive experimentation with and coordination of heterogeneous, off-the-shelf software tools. Using the functionality offered by the platform, *tool providers* can publish their tools on the Internet and get valuable feedback from the end users. *End users* can compare different tools within their application domain and can combine tools coming from different application domains to solve problems a single tool never would have been able to.

1.3.2. Meta-Level

In addition to the platform, the ETI project offers meta-level support. It is realized by means of the *ETI Community Online Service* at www.eti-service.org. This personalizable Web application

- organizes the collaborative effort to develop and enhance the platform,
- supports the people who make new tools and the corresponding data types accessible as ETI activities and types, and
- helps people who (want to) host an ETI site.

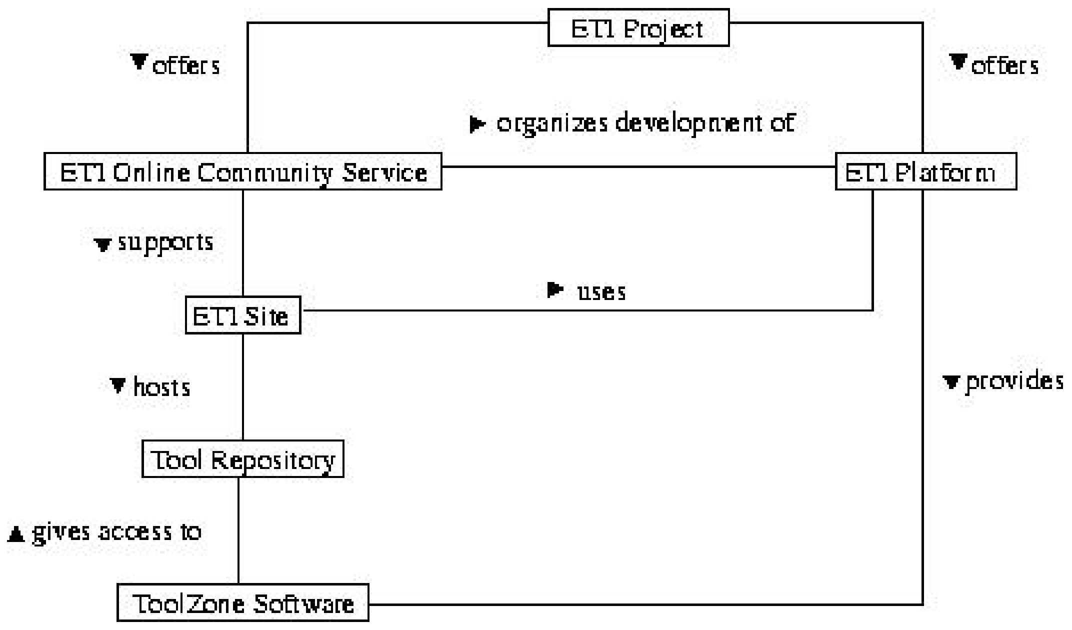
Via this virtual meeting point, the people involved in the development and maintenance of the platform as well as people being responsible for an ETI site may exchange information and discuss topics of their interest. Of course, end users and tool providers are also invited to comment on the platform via the community site to influence future directions. For this, registered users get access to mailing lists, discussion groups, frequently asked questions, platform updates and documentation. To ensure that everyone finds the right information in a fast and easy way, this application is customizable by the user. Additionally, a profile-based permission concept controls the access to the offered functionality and data.

In addition to the ETI Community Online Service, the project infrastructure comprises:

- access to the platform's source code via a version control system which offers concurrent access and secure data transfer,
- coding guidelines to ease the maintenance of code written by other developers,
- SGML-based documentation supporting automatic generation of online and printed versions of the documentation out of *one* source base,
- tools for automatic generation of source code level documentation similar to Javadoc [Javadoc],
- a problem report tracking system for filing and maintaining problem reports, and change requests.

The easiest way to explain the relation between the ETI project, the platform, and the ETI sites, is to present it as a producer-consumer relation (see Figure 1-1). Here, the ETI project is the producer which offers a product called the ETI platform. The ETI sites are the customers of the ETI project. They use the product to provide services to their customers, i.e. the tool providers and the end users. The ETI Community Online Service then organizes the product development and maintenance. In addition it offers product support for the ETI sites.

Figure 1-1. ETI Project Organization



1.3.3. Non-functional Platform Characteristics

The architecture and the concrete implementation of the ETI platform are driven by the four non-functional key requirements mentioned in Section 1.2: *Internet-based*, *secure*, *performant* and *failsafe*. The following paragraphs document how these aspects have been addressed within the ETI platform.

Internet-based

The access to the tool repository and the experimentation facilities is provided by a client-server application, called the ToolZone software. The client, a Java application, communicates with the server-side software components using the Hyper-Text Transfer Protocol (HTTP) protocol [HTTP] and Java Remote Method Invocation (RMI) [Dow98, RMI].

Secure

The ToolZone software offers protected home areas to store personal case studies and data. Additionally, a profile-based security concept is employed to control the access to personal data and functionality. Sensible parts (like access to the user's profile data) of the Web applications hosted by the ETI sites are only accessible using Secure Socket Layer (SSL) based encryption [Res00, SSL].

Performant and failsafe

The ToolZone software uses a load-balancing component to distribute the client sessions on several server machines. This ensures a performant and failsafe access to the provided functionality. The only restriction currently is that ToolZone-users sessions which were served by the machine that went down are lost. They cannot be recovered.

Beside the four main requirements which are specific to the solution we propose, there are three generic requirements every software system should address, i.e. usability, openness and maintainability. The next three paragraphs document how they have been realized within the ETI platform.

1.3.3.1. Usability

Usability of the ETI platform is mainly important for the end user. It is the main aspect which influences the acceptance of the software and by this of the whole project, since the software is represented by the end user's view, here the ToolZone client, on it. Beside others, the software components offered by the ETI project provide:

Intuitive graphical user interface for the client applications:

The ToolZone client application giving access to the tool repository provides an elaborate GUI with help functionality and hypertext support.

Easy and structured access to the activities located in the tool repository:

A taxonomy-based retrieval mechanism offered by the ToolZone software in combination with the synthesis of coordination sequences from loose specifications enable a goal-oriented access to the activities contained in the tool repository.

Minimal set of requirements for client machines:

The ETI Community Online Service as well as the Web applications hosted by the ETI sites can be accessed with a standard Web browser. No additional browser plugins are required. The access to the tool repository is provided by the Java-based ToolZone client which requires a Java Runtime Environment to be installed on the client machine. To fully automize its installation on the client machine, the Java Web Start application launcher [JavaWebStart] must also be available.

1.3.3.2. Openness

"A-posteriori"-integration is another key characteristic of the ETI platform: every feature of an off-the-shelf tool can be made available in the tool repository as an activity. In fact, the ETI platform makes no assumptions on the design, the availability (e.g. source code or binary version, operating system), and the accessibility (e.g. system calls, Java RMI, CORBA [Sie00, CORBA]) of the tool functionalities.

1.3.3.3. Maintainability

This aspect is mostly of interest for the people maintaining the general platform, extending it by new tools and hosting an ETI site as well as for the people developing the ETI Community Online Service. They work on the applications in different phases (development, tool integration, and installation).

With respect to the maintenance of the ETI platform, one important aspect is the availability of an infrastructure which allows fast communication between the platform developers. In the context of the ETI project, this is implemented beside others via

- the ETI Community Online Service providing up-to-date information on the project and communication facilities, like discussion groups and mailing lists,
- remote, secure and concurrent access to the platform's source code via a version control system which provides user-based authentication and encrypted data transfer,
- coding guidelines and standardized processes ensuring the uniformity of the platform's architecture, source code and installations.

Additionally, the ETI platform defines several roles which help to identify the tasks performed by various parties during the development, extension and hosting of the platform: *Tool Provider*, *Tool Integrator*, *Platform Developer*, and *Site Manager*. Each role can be fulfilled by any person having the skills which are necessary to perform the associated task. The following paragraphs give a short introduction into each role defined by the ETI platform, including the tasks and mandatory skills:

Tool Provider:

> A tool provider is an end user with special focus. Whereas the standard end user looks for candidate tools or a combination of tool functionalities to solve a certain problem, the tool provider is mainly interested in publishing his tool and getting valuable feedback from the end users. In addition to the tool itself, the tool provider supplies benchmarks defining the tool's profile.

Tool Integrator:

> The tool integrator makes new activities and data types available within the tool repository. He investigates the tool to be integrated, identifies new activities and types, and establishes connections between the new activities and already available ones. Thus he needs a good understanding of the tool to be integrated and of the application domain modeled in the tool repository. In addition, he must be an expert in the C++ programming language [Str97], which is used to implement the tool management application (see Figure 2-5), the core of the ToolZone software. We support the integration task by providing a tailored process as well as some utility tools (see Chapter 7).

Platform Developer:

> The platform developer adds new functionality to the ToolZone software. He must have Java and C++ knowledge. In addition to that, ToolZone software developers need a good understanding of the platform's architecture which is documented in Part II of this document.

Site Manager:

> A site manager sets up and maintains an ETI site. This includes setting up the servers, installing the required software, and customizing the site-specific Web application. He needs expertise on a UNIX-based operating system (e.g. Linux or SUN's Solaris) for the installation process.

The development of the ETI Community Online Service is supported by a process and a software development tool which are customized for the development of reliable Web applications (see Part III of this thesis). On the basis of a five-layer architecture of the Web application, the process organizes HTML experts, OO specialists, component integrators and application experts to deliver the application within a short time frame. Within this process, the software development tool allows the construction, the validation and the deployment of the Web application in a graphical manner. No programming skills are required for these tasks.

1.4. Background

As will be explained in Chapter 3, there was and still is no existing approach which completely satisfies our needs with respect to the features that should be offered by the platform as well as the ETI Community Online Service. Consequently, we started to realize the platform and in parallel established the corresponding infrastructure in form of a personalizable Web application on our own. For this, two projects provided a good starting point:

- the METAFrame environment (see Section 1.4.1) and
- the Service Definition Environment of the METAFrame project (see Section 1.4.2).

1.4.1. The METAFrame Environment

METAFrame [SMCB96a, MS97, Cla97, CSMB97] "is a meta-level framework designed to offer a sophisticated support for the systematic and structured computer aided generation of application-specific complex objects from collections of reusable components" [MF]. This framework implements a flexible, open component integration architecture (see [Cla97]). As soon as the components are integrated into the METAFrame framework, they can be accessed via *High-Level Language* (HLL) programs. HLL is a coordination language with Pascal-like syntax whose design was influenced by the following four requirements:

- *interpreted*: HLL programs can be easily modified and run without any time-consuming compilation process.
- *simple*: HLL is not a full procedural programming language. Its syntax is in principal restricted to variable declarations, an assignment operator, statements (like while-loops and if-then-else), and procedures/functions. Thus, it is on the one hand simpler to use than other well-known coordination languages like Perl [WCO00]. On the other hand it is expressive enough to realize the coordination tasks we aimed at.
- *typed*: In contrast to other coordination languages, where in most cases only one data type (often the type String) exists, HLL variables can be arbitrarily typed. A central characteristic of the HLL-type concept is that new *basic* types
  - can be added to the language at runtime and
  - may represent complex data structures like directed graphs.

  This contrasts other programming languages like Java and C++ where basic types are hard coded into the compiler. The flexibility of the HLL-type concept is central to the interoperability of activities, since it allows us to define data compatibility on a symbolic level.

- *extensible*: the language can be extended dynamically by new basic types and functions. The types and functions are provided by so-called METAFrame Modules which can be loaded into the interpreter at runtime [Hol97b]. The implementation of the METAFrame Modules is provided by C++ classes. Thus, even third-party functionality provided e.g. by local libraries, remote services like CORBA or binary executables can be made available within HLL.

HLL programs can be run using the HLL interpreter presented in [Hol97b] in detail.

In addition to the HLL interpreter, the METAFrame framework offers the *synthesis component* which generates sequences of functional components out of abstract descriptions [SMF93]. This is central for a non-expert tool coordination.

Consequently, the HLL in combination with METAFrame's synthesis functionality matched our ideas on the coordination aspects. Thus, the METAFrame framework was ideal as the basis for the ToolZone software.

1.4.2. The Service Definition Environment

To build the ETI Community Online Service, we looked for support in terms of tools and processes. But in 1996, Web applications were rather simple. We found none of the available techniques to implement and maintain them (like Common Gateway Interface (CGI) scripts [CGI] and Server Side Includes (SSI) [SSI]) adequate to support the development of complex

Web applications like the one we aimed at. We wanted a solution which supported the structured, cooperative (many developers having complementary skills) development of reliable Web applications. For this, we set up an environment which comprises

- a software development process,
- a Web-application architecture and
- a workflow design tool

The workflow design tool is based on the Service Definition Environment of the METAFrame project. This programming environment had already successfully been used in the area of telecommunications in 1995: in a cooperation with the University of Passau and the Siemens Nixdorf Informationssysteme AG in Munich this tool has been used to realize value-added telephony services in a fast and reliable way [SMCB96b, SMCBR96a, SMCBR96b, SMCBRW96, SMCBNR96, BMSB97, SMBK97, MB98]. There

1. the control flow of the service was graphically configured on basis of a set of components,
2. it was checked using the validation features offered by the Service Definition Environment and
3. once the configuration and validation of the control flow was finished, a compiler generated code out of the description of the control flow which implemented the intended telephony service.

To adjust the Service Definition Environment to the Web-application domain, we generalized the telecommunication-focused framework and developed a compiler which generates Java code mapping the specified control flow into an application (step 3 documented above).

Today, the Service Definition Environment and the associated process are not only used to enhance the ETI Community Online Service. A commercial version of the Service Definition Environment and the development process are used in projects aiming at the realization of reliable, personalized Web applications (see [MFTech]).

1.5. The History of the ETI Project in Dates

1993

Start of the METAFrame project at the University of Passau, Germany. In the initial phase an interpreter for the interpreted, procedural coordination language HLL (High-Level Language) and a graph library was implemented by 4 students.

1994

The initial version of the synthesis component was added to the METAFrame environment. It offers sophisticated support for the systematic and structured computer aided generation of application-specific complex objects from collections of reusable components .

03.1995 - 10.1995

The first version of the Service Definition Environment was implemented in cooperation with the University of Passau and Siemens Nixdorf Informationssysteme AG in Munich.

1996

The realization of the ETI platform was initiated in the context of the *Springer International Journal on Software Tools for Technology Transfer* (STTT)[STTT], with the goal to provide a platform for the interactive experimentation and coordination of heterogeneous software tools.

31.03. - 2.4.1998

First official presentation of STTT/ETI at ETAPS' 98, the first *European Joint Conferences on Theory and Practice of Software*, in Lisbon, Portugal, including online ETI demonstrations.

1999

Start of the platform's re-engineering process, motivated by feedback of the platform users. One important result of this re-engineering phase was the decoupling of the ETI platform from STTT. Whereas the first version of the platform was tighten to STTT, the Web site hosting the electronic component of STTT today is one of the available ETI sites.

This generalization was the starting point of an international open-source project, called the *ETI Project*. Within this context the ETI community was founded which is organized via the ETI Community Online Service.

06.2000

Launch of the ETI Community Online Service and of the documented version of the ETI platform.

Chapter 2. Technology Overview

Beside some utility programs, which will not be documented in this thesis, the ETI project offers two main applications (see also Figure 1-1):

1. the *ToolZone software*, which is part of the ETI platform (see also [SMB97, SMB98]), gives Internet-based access to the tool repository which contains the activities and types representing functionalities offered by off-the-shelf tools and the data types they work on.
2. the *ETI Community Online Service* organizes the development of the ETI platform and provides support for the people operating an ETI site. This Web application offers access to discussion groups, mailing lists, a problem report management system, and platform documentation and source code etc.

This chapter provides some fundamental knowledge about the two key-applications offered by the ETI project and gives references to later chapters which go into the details of the illustrated aspects. Section 2.1 presents the concepts of the main software component of the ETI platform, the *ToolZone software* (the end user's and developer's view on the software is documented in Chapter 5 and Chapter 6, respectively). This component gives direct access to the activities and types contained in the tool repository of an ETI site. Using the ToolZone client, the end user can via the Internet

1. retrieve information on the available tool-repository entities,
2. execute single activities, and
3. combine activities to programs and then execute the programs.

Section 2.2 then introduces the Web-development environment. Here, we give a short overview of the Web-application architecture and the software development process in Section 2.2.1. Section 2.2.2 then introduces the Service Definition Environment of the METAFrame project. Finally, Section 2.2.3 presents how the software components implementing the Web application, which has been built with the proposed environment, can be distributed on several server machines to ensure performant and failsafe access to the application's features.

> **Note:** Besides the maintenance of the ETI Community Online Service, the Service Definition Environment and the associated software development process are also used to build parts of the Web application hosted by each of the ETI sites.

2.1. Introducing the ToolZone Software

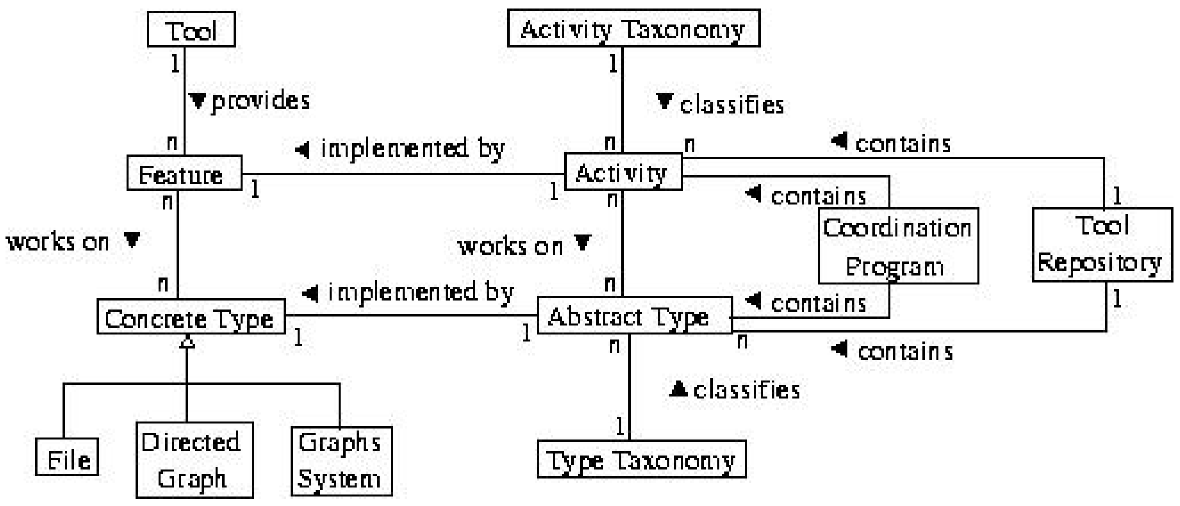
The ToolZone software gives structured access via the Internet to the activities contained in an ETI site's tool repository, with the ability to

1. browse through the available activities and the data types they work on,
2. execute single activities,
3. combine activities to programs (called *Coordination Programs*), and run the programs.

Every functionality of an off-the-shelf software tool can be integrated into the tool repository as an ETI activity. After their integration, the activities can either be executed stand-alone or they can be combined to programs. The ToolZone software provides two means for activity combination (in the following called coordination): *HLL programming* and *program synthesis*.

The implementation of the ToolZone software is based on the part of the ETI meta model shown in Figure 2-1 as a UML class diagram [BRJ98, FS99, UML]. Within this figure, a "Coordination Program" denotes either a hand-made HLL program or a program generated by the synthesis component.

Figure 2-1. The conceptual Model of the ToolZone Software



The right part of the meta model shown in Figure 2-1 covers the platform's view on the tool features. It presents the relation between the activities, the types they work on, the taxonomies and the tool repository. The left part of the same figure models the world of the software-tools. The link between the software tools and the platform's view on them in form of the activities is modeled by the "implemented by" associations. This link is established during the tool integration task, which makes a certain tool feature and the concrete data types they work on accessible by the platform as activities and types. Chapter 7 goes into the details of this process.

In this section, we first go into the details of the functional entities contained in the tool repository, i.e. the ETI Activities. Subsequently, Section 2.1.2 presents how activities and the data types they work on can be classified within the *Activity Taxonomy* and *Type Taxonomy*, respectively. Section 2.1.3 and Section 2.1.4 then cover the coordination facilities offered by the ToolZone software: HLL programming and program synthesis. Finally, Section 2.1.5 gives an overview of the architecture of the ToolZone software.

2.1.1. Activities

In general, tools are not integrated into the tool repository as monolithic blocks. Rather, single tool features are identified and prepared to be accessible by the platform. Within the tool repository, a single tool functionality is represented by an *ETI Activity* which is the elementary functional component of the ETI platform.

An activity definition comprises

- the *name* that is beside others required to reference the activity in a coordination-task description (see Section 2.1.4.2).
- the *tool* including the parameters to access the feature represented by this activity.
- the *documentation* which provides beside others the information on the tool feature that is represented by this activity.
- the *classification constituent* in terms of predicates which characterizes the activity within the tool repository. This is the basis for the structured activity access by means of the taxonomies (see Section 2.1.2). Additionally this information can be used to specify an activity using abstract properties instead of its name in a coordination-task description (see Section 2.1.4.2).
- the *implementation constituent* in terms of a HLL function which implements the activity on the basis of the corresponding tool feature.
- the *stand-alone constituent* which defines the execution behavior of the activity in stand-alone execution mode. This HLL program is run, when a single activity is executed by the user. Whereas the implementation constituent is defined by one single HLL function, the stand-alone constituent provides the implementation of the context in which this HLL function can be executed in stand-alone execution mode.

After an activity has been made available within the tool repository, it can be combined with other activities. For this, the end user can write an HLL-program implementing the intended coordination task (see Section 2.1.3). Alternatively, the glue-code coordinating the activities can be generated automatically on the basis of an abstract coordination-task description, called *loose specification* (see Section 2.1.4). A planned extension of the ToolZone software will also allow a mixed approach.

In contrast to HLL-based coordination which can be used to combine arbitrary activities contained in the tool repository, only activities having a certain profile can be used for program synthesis. These *synthesis-compliant activities* look at a tool feature as a “transformational” entity. This means that a tool feature is seen as a component taking an object of type T_1 as input and delivering an object of type T_2 as output.

In addition to the information about standard activities synthesis-compliant activities provide

- the *interface constituent* specifying the abstract input/output behavior of the activity, i.e. its input and output type. This is used by the synthesis component to combine activities to programs.
- the *tool-coordination constituent* which defines the execution behavior of the activity in tool-coordination execution mode. Similar to the stand-alone constituent, this activity constituent provides the implementation of the context in which the HLL function can be executed in tool-coordination execution mode.

As an example, Table 2-1 shows the names and the interface constituent of the definition of simple synthesis-compliant activities identified in the world of text processing tools. It presents the name of the activities, their input and output types as well as the tools their implementation is based on.

Table 2-1. Simple Text Processing Activities

Activity Name	Input Type	Output Type	Tool	Description
latex	TEXFile	DVIFile	**latex**	Structured text formatting and typesetting program.
dvips	DVIFile	PSFile	**dvips**	Converts a TeX DVI file to PostScript.
gv	PSFile	Display	**gv**	A PostScript viewer.
lpr	PSFile	Printer	**lpr**	Prints files.

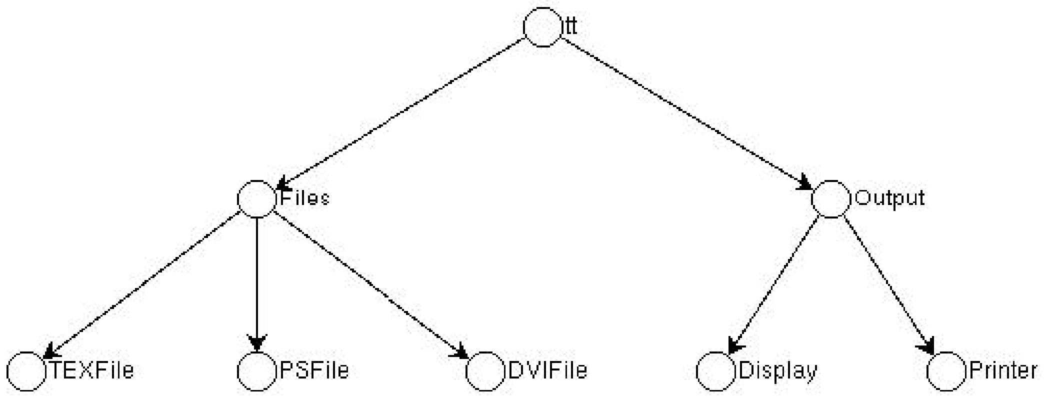
2.1.2. Taxonomies

For a flexible handling (retrieval, loose object specification, abstract views), activities and the data types they work on are classified by means of the *Activity Taxonomy* and *Type Taxonomy*, respectively. A taxonomy is a hierarchical structure of predicates over a set of atomic elements, here the activities and types respectively. Formally, a taxonomy (defined over a set of atomic objects S) is a sub-lattice of the power-set lattice over S which comprises elements with a particular profile which are identified by names.

Taxonomies can be represented as directed acyclic graphs (DAGs) (see Figure 2-2) where each leaf represents an atomic entity (here activity or type) and each intermediate node represents a set of entities, called *group*. Conceptually, edges reflect an “is-a” relation between their target and source nodes. The semantics of an intermediate node is the set of all atomic entities which are reachable within the taxonomy DAG from this node. With respect to this semantics, edges reflect the standard set inclusion.

Figure 2-2 shows a simple classification of the types introduced in Table 2-1. Here, the type group tt represents all types which are available in the tool repository, i.e. TEXFile, PSFile, DVIFile, Display, and Printer. The type group Files represents the types TEXFile, PSFile, and DVIFile. The activities are organized within the activity taxonomy analogously.

Figure 2-2. A simple Type Taxonomy



2.1.3. HLL Programming

Once an activity is available in the tool repository, it can be accessed via the HLL (High-Level Language) [Hol-a] function defined by the activity's implementation constituent. On the basis of this HLL function, HLL programs are used to manually combine activities representing heterogeneous functionalities coming from different tools in order to perform complex tasks. Example 2-1 shows an HLL-based coordination program.

Example 2-1. An HLL Coordination Program

The HLL-based coordination program presented in this example minimizes a labelled transition system (LTS) stored in the `aut`-format, a file format defined by the Caesar/Aldebaran Development Package (CADP) [FGK96, CADP]. After that, the `xtl` model checker, a tool contained in the CADP toolkit, is invoked on the minimized labelled transition system.

In detail, this is done by requesting from the user the files storing the labelled transitions system and the formula to be checked via the `fsBoxLoad` function of the `ETI` HLL-library (see **(1)** and **(2)** of Example 2-1). Then the model is minimized with respect to observational equivalence [Mil89] using the `aldebaranMIN_STD_I` function contained in the HLL-library `CADP` (see **(3)**). Since the `xtl` model checker requires the model to be provided in the `bcg` file format, the `aut`-representation of the minimized labelled transition system is transformed into this format by the HLL function `autF2bcgF` (see **(4)**). Finally, the `xtl` model checker is called using the `xtl` function of the HLL-library `CADP` getting the model and the formula as arguments (see **(5)**).

```
var String: aut_model;
var String: min_aut_model;
var String: bcg_model;
var String: formula;
var ETIResult: result;

aut_model := ETI.fsBoxLoad ("Select Model", "*.aut");         (1)
formula := ETI.fsBoxLoad ("Select Formula", "*.xtl");         (2)

result := CADP.aldebaranMIN_STD_I (aut_model, min_aut_model); (3)
result := CADP.autF2bcgF (min_aut_model, bcg_model);          (4)
result := CADP.xtl (bcg_model, formula);                      (5)
```

2.1.4. Program Synthesis

In addition to the HLL-based coordination, the ETI platform provides automated coordination support by means of its synthesis component. Here, the glue-code combining the activities is automatically generated. For this, the user *specifies* a coordination task via an abstract description called *loose specification*, instead of programming it using the HLL. From the coordination-task description, the synthesis component then generates sequences of activities (called *coordination sequences*) each implementing the specified task. Using loose specifications, the user characterizes *what* he wants to achieve instead of *how* to achieve it. This goal-oriented approach is the main difference between ETI's synthesis-based coordination facility and other coordination approaches like UNIX piped commands, scripting languages (e.g. Perl [WCO00], Python [Lut01]) or the **ToolBus** [BK96, KO96, ToolBus]: there the user is forced to precisely specify the coordination process like in HLL programs.

But there are two limitations with respect to the programs generated by the synthesis component, which do not apply to HLL-based coordination:

- Whereas a full procedural programming language can be used to implement the coordination task in HLL, only sequential compositions of activities can currently be generated by the synthesis functionality.
- The synthesis algorithm can only combine synthesis-compliant activities (see Section 2.1.1), whereas the HLL can be used to coordinate arbitrary activities available in the tool repository.

In the following sections we go into the detail of the program synthesis.

2.1.4.1. Coordination Sequences

On the basis of the available synthesis-compliant activities, end users can build and then execute sequential programs called *coordination sequences* which are finite paths of the form

$$\begin{tikzcd} T_1 \arrow[r, "a_1"] & T_2 \arrow[r, "a_2"] & T_3 \arrow[r, dashed] & T_n \arrow[r, "a_n"] & T_{n+1} \end{tikzcd}$$

where T_i is a type and a_i is a synthesis-compliant activity which transforms an object of type T_i into an object of T_{i+1}. For illustration, we provide a concrete example of a coordination sequence based on the activities introduced in Table 2-1.

Example 2-2. A Coordination Sequence Example

The following coordination sequence starts with a `tex`-file, runs the **latex** command on it, transforms the DVI result into a PostScript file and displays the PostScript file on the screen using the PostScript viewer **gv**.

$$\begin{tikzcd} \text{TEXFile} \arrow[r, "\text{latex}"] & \text{DVIFile} \arrow[r, "\text{dvips}"] & \text{PSFile} \arrow[r, "\text{gv}"] & \text{Display} \end{tikzcd}$$

2.1.4.2. Loose Specifications

Building a coordination sequence manually may be a non-trivial task: the end user must know the synthesis-compliant activities contained in the tool repository and must solve tool interfacing issues like finding the right data-type transformer to connect features possibly provided by different tools. To ease this task, the ETI platform offers a synthesis component which generates the coordination sequences out of abstract descriptions, called *loose specifications*.

These abstract descriptions which are based on the Semantic Linear-time Temporal Logic [SMF93] (see Section 5.2.2 for details) are loose in two orthogonal dimensions:

Local Looseness:

The characterization of types and activities is done at the abstract level of the taxonomies, instead of enumerating them explicitly. Here names contained in the taxonomies are interpreted as propositional predicates. They can be combined by the Boolean operators & (*and*), | (*or*) and ~ (*not*) to specify sets of activities and types. With respect to the type taxonomy presented in Figure 2-2, the formula `tt & ~Files` characterizes the types Display and Printer.

Global/Temporal Looseness

The characterization of whole coordination sequences is done in terms of abstract constraints specifying precedences, eventuality, and conditional occurrence of single taxonomy entities, rather than specifying the precise occurrence of the types and activities (see e.g. the *before* operator < in Example 2-3).

Example 2-3. A loose Specification

On the basis of the activities shown in Table 2-1 and the type taxonomy presented in Figure 2-2, the user could write a loose specification of the form of

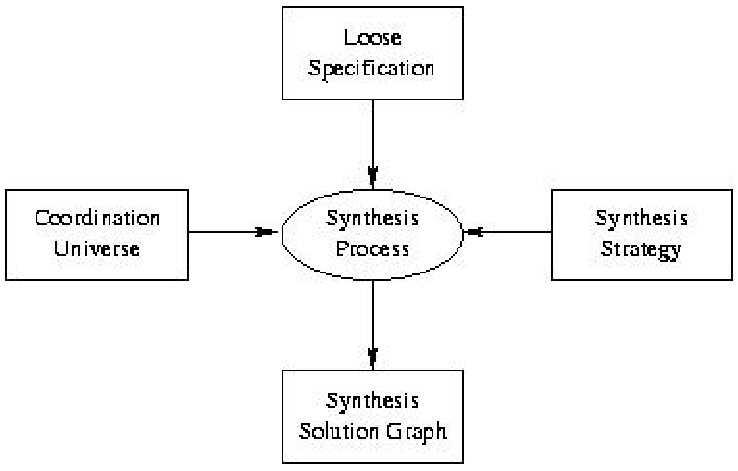
```
TEXFile < Output
```

to query all coordination sequences able to bring a TEXFile on some Output device. Note, the *before* operator < contained in the formula. It specifies that an Output object must follow a TEXFile object without fixing when (as direct successor or anywhere later in the future) this occurrence should take place.

2.1.4.3. The Synthesis Process

On the basis of the available synthesis-compliant activities and a loose specification, the synthesis component (see Figure 2-3) delivers a *synthesis solution graph*. Within this directed graph (see Figure 2-4 as an example), each path starting at the unique start node and ending at the unique end node represents one coordination sequence which satisfies the given loose specification.

Figure 2-3. The Synthesis Process



The synthesis solution graph is determined on the basis of the *coordination universe* which is the mathematical model of the synthesis process. This directed graph represents all possible combinations of the activities contained in the tool repository. Similar to the synthesis solution graph, nodes contained in the coordination universe model types and edges model synthesis-compliant activities whose input type is represented by the source node of the edge and whose output type is represented by the target node. In order to determine the synthesis solution graph, the synthesis process “searches” for paths within the coordination universe that satisfy the loose specification. This set of paths (if not empty) is then returned in form of the synthesis solution graph as result of the synthesis process. The coordination universe is automatically generated by the synthesis component. This process uses the specification of the activities' input/output behavior defined by the activities' interface constituent (see Section 2.1.1).

The result of the synthesis process is influenced by a *solution strategy* which specifies the kind of the coordination sequences the end user is interested in. Four strategies are available:

All Solutions

Returns all coordination sequences which satisfy the given loose specification. Since the coordination universe may contain cycles, this is also true for the generated solution graph.

All Minimal Solutions

Returns all minimal (w.r.t. their length) coordination sequences which satisfy the given loose specification. In contrast to the all-solutions strategy, the minimal-solutions strategy handles loops contained in the coordination universe by unrolling. This means that the resulting solution graph does not contain any cycles. Minimal-solutions may have different length.

All Shortest Solutions

Returns all shortest (w.r.t. their length) coordination sequences which satisfy the given loose specification. The set of the shortest coordination sequences is the subset of the minimal coordination sequences where each element has a minimal number of activities. Thus, all generated coordination sequences have the same length.

One Shortest Solution

Generates one shortest coordination sequence which satisfies the given loose specification. If there exists more than one shortest coordination sequence which satisfies the specification, the algorithm randomly selects one.

With respect to the loose specification introduced in Example 2-3 ETI's synthesis component delivers *two* coordination sequences as result:

$$\begin{tikzcd}
\text{TEXFile} \arrow[r, "\text{latex}"] & \text{DVIFile} \arrow[r, "\text{dvips}"] & \text{PSFile} \arrow[r, "\text{gv}"] & \text{Display}
\end{tikzcd}$$

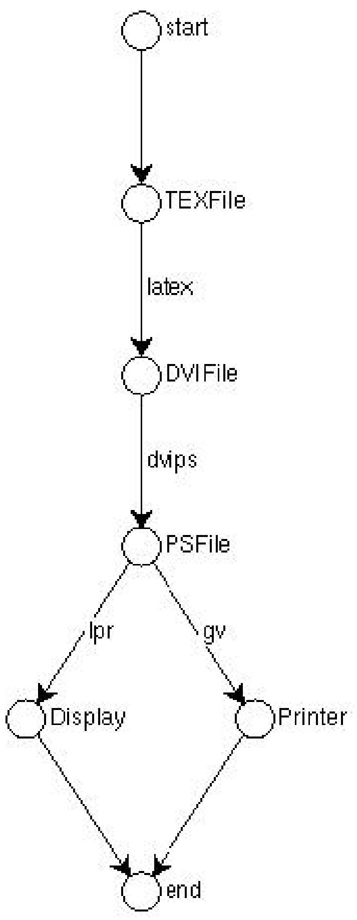
and

$$\begin{tikzcd}
\text{TEXFile} \arrow[r, "\text{latex}"] & \text{DVIFile} \arrow[r, "\text{dvips}"] & \text{PSFile} \arrow[r, "\text{lpr}"] & \text{Printer}
\end{tikzcd}$$

This is due to the fact that the output device has been loosely specified by using the type group Output instead of one of the atomic types Display or Printer (local looseness). Since there is no activity which directly transforms a TEXFile object into a Display or Printer object, the synthesis component automatically inserts three activities which in combination perform this transformation.

The two coordination sequences are represented by the coordination graph shown in Figure 2-4 which is the result of the synthesis process. Note that the “start” and “end” nodes are automatically added for technical reasons.

Figure 2-4. The “TEXFile < Output” Solution Graph



As documented in Section 5.2, using the ToolZone client the end user can select and then execute a path within the solution graph.

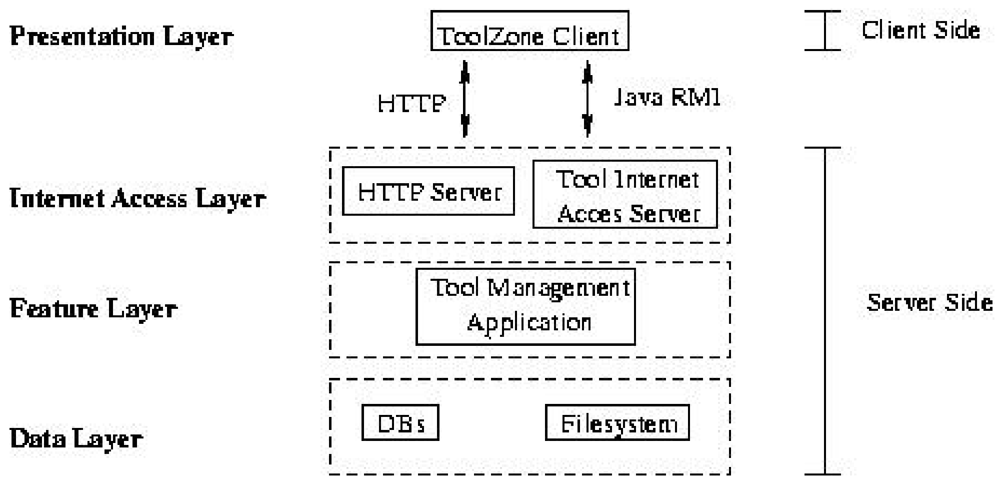
2.1.5. The ToolZone Architecture

Logically, the ToolZone software is built using four layers (see Figure 2-5): the data layer, the feature layer, the Internet access layer and the presentation layer.

The *Data Layer* stores information on the tool repository, i.e. the available activities and types as well the user-owned (private) and common case studies.

On top of the data layer, the *Feature Layer* provides the real functionality offered by the Tool-Zone software. This layer contains the C++-based [Str97] *Tool Management Application*, the core of the ToolZone software. All end-user features offered by the ToolZone software are implemented by this application. It gives access to the activities and types contained in the tool repository, it provides the coordination means, and controls the execution of single activities and coordination programs.

Figure 2-5. The logical Architecture of the ToolZone Software



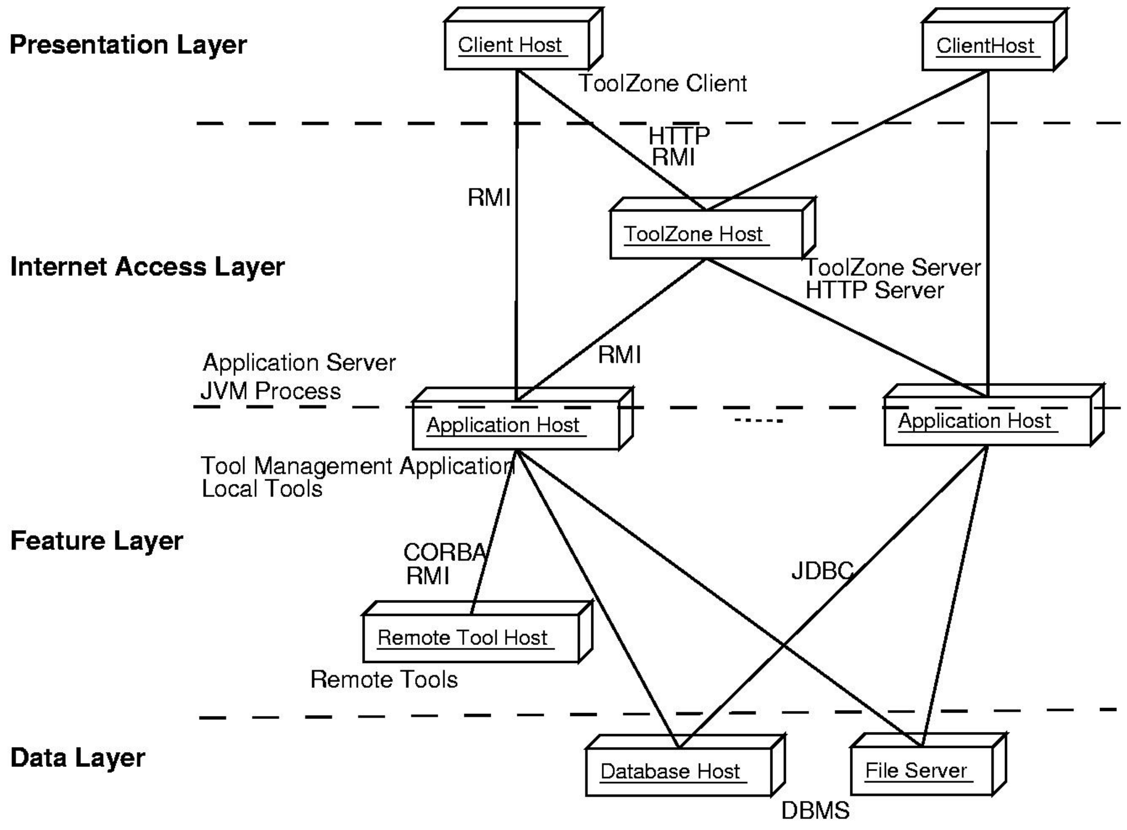
The *Internet Access Layer* provides the facilities to make the features of the tool management application accessible via the Internet. For its implementation we use a standard HTTP (Hypertext Transfer Protocol [HTTP]) server, e.g. the Apache server [Apache], and the *Tool Internet Access Server*. Here, the HTTP server provides the activity and type documentation, and an ETI site specific configuration file (see Section 5.1 for more details). The functionality provided by the tool management application is encapsulated by the tool Internet access server which is connected via Java Remote Method Invocation (RMI) [Dow98, RMI] to the ToolZone client.

The *Presentation Layer* contains software components to access the functionality provided by the feature layer via a graphical user interface. Here, the Java-based ToolZone client is used to gain access to the activities and types contained in the tool repository. It can be launched from the Web browser using the Java Web Start technology [JavaWebStart].

Physically, the components implementing the ToolZone software can be distributed on several machines to provide performant and failsafe access to the offered features. Figure 2-6 shows a UML deployment diagram documenting the distribution of the ToolZone software components on different server machines. This illustrates a possible physical organization of an ETI site. Here, the ToolZone server running on the ToolZone host and the application server running on the application host in combination with the Java Virtual Machine (JVM) process implement the tool Internet access server introduced in Figure 2-5.

> **Note:** In a minimal scenario, all server-side software components may be deployed on one single host.

Figure 2-6. The physical Organization of an ETI Site



In Figure 2-6, the blocks denote the hosts being part of an ETI site. The software components running on each of the hosts are listed near the corresponding blocks. The links in conjunction with their labels specify the kind of communication taking place between the hosts.

The responsibilities of the hosts and the require software components are in detail described in Section 5.1.

2.2. Building the Community Online Service

The ToolZone software documented in the previous section gives access to the tool repository provided by an ETI site. Using this application, end users can experiment with the available activities via the Internet. In contrast to that, the ETI Community Online Service at www.eti-service.org organizes the collaborative effort to enhance the ToolZone software, to extend it by new activities and types, and to host an instantiation of the platform at an ETI site. This personalizable Web application implements a virtual meeting point where ETI platform developers, tool integrators as well as ETI site managers may exchange information and discuss topics related to the general platform. Of course, end users are also invited to comment on the platform, to file bug reports and to propose future enhancements.

The ETI Community Online Service and the Web applications of the ETI sites have been implemented and are continuously being extended using the Service Definition Environment of the METAFrame project as workflow design tool and a software development process in which the tool is embedded.

Whereas the already available Java technology [CW98, Java] and appropriate development tools, like Borland's JBuilder [JBuilder], were sufficient to build the ToolZone software, there was no adequate development support in terms of processes and tools customized to realize the ETI Community Online Service in 1996. At that time, Web applications were implemented by simple CGI (Common Gateway Interface) [CGI] scripts, like visitor counters on Web pages or scripts presenting the content of a database within a Web browser. There was neither a process nor tool support required to bring a Web application into operation. In most cases, the CGI scripts were programmed by a single developer just using a standard text editor like XEmacs [XEmacs].

The complexity of the ETI Community Online Service required a more structure approach. We developed a general environment which was intended to build reliable Web applications based on the Java technology. This environment which supports distributed development teams comprises

- a role-based software development process,
- a Web-application architecture, and
- a workflow-design tool.

The results of the workflow-design tool can automatically be translated into a server-side Java application in form of a Servlet [HC98, Servlet].

Section 2.2.1 gives an overview of the process and the Web-application architecture the process is based on. After that, Section 2.2.2 introduces the workflow design tool, i.e. the Service Definition Environment of the METAFrame project (Details can be found in Part III of this thesis). Afterwards, Section 2.2.3 presents how the software components implementing the Web application may be distributed on several machines to provide performant and failsafe access to the features provided by the ETI Community Online Service.

2.2.1. The Key Concepts of the Process

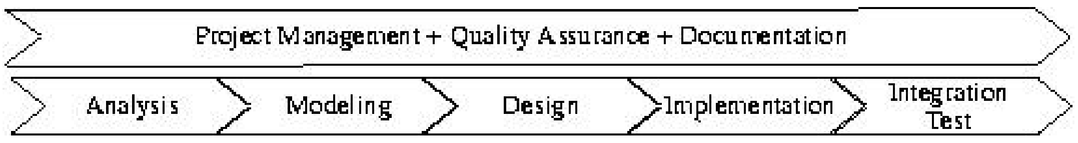
To successfully roll-out an application within a given time frame the required tasks have to be centrally coordinated. This is done by a software development process which defines

- which *task* has to be performed by
- which *person* at

- which *time*.

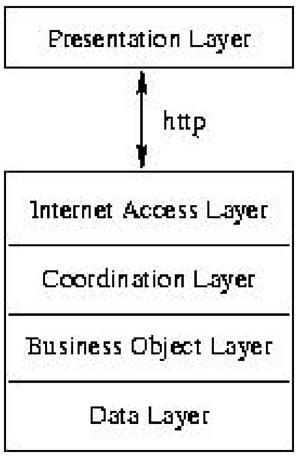
The progress of this process is supervised by the project manager. In general, a software development process is defined along the five core phases: Analysis, Modeling, Design, Implementation, (Integration) Test. Additionally, there are at least three tasks which accompany these core phases: Project Management, Quality Assurance and Documentation (see Figure 2-7). Note that the figure illustrates an idealized structure of a software development process. Within an iterative process, the phases are typically traversed several times.

Figure 2-7. The idealized Structure of a Software Development Process



The process presented in detail in Part III of the thesis, focuses on the core development phases. It is based on a five-layers architecture of a Web application shown Figure 2-8.

Figure 2-8. Architectural Layers of a Web Application



As shown in Figure 2-8, we distinguish the application's GUI (*Presentation Layer*), the communication layer (*Internet Access Layer*), the application's logic (*Coordination Layer*), the base functionality implemented by the *Business Object Layer* and a layer to store persistent data (*Data Layer*):

The lowest layer is the *Data Layer*. It provides the functionality to store the persistent data of the business objects contained in Layer Two into a persistent, electronic medium like a database or the file system.

On top of the data layer, the *Business Object Layer* of the Web application is located. Business objects are instances of business classes which represent things of the chosen application domain, like a user or a tool within the ETI Community Online Service. In general, this layer provides the features on which the implementation of the business processes is based.

The software components contained in the *Coordination Layer* implement the business processes provided by the application, like the registration procedure of the ETI Community Online Service. They coordinate the features offered by the business object layer to perform their task. Whereas the business classes can in most cases be reused for other applications within the same application domain, the processes are specific to the application to be built. Additionally, unlike the business classes, the represented processes may change over the time. For this a flexible handling of the coordination layer is required.

The *Internet Access Layer* "just" connects the presentation layer and the coordination layer using the HTTP (Hypertext Transfer Protocol) protocol [HTTP]. In our scenario, its implementation is provided by a Java 2 Platform Enterprise Edition (J2EE) [J2EE] compliant Web container like tomcat [tomcat].

The *Presentation Layer* contains components implementing the Web application's GUI. These are mostly HTML pages which are rendered by the client's Web browser. In general, the HTML (Hypertext Markup Language [HTML]) pages may contain Applets [Applets], JavaScript [Fla98] functions or Flash [Flash] elements to integrate functionality, or graphic and sound elements to animate the presented content. Client-side components like these should only be used in a restricted way with respect to the presented architecture. It must be ensured that the software components located in the coordination layer always keep the full control on the Web application.

Other tools support the realization of the software components located in the application's presentation layer (e.g. Macromedia Dreamweaver [Dreamweaver]) and business object layer (e.g. the JBuilder). Our coordination-centric approach, which is realized by the Service Definition Environment, does not require manual programming. Instead, the implementation of the coordination layer is provided by the following steps:

1. On the basis of the business classes a library of components called *Service-Independent Building Blocks* (SIBs) is built.
2. The coordination layer is modeled as a *Service Logic Graph* using the Service Definition Environment on the basis of the available SIBs.
3. The service logic graph is checked with respect to quality requirements using the validation features offered by the Service Definition Environment.
4. The implementation of the coordination layer is automatically generated from the service logic graph on the basis of the SIBs' implementations.

Once the implementation of the business classes and the required components have been provided, the construction, the validation, and the installation of the Web application (steps 2 - 4) are done via the GUI of the Service Definition Environment. Neither Java-programming skills, nor system administration knowledge (like configuring and restarting the Web container) are required to perform these steps.

Compared to other widespread techniques for the development of Web-applications (like e.g.

Allaire's Cold Fusion Scripts [ColdFusion] and Java Server Pages[FI00, JSP]) where features are implemented by scripting elements embedded into the HTML pages, the proposed concept has two advantages:

1. Due to the explicit representation of the process layer, the application development is workflow-centric and not document-centric. Change requests concerning the processes implemented by the Web application can be realized without any programming effort.
2. The strict architectural separation of the GUI and the process layer allows the independent implementation and maintenance of the features provided by the application and the application's look and feel. In consequence, another or a new GUI can be implemented without modifying the implementation of the features provided by the process layer and the business object layer. This contrasts HTML-embedded scripting techniques where the implementation of the GUI and the offered features are maintained in the same document. There, providing another GUI for the same application often results in

    1. making a copy of the original application,
    2. modifying the GUI portion of the copy and
    3. finally maintaining two versions of the same application which only differ in their look and feel.

The logical layers of a Web application induce several task-specific roles:

- *System Engineers* analyze the problem which should be addressed by the Web application.
- *OO Specialists* design and implement the business classes required to realize the SIBs.
- *SIB Integrators* provide the new Service-Definition-Environment-compliant components and
- *Application Experts* configure the application using the Service Definition Environment,
- *HTML Designers* build the GUI of the application,

In combination with the validation features offered by the Service Definition Environment (see next section for an overview), this division of labor is central for developing complex, reliable Web applications with a cooperative, distributed effort within a short time frame.

The next section gives a short introduction to the Service Definition Environment. It is organized along the steps 2 to 4 which have to be performed to construct the coordination layer of a Web application on the basis of the SIBs. Step 1 of the procedure documented above is detailed in Section 13.2.

2.2.2. An Overview of the SD Environment

The Service Definition Environment is a workflow design tool which allows the graphical configuration of workflows on the basis of components called *Service-Independent Building Blocks* (SIBs).

2.2.2.1. Service-Independent Building Blocks

Conceptually, SIBs can be seen as functions which perform specific tasks and deliver a return value as result. Their specification comprises all information which are used by the features of the Service Definition Environment, like symbolic execution, local and global consistency checking (see Section 12.2), and code generation (see Chapter 13):

The SIB Definition

describes the logical view of a component of the Service Definition Environment. It specifies the identifier of the component, classification information, the parameters (which can be used to customize instances of a specific SIB) and a collection of possible return values. Beside others, this information is the basis for the global validation of the application logic in terms of formal verification.

The SIB Documentation

gives some information on the task performed by this SIB. Beside others, it documents under which circumstances a specific return value is delivered.

The Simulation Code

is used by the trace-feature of the Service Definition Environment to symbolically execute the modeled application logic at design time.

The Local-Check Code

specifies local consistency rules. In contrast to global validation, which checks the interaction between SIBs, local validation checks the consistency of the configuration of single SIB occurrences.

The Implementation Code

defines the physical aspect of a SIB. It is used by the compiler during the code-generation process. In general, the SIBs' implementation code may be written in any programming or scripting language like C/C++ [Str97] or Perl [WCO00]. However in the Web-application area, the implementation of a SIB is provided by a Java class.

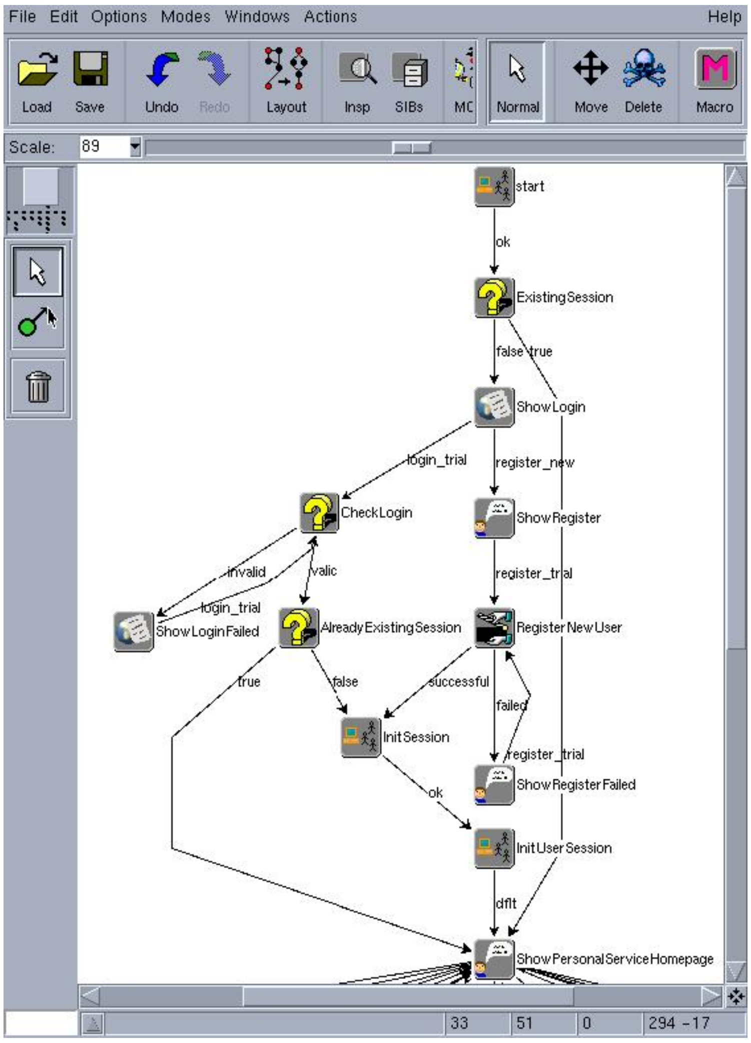
This strict separation of the aspects of a SIB specification was motivated by the requirement that the modeling and validation of the application logic can already be performed *before* the implementation of the business objects and SIBs is available (see Figure 9-1).

2.2.2.2. Modeling the Application's Coordination Layer

The coordinating workflows are modeled as directed graphs called *Service Logic Graphs* (SLGs). Figure 2-9 shows an SLG fragment which models the login and registration procedure of a Web application. The nodes of the graph represent functional entities, i.e. the SIBs in this process. Within the SLG each of the possible return values of a SIB can be attached to an edge starting at the node which represents the corresponding SIB to steer the flow of control.

In the SLG shown in Figure 2-9 the edge which starts at the node `RegisterNewUser` and ends at the node `InitSession` specifies that the execution of the application should be continued with the SIB `InitSession`, if the execution of the SIB `RegisterNewUser` delivers the results `successful`. This way the edges represent the flow of control through the coordination layer. Each service logic graph must have a unique start node which defines the starting point of the application.

Figure 2-9. The Service Logic Graph Editor



2.2.2.3. Validation

During the design, the Service Definition Environment supports the end-users in checking the modeled application on the level of the coordination layer using the validation features documented in Section 12.2. This means, that the validation components work on the service logic graph by checking the proper configuration and interaction of the SIBs. They do not consider the implementation code of a SIB which is viewed as an atomic entity and assumed to be correct. Beside symbolic execution and local consistency checking, formal verification in terms of model checking [VW86, SO97, MSS99, Cle99] is used to validate the global interaction of the SIBs. With respect to the model checking, the correctness assumption of the SIBs' implementation code was an explicit design criterion settled at the beginning of the realization of the Service Definition Environment. This was motivated by the following reasons:

The profile of the end-user:

> The targeted end-user group of the Service Definition Environment is the application expert who knows the workflows of his application domain very well, but in general has little or no programming skills. Confronting an end-user with error messages on (e.g. Java-) code level would make no sense, since he is unable to interpret the error reports.

The complexity of exhaustive formal verification:

> Proving the correctness of complex Web applications on code level is in general infeasible. There are techniques which allow the formal verification of Java programs with respect to certain properties (see e.g. the Bandera project [CDHLPRZ00, Bandera]). But all these techniques require a deep understanding of the program and the problem to be verified, and the technique which is used for the program analysis. Additionally, a lot of properties cannot even be verified automatically. Here, the end-user must interact with the verifier to provide auxiliary information required for the verification process.
>
> The application expert, the end-user of the Service Definition Environment, would of course never accept such a use of formal verification. He wants a seamless integration of formal methods at his level of understanding (see also [MBS01]). For Web applications, service logic graphs turned out to be adequate for modeling. Moreover, since the model-checking problem is decidable for finite state systems [CES83, Eme90, CPS93, SC93], like service logic graphs, checking whether a service logic graph satisfies a certain property can even be done fully automatically without any interaction of the end-user.

Of course, the validation features provided by the Service Definition Environment cannot guarantee the correctness of the whole Web application, since they focus on the process layer. But they are a valuable mechanism to detect errors already in an early phase of the software development cycle. Additionally, in combination with the component model of the Service Definition Environment, the SIBs, the validation features realize the means which allow even non-programmers to build reliable Web applications.

2.2.2.4. Automatic Code Generation

After the design of the workflow has been finished and validated, the Service Definition Environment is able to generate the implementation of the application's coordination layer out of the service logic graph on the basis of the provided SIBs. The implementation of the corresponding compiler depends of course on the application domain in which the Service Definition Environment is used. With respect to the Web, the offered compiler generates a Java Servlet [HC98, Servlet] which implements the process layer of the intended Web application. Since the code generation is performed automatically, there are only two possible classes of errors:

- process errors, which can be detected using the validation features, and
- errors located in the implementation of the business classes.

In fact, as we will see in the next section, the coordinating workflows can be put into practice without introducing errors.

2.2.3. Component Deployment and Packaging

A Web application which has been built using the Service Definition Environment is implemented by three kinds of components:

- the servlet which implements the service logic graph,
- the components implementing the SIBs and
- the components implementing the business classes.

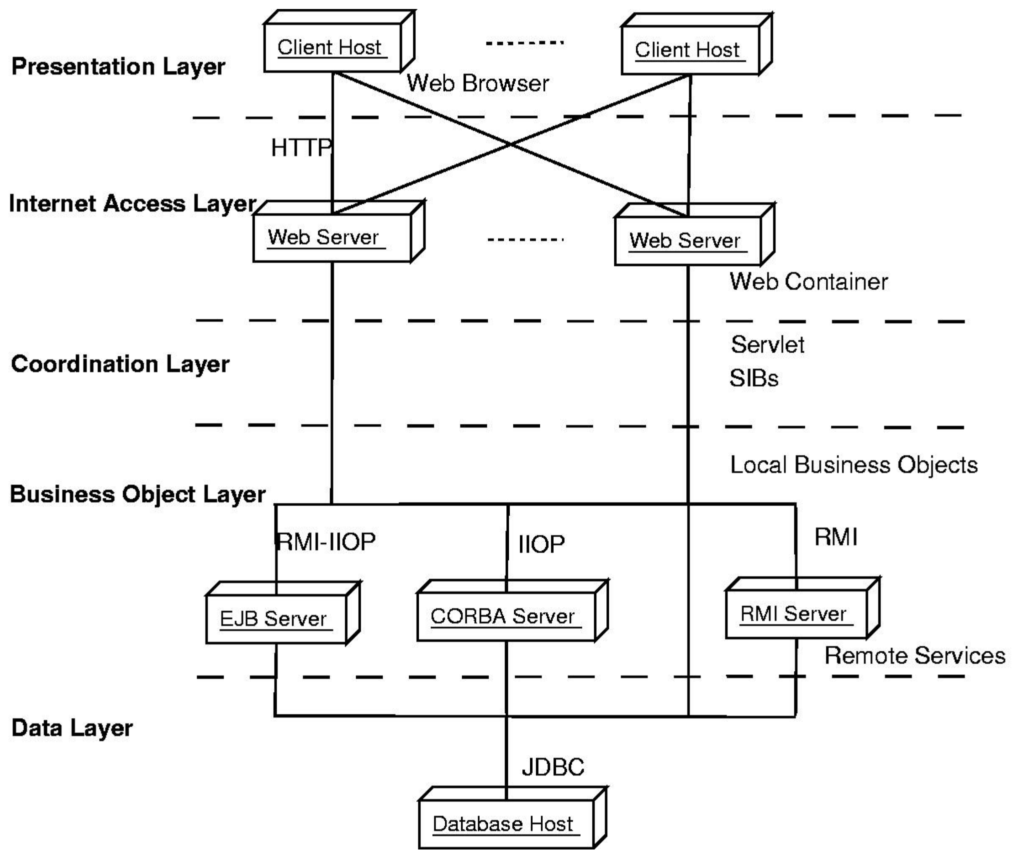
The servlet and the components implementing the SIBs are always executed on the host running the Web container which is located in the Internet access layer of the Web application.

Concerning the components implementing the business classes, the Service Definition Environment places no restrictions on their location. Since the SIBs are realized by Java classes, their implementation

- can be based on local components, i.e. components located on the same host running the Web container.
- can connect to remote services (like Enterprise JavaBeans [Mon00, EJB], Java Remote Method Invocation (RMI) [Dow98, RMI] and CORBA [Sie00, CORBA] services) to perform their task.

Figure 2-10 shows an overview of the deployment of the components implementing the Web application. Note that the Web container, the servlet, the SIBs and the local business objects are all physically located on the same Web server, but belong to different logical layers.

Figure 2-10. The Deployment of the Components implementing the Web Application



The packaging and the installation of the local software components is also performed by the Service Definition Environment. The corresponding process which depends on the available infrastructure, e.g. the kind of the Web container, can be flexibly configured. In particular, the Service Definition Environment supports the creation of Web Archive (WAR) files which are the J2EE standard for packaging the resources needed by a Web application.

The packaging and the installation of the components implementing the remote services can currently not be performed by the Service Definition Environment. It has to be done using other tools before the Web application which uses these services can be made available.

Chapter 3. Contributions and State of the Art

3.1. Contributions

Main contribution of this thesis is a new process for application development which puts the application expert into the center. Application experts

1. can easily compose coarse-granular components to applications,
2. are controlled by formal methods during the composition process,
3. can generate code on the basis of a specification of the components' interaction, and
4. can automatically deploy the application within a specifically designed runtime environment

within an Internet-based infrastructure.

This goal to software development requires a strict architectural separation of

- elementary services which provide the basic functionality of the application,
- coarse-granular components wrapping the elementary services into building blocks on the level of the application expert's understanding, and
- coordinating workflows which specify the behavior of the application on the level of the components' cooperation.

Our approach extends "traditional" software development, in the sense that the application's coordination layer is not coded using programming languages like Java [CW98, Java] or C++ [Str97]. Instead it is graphically designed using coordinating workflows, which afterwards can be validated, compiled into Java or C++ source code and put into operation by the application expert himself. The implementation of the elementary services and the components is performed by system developers using standard techniques and tools e.g., as described in Section 3.2.3.

The process is built on top of METAFrame's integration architecture [Cla97] including the synthesis tool [SMF93], the Service Definition Environment, as well as public developments like Java and Web technology like SUN's Java 2 Enterprise Edition (J2EE) [J2EE] and the Jakarta [Jakarta] project. These technologies are integrated into an entire framework which comprises:

- a component model representing functionalities on the level of the application expert's understanding,
- a coordinating workflow model specifying the components' interaction based on extended finite automata,
- a tooling with an intuitive graphical user interface offering the seamless integration of formal methods to control the component composition,
- a compiler which generates Java source code out of the coordination model,
- a deployment tool which packages the application and deploys it in an appropriate runtime environment, and
- a runtime environment which, in particular, hides details of process distribution, hardware configuration and server locations from the application experts.

This thesis also presents two applications of the proposed approach to software development: the Electronic Tool Integration Platform and a development environment for reliable Web applications.

The Electronic Tool Integration Platform

The Electronic Tool Integration (ETI) Platform, in particular the ToolZone software, allows users to build complex, heterogeneous software tools without any programming knowledge in an Internet-based, secure, performant, and failsafe environment:

1. Users combine components called ETI activities, which are implemented on the basis of a set of pre-installed software tools.
2. Component composition is either performed manually using the High Level Language or graphically using coordination sequences, the coordinating workflow model of the ETI platform.
3. Users are guided during the composition process by ETI's synthesis service which generates coordination sequences out of goal-oriented, loose specifications.
4. Users easily execute HLL programs and coordination sequences on a remote server farm.

A specifically designed runtime environment which is part of the ETI platform takes care of the inter-tool communication, and the details of the hardware and software requirements when executing a customized tool.

A Development Environment for reliable Web Applications

The Web development environment, in particular, the extension of the Service Definition Environment enables an application expert to build reliable Web applications and to put them into operation without any knowledge of Java and server technology:

1. Application experts easily build applications on the basis of components called Service-Independent Building Blocks (SIBs).
2. The cooperation of the SIBs is modeled in terms of service logic graphs, the coordinating workflow model of the Service Definition Environment.
3. Application experts are supported during the design of the service logic graph (beside others) by model checking which guarantees the global consistency of the SIBs' cooperation.
4. Application experts graphically generate and package the Web application, and finally deploy it in a Java-based runtime environment.

The runtime system of the Web development environment uses Java Servlet technology [HC98, Servlet] in combination with a J2EE compliant Web container like the tomcat container [tomcat] of the Jakarta project.

3.2. State of the Art

This section discusses state of the art technology and projects. It is split into three subsections which cover

- technology which is related to our software development process (see Section 3.2.1),
- projects which are related to the ETI platform (see Section 3.2.2), and
- technology for Web application development (see Section 3.2.3).

3.2.1. Component Models and Process Modeling Tools

"Standard" Component Models

Our application expert-centric development process defines a component model which is on the application expert's level of understanding. In consequence is does not compete with defacto standards like JavaBeans [JavaBeans], Enterprise JavaBeans [Mon00, EJB] and the Microsoft (D)COM model [Rog97, COM], since these models address developers, but not application experts. In fact our technology is built on top of these well-known component models: our development process uses the above-mentioned widespread models in combination with interoperability services like CORBA [Sie00, CORBA] to implement the elementary services of the applications.

Component Frameworks

The component frameworks we discuss in this section can be classified in frameworks which can handle generic components and frameworks offering a customized component model. The first class of frameworks can be used in several application scenarios. In contrast to that, frameworks providing a customized component model are typically restricted to a specific application domain.

Frameworks supporting generic components

Component frameworks like the SanFrancisco framework of IBM [SanFrancisco] or, more generally, the WebSphere Business Components product family [WBC] as part of the IBM framework for e-business [IBMebusiness] typically offer:

- a set of low-level components (e.g. business classes implementing things like company, address, currency, business partner, unit of measure, cash balances),
- coarse-granular components implementing common business processes such as general ledger, order processing, inventory management, product distribution, and often
- a runtime environment in which applications built with the framework are executed.

Developers can then extend the framework by adding new low-level and coarse-granular components, and afterwards build applications on top them. For example, application development within the SanFrancisco framework is based on standard tools like Rational Rose [Rose] and Java development environments like Borland JBuilder, and IBM VisualAge for Java [vajava] (see [SanFranciscoAD]). Applications using WebSphere Business Components are realized with the WebSphere Business Components Composer [WBCC]. All frameworks of this class address developers and are not adequate for application experts without a deep programming knowledge.

Frameworks supporting customized component models

Component frameworks like WaterBeans [WaterBeans] and Sally [SBFMMS98, Sally] are customized for a specific application domain, i.e. water quality modeling and signal processing, respectively. They offer

- a customized component model,
- a tool to graphically build applications on the basis of the components as well as
- a runtime environment for application execution.

Characteristic to these approaches is that they focus on the data flow between the components. This means that the components have several typed input ports and output ports, and they “fire” when all data is available. The control flow is described implicitly and in a distributed fashion. This Petri-Net like execution semantics is difficult to comprise and inadequate for the application experts we have in mind. In addition, the validation facilities offered by the frameworks are limited to local type compatibility.

Generic Coordination Approaches

The previous sections presented approaches where either

- generic components are composed using standard development environments (see e.g. the WebSphere Business Components), or
- more customized components are graphically coordinated (see e.g. Sally).

All techniques have in common that they focus on the components rather than on component coordination. Projects like ToolBus (see Section 3.2.2), or coordination languages like STL++ [SCH99, STLPP] and Linda [CGM93, Linda] concentrate on the communication between components (see also the Coordination conference series on coordination models and languages, e.g. [Coordination2002]). Similar to process modeling tools their strength is the specification of complex, distributed systems, which is, on this level, far beyond the skills of the application experts we address.

Process Modeling Tools

Advanced process modeling tools like LEU [DGSZ94] or projects like MOKASSIN [mokassin] focus on modeling complex processes as well as process cooperation and distribution typically on the basis of Petri-Net based software modeling languages. They are designed to support software development that requires to model at this level of detail, but they are far to complicated for the people we address. Our goal is to enable application experts to build reliable applications in a controlled and easy way without concerning about architectural and distribution aspects. This requires

1. a coordinating workflow model which is on the application expert's level of understanding, and which can automatically be checked for implementability and guaranteed service properties, and
2. to hide hardware and software details as well as process distribution and communication issues within the components' implementation or an (appropriate) runtime environment.

Even if process modeling tools are used in a restricted way where only a single process (no process cooperation and no distribution) is specified, source code generation is only possible, if the complete system is (hierarchically) modeled within the tool. Typically, there is no support for code generation on the basis of coarse-granular components whose implementation is done outside the modeling tool.

3.2.2. Projects related to the ETI Platform

We do not know of any project combining the features offered by the ETI platform, which enables users to

- search for software tools using syntactic criteria as well as abstract properties specifying the profile of the tool they are interested in,
- execute single tool features remotely,
- combine features coming from heterogeneous software tools to complex applications and
- execute the applications via the Internet.

Projects related to the ETI platform can be classified by Web sites giving access to tools and tools which focus on tool coordination.

Web Sites giving Access to Software Tools

Link Collections and Software Archives:

This class of Web sites provides passive access to software tools. They

- offer links to software tools in a specific application domain, like the Petri Nets Tool Database [PNTD], the Formal Methods Europe [FME] database or the UM weather software library [UMW], or
- give direct access to the software in source code or in binary format like the comp.simulation software archive [CSSA], the download.com site [DownloadCOM], etc.

Users which access software tools via these sites are still confronted with the full download and installation burden. ETI sites overcome these burdens by giving access to a repository of pre-installed software tools with remote execution and coordination facilities.

Web sites providing execution facilities of a single tool:

Some Web sites like the **HyTech** [HyTech] home page or the **smv** guided tour [SMV] give remote access to a specific tool functionality via an HTML form. In contrast to ETI sites, they are restricted to a single tool and do not support a fair evaluation of different tools.

Project focusing on Tool Coordination

PROSPER [DCNB00, Prosper]:

The **PROSPER** (Proof and Specification Assisted Design Environments) toolkit provides an infrastructure based on the HOL98 theorem prover [GM93]. Existing verification tools can be integrated into nearly any application (like CAD and CASE tools), thus extending this application by a new validation feature. For this, verification tools are encapsulated into components called plugins. Afterwards, system developers may implement customized verification procedures on the basis of the plugins' features at the level of ML programming [MTHM97]. In contrast, ETI offers programming-free tool coordination facilities and it supports the tool coordination process by a synthesis service which in particular takes care of incompatible tool compositions.

ToolBus [BK96, KO96, ToolBus]:

> Closest to our approach is the **ToolBus** project since it offers a coordination infrastructure as well as a language for process communication. The cooperation of processes is specified via a process-algebra based scripting language. ToolBus scripts specify the behavior of each process as well as their interaction with each other and with external tools. Similar to process modeling tools (see Section 3.2.1), this approach to software development is very well suited for (process algebra) experts to solve complex tool coordination task, but it is far to complicated for the people we address.

3.2.3. State of the Art Web Application Development

Current state of the art Web development is based on SUN's Java 2 Enterprise Edition (J2EE) technology or Microsoft's .NET initiative [TL01, NET]. Both approaches define several component models and a runtime environment which provides inter-component communication, transaction and communication services. Application development is performed by

1. developing the components using standard tools like Borland's JBuilder [JBuilder], the Together ControlCenter [Together] or Microsoft's Visual Studio [VisualStudio], and
2. deploying and configuring the components within an implementation application server like BEA WebLogic Server [WebLogic], Borland AppServer [AppServer], IBM WebSphere [WebSphere], or Microsoft .NET Enterprise Servers [MSNetServers].

Our approach uses standard development tools and J2EE technology to implement the elementary services of the Web application. But since this requires expertise that is far beyond the skills of the application experts, our approach wraps this technology in order to hide all the technical details. This enables application experts without knowledge of the component models, or distribution and interoperability facilities to graphically build reliable Web applications in a controlled manner.

Within the traditional scenario, the implementation of the graphical user interface (GUI) of the Web application is typically performed using Java Server Pages (JSPs) [JSP] or standard HTML with integrated scripting. In the later case, HTML designers use tools like Macromedia Dreamweaver [Dreamweaver] and FrontPage [FrontPage] in combination with Velocity [Velocity] or WebMacro [WebMacro] scripts. Our approach to Web application development is based on standard HTML with integrated Velocity scripting to realize the GUI of the application. JSPs are not adequate for our purpose, since they focus on a document view of the application (instead of a control flow view) and they too often require Java programming skills.

II. The ToolZone Software

Chapter 4. Introduction

The purpose of the ToolZone software (see Section 2.1 for an overview) is to give an end user Internet-based, secure, performant and failsafe access to the activities and types available in the tool repository with the ability to

1. get detailed information on each activity and type,
2. execute single activities,
3. combine activities to programs, and
4. run these programs via the Internet.

This part of the thesis presents the ToolZone software from several points of view corresponding to the roles introduced in Section 1.3.3.3. First, the end user's view is illustrated in Chapter 5. Afterwards we go into the details of the design of the ToolZone software in Chapter 6 (platform developer's view). Chapter 7 then focusses on the tool-integration process which makes new activities and types available in the tool repository.

But before we present the ToolZone software in detail, we give a short introduction to two software tools which are used to illustrate the presented ideas. The tools are taken from the instantiation of the ETI platform customized for the *Springer International Journal on Software Tools for Technology Transfer* (STTT) [STTT]. This ETI site, which can be found at eti.cs.uni-dortmund.de, provides tools focusing on the analysis and verification of distributed (real-time) systems.

4.1. Sample Tools

In this part of the thesis we will give several examples which should help the reader to understand the concepts. They are based on the

- the Caesar/Aldebaran Development Package (CADP) [FGK96, CADP] (see Section 4.1.1) and
- the Hybrid Technology Tool (HyTech) [HHW97, HyTech] (see Section 4.1.2).

4.1.1. The Caesar/Aldebaran Development Package

CADP is a "Software Engineering Toolbox for Protocols and Distributed Systems" [CADP]. It provides tools for the simulation, testing and verification of systems specified in the ISO [ISO] language LOTOS [BB87, LOTOS]. In addition to other tools, it ships with **aldebaran**, a tool for minimizing and comparing (edge-)labeled transition systems (LTSs) [BFKM97]. One of the file formats which can be used to store LTSs is the `aut`-format.

Each `aut` file starts with a header line, which stores an integer number representing the start node of the LTS as well as the number of the nodes and edges contained in the LTS (see **(1)** in Example 4-1). The subsequent lines represent the edges contained in the LTS (see e.g. **(2)** in Example 4-1). They are triples of the form of

(n_1, a, n_2),

where n_1 is an integer number representing the start node of this edge, a is the information associated to this edge, and n_2 is an integer number representing the end node of this edge. Example 4-1 shows a specification of a simple LTS in the `aut`-format, which contains four nodes and four edges.

> **Note:** Within the `aut`-format, information can only be associated to edges. No node information can be stored.

Example 4-1. An `aut`-format Specification of an LTS

```
des (0,4,4) (1)
 (0,a,2)    (2)
 (0,a,1)
 (2,b,3)
 (1,b,3)
```

Since most of the tools provided by the toolbox are command-line tools, i.e. they offer no graphical user interface (GUI) to access the offered functionality, CADP ships the **xeuca** application [Gar96] which allows to uniformly access the features implemented by the command-line tools via a TCL/TK-based [Wel97] GUI.

4.1.2. The Hybrid Technology Tool

The purpose of the HyTech tool is documented in [HyTech] as follows: “HyTech is an automatic tool for the analysis of embedded systems. HyTech computes the condition under which a linear hybrid system satisfies a temporal requirement. Hybrid systems are specified as collections of automata with discrete and continuous components, and temporal requirements are verified by symbolic model checking. If the verification fails, then HyTech generates a diagnostic error trace”. Hybrid systems which can be analyzed using HyTech are stored in files being conformant to the `hy`-format. A `hy`-file is split into three sections:

1. The *declaration section* provides an entry for each variable which is needed to define the automata of the system.
2. The *automata section* specifies each hybrid automaton which is part of the system to analyze.
3. The *commands section* defines the analysis which has to be performed by the HyTech tool on the given system.

A detailed description of this format can be found in the HyTech user guide available at [HyTech].

Chapter 5. End User Aspects

Since the end user is primarily confronted with the graphical user interface of the ToolZone client, we begin with a brief history of its implementation. This illustrates the meanders of the ToolZone client which were induced by the evolution of the Java technology [Java].

When we started to plan the first version of the ETI platform in 1996, we wanted to give access to its features via remote login on the server machine. Though this access method is available on a wide range of operating systems and hardware architectures, it would not be accepted by the users since the user interface would have been textual. This is of course not intuitive, especially since working on directed graphs representing labeled transition systems is one of the main features of ETI's first application domain, the analysis and verification of distributed (real-time) systems. Additionally this solution would not scale with a huge number of users, since there must be a real account maintained on the server machine for every user.

Fortunately these were the days when Java came up. First released in 1995, Java is an object-oriented programming language which "itself is closely based on C++, but simplified somewhat and with automatic garbage collection added. In addition it has been made so that it is completely architecture neutral, with the quirks of the particular platform being handled by the interpreter."[JavaPress]

Believing in the promises of its inventor, SUN Microsystems Inc. [SUN], we implemented the first version of the ToolZone client as an applet with the Java Development Kit (JDK) version 1.0.2 using the Java Abstract Window Toolkit (AWT) classes to build the GUI. "An applet is a program written in the Java programming language that can be included in an HTML page, much in the same way an image is included. When you use a Java technology-enabled browser to view a page that contains an applet, the applet's code is transferred to your system and executed by the browser's Java Virtual Machine (JVM)." [Applets] But testing the applet on Java technology-enabled Web browsers from different vendors having different versions was more a "Write once, runs only on the browser of a specific vendor having a dedicated version number" experience rather than SUN's promoted "Write Once, Run Anywhere" slogan. The main problems we faced were the different implementations of the AWT classes and the security policy within the browsers' JVMs.

Even SUN admitted these problems and launched the first version of the Java Activator (today known as the Java Plug-in [JavaPlugIn]) a few month before the initial ToolZone software release was planned. This Web browser plug-in disables the JVM shipped with the browser and redirects the execution of an applet to the virtual machine bundled with the plug-in. By this the Java virtual machines available to run the applet became more compatible. Consequently, to get rid of the browsers' vendor and version problems we adapted the ToolZone client to be plug-in conform, and finally released and presented the first version in March 1998.

But the experience showed that only a few users were willing to download and install a nearly 11 MByte browser plug-in just to get an impression of the ToolZone software. Additionally, the AWT based GUI had several drawbacks, and we observed some performance problems when displaying huge graphs. Consequently we started to refactor the software in 1999.

Today, after the Java technology has been improved and we have adapted the software several times to the updated versions of the JDK, the ToolZone client runs as a Java application (not as an applet anymore) which has an intuitive Swing-based GUI [WC99, Swing]. The installation of the ToolZone client is done fully automatically by using the Java Web Start application launcher [JavaWebStart]. The current ToolZone client implementation has been tested on a wide range of platforms which support the Java 2 runtime environment. See [JavaPorts] for information on available ports.

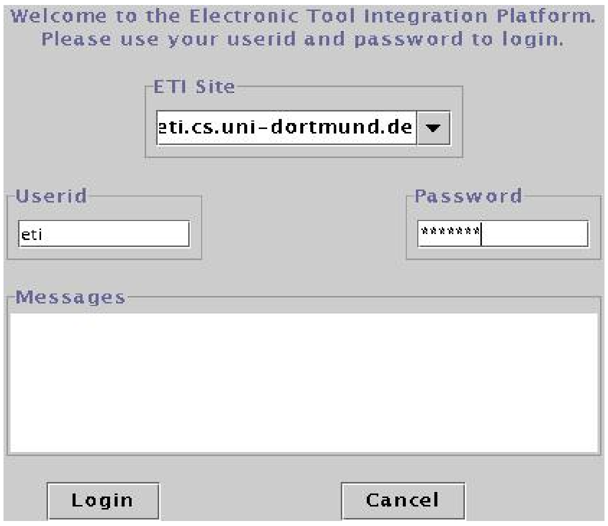
In the rest of this chapter we show some client's features and give some information on their implementation. For this, we start with the login process in Section 5.1. Here we illustrate the responsibilities and the collaborations of the software components introduced in Section 2.1.5. Afterwards, Section 5.2 gives an overview of the features provided by the ToolZone client. This is done by presenting a guided tour which shows a typical end-user session.

5.1. Logging In

To provide Internet-based, performant and failsafe access to the tool repository, the ToolZone software is deployed on four logical layers (see also Figure 2-6): the presentation layer, the Internet access layer, the feature layer and the data layer.

The client host located in the presentation layer runs the ToolZone client software. The ToolZone client gives remote access to the features offered by the ToolZone software. To use the features offered by the ToolZone software, the user must login to an ETI site. For this, he must specify the ToolZone host to connect to, and a user id and password valid for this site. He can supply this information using the login screen shown in Figure 5-1. This screen appears when the ToolZone client is started.

Figure 5-1. The Client Login Screen



Welcome to the Electronic Tool Integration Platform.
Please use your userid and password to login.

ETI Site: eti.cs.uni-dortmund.de

Userid: eti

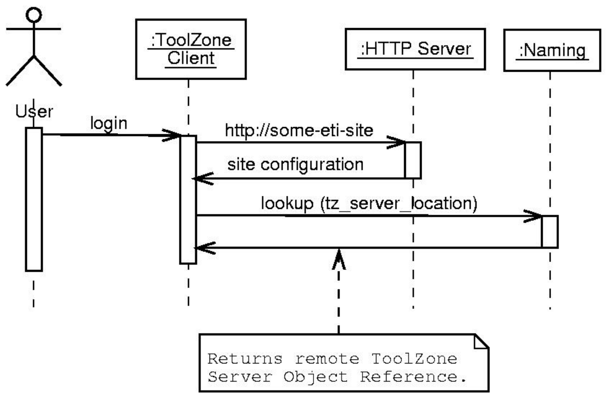
Password: *******

Messages

5.1.1. Initializing a Session

As soon as the user has entered the required data and has pressed the Login button, the client contacts the specified ToolZone host via the Hypertext Transfer Protocol (HTTP) [HTTP] to download a file characterizing the configuration of the chosen ETI site (see Figure 5-2). Beside others, the file contains information necessary to configure the client GUI properly. The GUI specific information is mainly used to build the File menu of the ETI Shell Window (see Figure 5-4). The items shown in the New Graph and New System sub-menus depend on the types available in the tool repository hosted by the chosen ETI site (see Section 5.2.3). In addition to the GUI information, the configuration file provides the location of the site's ToolZone server, which is responsible to accept the initial connection from the client. The location of the server component is specified as a URL which conforms to the Java Remote Method Invocation (RMI) format [Dow98, RMI]. Via this URL, the client obtains a reference to the ToolZone server object located on the ToolZone host. For this it uses the `Naming` class contained in the `java.rmi` package of the Java 2 runtime environment. Figure 5-2 shows the first phase of the login process.

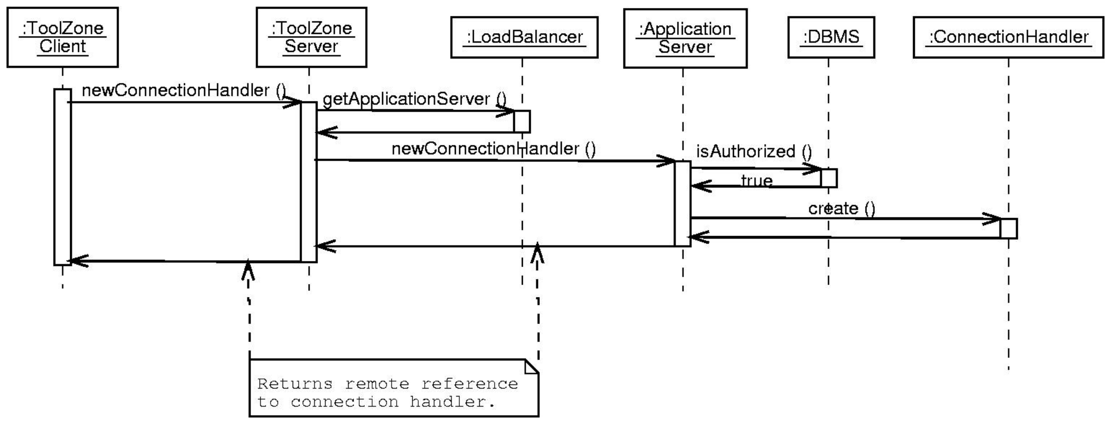
Figure 5-2. The first Phase of the Login Process



After the client has got the reference to the remote ToolZone server object, it connects to the server via Java RMI to obtain a connection handler object, which is responsible to handle the session of this client.

This connection handler object is obtained as follows (see Figure 5-3): First, the ToolZone server accepts the `newConnectionHandler` request by the ToolZone client. Then it delegates it to an application server running on one of the ETI site's application hosts (see Figure 2-6). The appropriate application server is selected by the ToolZone server using a load balancer. This software component ensures that incoming client requests are distributed on the site's application hosts to share the available computing power amongst the connected clients in an optimal fashion. In the current version of the ETI platform a simple load balancing strategy has been implemented which just selects the "next" application server based on a fixed server enumeration (round robin strategy).

Figure 5-3. The second Phase of the Login Process



The application server is then responsible for authoring the request by validating the passed user id and password. For this, it uses the database management system (DBMS) running on the database host. Here the database is located which stores the user profiles. If the check is successful, the application host creates a new connection handler object and returns a remote reference to it back to the client.

Once the user is successfully logged in, the ToolZone server object will not be connected by this client again. Further client requests will then be processed by the connection handler.

If the client has successfully logged in to the ETI site, the ETI Shell Window is shown (see Figure 5-4) which gives access to the ToolZone client functionality (see Section 5.2 for further details). Otherwise, a message will be displayed in the Messages text area of the client login screen (see Figure 5-1) giving information and possible solutions to the problem.

5.1.2. Client-Request Handling

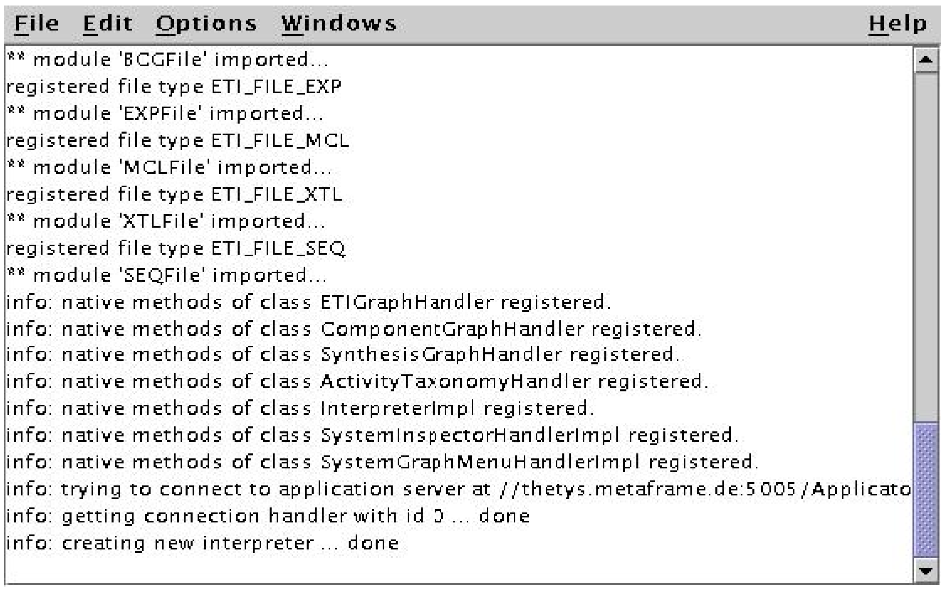
As already mentioned, the session of a client is maintained by the connection handler object returned by the ToolZone server during the login procedure. But the connection handler does not provide the implementation of the ToolZone-software features. In combination with other objects documented in Chapter 6 the connection handler is only the mediator between the client and the tool management application (see Section 6.1.1). For every session which is created a corresponding instance of the tool management application is started on the application host. An instance of the application then provides the implementation of the ToolZone-software feature for the associated client.

The tool management application is based on the METAFrame environment introduced in Section 1.4.1. It gives uniform access to the activities and types available in the tool repository, regardless whether the tool functionality is available locally on the application host or remotely on a remote tool host. In the current status of the project remote tools can be accessed via the interoperability services RMI [Dow98, RMI] and CORBA [Sie00, CORBA].

5.2. A User Session

After a successful login, all features provided by the ToolZone client can be accessed via the ETI Shell Window shown in Figure 5-4.

Figure 5-4. The ETI Shell Window



```
** module 'BCGFile' imported...
registered file type ETI_FILE_EXP
** module 'EXPFile' imported...
registered file type ETI_FILE_MCL
** module 'MCLFile' imported...
registered file type ETI_FILE_XTL
** module 'XTLFile' imported...
registered file type ETI_FILE_SEQ
** module 'SEQFile' imported...
info: native methods of class ETIGraphHandler registered.
info: native methods of class ComponentGraphHandler registered.
info: native methods of class SynthesisGraphHandler registered.
info: native methods of class ActivityTaxonomyHandler registered.
info: native methods of class InterpreterImpl registered.
info: native methods of class SystemInspectorHandlerImpl registered.
info: native methods of class SystemGraphMenuHandlerImpl registered.
info: trying to connect to application server at //thetys.metaframe.de:5005/Applicato
info: getting connection handler with id 0 ... done
info: creating new interpreter ... done
```

To present an overview of the ToolZone-software functionality, the next two sections (Section 5.2.1 and Section 5.2.2) are organized along the typical workflow which is performed by an end user (see also [MBS98, BKMS99]):

1. Browsing through the available activities (see Section 5.2.1).
2. Executing single activities in stand-alone mode (see Section 5.2.1).
3. Combining activities to programs (see Section 5.2.2).
4. Executing the coordination programs (see Section 5.2.2).

Note: A complete user manual can be found in [TZUser].

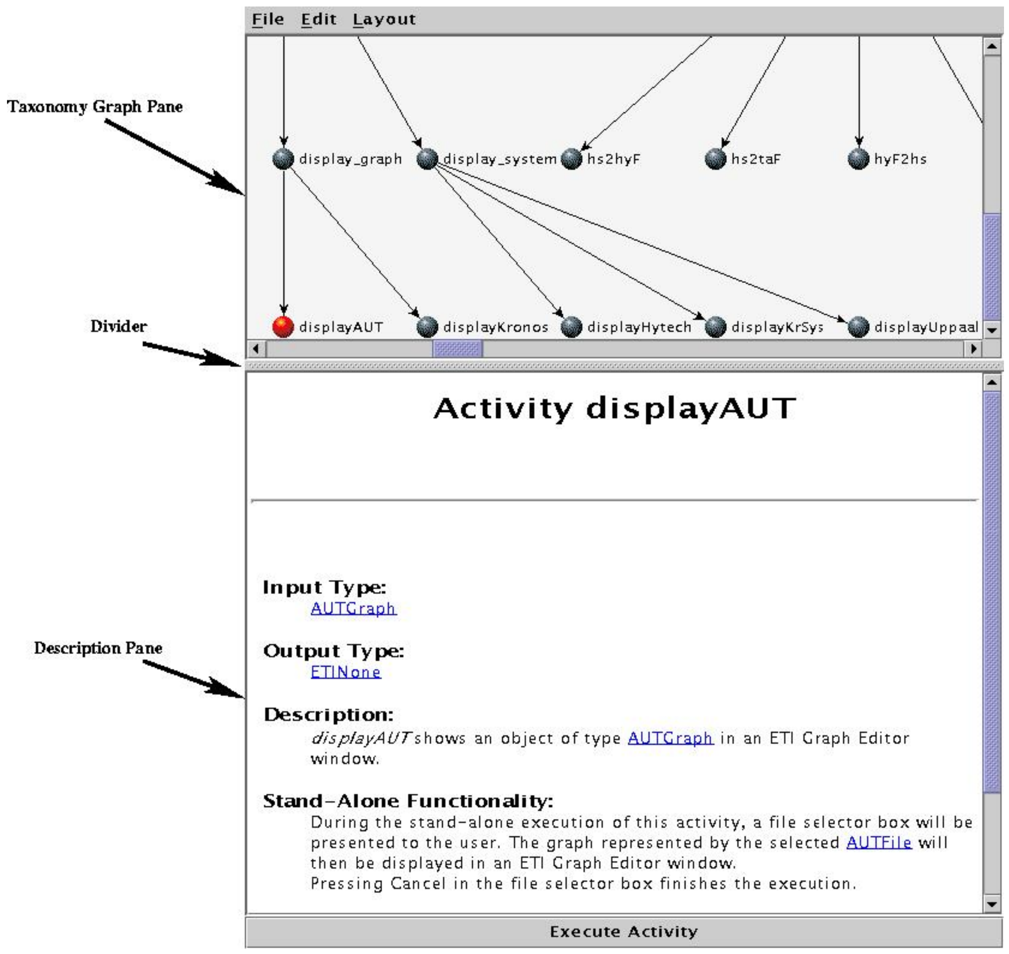
Afterwards, Section 5.2.3 shows how the data, which is needed to execute the activities, can be accessed via the ToolZone client.

5.2.1. Browsing through the available Activities

When the user connects to an ETI site, he wants to get an overview of the activities and types which are available in the corresponding tool repository. As already mentioned in Section 2.1.2 the activities and types are organized in the *ETI Activity Taxonomy* and *ETI Type Taxonomy*, respectively. In the rest of this section we will focus on the activity taxonomy. The type taxonomy can be accessed analogously.

To visualize the activity-taxonomy DAG, the user obtains the Activity Taxonomy window (see Figure 5-5) by selecting Options⟶Activity Taxonomy (**Ctrl-A**) in the ETI shell main menu.

Figure 5-5. The Activity Taxonomy Window



Activity displayAUT

Input Type:
AUTGraph

Output Type:
ETINone

Description:
displayAUT shows an object of type AUTGraph in an ETI Graph Editor window.

Stand-Alone Functionality:
During the stand-alone execution of this activity, a file selector box will be presented to the user. The graph represented by the selected AUTFile will then be displayed in an ETI Graph Editor window.
Pressing Cancel in the file selector box finishes the execution.

The activity taxonomy window is split up into two parts: the taxonomy graph pane and the description pane (see Figure 5-5). By dragging the divider that appears between the two panes, the user can specify how much of the window's total area goes to the graph or the description pane.

The taxonomy graph pane of the window shows the ETI activity taxonomy represented as a directed acyclic graph (DAG). Here, leaf nodes model available atomic activities. The other nodes denote activity groups, which represent all atomic activities that are reachable by following the edges within the taxonomy DAG which start at this node (see Section 2.1 for more information on the taxonomies).

The description of an atomic activity or activity group can be obtained by selecting the corresponding node in the graph pane. In this case the associated documentation will be displayed in the description pane of the Activity Taxonomy window.

If the chosen node represents an atomic activity, i.e. it is a leaf node of the activity taxonomy DAG, the Execute Activity button located at the bottom of the window will be enabled, and the associated activity can be executed by selecting this button. If the button is pressed by the user, the tool management application executes the chosen activity in stand-alone mode. The associated behavior is specified by the stand-alone constituent of this activity (see Section 2.1.1). It is documented in the Stand-Alone Functionality section of the activity description (see Figure 5-5).

Similar to the activities, their input and output types are organized in the *ETI Type Taxonomy*. The corresponding Type Taxonomy window can be obtained by selecting Options⟶Type Taxonomy (**Ctrl-T**). Note that the Type Taxonomy window does not offer an Execute button, since only activities can be executed.

5.2.2. Combining Activities

As documented in Section 5.2.1 *single* activities can be executed via the Activity Taxonomy window shown in Figure 5-5. But the real power of the platform is the *combination* of activities. This can be done either by

- combining activities manually using the coordination language HLL (see also Section 2.1.3) or by
- automatic generation of coordination sequences from abstract coordination-task descriptions, called *Loose Specifications* (see also Section 2.1.4).

In the following two sections, we focus on the usage of the synthesis feature provided by the ToolZone software. For this, we present in Section 5.2.2.1 the syntax and the semantics of the coordination formulae which are used to formalize the loose specifications. Afterwards Section 5.2.2.2 illustrates how the synthesis functionality can be accessed via the ToolZone client.

Using ETI's synthesis component, synthesis-compliant activities (see Section 2.1.1) can be combined to *Coordination Sequences*. Conceptually, a coordination sequence is a finite path of the form of

$$T_1 \xrightarrow{a_1} T_2 \xrightarrow{a_2} T_3 \dashrightarrow T_n \xrightarrow{a_n} T_{n+1}$$

where T_i is an atomic type and a_i is an atomic activity which transforms an object of type T_i into an object of T_{i+1}. This means that two atomic activities a_i and a_j can only be combined directly, if the output type of the activity a_i is the same as the input type of the activity a_j or if the output type of the activity a_j is the same as the input type of the activity a_i.

> **Note:** In the context of the ETI project, every coordination sequence starts with the type ETINone and it ends with the type ETIResult. These two special types ensure that the coordination sequences are well-defined, in the sense that the execution of a coordination sequence always starts with “nothing” and ends with a result which can be shown graphically on the screen of the client host.

5.2.2.1. Writing Coordination Formulae

Coordination sequences are generated out of abstract descriptions (called *loose specifications*) using the synthesis component of the METAFrame project (see [SMF93] for information on the implemented algorithm). The loose specifications are defined by *coordination formulae* whose syntax is derived from the Semantic Linear-time Temporal Logic (SLTL) (see also [SMB97]) .

Syntax

The actual syntax of the coordination formulae is defined by the BNF shown in Table 5-1. Here `T` and `A` are identifiers contained in the type taxonomy and activity taxonomy, respectively.

Table 5-1. The Syntax of the Coordination Formulae

f	::=	f_{type} |	$f_{activity}$ |
		(f & f) |	(f | f) |
		F (f) |	G (f) |
		(f $<$ f) |	F {f, ..., f}
f_{type}	::=	T |	~(f_{type}) |
		(f_{type} & f_{type}) |	(f_{type} | f_{type})
$f_{activity}$	::=	A |	~($f_{activity}$) |
		($f_{activity}$ & $f_{activity}$) |	($f_{activity}$ | $f_{activity}$)

Semantics

The semantics of a coordination formula is a set of coordination sequences which is determined on the basis of the *coordination universe*. The coordination universe is a directed graph which represents all possible combinations of the available synthesis-compliant activities. It is automatically generated by the synthesis component on the basis of the taxonomies and the input/output-behavior description of the activities.

Using the *synthesis strategy*, the user can define a filter on the solutions he is interested in. The following four strategies are available: all solutions, all shortest solutions, all minimal solutions, one shortest solution. See Section 2.1.4.3 for a detailed description.

The set of coordination sequences

- which satisfy a given loose specification, i.e. the semantics of a coordination formula, and
- which are conform to the provided synthesis strategy

is represented as a directed acyclic graph called *Synthesis Solution Graph*. In the context of the ToolZone software, each path within this graph which starts at the node ETINone and ends at the node ETIResult represents a solution with respect to the given formula and the chosen synthesis strategy. The left part of Figure 5-7 shows an example of a synthesis solution graph.

Formally, the semantics of a coordination formula is based on the SLTL semantics. But intuitively, it can be explained as follows:

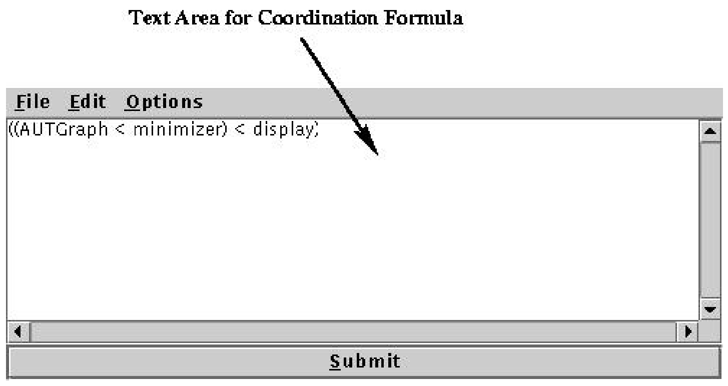
- A type formula f_{type} is satisfied by every path in the coordination universe whose first element (a type) is contained in the set of types specified by f_{type}. The set of types defined by a type formula is determined on the basis of the type taxonomy. For this, the non-atomic elements of the taxonomy (which may be used to build the type formula) are interpreted as sets over the atomic types. In consequence, the semantics of the operators `&`, `|` and `~` are defined by set intersection, union and complement. See Section 2.1.2 for an example.
- An activity formula $f_{activity}$ is satisfied by every path in the coordination universe whose first activity is contained in the set of activities specified by $f_{activity}$. The set of activities defined by a activity formula is determined in analogy to a type formula on the basis of the activity taxonomy.
- The operators `&` and `|` are interpreted in the usual fashion as intersection and union of sets of paths.
- `F (`f`)` (the *finally* operator) is satisfied by every path in the coordination universe which has a suffix satisfying f.
- `G (`f`)` (the *generally* operator) is satisfied by every path in the coordination universe where f is satisfied for every suffix.
- $(f_1 < f_2)$ is defined by the formula `F (`f_1 `& F (`$f_{2)}$`))` which means that it is satisfied by every path in the coordination universe having a suffix satisfying f_1, which itself possesses a suffix satisfying f_2. By this the operator < is in particular a simple means for ordering the occurrences of property satisfaction within a coordination sequence.
- `F {`f_1`, ..., `f_n`}` is defined by the formula `F (`f_1`) & ... & F (`f_n`)` which is satisfied by every path in the coordination universe having a suffix for each of the sub-properties f_i.

5.2.2.2. Experimenting with ETI's Synthesis Feature

To access the synthesis feature, the user must obtain the Synthesis Editor window (see Figure 5-6) by selecting Options⟶Synthesis Editor (**Ctrl-Y**) in the ETI shell main menu. Here he can directly enter the coordination formula which specifies his coordination request using the syntax defined in Table 5-1. Alternatively, he can load a formula into the editor via the File⟶Load (**Ctrl-L**) menu item. The internal types ETINone and ETIResult should never be used when defining a coordination formula. The software automatically extends the formula supplied by the user in a way that the resulting coordination sequence(s) are well-defined.

The synthesis strategy can be selected via the Options menu provided by the Synthesis Editor window. By default the *All Shortest Solutions* strategy will be used.

Figure 5-6. The Synthesis Editor Window

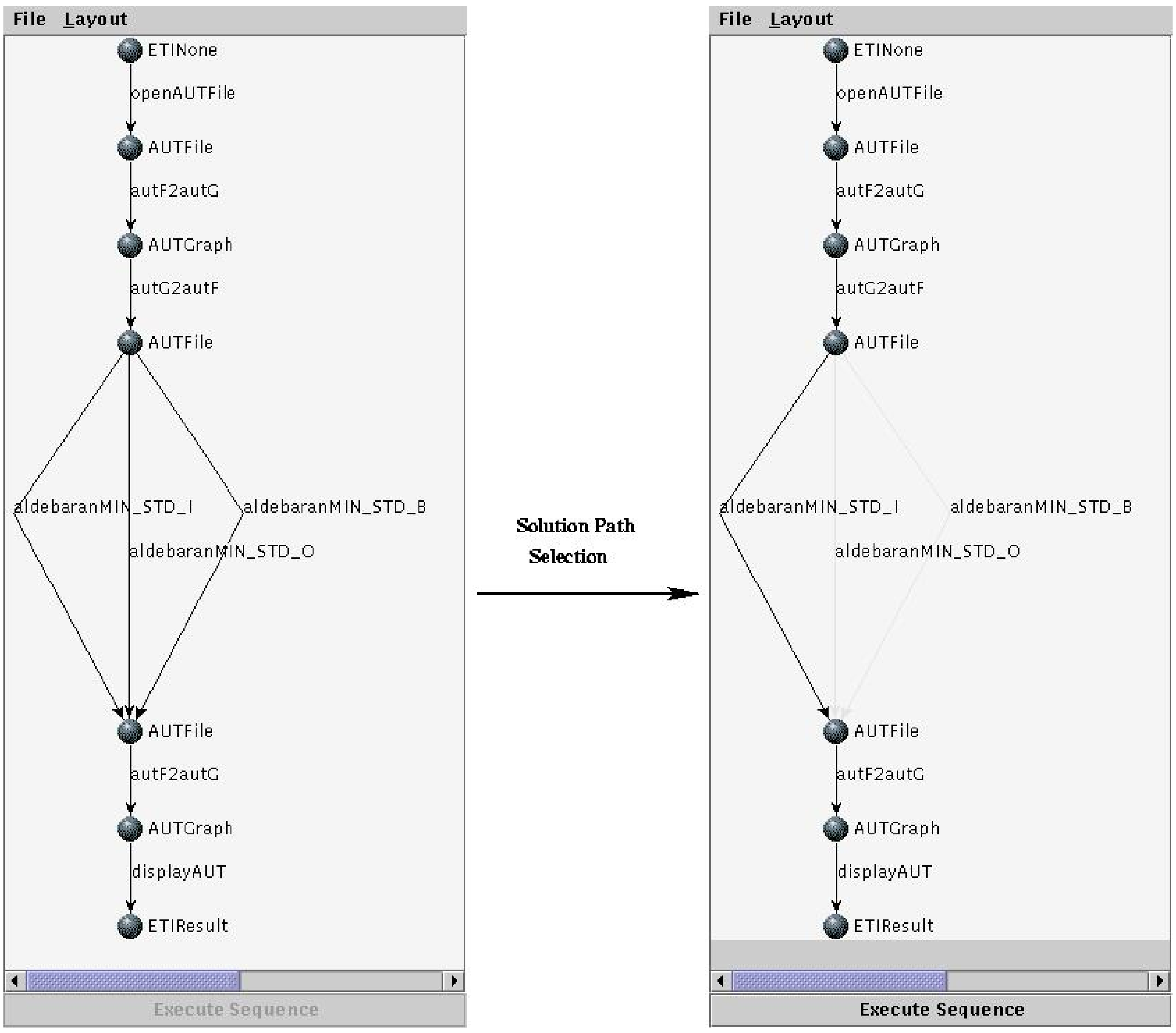


((AUTGraph < minimizer) < display)

The formula `((AUTGraph < minimizer) < display)` shown in Figure 5-6 is taken from a scenario where the the CADP toolkit is available within the tool repository. It specifies the following request: "Within the returned sequences an LTS represented by an `AUTGraph` object should be available. On this LTS a `minimizer` algorithm should be invoked. After that, the result of the minimization should be `displayed` on the screen". Note, that the formula does not specify the exact minimizer and display features to be used, since in this example the identifiers `minimizer` and `display` represent activity groups, and not atomic activities.

When the user submits the coordination request by selecting the Submit button at the bottom of the Synthesis Editor window, the synthesis component provided by the tool management application solves this specification. If a solution can be found, the result of the synthesis invocation is a synthesis solution graph where each path of the graph starting at the node ETINone and ending at the node ETIResult represents a solution with respect to the given formula that is conformant to the selected synthesis strategy. Otherwise, a window will be shown giving the information that no coordination sequence satisfying the formula could be found. The left part of Figure 5-7 shows the synthesis solution graph representing all shortest paths which satisfy the formula shown in Figure 5-6.

Figure 5-7. A Synthesis Solution Graph



With respect to the formula presented in Figure 5-6 the synthesis solution graph shown in Figure 5-7 looks a little bit long winded, in the sense that

1. first an AUTFile object is created,
2. then the AUTFile object is transformed into an AUTGraph object which is
3. finally converted into an AUTFile object again.

But a detailed view on the formula explains this fact. It is explicitly specified in the formula, that an AUTGraph object should be reached before a `minimizer` activity is executed. Thus, the surrounding activities must be automatically inserted by the synthesis component to make the coordination sequences type-correct. The first two activities (`openAUTFile` and `autF2autG`) are introduced since

1. every valid coordination sequence must start with the type ETINone and

2. (with respect to this example) there is only the `autF2autG` activity which transforms an AUTFile object into an AUTGraph object.

The `autG2autF` (third) activity has been added due to the fact that all suitable `minimizer` activities, i.e. `aldebaranMIN_STD_I`, `aldebaranMIN_STD_O` and `aldebaranMIN_STD_B`, can only work on AUTFile objects as their input type, and not on AUTGraph objects. Note that concerning the activity group `display` exactly one alternative (`displayAUT`) matches in this context.

The user can now select his favored coordination sequence within the coordination graph window, and execute it by selecting the Execute Path button as soon as the selection is unique. The selection of the sequence is done by clicking with the left mouse button on the edges that should be contained in the favored solution. As shown in the right part of Figure 5-7, the edges contained in the favored solution remain black, all other edges will be colored in light gray. See [TZUser] for detailed information on the selection process.

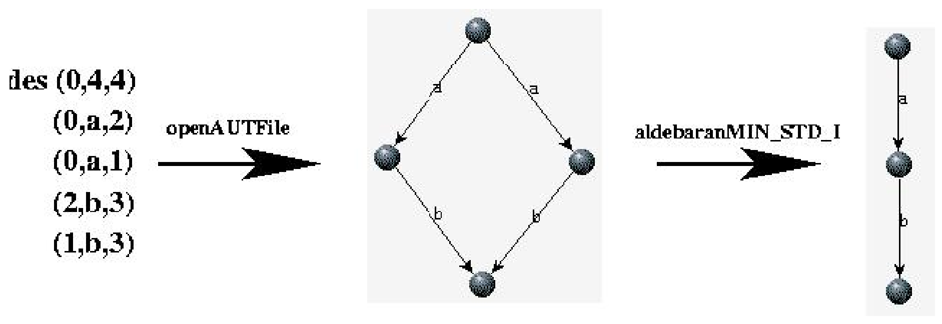
When the user executes the coordination sequence, the tool management application runs the corresponding activities in tool-coordination mode. The associated behavior is specified by the tool-coordination constituent of the activity (see Section 2.1.1).

With respect to the coordination sequence selected in Figure 5-7 (black path in the right part of the figure), the execution will perform the following steps in order:

1. First, the activity `openAUTFile` displays a file selector dialog (see Section 5.2.3) where the user can select the file which stores the graph to be minimized. This activity will generate an object of type AUTFile as output.
2. The object of type AUTFile gets now automatically transformed into an object of type AUTGraph by the activity `autF2autG`.
3. Then the AUTGraph object is transformed into an AUTFile object again, using the activity `autG2autF`.
4. After that, the minimization algorithm `aldebaranMIN_STD_I` is invoked on the AUTFile object. This algorithm minimizes the graph with respect to the *tau*.a* bisimulation [Mil89] using the Paige/Tarjan algorithm [FM90, PT87].
5. The resulting AUTFile object is then transformed into an AUTGraph object by the activity `autF2autG`.
6. Finally, the AUTGraph is displayed on the screen using the `displayAUT` activity.

Starting with an AUTFile having the content shown in the left part of Figure 5-8, the minimized AUTGraph looks like the one shown in the right part of the same figure.

Figure 5-8. A Minimizer Example



5.2.3. Data Access

This section presents the ToolZone-client features which can be used to access the data areas offered by an ETI site. For this the ToolZone client provides three editor-classes:

1. a general text file editor,
2. an editor to work on files representing directed graphs and
3. an editor to work on files representing graphs systems.

> **Note:** Graphs systems are collections of directed graphs plus some "global" information. The interpretation of the collection and the global information is tool specific. In one case, the collection may be interpreted as a set of procedures with the global information being a set of global variables. In another case the collection may represent a set of LTSs which model the components of a large software system, where the whole system is defined by the parallel composition of the single components.

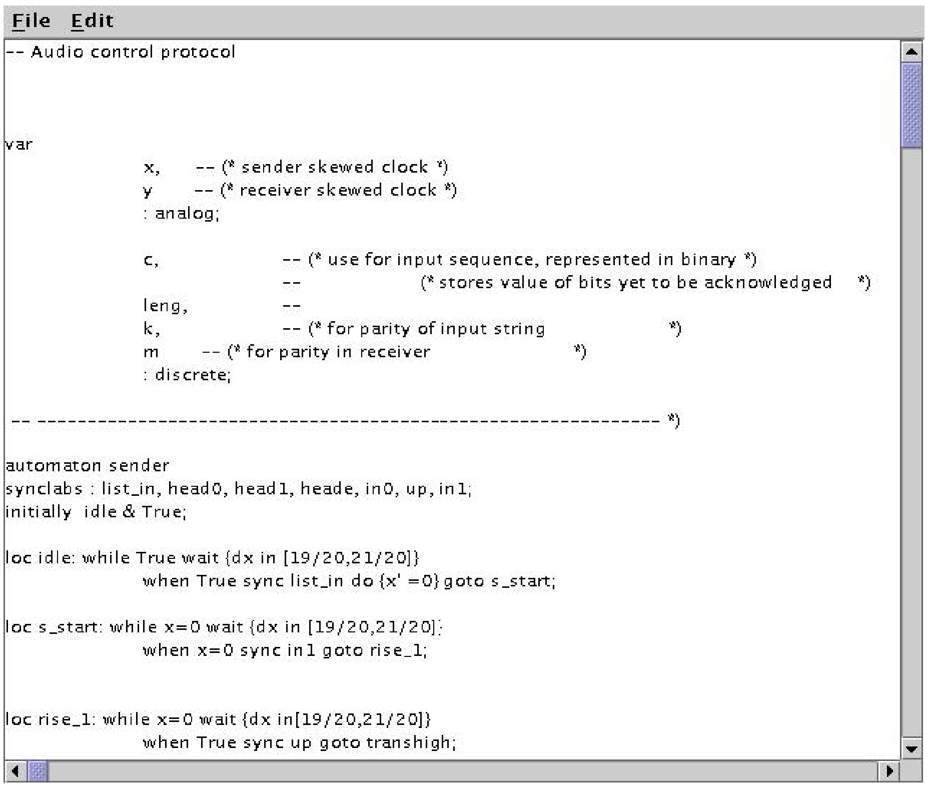
All editors can be accessed via the File menu contained in the menubar of the ETI Shell Window (see Figure 5-4). Whereas the New Text File menu item can be used to obtain the text file editor window, the menu items offered by the New Graph and New System submenues give access to the graph and graphs-system editors.

In the following two sections (Section 5.2.3.1 and Section 5.2.3.2) we give a short introduction to the editors offered by the ToolZone software. Afterwards, Section 5.2.3.3 present the ToolZone-client file dialog which can be used to load and store the objects visualized by the editors into a file. Please take a look at [TZUser] for a detailed description of the available features.

5.2.3.1. The General Text File Editor

The most universal editor is the one for general text files. It can be used to edit text files having an arbitrary format. Figure 5-9 presents an ETI Text Editor window which shows a fragment of a real-time system specification in the HyTech [HHW97, HyTech] format. The ETI editor provides all features a standard text editor offers. Thus, there is nothing specific with respect to the ETI text editor.

Figure 5-9. The ETI Text Editor Window



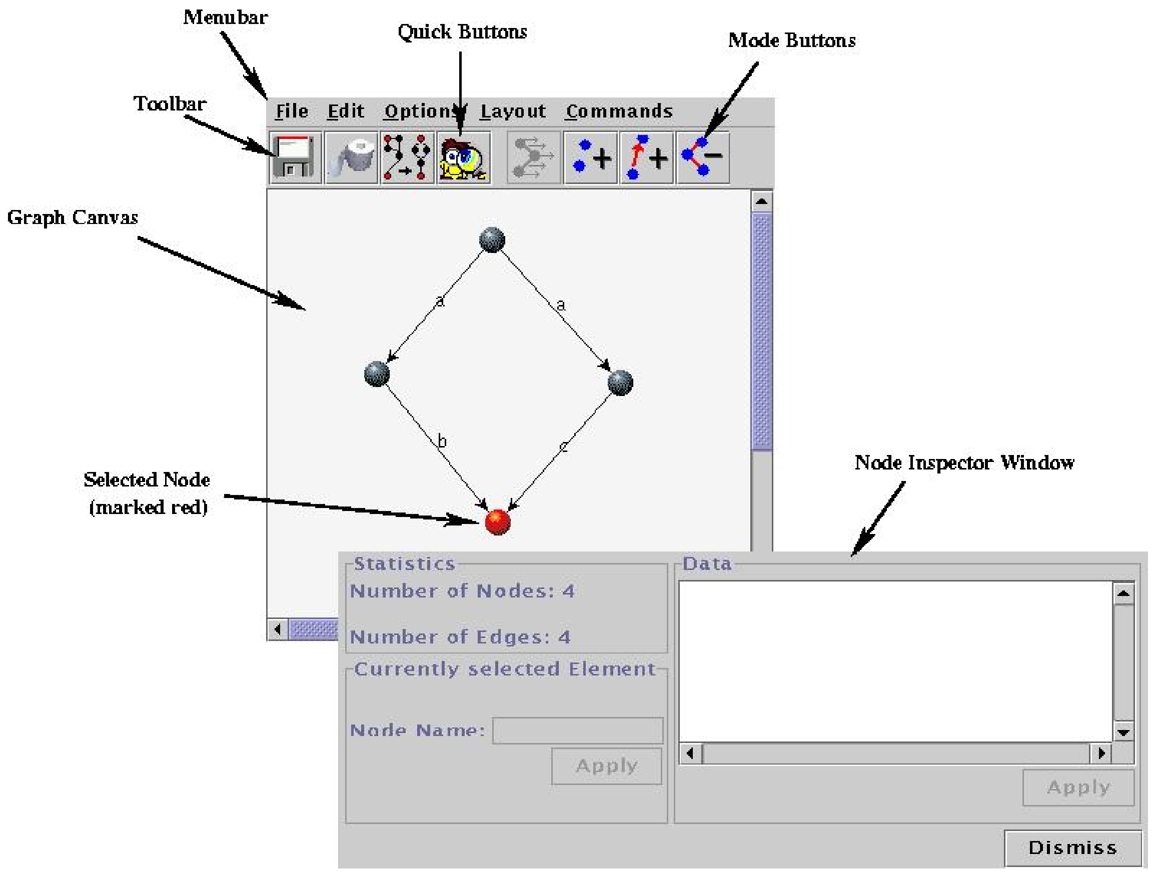
```
-- Audio control protocol



var
            x,     -- (* sender skewed clock *)
            y      -- (* receiver skewed clock *)
            : analog;

            c,               -- (* use for input sequence, represented in binary *)
                             --               (* stores value of bits yet to be acknowledged    *)
            leng,            --
            k,               -- (* for parity of input string                 *)
            m       -- (* for parity in receiver                   *)
            : discrete;

 -- ---------------------------------------------------------------- *)

automaton sender
synclabs : list_in, head0, head1, heade, in0, up, in1;
initially  idle & True;

loc idle: while True wait {dx in [19/20,21/20]}
            when True sync list_in do {x' =0} goto s_start;

loc s_start: while x=0 wait {dx in [19/20,21/20]:
            when x=0 sync in1 goto rise_1;


loc rise_1: while x=0 wait {dx in[19/20,21/20]}
            when True sync up goto transhigh;
```

5.2.3.2. The Graph and Graphs System Editors

Whereas there exists just one editor implementation for text files, which supports any textual file format, the ToolZone software provides a generic editor for directed graphs (see Figure 5-10) and a generic editor for graphs systems (see Figure 5-11). In the following, we give an overview of the the graph editor. The graphs system editors are handled a similar way.

The generic graph editor can be customized to support a specific file format which is used to store a graph object into a file. Here, an example of a file format may be the `aut`-format introduced in Chapter 4. Of course, textual LTS-descriptions in the `aut`-format can also be edited using the general text editor (see Section 5.2.3.1). But using the graph editor, the end user can specify the LTS in a graphical manner which is more comfortable. Instances of the generic graph editors mainly differ in the implementation of the load and save functionality (see Section 7.5.2 and Section A.2 for details) which is of course specific to the supported file format. The GUI, i.e. the end user's view on the editors, is the same for all instances.

Figure 5-10. The ETI Graph Editor Window



Since the instances of the graph editors which are accessible via the ToolZone software depend on the graph types available in the tool repository, the menu items provided by the New Graph submenu are specific to the connected ETI site. This means that dependent on the available graph types a menu item giving access to the corresponding graph editor has dynamically to be generated. This is done on the basis of the configuration file loaded from the ETI site during the startup of the client application (see Section 5.1).

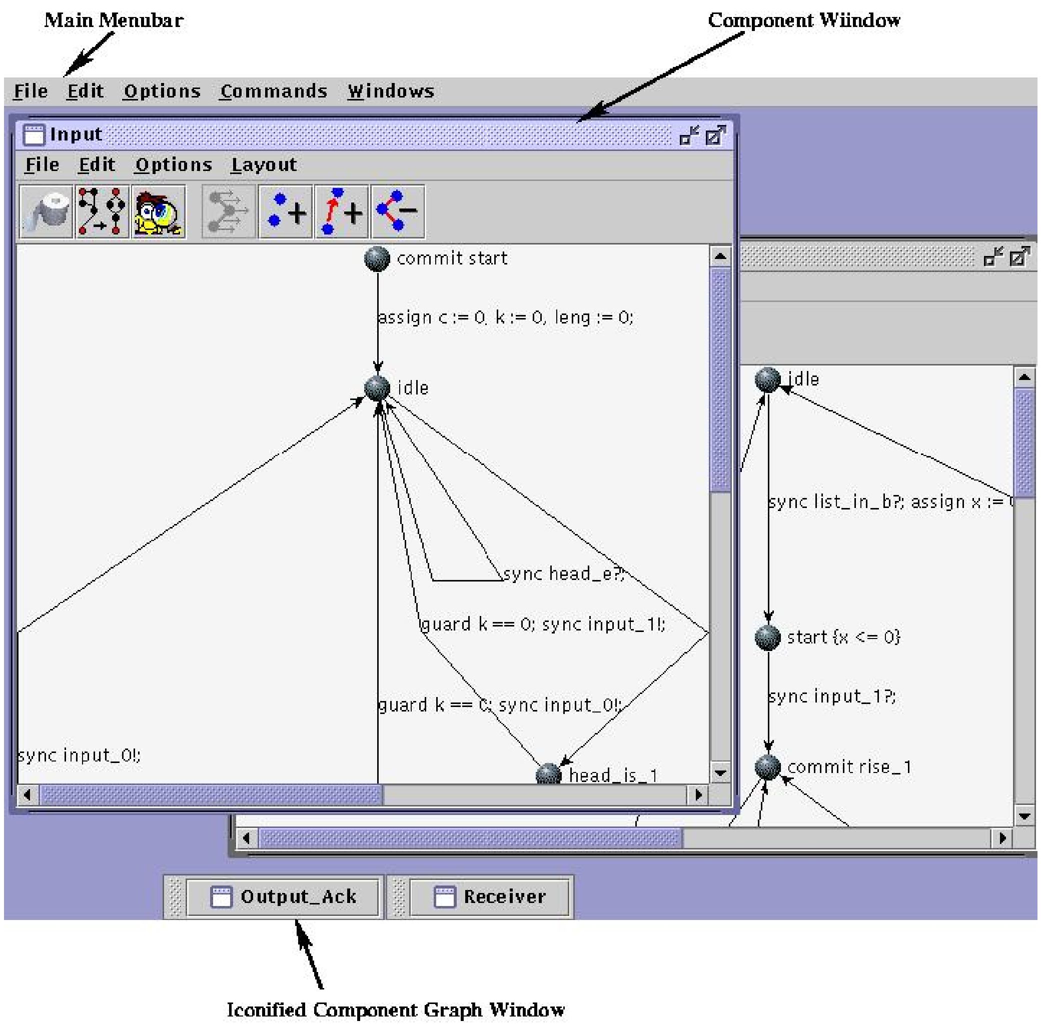
The ETI site associated to the Springer International Journal on Software Tools for Technology Transfer [STTT] currently provides two concrete editors for directed graphs:

- one supports the `aut`-format which is used to store labelled transition systems within the Caesar/Aldebaran Development Package [FGK96, CADP],
- the other gives access to `tg`-files representing real-time systems which can be analyzed by the Kronos [Yov97, Kronos] tool.

With respect to graphs systems, this version of the software provides editors for real-time systems stored in the formats which can be used by

- the Uppaal [LPY97],
- the HyTech [HHW97], and
- the Kronos tools.

Figure 5-11. The ETI System Editor Window


Main Menubar
Component Wiindow
File Edit Options Commands Windows
Input
File Edit Options Layout
commit start
assign c := 0, k := 0, leng := 0;
idle
sync head_e?;
guard k == 0; sync input_1!;
guard k == 0; sync input_0!;
sync input_0!;
head_is_1
idle
sync list_in_b?; assign x :=
start {x <= 0}
sync input_1?;
commit rise_1
Output_Ack
Receiver
Iconified Component Graph Window

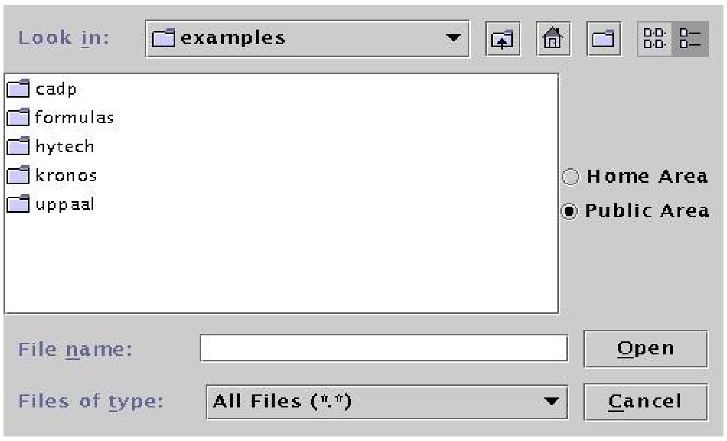
5.2.3.3. The File Dialog

The objects visualized by the editors can be loaded or saved via the file dialog (see Figure 5-12). Using this window, the user can select the file which contains the required information.

There are two data areas which can be accessed via the file dialog:

- the *public area* which contains examples common to all users, and
- the *home area* which contains files which can only be accessed by a specific user.

Figure 5-12. The Load File Dialog



In contrast to the public area which can only be accessed in read-only mode, the home area can also be used to upload files on the server. Both data areas are located on the file server which is accessible by the application host (see Figure 2-6).

The implementation of the file dialog is based on the standard Swing file chooser [WC99, Swing]. For our purposes it has been extended by an accessibility component which can be used to switch between the home and the public data area. Selecting either the Home Area or the Public Area radio button placed in the right part of the dialog, will show the files contained in the user's home area and the public area, respectively.

Chapter 6. The ETI Developer’s View

This chapter documents the developer’s view on the ETI ToolZone software. For this, it is organized as follows.

First, Section 6.1 gives a coarse-granular overview of the software components located in the Internet access layer (see Section 6.1.1) and the feature layer (see Section 6.1.2) of the ToolZone software. This will provide information which is required to understand the ETI-platform design documented in Section 6.2.1 and Section 6.2.2. Beside others

- Section 6.1.1 illustrates the general communication mechanism which is used to connect the client software and the tool management application via objects located in the Internet access layer.
- Section 6.1.2 presents facts on the structure of the METAFrame environment which are important to understand the ToolZone software design.

Section 6.2 then illustrates the design of the main conceptual business classes:

- types (Section 6.2.1) and
- activities (Section 6.2.2).

The types and activities are implemented by C++ and Java classes depending on the layer the associated object are located in: C++ classes provide the implementation for the objects within the feature layer, and objects within the Internet access layer and presentation layer are realized by Java classes.

In particular, Section 6.2.1 contains two sub-section. The first sub-section is dedicated to the generic file types (Section 6.2.1.2). The second one (Section 6.2.1.3) present the design of graphs. Since the design concepts illustrated in the two sections are reused in the context of the graphs system design, this aspect is not covered by this chapter. The interested reader can find it in Appendix A.

6.1. Zooming into the ToolZone Architecture

Before we start to present the design details of the ToolZone software, we need a better understanding of the software components and their responsibilities located in the Internet access layer and the feature layer.

6.1.1. The Internet Access Layer

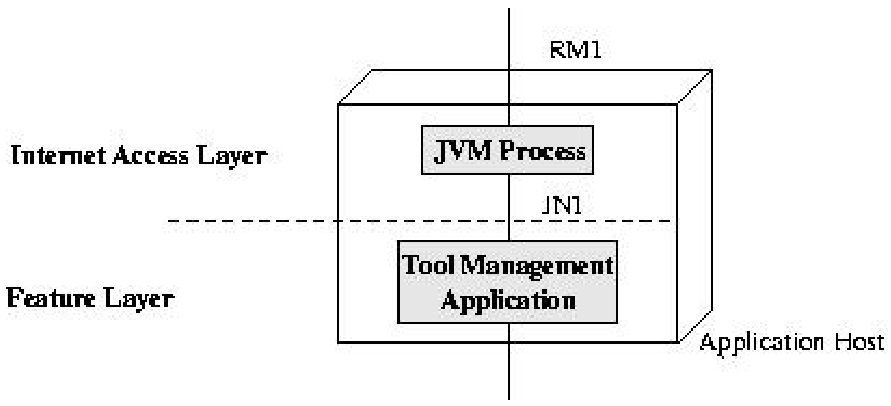
The Internet access layer of the ToolZone software is the mediator between the client software and the tool-management application running on the application host. As already mentioned in

Section 2.1.5, its implementation comprises

- an HTTP server,
- the ToolZone server,
- the application server, and
- a Java Virtual Machine (JVM) process running on the application host.

Here the ToolZone server, the application server, and the JVM process implement the tool Internet access server introduced in Figure 2-5. Whereas the ToolZone server and the application server are only relevant during the login process (see Section 5.1), the JVM process provides a set of remotely accessible objects that manage the communication between the client and the tool management application during a user session. The objects deployed in the JVM communicate via Java Remote Method Invocation with the ToolZone client and via the Java Native Interface (JNI) [Gor98, JNI] with the tool management application (see Figure 6-1). Beside others, Section 6.2.1.3.1 and Section A.1.1 give concrete examples of objects deployed in the JVM and present their collaboration in certain scenarios.

Figure 6-1. Central Software Components of the Internet Access Layer



As already mentioned in Chapter 4, neither the ToolZone client nor the implementation of the Internet access layer provide direct tool access. The client-side software implements the remote graphical user interface (GUI) giving access to the features provided by the platform. The features in terms of data objects and business logic are all implemented by the tool management application, i.e. within the feature layer. The Internet access layer implements the communication facilities. Its purpose is to delegate client requests to objects within the tool management application and vice versa. This strict separation of GUI, communication layer and feature layer ensures that the tool management application fully controls the access to the activities. At the same time this separation allows us to deploy a "thin" client application. This keeps the system requirements on the client host at a minimum which is an important aspect of wide software usability (see Section 1.3.3.1).

The communication between the client and the feature layer via the Internet access layer conforms to the following pattern:

1. A GUI event invokes a method on an object provided by the ToolZone client.
2. The client side object delegates the method invocation to its server side opponent located in the Internet access layer via Java RMI.
3. The corresponding method of the server side object is implemented as native method. In consequence it is mapped to a function provided by the tool management application using JNI.
4. The tool management application then performs the computation and finally initiates modifications on the client GUI by calling methods on Java objects located in the Java Virtual Machine of the Internet access layer.
5. The method calls are then delegated to the equivalent object on the client side using Java RMI.
6. The result of the initial GUI event is displayed on the client screen.

Figure 6-12 and Figure A-3 show concrete instantiations of this generic communication procedure.

The next section now provides more information on the software components located in the feature layer of the ToolZone software.

6.1.2. The Feature Layer

The feature layer of the ToolZone software is implemented by the tool management application running on the application host. This application, which controls the access to the tools located in the tool repository, is based on the METAFrame framework. It is implemented in C++ [Str97], and consists of an application-independent kernel managing the application-specific components (here the activities and types) contained in a component database (the tool repository). Amongst others, the METAFrame kernel provides

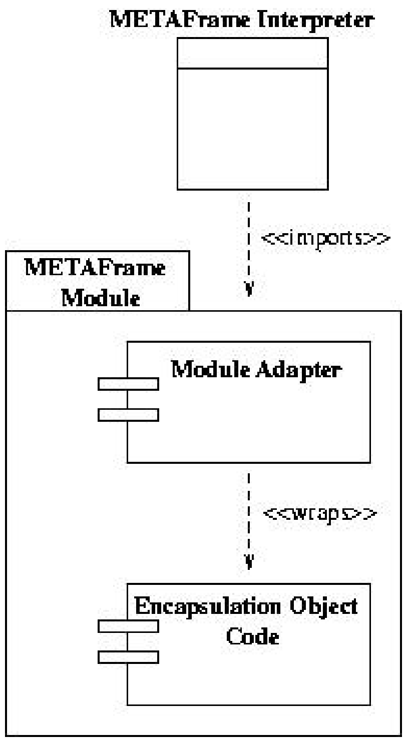
- the *hypertext system* [Dan98] giving access to the documentation and classification of the activities and types,
- the *synthesis component* [SMF93] that allows the end user to generate coordination sequences from the synthesis-compliant activities contained in the tool repository,
- the *HLL interpreter* [Cla97, Hol97b] which controls the execution of single activities as well as coordination programs, and
- the *PLGraph library* [BBCD97, PLGraph], that implements a flexible, polymorphic graph data structure.

With respect to the ToolZone software, the HLL interpreter and the PLGraph library are relevant to understand its design and the extension capabilities. Whereas extending the HLL by new functions and types is the main task during the implementation of new activities and types, the implementation of the generic graph and graphs system types are based on the PLGraph graph library. The following two sections give a basic introduction to the two software components, presenting mainly aspects which are relevant to the design of the ToolZone software.

6.1.2.1. The HLL Interpreter

Within the ETI project the HLL interpreter is used to control the execution of the ETI activities. It can be programmed using the coordination language HLL. This imperative coordination language can be extended by new data types and functions at runtime. The additional types and functions are provided by *METAFrame Modules* (implemented as shared libraries) that can be imported into the interpreter core. Figure 6-2 shows the structure of a METAFrame module.

Figure 6-2. The Structure of a METAFrame Module



As presented in [Cla97], it consists of

- the *Encapsulation Object Code* providing the C++ classes implementing the chosen function or type, and
- the *Module Adapter* wrapping the C++ classes into HLL functions and data types.

The adapter wraps and unwraps HLL data objects into C++ objects and maps the execution of an HLL function to the corresponding encapsulation code. Besides some error handling, there is no additional functionality provided by the adapter.

Building a METAFrame Module

Building METAFrame modules (see also [Hol97b]) is one of the main tasks of the integration process which makes new activities and types available in the tool repository (see Chapter 7 for more information). In this process the METAFrame module is automatically generated on the basis of

1. the C++ encapsulation code and
2. the METAFrame Module Adapter Specification

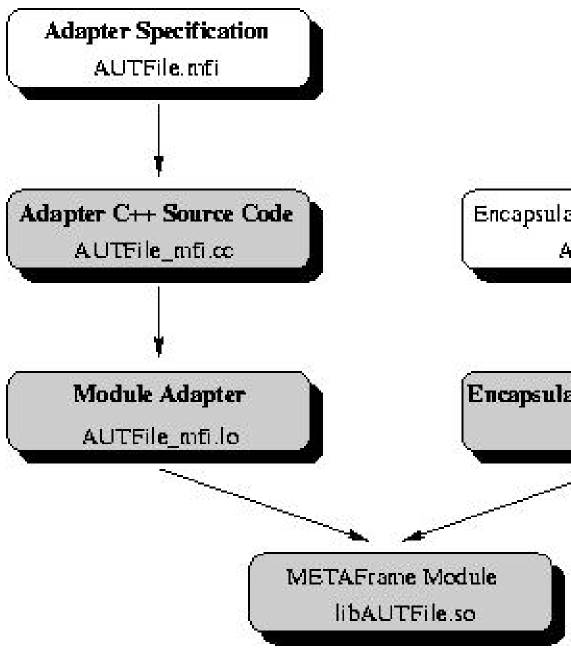
which provide the implementation of the addressed functions and types, and specify their linkage to HLL functions and types, respectively. In detail, the automatic generation of the METAFrame module on the basis of the encapsulation source code and the adapter specification is performed in four steps:

1. generating the C++ source code of the module adapter out of the adapter specification,
2. compiling the C++ source code of the module adapter which results in the actual *META-Frame Module Adapter*,
3. compiling the C++ source code of the encapsulation code which results in the *Encapsulation Object Code*, and finally
4. building the *METAFrame Module* (a shared library) by linking the METAFrame Module Adapter and the encapsulation object code.

Figure 6-3 exemplarily shows how the general workflow has been applied to build the `AUTFile` METAFrame module. This module provides the HLL type AUTFile which gives access to labelled transition systems stored in the `aut`-file format (see Section 4.1.1). Within Figure 6-3 the automatically generated files are represented by gray rectangulars whereas the files that have to be provided by the developer are colored white. Edges denote automatic transformations which are performed either by tools bundled with the METAFrame environment or by standard C++ compilers.

In the context of the ETI platform, this generic wrapping-process has been customized for the integration of off-the-shelf software tools (see e.g. Chapter 7).

Figure 6-3. Building the `AUTFile` Module



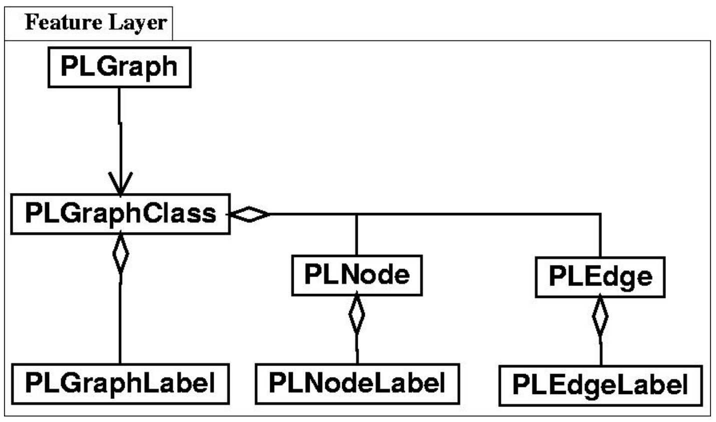
6.1.2.2. The PLGraph Library

The PLGraph library realizes a flexible, polymorphic graph data structure implemented in C++. The main concept of the library is the strict separation of

1. the graph structure represented by the nodes and edges, and
2. the application-specific data and functionality, which can be associated to these objects.

Figure 6-4 shows a UML class diagram [BRJ98, FS99, UML] presenting important classes of the library.

Figure 6-4. The PLGraph Design

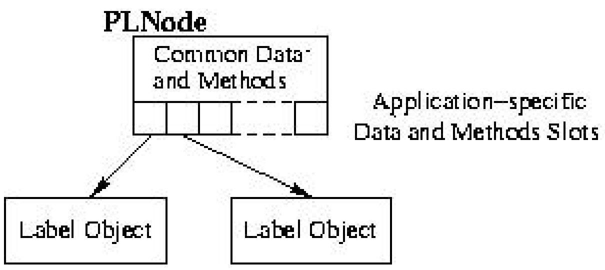


Beside others the PLGraph library contains

- the class `PLGraphClass` gives access to the graph data structure by providing beside others methods for obtaining the node and edge sets,
- the classes `PLNode` and `PLEdge` model node and edge objects, and
- the classes `PLNodeLabel`, `PLEdgeLabel` and `PLGraphLabel` model so called *labels* which encapsulate the application-specific data and functionality.

Figure 6-5 shows the conceptual structure of a node object in the PLGraph library. It consists of a part which manages common data and functionality primarily concerned with providing access to the graph structure, like access to predecessor and successor nodes. In addition to that, it contains slots where application-specific data and functionality encapsulated into a label object can be attached. Edge and graph objects are structured analogously.

Figure 6-5. Conceptual Structure of a PLGraph Library Node



Using the PLGraph library, a new graph type is *not* obtained via inheritance from the graph, node and edge classes. Instead the additional data and functionality is provided by label classes inheriting from the classes `PLNodeLabel`, `PLEdgeLabel` and `PLGraphLabel`. A graph object of the addressed type is then obtained by adding the corresponding label objects to the graph at runtime.

The label classes are the key for the flexibility of the PLGraph library which in turn is important for the ToolZone software. This is because the kind of data associated to nodes, edges and graphs may change at runtime due to the different transformations which are applied to a graph object. If we would obtain specific node types via inheritance instead of delegation the kind of information associated to a node cannot be changed dynamically. It is fixed at the time the corresponding graph object is created. To provide the required flexibility one could alternatively

- provide *one node class* which is able to store all kind of data. This would solve the problem but waste a lot of memory, since at a given time only a small part of the allocated node-memory is used. This approach would not scale, since graph objects may contain a huge number of nodes.

- implement *several node classes* one for each possible combination of node information. This would also not scale due to an explosion of the number of node classes. Another disadvantage of this approach is that we must know the different kind of information needed for the actual transformation at the creation time of the graph object.

The trade-off to the flexibility are the following three minor drawbacks:

1. The type of a graph cannot be determined at compile time. This is due to the fact that the type of a graph is implicitly defined by the type of the label objects attached to the nodes, edges and the graph. But this information cannot be computed at compile time.
2. During runtime it is not ensured that the proper label objects are attached to the graph, when required. This is the responsibility of the application developer.
3. The concept of specializing a graph object by encapsulating the new functionality into special classes and not by inheritance from a base graph class is unusual to most programmers. Additionally, the graph’s functionality is not accessible via one class. Instead it is distributed over several label classes each providing a dedicated set of methods. This means that the usage of the library within an application gets more complicated.

Within a specific context like the ETI platform these drawbacks can be hidden from the developer using certain design patterns (see Section 6.2.1.3).

6.2. The Design of the Business Classes

The next two sections go into the details of the design of the main business classes of the ToolZone software:

1. types (Section 6.2.1)
2. activities (Section 6.2.2).

6.2.1. The Design of the generic Types

This section explains the design of two generic types which are currently supported by the ToolZone software:

1. files (see Section 6.2.1.2),
2. graphs (see Section 6.2.1.3), and

The third supported type, i.e. the graphs system, is handled in Appendix A since its design does not introduce new concept.

In particular, the following two sections present the realization of the communication between the presentation layer and the feature layer introduced in Section 6.1.1 on a detailed level. Thus they document the realization of an Internet-based, secure, performant and failsafe infrastructure to provide access to the coordination features offered by METAFrame environment, was put into practice. The sections do not introduce new concepts. For this, they mainly address

- readers who want to extend the ToolZone software and
- readers who look for a solution in a similar application domain.

Consequently, readers who are not interested in the design details may continue with Section 6.2.2.

6.2.1.1. Basic Features

The generic types are realized by the classes `ETIFile`, `ETIGraph` and `ETISystem`. New, specific file, graph, and graphs system types are (mainly) obtained via inheritance from these base classes.

All generic types provide basic features which help to maintain their concrete subtypes. For this a unique type identifier is used (see Chapter 7). This management functionality includes beside others

- methods to register new concrete types, like the `registerFileType` method of the class `ETIFile`,
- methods to obtain all currently available subtypes of a specific generic type, like `getRegisteredGraphTypes` method of the class `ETIGraph`,
- methods to obtain a fresh object of a specific subtype via the unique type identifier (e.g. provided by the method `getSystemByType` of the class `ETISystem`).

Note that the following sections do not go further into the design of graphs systems. This is presented in Appendix A.

6.2.1.2. Access to the File System

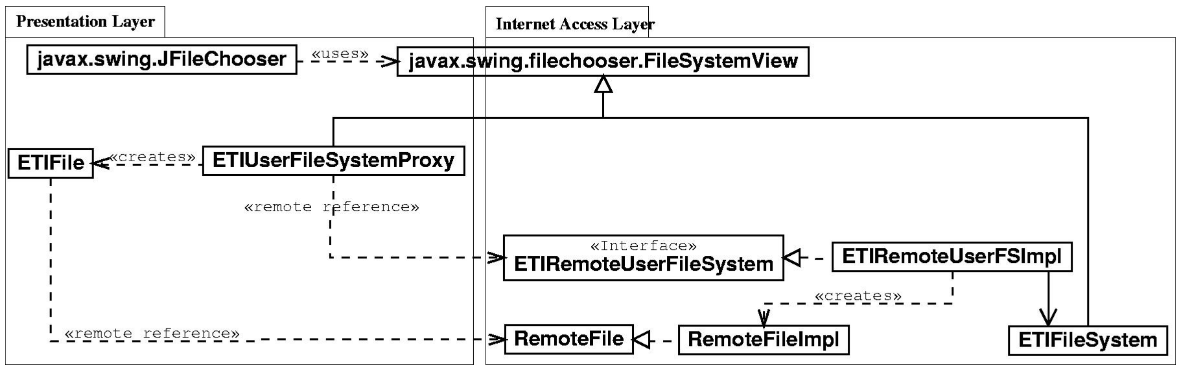
The design of the generic file type is illustrated along the two steps that have to be performed to access the data which is stored in a file:

1. Selecting the name of the file via the file dialog (see Section 6.2.1.2.1).
2. Accessing the file within the tool management application (see Section 6.2.1.2.2).

6.2.1.2.1. Selecting a File Name

The selection of a file name is done using the file dialog shown in Figure 5-12. The window extends the standard Swing file chooser by two radio buttons, which are used to switch between the public and the home area. According to the design of the Swing file chooser, the functionality required to browse through a file system is provided by the class `FileSystemView` contained in the `javax.swing.filechooser` package. Here methods like `getFiles` and `createNewFolder` give access to the underlying file system (see [JavaAPI] for more information).

Figure 6-6. The ToolZone User File System Classes



Note: Since objects of the type `javax.swing.filechooser.FileSystemView` are located in the client layer as well as the Internet access layer, this class is contained in both package elements.

The ToolZone file system is split up into a client side and a server side file system which communicate via Java RMI. Figure 6-6 shows the classes implementing the home area (user file system).

The implementation uses the *Proxy Pattern* which is a standard design guideline to "provide a surrogate or placeholder for another object to control access to it" [GHJV95]. The purpose of the pattern within the ToolZone software is

1. to provide a client side representative of the file system located on the server, thereby hiding the details of the client server communication to the client application. In consequence, changing the communication mechanism does not affect the code of the client application. This characteristics of the proxy pattern is also called *remote proxy*.
2. to control the access to the "real" files located on the server. This ensures, that clients
   - can only access the specific parts of the whole file system which are dedicated to the ToolZone application.

- can access the public area only in read-only mode.
- can access the home area in read and write mode.

This means that a central part of the security policy of the ToolZone software is implemented via this particular form of the proxy pattern, also known as *protection proxy*.

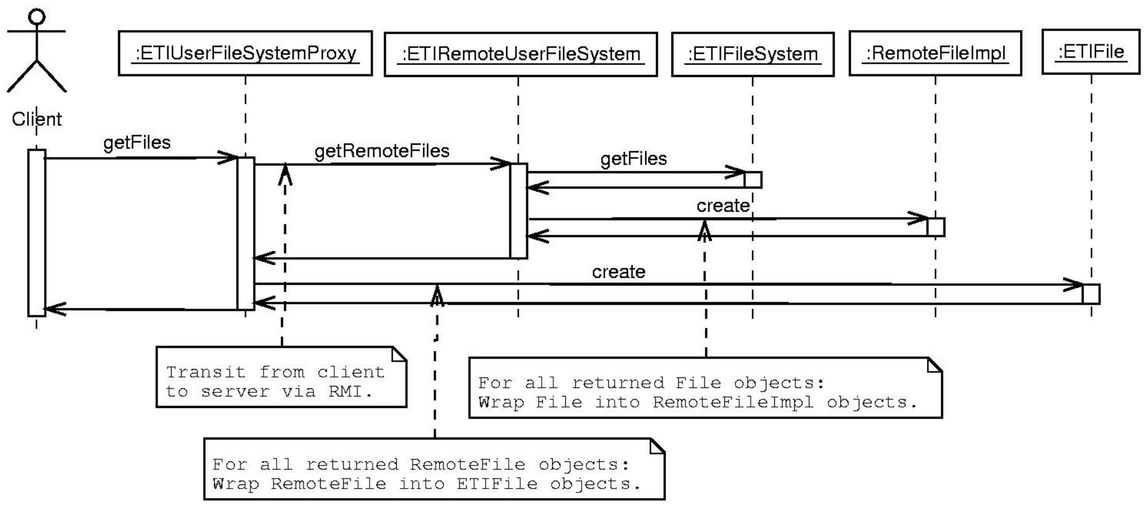
The remote proxy is realized by the class `ETIUserFileSystemProxy` which gives the client-side classes access to the server-side file system. The object providing the server-side access to the file system is the class `ETIRemoteUserFSImpl`. To control the operation on the file system, the class `ETIRemoteUserFSImpl` does not use the standard `FileSystemView` class contained in the `javax.swing.filechooser` package. Instead of this the access policy is implemented by a protection proxy (class `ETIFileSystem`) which controls the real file access provided by the `FileSystemView` class.

Note: The design of the public file system is similar to the design of user file system.

Within the selection process only client side objects and objects contained in the Internet access layer are involved. No tool management functionality is required.

Figure 6-7 shows the collaboration between the client side and server side classes implementing the user file system as UML sequence diagram. Here the scenario where the client file chooser requests the name of the files contained in a directory is documented. This complex process, where client side request are delegated via the remote proxy (`ETIUserFileSystemProxy`) and the protection proxy implemented by the `ETIFileSystem` class, ensures that the client code is independent of the particular communication mechanism and that the data is securely accessed.

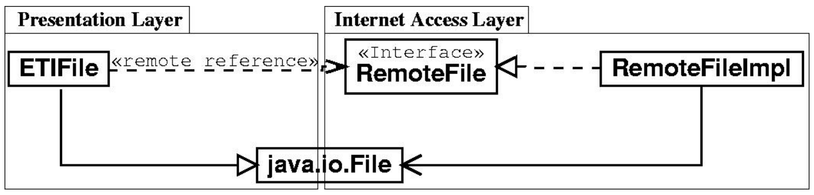
Figure 6-7. Requesting File Names of a Directory



Client
:ETIUserFileSystemProxy
:ETIRemoteUserFileSystem
:ETIFileSystem
:RemoteFileImpl
:ETIFile
getFiles
getRemoteFiles
getFiles
create
create
Transit from client to server via RMI.
For all returned File objects: Wrap File into RemoteFileImpl objects.
For all returned RemoteFile objects: Wrap RemoteFile into ETIFile objects.

In detail, a client request for a file name is handled as follows. Whenever the client wants to get the list of the files contained in a directory on the server, it calls the `getFiles` method on the `ETIUserFileSystemProxy` object. This object is an implementation-independent representative of the real file system. It delegates the method invocation to its opponent on the server side (object of type `ETIRemoteUserFileSystem`), here using Java RMI. The server side object then uses the `ETIFileSystem` to get the list of the files represented as `java.io.File` objects. To make these objects accessible by the client, the `ETIFileSystem` wraps the `File` objects into `RemoteFileImpl` objects which implement the remote interface `RemoteFile`. After that, the skeletons of the `RemoteFile` objects are returned to the client, i.e. to the `ETIUserFileSystemProxy` object. Since the class `ETIUserFileSystemProxy` is derived from the class `FileSystemView` (see Figure 6-6), the `getFiles` method must return objects derived from `java.io.File`. Hence, the `ETIUserFileSystemProxy` object encapsulates the `RemoteFile` objects into `ETIFile` objects, which are derived from the class `java.io.File`. Figure 6-8 shows the classes implementing the file handling in the Internet access and presentation layer.

Figure 6-8. The Java File Class Hierarchy



6.2.1.2.2. Accessing the File within the Tool Management Application

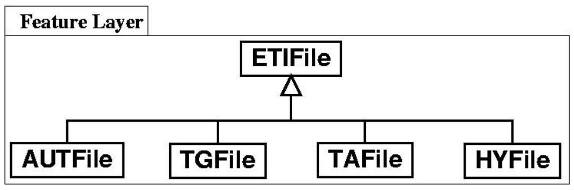
The result of the file-selection process is a string which locates the file on the server. It is passed by the client to the tool management application for further processing. There are mainly two scenarios for this processing task:

1. The content of the file is loaded into a corresponding editor.
2. The content of the file is used to initialize an object which is itself processed by an activity.

Within the tool management application, files are modeled as classes derived from `ETIFile`. This means, that e.g. a file in the CADP `aut`-format is represented by a C++ class with name `AUTFile`, which inherits from the generic file class `ETIFile`. Figure 6-9 shows a part of the `ETIFile` class hierarchy within the platform instantiation for STTT. Here the classes `TGFile`, `TAFile` and `HYFile` model text files which store graphs systems of the tools Kronos, Uppaal, and HyTech respectively.

Similar to the `java.io.File` class, the file objects only provide meta information on the chosen file. Within the ToolZone software, the meta information comprises the file name and its type, and the unique file type identifier. It is the responsibility of the application to open, close, etc. the associated file on the hard disk.

Figure 6-9. Part of the STTT File Classes Hierarchy

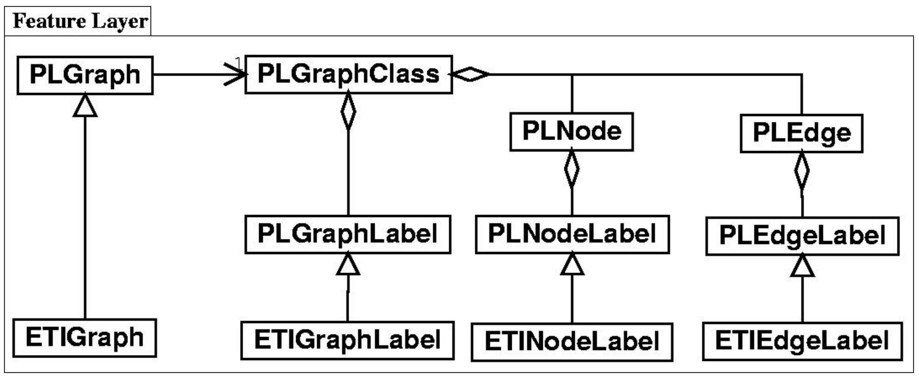


6.2.1.3. Directed Graphs

Verification tools like [FGK96, Yov97, HHW97, LPY97] were in the focus of the first application domain of the ETI platform. These tools (semi) decide whether a system satisfies a certain property. In most cases the system is modeled as labelled transition system, i.e. a directed graph where atomic information can be associated to the nodes and edges, or a collection of them called *graphs system*. The corresponding property is defined by a formula in a certain logic, e.g. the modal mu-calculus [Koz82]. In consequence, directed graphs and collections of them are very important data structures in this field of application.

As already mentioned, the ToolZone software uses the PLGraph library of the METAFrame project to implement the generic graph type. This was done by writing node, edge and graph label classes which provide the ETI specific functionality (see Figure 6-10). On the basis of the label classes a new graph type can be obtained by providing label classes which inherit from the classes `ETINodeLabel`, `ETIEdgeLabel` and `ETIGraphLabel`, respectively. Section 7.5.2 goes into the details of this process.

Figure 6-10. The ETI Graph Label Hierarchy



In the context of the ToolZone software the PLGraph library can be controlled by introducing a facade class [GHJV95] (class `ETIGraph` of Figure 6-10) and coding guidelines in a way that the usage of a graph type is simplified without loosing its flexibility. In combination, the two concepts tackle all drawbacks mentioned in Section 6.1.2.2:

1. The facade class makes the type of a graph object explicit. Consequently, we can check the type of a graph at compile time without loosing the runtime-flexibility.
2. The coding guidelines recommend to initialize the appropriate label objects at the creation time of the graph object (see Section 7.5.2.1.3). This ensures that the proper label objects are attached to the graph, which prevents runtime errors.
3. The ToolZone software extends the class `PLGraph` to implement the base class (class `ETIGraph`) for the specific facade classes. The facade classes themselves are derived from the class `ETIGraph`.

Every implementation of a specialized graph type comprises a facade class which is derived from the class `ETIGraph` (see Section 7.5.2.1.5). The facade class does not provide any functionality itself. It delegates all method invocations to the corresponding graph label object.

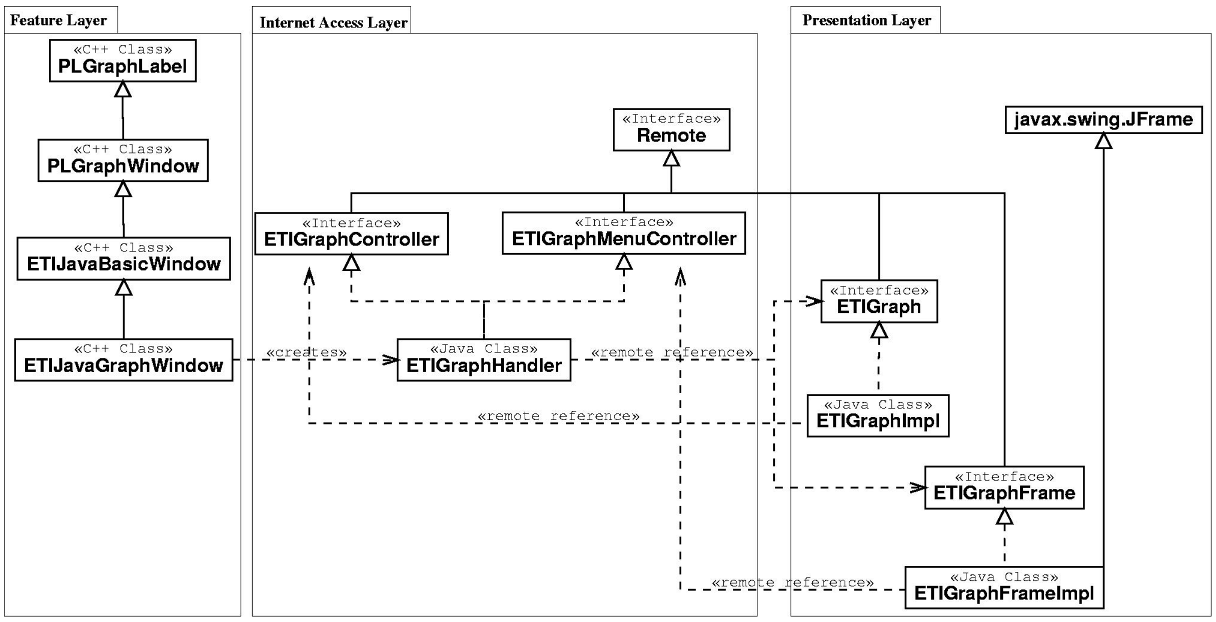
In the rest of this section we present the implementation of the graphical user interface of the graph types (Section 6.2.1.3.1). This illustrates the communication between client and server side components of the ToolZone software in the graph scenario (see also Section 6.1.1). Finally, we explain the basic design of the file formats in Section 6.2.1.3.2. In combination with the label classes, the file-format classes are the basis for the graph-integration process documented in Section 7.5.2.

6.2.1.3.1. The Graphical User Interface

Whereas the objects which store the graph data structure are all located in the feature layer, the objects which display the graph are distributed over all the layers of the ToolZone software depicted in Figure 2-5. Depending on the layer in which the objects are located, they communicate via Java RMI or JNI with each other, and the graph object which is shown on the screen. Figure 6-11 shows all classes which are responsible for displaying a graph object on the client screen.

Within the feature layer, the main class representing the GUI is the class `ETIJavaGraphWindow`. It is derived from `PLGraphWindow` which is a special graph label, i.e. it is derived from the class `PLGraphLabel`. To display a graph on the client screen, first a new `ETIJavaGraphWindow` object is created on the server side, and then this object is attached to the graph object to be shown using the `addGraphLabel` method of the class `PLGraph`. Figure 6-12 shows the details of this process. The `ETIGraphHandler` object residing in the Internet access layer of the software is the mediator between the server side and the client side GUI objects. It communicates with the server object via JNI and with the client objects of type `ETIGraphFrameImpl` and `ETIGraphImpl` via Java RMI. The remote views of the objects of the two classes are defined by the interfaces `ETIGraphFrame` and `ETIGraph`, respectively. The `ETIGraphFrameImpl` implements the window in which the graph is shown, i.e. it provides the menu bar, the tool bar including the quick and mode buttons, and the graph canvas (see Figure 5-10). The graphic representation of the nodes and edges of the graph, i.e. the content of the graph canvas, is implemented by the `ETIGraphImpl` class.

Figure 6-11. Classes building the GUI of a Graph Object


Feature Layer
Internet Access Layer
Presentation Layer
«C++ Class» PLGraphLabel
«C++ Class» PLGraphWindow
«C++ Class» ETIJavaBasicWindow
«C++ Class» ETIJavaGraphWindow
«Interface» Remote
«Interface» ETIGraphController
«Interface» ETIGraphMenuController
«Java Class» ETIGraphHandler
«creates»
«remote reference»
«remote reference»
«remote reference»
javax.swing.JFrame
«Interface» ETIGraph
«Java Class» ETIGraphImpl
«Interface» ETIGraphFrame
«Java Class» ETIGraphFrameImpl

The `ETIGraphHandler` object provides two remote views to the client defined by the interfaces `ETIGraphController` and `ETIGraphMenuController`. The methods of the `ETIGraphController` interface are used by the `ETIGraphImpl` objects to get access to the corresponding `ETIGraph` object located in the feature layer. The `ETIGraphMenuController` interface allows the `ETIGraphFrameImpl` object to access `ETIJavaGraphWindow` functionality.

Figure 6-12. Creating a remote Graph Editor Window

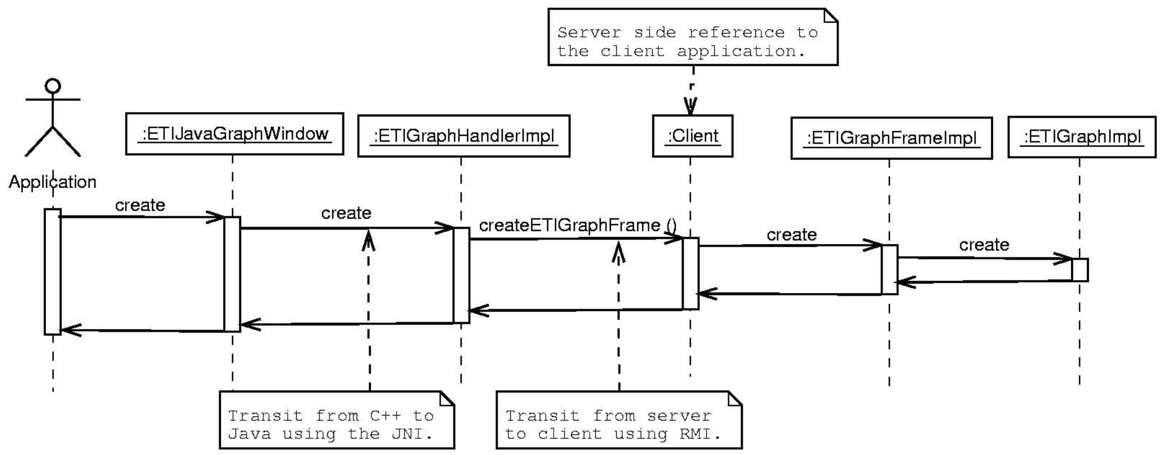


Figure 6-12 shows the scenario where a remote graph window is created by the server side application in order to display a graph object on the client. In accordance to the communication pattern documented in Section 6.1.1, the creation of the `ETIJavaGraphWindow` object has been initiated by the client GUI, and passed via the Internet access layer to the tool management application (see steps one to three of the communication pattern). As already mentioned in Section 6.1.1, this complex process has been realized to let the application logic been executed on the server, thus allowing us to deploy a thin client application.

The GUI of the ToolZone software is implemented in a generic way. This means that it is the same for all graph types available in the ETI platform. Adding a new graph type to the platform has no impact on the GUI classes.

6.2.1.3.2. File Formats

A file-format class provides the implementation of the functionality which is needed to load a graph object from and to store it to a file. Beside others, it comprises the scanner and the parser which are able to handle the syntax of a specific format. The ToolZone software distinguishes two kinds of file formats:

The (unique) *native file format*:

> The native format is the default format which is used to make a graph object persistent on the disk.

Optional *import/export filters*:

> The filters store graph objects into and load graph objects from files whose content is not conform to the native format. Thereby they provide a link to other graph libraries.

A file-format class is implemented as a graph label. More precisely, it must be derived from the class `PLFileFormat` contained in the PLGraph library. Before a graph can be stored to or loaded from a file of a certain format, a corresponding file format object has to be created and registered to the graph object. This is done in the constructor of the graph facade class using the `addFileFormat` method of the class `PLGraphClass`. Example 6-1 shows the registration of the `AUTFileFormat` to an object of type `AUTGraph`.

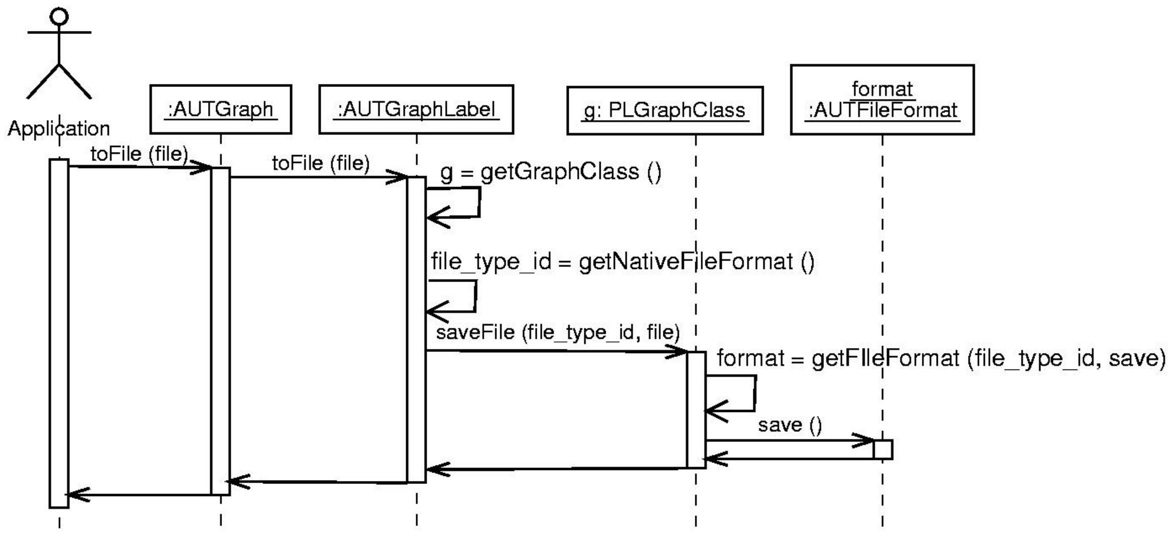
Example 6-1. Registering the `AUTFileFormat`

```
AUTGraph::AUTGraph ()
{
             :
             :

    addFileFormat (new AUTFileFormat ());
}
```

The interaction between a graph and a file-format object is exemplarily illustrated in a scenario where a graph is stored into a native file (see Figure 6-13). In the case that a graph is loaded from a native file, or that non-native export and import filters are used, the communication between the objects is similar. Only the requested methods vary.

Whenever the application stores a graph into a native file, the method `toFile` is called on the graph facade object (see `AUTGraph` object in Figure 6-13). Since this facade class does not provide the requested functionality, the method invocation is first delegated to the corresponding graph label object. Now, the graph label object determines the identifier of the native file format via a `getNativeFileFormat` call. Knowing the file type identifier, the graph label object requests the save functionality from the corresponding graph object. The `saveFile` method of the `PLGraphClass` object then uses the passed file format identifier to determine the corresponding file format object, and invokes the `save` method on it. This finally stores the graph object into a file. Figure 6-13 shows a UML sequence diagram documenting this scenario.

Figure 6-13. Saving an `AUTGraph` Object to a File



Note that most parts of the process are implemented in the core of the ToolZone software, in particular in the class `PLGraphLabel` from which the `AUTGraphLabel` class is derived. This means that only the specific file format class has to be provided and registered to the graph object when a new file format should be supported (see Section 7.5.2). In consequence, the developer can focus on his expertise, here writing the chosen file format (see also Section 7.5.2.1.4).

6.2.2. ETI Activities

An activity represents a single functionality of which many may be provided by a single integrated tool. This tool feature can be accessed via the HLL function defined by the activity's implementation constituent (Section 2.1.1). The signature of the HLL function is of the form of

$$\texttt{f1 (Type}_1\texttt{: input}_1\texttt{, ..., Type}_m\texttt{: input}_m\texttt{,}$$
$$\texttt{Type}_{m+1}\texttt{: result}_{m+1}\texttt{, ..., Type}_n\texttt{: result}_n\texttt{): ETIResult,}$$

where

- `f1` is the name of the HLL function,
- $\mathtt{input}_1$ to $\mathtt{input}_m$ are the arguments representing the *input data*,
- $\mathtt{result}_{m+1}$ to $\mathtt{result}_n$ are the arguments which contain the *computational result* after the activity execution, and
- the return value of type ETIResult provides *diagnostic information about the activity execution*. It comprises the termination status of the activity (ok, fail, etc.) and optional diagnostic output generated by the wrapped tool feature.

If it is a synthesis-compliant activity, the signature of the HLL function contains only *one* input parameter and *one* output parameter (which may be composite).

6.2.2.1. Wrapping Tool Features

As already mentioned in Section 6.1.2.1, the HLL function is implemented by a METAFrame module which consists of two components:

- the *encapsulation code* which comprises several C++ classes giving direct access to the tool feature, and
- the *module adapter* wrapping the C++ functionality into the HLL function.

Example 6-2 shows how the the HLL function `aldebaranMIN_STD_I` has been realized on the basis of the **aldebaran** tool. Note that the example only presents an overview of the wrapping concept. Section 7.5.3 goes into the details of the techniques which are used to provide the implementation constituent of an activity.

Example 6-2. Realization of the `aldebaranMIN_STD_I` HLL function

The activity `aldebaranMIN_STD_I` represents the feature of the **aldebaran** tool which minimizes a labelled transition system with respect to the *tau*.a* bisimulation [Mil89] using the Paige/Tarjan algorithm [FM90, PT87]. The feature can be accessed by executing a command of the form of

```
aldebaran -std -imin in.aut > out.aut,
```

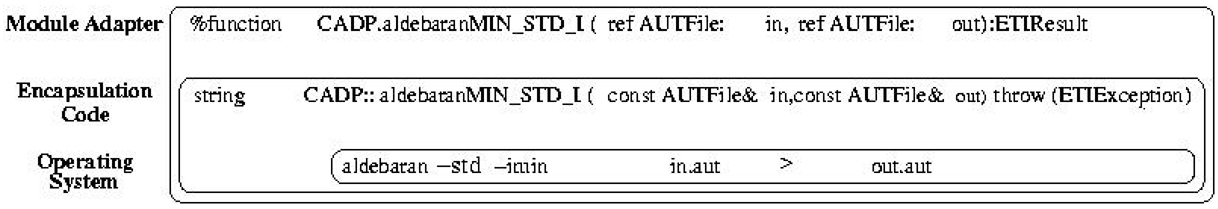
where

- the command line options `-std` and `-imin` specify the kind of the minimization procedure which has been chosen, here *tau*.a* bisimulation using the Paige/Tarjan algorithm,
- *`in.aut`* specifies the `aut` file which stores the LTS to be minimized, and
- *`out.aut`* defines the file into which the minimized LTS is stored.

Figure 6-14 shows how the chosen aldebaran feature is accessed via the HLL function. For this,

- the encapsulation code comprises the C++ class `CADP` which offers the method `aldebaran-MIN_STD_I`. This method executes the system call to access the **aldebaran** feature.
- the module adapter provides the implementation of the HLL function `aldebaranMIN_STD_I` of the `CADP`. The implementation uses the `aldebaranMIN_STD_I` of the class `CADP`.

Figure 6-14. Wrapping Tool Functionality

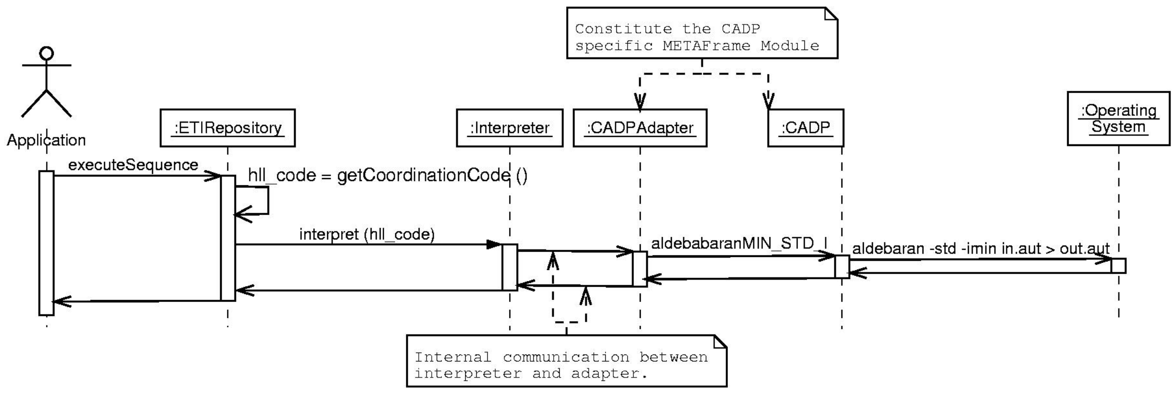


```
%function   CADP.aldebaranMIN_STD_I ( ref AUTFile:   in, ref AUTFile:   out):ETIResult
string   CADP:: aldebaranMIN_STD_I ( const AUTFile&  in,const AUTFile&  out) throw (ETIException)
aldebaran -std -imin   in.aut   >   out.aut
```

6.2.2.2. Activity Execution

An activity can be executed either in stand-alone mode or in tool-coordination mode. The corresponding HLL code fragment, which uses the HLL function defined by the activity's implementation constituent, is run by the METAFrame interpreter. Here, the interpreter only controls the execution flow. Neither the adapter code nor the encapsulation code provide the implementation of the real computation. This is performed by the tool itself. For this, the execution flow is delegated by the interpreter via the module adapter and the encapsulation code to the tool. Figure 6-15 shows how the `aldebaranMID_STD_I` activity is executed by the interpreter in the context of a coordination sequence.

Figure 6-15. Executing an Activity in tool-coordination Mode

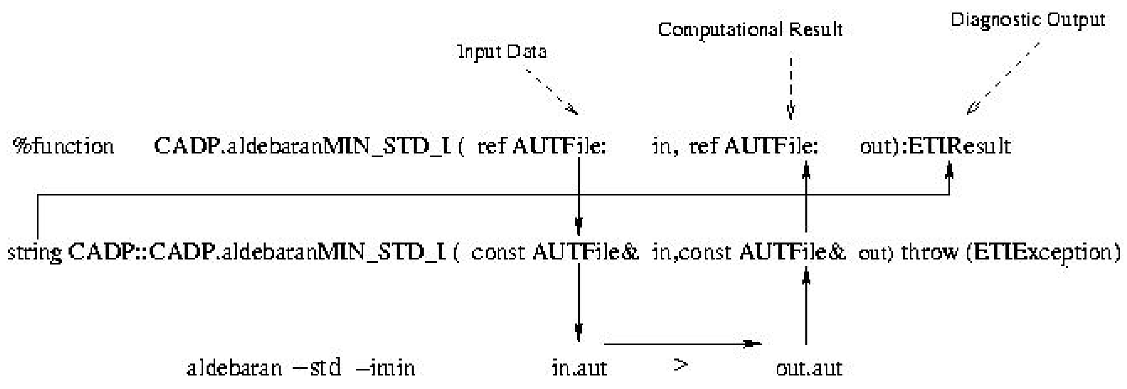

Constitute the CADP specific METAFrame Module
Application
:ETIRepository
:Interpreter
:CADPAdapter
:CADP
:Operating System
executeSequence
hll_code = getCoordinationCode ()
interpret (hll_code)
aldebabaranMIN_STD_I
aldebaran -std -imin in.aut > out.aut
Internal communication between interpreter and adapter.

When a coordination sequence is executed by the client, the tool management application calls the `executeSequence` method on the `ETIRepository` object, which models the tool repository in the ToolZone software. For every activity contained in the coordination sequence, the `ETIRepository` object first fetches the tool-coordination HLL code of the activity. Then the interpreter is asked to run the corresponding HLL code. In our example the HLL code comprises the `aldebaranMIN_STD_I` function. When this HLL function is run, the interpreter passes the execution flow via the `CADPAdapter` and the `CADP` objects to the **aldebaran** tool, which provides the implementation of the chosen functionality.

During the execution of the HLL function, the input and output data is handled as follows (see Figure 6-16). The `CADPAdapter` is responsible for unwrapping the real C++ data object from the HLL object passed as the first argument of the HLL function (the input data of the activity) and calling the `aldebaranMIN_STD_I` method of the `CADP` encapsulation object. The implementation of the `aldebaranMIN_STD_I` method then builds the appropriate system call and executes it. The diagnostic output of the system command execution is collected and passed back to the `CADPAdapter` as return value of the C++ method invocation. The computational result, i.e. the file representing the minimized graph is encapsulated into a C++ `AUTFile` object. After the execution of the `aldebaranMIN_STD_I` method within the `CADP` object has been finished, the control flow gets back to the `CADPAdapter`. Now the adapter

1. wraps the computational result (obtained via the second parameter of the C++ method) into an HLL AUTFile object,
2. wraps the diagnostic result into a ETIResult object, and
3. finally passes the control flow back to the interpreter.

Figure 6-16. Handling Input/Output Data

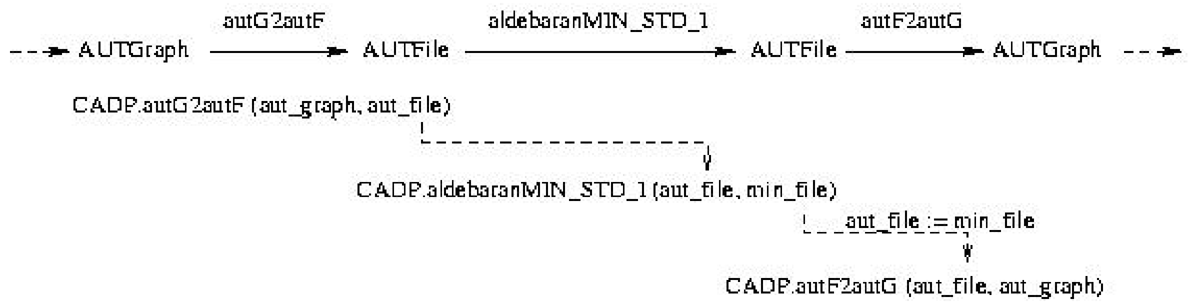

Input Data
Computational Result
Diagnostic Output
%function CADP.aldebaranMIN_STD_I (ref AUTFile: in, ref AUTFile: out):ETIResult
string CADP::CADP.aldebaranMIN_STD_I (const AUTFile& in,const AUTFile& out) throw (ETIException)
aldebaran −std −imin in.aut > out.aut

6.2.2.3. Inter-activity Communication

Inter-activity communication is realized by means of data passing. This means, that the two activities can share information by using the same variable identifier to store the data for the transfer. Depending on the coordination facility, the activity communication must either be programmed by hand (HLL-based coordination) or it is provided by the activity execution context (synthesis-based coordination). Since we have already presented HLL-based coordination in Section 2.1.3, we will now focus on inter-activity communication in the context of the execution of a coordination sequence.

After the execution of an activity has been finished, its computational result is passed, via the second argument of the corresponding HLL function call, back to the interpreter. The interpreter can now pass this object to the subsequent activity as first parameter, if the HLL code uses the same variable identifier to reference the data. The upper part of Figure 6-17 shows a fragment of the coordination sequence documented in Figure 5-7. In the lower part of the same figure the corresponding HLL code is presented. Here, the dotted arrows document how the data is passed from one activity to another.

Figure 6-17. Passing Data between Activities


autG2autF
aldebaranMIN_STD_1
autF2autG
AUTGraph
AUTFile
AUTFile
AUTGraph
CADP.autG2autF (aut_graph, aut_file)
CADP.aldebaranMIN_STD_1 (aut_file, min_file)
aut_file := min_file
CADP.autF2autG (aut_file, aut_graph)

Conceptually, this procedure is well-defined, since the synthesis component only delivers type-safe coordination sequences, i.e. sequences of activities where the output type of an activity is the same as the input type of the subsequent one. From the HLL programming point of view, this requires some guidelines for writing the tool-coordination HLL-code (see Section 7.5.3.3.1).

Chapter 7. The Tool Integrator’s Tasks

Tool integration (see also [BMW97]) is the task of adding new activities and types to the ETI tool repository. After the integration the activities and types can be accessed via the ToolZone client. In general, integrating software tools is a complex task. But the ETI platform provides an environment where the integrator can focus on his expertise (see e.g. Section 6.2.1.3.2), i.e. the knowledge on the tool to be integrated.

To make tool integration as easy as possible it is important, that the tool integrator is guided by a clearly defined process which beside others

- documents the specific hooks which are offered by the ETI platform for the integration of new types and activities and
- presents solutions for frequently occurring integration situations.

This also ensures that the source code of the platform extensions in form of new activities and types remains maintainable.

This chapter covers the ETI tool integration process. It uses the flexible component-integration facilities offered by the METAFrame project (see [Cla97]). In the context of the ETI platform, we embed this technology into workflows, utility tools and guidelines customized for the integration task.

Process Overview

To add a new software tool to the ETI tool repository, the integrator has to

1. split up the tool to be integrated conceptually into a set of ETI activities, and to choose or invent appropriate input and output types,
2. classify the activities and the associated types within ETI’s taxonomies,
3. define the activities’ implementation constituent,
4. specify the stand-alone constituent of the activities.

If the chosen activity is synthesis-compliant, the tool integrator must also

1. specify the activity’s interface constituent and
2. define the tool-coordination constituent of the activity.

The easiest way to integrate a tool into the ETI platform is the *ETI Integration Light*. This process applies primarily to tools providing their own GUI. The big disadvantage of this kind of tool integration is that lightly integrated activities cannot be used for tool coordination. They can only be executed in stand-alone mode. This means, that the tool is integrated as one monolithic activity, which cannot be connected to other ETI activities. In contrast to the ETI

Integration Light, the *ETI Advanced Integration* needs more integration effort. But it results in activities which can be coordinated with any other activity available in the tool-repository.

The integration process, in particular the programming in the context of the advanced integration, seems to be a time-consuming and often not challenging task. But adding new activities over and over again is indispensable for the success of the ETI platform. To ease this procedure, the ETI platform provides a process including coding guidelines and some utility tools to make the labor of integration more convenient.

Outline of this Chapter

The focus of this chapter are the steps which require programming effort, i.e.

- the realization of an activity's *implementation constituent*,
- the definition of an activity's *stand-alone constituent*, and
- if required, the definition of an activity's *tool-coordination constituent*.

Whereas the realization of an activity's implementation constituent is done by *creating METAFrame modules* providing the corresponding HLL function and data types (see Section 6.1.2.1), the specification of the stand-alone and tool-coordination constituent is done via *HLL programming*.

To present the integration process and to point out where the tool integrator is supported during his task, this chapter is organized as follows.

From a coarse granular point of view, Section 7.1, Section 7.2, and Section 7.3 cover aspects that are relevant for both, the light and the advanced integration processes. Section 7.4 and Section 7.5 then apply the presented concepts to the two kinds of tool-integration, i.e. Integration Light and Advanced Integration, respectively. Most of this chapter is concerned with the advanced integration. Whereas the light integration is rather simple, there are a lot of aspects that have to be presented with respect to the advanced integration process.

Concretely, *Section 7.1* illustrates the conceptual tasks.

> Here
>
> - the taxonomy extension (Section 7.1.1) and
> - the specification of the interface constituent of an synthesis-compliant activity (Section 7.1.2)
>
> are presented.

Section 7.2 gives an overview of the implementation tasks of the integration process.

> For this, the section introduces the subtasks which are concerned with the concrete implementation of an activity or type:

- encapsulation and
- HLL extension

result in the METAFrame modules providing the activity’s implementation constituent including the required types and

- HLL programming

is used to define the activity’s stand-alone and tool-coordination constituents.

Section 7.3 presents the integration support offered by the ETI platform.

It gives an overview of the utility programs and the coding guidelines that ease the advanced integration process.

Section 7.4 covers the ETI Integration Light.

This section points out that the light integration is on the one hand much easier than the advanced integration. But on the other hand this process does not prepare the chosen tool’s features for coordination.

Section 7.5, the focus of this chapter, goes into the details of the ETI Advanced Integration.

Here, we cover the integration of new data types in Section 7.5.1 (files), and Section 7.5.2 (graphs). Since the integration of graphs systems is very similar to graph integration, this aspect is delegated to Appendix A. Finally, we go into the details of integrating activities in Section 7.5.3.

Section 7.6

summarizes this chapter by talking about the effort which is needed to integrate tools into the ETI platform.

Still more technical information on the ETI integration process can be found in [Bra99]. Note that a thorough understanding of the design of the ToolZone software is essential. See Chapter 6 for more details on this issue.

7.1. Conceptual Modeling

After the chosen tool has been split up into a set of activities and adequate types have been invented (see Step 1 of the integration process), the activities and the associated types have to be classified in the activity and type taxonomy, respectively. Additionally, the interface constituents of all synthesis-compliant activities have to be defined. This procedure extends ETI’s coordination language by new identifiers. Subsequently, end users are able to define loose specifications which contain the new types and activities (see Section 2.1.4.2). However, since the concrete implementation of the new repository entities is still missing, neither HLL programs nor coordination sequences using the new types and activities can be executed at this stage.

7.1.1. Classification within the Taxonomies

To modify the type and activity taxonomies, currently a text file storing the type and activity information of the tool repository has to be edited. In a future version of the platform, this information will be editable via a graphical user interface (see Chapter 16). To add a new entity to a taxonomy, first of all a group has to be chosen to which the new taxonomy entry should be "connected". Depending on the current status of the taxonomies, either an adequate group can be found, or it may have to be invented and added to the taxonomy.

> **Tip:** If the tool to be integrated provides more than one type or activity, one entry should be added to the taxonomy that defines a group representing all types or activities provided by this tool.

Example 7-1shows the general process of classifying a new type by means of the type AUTFile.

Example 7-1. The AUTFile Entries in the STTT Repository File

In the following we will add the type AUTFile to the type taxonomy as an example. After an appropriate group has been chosen to connect the new type to (here the group CADPFile shown in Example 7-1 line **(1)**), at least the following two entries have to be added to the repository file to extend the taxonomy:

1. An entry that declares the entity to be added to the taxonomy and establishes an is-a relation to the previously chosen group (see **(2)** in Example 7-1).
2. An entry which establishes a link between the new entity and the HTML file containing its documentation (see **(3)** in Example 7-1).

```
% Declaration of the type CADPFile as a descendant
& of the type group File.
% The type group CADPFile represents all
% file types provided by the CADP Tool Kit
tax (type, File, CADPFile).                (1)

% Declaration of the type AUTFile as a "special" CADPFile type
tax (type, CADPFile, AUTFile).             (2)

% Reference to the AUTFile type documentation page
tax_info (type, AUTFile, ’AUTFile.html’).  (3)
```

Note: If the entity to be added is an activity, the keyword `module` must be used instead of `type`. This is due to the fact that the ToolZone software uses the synthesis component of the METAFrame environment, which offers types and modules. Whereas ETI types are represented as types in the synthesis component, ETI activities are modeled as modules.

The documented entries specify only the repository file entries which are required to add a new entity to one of ETI’s taxonomies. Additionally, entries establishing further useful *is-a* relations between the new entity and other entities of the taxonomy can be added. To define a new relation representing that a type T_1 *is-a* type T_2, the repository file must be extended by the following entry:

tax (type, T_2, T_1).

With respect to the DAG presentation of the taxonomy, each entry of this kind models an edge starting at the node T_2 and ending at T_1.

7.1.2. Defining the Activity’s Interface Constituent

If a synthesis-compliant activity is to be declared within the tool repository, an entry which specifies the activity’s interface constituent has to be added. This is of the form of

module (*activity_name*, *input_type*, *output_type*).

Example 7-2 shows the specification of the interface constituent of the `aldebaranMIN_STD_I` activity introduced in Section 6.2.2.

Example 7-2. Specification of the `aldebaranMIN_STD_I`’s Interface Constituent

```
% interface constituent of aldebaranMIN_STD_I
module (aldebaranMIN_STD_I, AUTFile, AUTFile).
```

7.2. The Integration Process Distilled

After the conceptual modeling of the activity and the associated types (see Section 7.1) have been finished, two major tasks have to be performed to provide the implementation constituent of the activity:

1. Integration of the required data types.
2. Integration of the chosen tool feature.

Each task consists of two subtasks which make the data types and the tool feature available to the HLL interpreter as new HLL types and functions:

1. Encapsulation:

Encapsulate the data types and functionality provided by the tool to be integrated into C++ [Str97] classes. These classes constitute the encapsulation code shown in Figure 6-2.

2. HLL Extension:

Extend the HLL by writing a module adapter specification on the basis of the encapsulation code and build the METAFrame modules which make the chosen functionality and data types available as HLL functions and types.

If a tool feature (and not a type) is to be integrated, the stand-alone and tool-coordination constituent of the activity has to be defined. This is done in the HLL programming subtask:

3. HLL Programming:

Define the stand-alone constituent of the activity as an HLL program. If the chosen activity is synthesis-compliant, the tool-coordination constituent also has to be provided in form of a HLL program.

7.3. Integration Support

When we designed the ETI platform, one of our major concerns was a simple and standardized integration process, which should make the integration tasks as easy as possible.

For this the ETI platform offers

- *utility software* which generates C++ encapsulation code and METAFrame adapter specifications from specifications of the type and the activity to be integrated (see Section 7.3.1).
- *design-level and coding guidelines* that standardize the integration architecture and code (see Section 7.3.2).

7.3.1. Utility Software

The ETI platform provides several utility tools which generate C++ encapsulation code and METAFrame adapter specifications from specifications of the type and the activity to be integrated. These tools semi-automize the integration process, since they generate substantial parts of the code which is required to integrate a tool. Table 7-1 shows an overview of the tool support which is currently offered to simplify the advanced integration process.

> **Note:** A complete list of all functions provided by the generated C++ classes and adapter specifications can be found in [Bra99].

Table 7-1. Tool-based Integration Support

Entity	Encapsulation	HLL Extension	HLL Programming
File	fully automatic	fully automatic	not required
Graph	semi automatic	fully automatic	not required
Graphs System	manual	fully automatic	not required
Activity	manual	manual	manual

Although the functionality offered by the generated C++ classes and adapter specifications cover most integration scenarios, in some special cases the code must be implemented manually. In the latter case, the C++ classes building the encapsulation code are realized on the basis of the API of the ToolZone software documented in Chapter 6. The creation of METAFrame module adapters is detailed in [Hol-b].

7.3.2. Coding Guidelines

In addition to the tools documented in the previous section, the integration process is accompanied by design-level and coding guidelines. Similar to other well-known coding guidelines like the Code Conventions for the Java Programming Language [JavaCodeConv], they contain rules which beside others settle how identifiers of classes, methods and data attributes should be built. The ETI coding guidelines, which can be found at the ETI Community Online Service, also provide platform-specific suggestions which simplify the integration process and keep the platform's architecture uniform. The most important guidelines which are relevant to the examples presented in this thesis are listed below.

For every tool which is to be integrated into the platform:

1. Implement *one* C++ class that provides *static* methods giving access to the tool features to be integrated.
2. For each type needed by one of the tool features implement *one* C++ class encapsulating this type.
3. Build *one* METAFrame module that exports HLL functions wrapping the static methods of the C++ class. This module makes functionality offered by the encapsulated tool available as HLL functions.

4. Build *one* METAFrame module for each tool data type implemented in step 2. This HLL-enables the encapsulated tool data types.
5. Use the same identifier for a function/method or a data type within the taxonomies, module adapters and encapsulation code.

7.4. Integration Light

The easiest way to integrate a tool into the ETI platform is the *ETI Integration Light*. This process applies primarily to tools providing their own GUI. Here, no encapsulation and no HLL extension is required, since the required data types and functions are already available in the core of the ToolZone software. Even the tasks which have to be performed within the conceptual modeling and HLL programming phases are trivial.

As already mentioned at the beginning of this chapter, lightly integrated tools can only be executed in stand-alone mode. A lightly integrated tool always runs on the application host (see Figure 2-6). The GUI of the tool is transfered to the client machine using the X protocol [Nye95, XWindows]. Technically, the remote display of the GUI via the Internet is problematic. On the one hand a lot of data has to pass the Net, on the other hand most firewall configurations disallow the remote display for security reasons. But in an Intranet scenario, where bandwidth is not a big issue and where all machines run in a trusted environment, this feature can be very useful.

In the rest of this section we document

- the conceptual modeling (Section 7.4.1 and
- HLL programming (Section 7.4.2) tasks

in the context of the ETI Light Integration.

7.4.1. Conceptual Modeling

The modeling of a lightly integrated tool as an ETI activity is very simple. It is always represented by exactly *one* activity having input and output type ETINone. This activity has to be a descendant of the special activity group named `stand_alone`. This is due to the fact that lightly integrated tools should not be considered by the synthesis process for the generation of coordination sequences.

Example 7-3 shows the entries that were added to the ETI tool repository file to fully specify the classification and the interface constituent of the activity `xeuca`.

Example 7-3. The `xeuca` Activity

The activity `xeuca` represents the **xeuca** tool contained in the CADP toolkit (see Section 4.1.1). In this example, we put the `xeuca` activity into the activity group `misc` (see **(2)** in Example 7-3) which is a descendant of the group `stand_alone` (see **(1)** in Example 7-3). Line **(3)** of Example 7-3 defines the declarative view, i.e. input/output behavior, of the `xeuca` activity. It transforms an object of type ETINone into an object of the same type.

```
tax(module, stand_alone, misc).   (1)
tax(module, misc, xeuca).         (2)
module(xeuca, ETINone, ETINone).  (3)
```

7.4.2. HLL Programming

During the HLL programming phase, the integrator must provide two HLL code fragments which define the execution behavior of the activity. Since a lightly integrated tool can only be run in stand-alone execution mode, the HLL code specifying the tool-coordination runtime behavior is trivial. It consists of the empty statement denoted as the empty string.

The stand-alone HLL code of a lightly integrated tool is simple too, since the ToolZone software provides the METAFrame module `ETI` implementing beside others the functionality for remote execution. The remote execution feature can be accessed by the HLL function `rexec` contained in the `ETI` module. Example 7-4 shows the stand-alone execution code of the `xeuca` activity.

Example 7-4. The stand-alone Execution Code of the `xeuca` Activity

```
ETI.rexec("cd /eti/public/cadp; xeuca");
```

7.5. Advanced Integration

The next sections go into the details of the advanced integration process. They present the extension capabilities of the ToolZone software, sometimes on the level of its C++ application programming interface. Although the rest of this chapter is very technical, it is important to be presented since it emphasizes the flexibility of the ToolZone software and shows that tool-integration is easier than one might expect at first sight. Since a lot of information has to be provided concerning this kind of integration, this section covers the most part of the chapter.

This section is organized as follows. The first two sections present the integration of new data types: files (Section 7.5.1), graphs (Section 7.5.2). Afterwards the integration of tool functionality is illustrated in Section 7.5.3. All three sections are organized along the technical tasks which have to be performed to provide the implementation of a new repository entity:

1. encapsulation,
2. HLL extension and
3. HLL programming, if the chosen entity is an activity.

Note that the integration of graphs systems is presented in Appendix A.

7.5.1. Files

This section goes into the details on how to integrate a new file type into the tool repository. At the beginning of each integration process, the integrator should choose a reasonable, unique identifier for the file type to be integrated. According to the coding guidelines, this identifier is used to name C++ classes, METAFrame modules etc. A good guideline is to choose the file name suffix as identifier. But other identifiers work of course as well.

> **Note:** Similar to Java files, ETI file objects store only meta data of the corresponding file, i.e. they are containers encapsulating a file type and a file name amongst others. It is the responsibility of the application to open, close, etc. the associated files on the hard disk.

The ToolZone software distinguishes two kinds of file objects: objects representing temporary files and objects representing non-temporary files. They differ in their creation and deletion behavior.

temporary file objects:

> When an object that represents a temporary file is created, a temporary file name is automatically determined by the tool management application. It is the responsibility of the file object to create the associated file on the hard disk, too. At the time when the object is deleted, the corresponding file on the hard disk is also removed.

non-temporary file objects:

> The creation of a file object representing a non-temporary file requires the specification of the associated file name. It is required that the associated file on the hard disk already exists. The object deletion has no effect on the file located on the hard disk.

In this chapter we will be integrating as an example the file type AUT. It represents files with suffix `.aut` storing labelled transition systems that can be analyzed using the CADP toolbox (see also Section 4.1.1).

7.5.1.1. Encapsulation

After the conceptual modeling, the chosen file type has to be encapsulated into a C++ class. This class must be derived from the class `ETIFile` (see Section 6.2.1.2) provided by the core of the ToolZone software. To ease the maintenance of the software, the name of this class should be built out of the identifier, chosen at the start of the integration process, followed by the string "File" (here "`AUTFile`").

There are two ways to provide the encapsulation source code:

1. The code is implemented by the tool integrator.
2. The code is generated using a utility tool offered by the ETI platform (see Section 7.3.1).

If the C++ class encapsulating the new file type is programmed by hand it must at least offer:

A default constructor

> that creates a file object representing a *temporary* file of the chosen type. Since the handling of temporary files is already provided by the base class, a minimal implementation of this constructor simply calls the constructor of the base class passing an internal file type identifier and the file name suffix as arguments. Example 7-5 shows the default constructor of the class `AUTFile`.
>
> **Example 7-5. The default Constructor of the Class `AUTFile`**
>
> ```
> AUTFile::AUTFile ()
>     : ETIFile ("ETI_FILE_AUT", ".aut")
> {
> }
> ```
>
> The chosen file type identifier must be unique within the ETI environment. It is used to determine whether a file type is known (can be handled) by the tool management application and to get information on the file name suffix that is associated to this file type. As a guideline, this identifier should conform to the following format: the string "ETI_FILE_" followed by the identifier that has been chosen at the start of the integration process (in our example this identifier is "ETI_FILE_AUT").

A constructor getting a const string reference as argument

> that creates a file object of the chosen type representing a *non-temporary* file. The name of the file is passed as the argument to the constructor.
>
> A typical implementation of this constructor simply calls the constructor of the base class with the file name, the file type identifier and the file suffix as arguments (see Example 7-6)

Example 7-6. The "const string&" Constructor of the Class `AUTFile`

```
AUTFile::AUTFile (const string& f)
    : ETIFile (f, "ETI_FILE_AUT", ".aut")
{
}
```

The encapsulation code can also be generated using a utility program offered by the ETI platform. This program generates the encapsulation code in form of a C++ header file on basis of the chosen file type identifier. The header file only implements the two required constructors in the way documented in Example 7-5 and Example 7-6. In most cases the generated code is sufficient for the encapsulation of a new file type.

7.5.1.2. HLL Extension

After we have encapsulated the file type into a C++ class, we have to make the new file type including all operations accessible to the HLL interpreter. This is done by writing a METAFrame module adapter specification and generating the corresponding METAFrame module (see Section 6.1.2.1 for the general procedure).

Similar to the code of the encapsulation class, the integrator can write the adapter specification by hand or he can generate a default implementation using a utility program (see Section 7.3.1).

If the adapter specification is implemented manually, it is important to note that it must export the `init` function (see Example 7-7). This function registers the corresponding file type with the tool management application as soon as the METAFrame module is loaded into the interpreter. The registration is performed by invoking the static method `registerFileType` of the class `ETIFile` (see **(1)** in Example 7-7).

Example 7-7. The `init` Function of the `AUTFile` Module

```
%init
{
    try {
        ETIFile* new_type = new AUTFile ();
        ETIFile::registerFileType (new_type);     (1)
        cout <<  "registered file type " << new_type->getType ()
             << endl;
    } catch (const ETIException& e) {
        cout << "Module AUTFile: file type "
             << new_type->getType ()
             << " already registered . .." << endl;
    }
}
```

To complete the HLL extension, the METAFrame adapter C++ source code, the adapter itself and the corresponding METAFrame module have to be generated. As documented in Section 6.1.2.1, this process is fully automated.

7.5.2. Integrating a new Graph Type

Integrating a new graph type means "linking" a graph file format or a graph library to METAFrame's PLGraph library [BBCD97, PLGraph]. After the integration, graph objects of the associated format or library can be modified using ETI's Graph Editor (see Figure 5-10). The main effort of establishing the link is to implement

- the load and save functionality reading and writing the native file format and
- methods to modify the node, edge and graph information via the editor

Besides the "functional" connection to the ToolZone software, the PLGraph Library serves as some kind of Intermediate Representation Language to provide transformers between different graph file formats and libraries.

Similar to the process of integrating a file type, the first step when integrating a graph type is to define a unique identifier. This identifier will prefix all C++ classes, METAFrame module names, HLL types, etc. provided to integrate the chosen graph format.

In this chapter we will be integrating the AUT graph type as an example. This graph type "links" the `aut`-file format, which is used to store edge-labelled transition system, introduced in Section 4.1.1, to the PLGraph Library.

7.5.2.1. Encapsulation

To encapsulate a new graph type, the following steps have to be performed. Let <*`ID`*> be the identifier of the chosen graph type. Then the integrator has to provide

1. a node label class named <*`ID`*>`NodeLabel` encapsulating the node specific data and methods (see Section 7.5.2.1.1),
2. an edge label class named <*`ID`*>`EdgeLabel` encapsulating the edge specific data and methods (see Section 7.5.2.1.2),
3. a graph label class named <*`ID`*>`GraphLabel` encapsulating the graph specific data and methods (see Section 7.5.2.1.3),
4. a class named <*`ID`*>`FileFormat` encapsulating the graph specific native file format (see Section 7.5.2.1.4) and finally

5. generate the facade graph class <*`ID`*>`Graph` (see Section 7.5.2.1.5).

> **Note:** When implementing the required classes, the integrator may of course choose the names of the classes on his own. But following the ETI coding conventions keeps the software uniform and makes it easier to maintain.

7.5.2.1.1. Implementing a Node Label Class

The node label class must inherit from the class `ETINodeLabel`. It encapsulates data and methods specific to the node type associated to the graph type to be integrated. In addition, it has to provide the following set of methods which are mainly used by the GUI to read or modify the label data:

`virtual void parse (const PLLabelData& data)`

The method `parse` initializes the label data from the passed `PLLabelData` object *`data`*. It it used to modify the label data via the node inspector window of the ToolZone client (see Figure 5-10).

If the specific node type does not provide any information associated to a node, the `parse` method needs not to be implemented. Here it is important, that the method `isInspectable` of the chosen node label class returns `false` (see below).

`virtual void unparse (PLLabelData& data) const`

This method returns a representation of the label data encapsulated into the `PLLabelData` object `data`. It is used to show the label data in the node inspector window of the ToolZone client (see Figure 5-10).

Similar to the `parse` method, the `unparse` method must only be implemented, if the chosen node type provides information associated to a node.

`virtual PLNodeLabel* proto (void) const`

`proto` returns a new (initial) instance of a node label object of the chosen class. It is used internally, when adding a new node to the graph. A typical implementation of the method looks like the one shown in Example 7-8.

Example 7-8. The `AUTNodeLabel`'s `proto` Method

```
PLNodeLabel*
AUTNodeLabel::proto (void) const
{
    return new AUTNodeLabel ();
}
```

`virtual void assign (const PLLabelClass* l)`

The method `assign` copies the label data from the label object *`l`* to the `this` object. A standard implementation of this method would check, if the passed label object is not equal to `this` and then call a method `copy` which implements all the real copying of the label data. The `copy` method can then also be used when implementing the copy-constructor and the assignment operator. Example 7-9 shows the implementation of the `AUTNodeLabel`'s `assign` and Example 7-10 the associated copy method.

Example 7-9. The `AUTNodeLabel`'s `assign` Method

```
void
AUTNodeLabel::assign (const PLLabelClass* l)
{
    if (l != NULL && l != this) {
        ETINodeLabel::assign (l);
        const AUTNodeLabel* nl
            = static_cast<const AUTNodeLabel*> (l);
        copy (*nl);
    }
}
```

Example 7-10. The `AUTNodeLabel`'s `copy` Method

```
void
AUTNodeLabel::copy (const AUTNodeLabel& l , bool)
{
    node_id = l.node_id;
}
```

`virtual bool isInspectable (void) const`

Depending on the type of the graph, it may occur that there is no information associated to a node. In this case it is recommended to overwrite the method `isInspectable`. If this method returns `false` instead of `true`, the Apply button and the inspector window's text area (see Figure 5-10) will be disabled preventing that a user types in any data.

> **Tip:** If the method `isInspectable` returns `false`, the `parse` and `unparse` methods need not to be implemented.

7.5.2.1.2. Implementing an Edge Label Class

Implementing an edge label class is very similar to the process of providing a node label class. The label class must inherit from the class `ETIEdgeLabel` and implement at least the following methods:

- `virtual void parse (const LabelData& data)`,
- `virtual void unparse (LabelData& data) const`,
- `virtual PLEdgeLabel* proto (void) const`,
- `virtual void assign (const PLLabelClass* l)`,
- `virtual bool isInspectable (void) const`

which have the same meaning as the ones provided by the node label class.

7.5.2.1.3. Implementing a Graph Label Class

A graph label class (derived from `ETIGraphLabel`) provides three types of methods:

1. one that contains the basic label functionality (similar to the ones provided by the node and edge label classes),
2. one that links the label to the ToolZone software and
3. one that is specific to the chosen graph type.

Since we already went into the details of the basic functionality in the previous two section, we only list the basic label methods and their signatures for completeness:

- `virtual void parse (const PLLabelData& data)`,
- `virtual void unparse (PLLabelData& data) const`,
- `virtual PLGraphLabel* proto (void) const`,
- `virtual void assign (const PLLabelClass* l) `,
- `virtual bool isInspectable (void) const`.

In addition to the basic label functionality, each graph label class must implement methods specific to the ToolZone software. These methods provide functionality, which will be accessed by the ToolZone client, amongst others. These methods are:

`virtual void check (ETIGraphInfo& info) throw (ETIException)`

Changing node, edge or the graph information may result in an inconsistent graph object. For this, the method `check` must be provided, which checks the consistency of the graph information. This method should at least ensure that the file content generated by a save operation can be read without any errors by a subsequent load method invocation. The *`info`* argument can be used to return diagnostic information to the application.

`check` throws an `ETIException`, if the check failed. Note that the `ETIGraphInfo` argument can be used to pass detailed information on the source of the problem to the calling application.

Example 7-11 shows the implementation of the `check` method of the class `AUTGraphLabel`. Here, the graph object is stored into a temporary file which is analyzed using the **aldebaran -info** command. If the command execution results in an error, the check fails, otherwise it terminates successfully.

Example 7-11. The `AUTGraphLabel`’s `check` Method

```
void
AUTGraphLabel::check (ETIGraphInfo& i) const throw (ETIException)
{
    // 1. save this graph object to a temporare file
    // but do not forget to store ’old’ filename to reset it after
    // temporary save!
    string file_name
        = getGraphClass ()->getGraphFilename ().get().c_str();
    AUTFile aut_file;
    toFile (aut_file.getFileName ());
    getGraphClass ()->getGraphFilename ().set (file_name);

    // 2. check the generated file content using the command
    //    aldebaran -info <file_name>
    string system_call = "aldebaran -info "
                         + aut_file.getFileName ();
    string tool_output;

    cout << "Executing: " << system_call << " ... " << flush;
    int ret = SystemUtils::execCommand (system_call, tool_output);
    cout << "done (" << ret << ")" << endl;

    // 3. analyze the result of the system call
    if (ret > 0) {
        i.setMessage ("Could not execute\n" + system_call);
        return false;
    }

    if (tool_output.find ("error") != string::npos) {
        i.setMessage (tool_output);
        return false;
    }

    return true;
}
```

`virtual const char* getNativeFileFormat () const`

This method returns the file format identifier to which this graph type is linked to. Amongst others, it is used by the methods `fromFile` and `toFile` to determine the native file format filter (see Section 6.2.1.3.2). Example 7-12 shows the implementation of the `getNativeFileFormat` method of the class `AUTGraphLabel`.

Example 7-12. The `AUTGraphLabel`'s `getNativeFileFormat` Method

```
const char*
AUTGraphLabel::getNativeFileFormat () const
{
    return "ETI_FILE_AUT";
}
```

In addition to the mandatory basic and platform-specific methods, there are two methods whose implementation is optional, but strongly recommended:

`virtual void init (PLGraphClass* g)`

The `init` method is automatically called, whenever a graph label object is attached to the graph by calling the `addGraphLabel` method of the class `PLGraph`. Here, graph label data which depend on the concrete graph object, to which this label object is be bound, can be initialized (e.g. the number of nodes).

According to the ETI design style guideline the `init` method should be used to add the appropriate node and edge label objects to the graph. This can be done using the methods `addNodeLabel` and `addEdgeLabel` of the class `PLGraph`. This ensures

- that the label classes and the code generated for the corresponding facade class (see Section 7.5.2.1.5) work properly in combination and
- that during runtime of the tool management application the proper label objects are available, when required (see also Section 6.2.1.3).

Example 7-13 presents the `init` method of the class `AUTGraphLabel`. It shows that in general the implementation of this method consists of three steps in order:

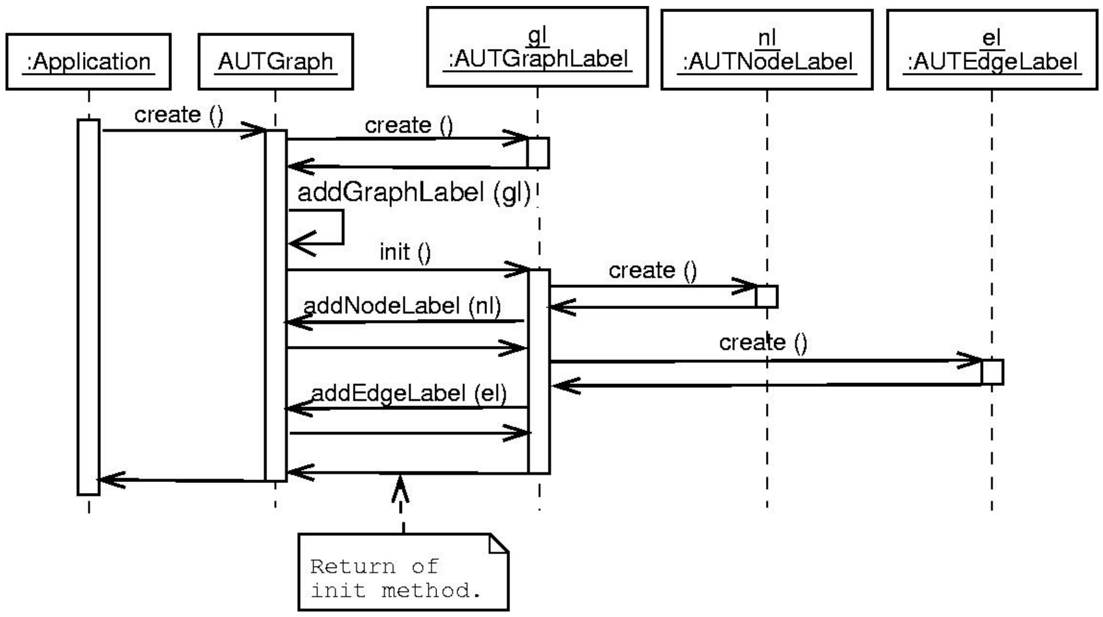
1. Call the `init` method of the super class `ETIGraphLabel`.
2. Add corresponding node and edge label objects to the graph using the method `addNodeLabel` and `addEdgeLabel`.
3. Initialize the label data that depends on the concrete graph object.

Example 7-13. The `AUTGraphLabel`'s `init` Method

```
void
AUTGraphLabel::init (PLGraphClass* g)
{
    // call init() method of super class
    ETIGraphLabel::init (g);

    // add corresponding node and edge label objects to the graph
    getGraphClass ()->addNodeLabel (new AUTNodeLabel ());
    getGraphClass ()->addEdgeLabel (new AUTEdgeLabel ());

    // initialize the data that depends
    // on the concrete graph object
    num_of_nodes = g->getAllNodes ().size ();
    num_of_edges = g->getAllEdges ().size ();
}
```

Figure 7-1 shows the interaction of the generated facade class and the label classes provided by the user in a UML sequence diagram [BRJ98, FS99]. In this example we focus on the graph class' `init` method and the triggered creation of the node and edge label object.

Figure 7-1. Creating an `AUTGraph` Object



`virtual void clear (void)`

The default implementation of the `clear` method within the class `ETIGraphLabel` removes all nodes and edges from the graph by calling `clear` on the associated graph object. If you want to clear the graph type specific data too, you have to overwrite the method `clear` of the class `ETIGraphLabel`. Example 7-14 shows the implementation of this method used for the integration of the AUT graph type.

Example 7-14. The `AUTGraphLabel`'s `clear` Method

```
void
AUTGraphLabel::clear (void)
{
    // clear graph label specific data
    first_state = 0;
    num_of_nodes = 0;
    num_of_edges = 0;

    // call clear method of the super class
    ETIGraphLabel::clear ();
}
```

7.5.2.1.4. Implementing a File Format Class

A file format class encapsulates the functionality which is needed to load a graph object from and to store it to a file. The ToolZone software distinguishes two kinds of file formats: the (unique) native file format of the graph type, and optional export and import filters (see Section 6.2.1.3.2).

The native file format of a graph type is the file format which it is linked to, in our example the `aut`-file format of the CADP toolkit. The associated file type identifier can be retrieved by invoking the `getNativeFileFormat` method on a graph object (see Section 7.5.2.1.3). When a graph object is loaded or stored using the `fromFile` and `toFile` methods, the control flow is delegated to the file format object referenced by the `getNativeFileFormat` method (see Figure 6-13).

Every file format class available in the ToolZone software must be derived from the class `PLFileFormat` which provides the following methods:

`void init (PLGraphClass* g)`

> The `init` method should perform initialization tasks. It is comparable to the `init` method of the graph label class.
>
> The `init` method gets called whenever the file format object is added to a graph object. The graph object to which the file format is added is passed as argument `g`.

`virtual const char* getFormatType (void) const = 0`

> This method returns the file type identifier of the file format which is encapsulated in this class. If this class encapsulates the native file format the returned value must be the same as the one returned by the `getNativeFileFormat` method of the graph object.

`virtual bool canLoad (void) const = 0`

> Checks whether this file format supports reading from a file. It returns `true`, if the encapsulated file format can be read, otherwise `false`. If this method returns `false`, the `load` method needs not to be implemented.

`virtual bool canSave (void) const = 0`

> Checks whether this file format supports writing to a file. It returns `true`, if the encapsulated file format can be written, otherwise `false`. If this method returns `false`, the `save` method needs not to be implemented.

`virtual void save (ostream& o, const set<PLNode*>& nodes, const set<PLEdge*>& edges) = 0`

> Writes the set of nodes *`nodes`* and the edges *`edges`* to the output stream *`o`*.

`virtual void load (istream& i) = 0`

Initializes the graph object from the input stream `i`.

Every file format which should be associated to a graph type must be registered to the corresponding C++ objects. This is done via an `addFileFormat` call in the constructor of the graph's facade class (see Section 7.5.2.1.5).

7.5.2.1.5. Generating the Facade Graph Class

In addition to the label classes, the encapsulation code of a graph type comprises a top level class (called *facade* class [GHJV95]) which is derived from `ETIGraph` (see Figure 6-10). The facade object can then be used to access the whole graph structure. The corresponding source code (only a C++ header file) is automatically generated by a utility program contained in the ETI platform (see Section 7.3.1).

The generated facade class implements only some management functionality, like a constructor, a destructor and an assignment operator. In addition to that, it makes the graph-type specicifc methods provided by the graph label available. The implementation of the graph type specific methods in the facade class just delegate the method invocation to the corresponding method of the associated graph label object. Example 7-15 shows the implementation of the `generateReceiverList` method of the `HytechGraph` facade class documenting the delegation principle.

Example 7-15. Method Delegation in the Facade Class

```
void
HytechGraph::generateReceiverList (const string& proc_name,
                                   const HytechSystem& system) const
{
    static_cast<HytechGraphLabel*> (getETIGraphLabel ())
        ->generateReceiverList (proc_name, system);
}
```

7.5.2.2. HLL Extension

The next step in the integration process is to provide a new HLL type <*ID*>Graph, where <*ID*> is the identifier chosen at the beginning of the graph integration process (here `AUT`). This is done by building a METAFrame module that exports the type and the associated set of functionality. For this, the integrator must provide a METAFrame module adapter specification. Similar to the process documented in Section 7.5.1.2, he can write the adapter specification by hand or generate one via a utility program. The generated module exports

- the HLL type,
- functions to load and save the graph objects of the exported type
- functions to display it in the current graphic context and
- functions to transform graph objects of the exported type into generic PLGraph objects.

Like the file modules, each graph module must implement the `init` function which registers the corresponding graph type in the tool management application (see **(1)** in Example 7-16), when the module is loaded into the interpreter. Example 7-16 shows the `init` function of the `AUTGraph` module.

Example 7-16. The `init` Function of the `AUTGraph` Module

```
%init
{
    try {
        ETIGraph* new_graph = new AUTGraph ();
        ETIGraph::registerGraphType (new_graph);   (1)
    } catch (const ETIException& e)
        cerr << "Module AUTGraph: " << e.getMessage ()
             << endl;
    }
}
```

To complete the HLL extension, the METAFrame adapter C++ source code, the adapter itself and the corresponding METAFrame module have to be generated. This task is like in the file integration process fully automated (see Section 6.1.2.1).

7.5.3. Feature Integration

After the data types have been made available within the tool repository, the tool functionality has to be prepared for coordination. Depending on the availability of the tool, the chosen functionality can currently be accessed within the encapsulation code by one of the following methods:

Black Box Integration

If the chosen tool is available in binary format only, and if it is installed on the application host (see Figure 2-6), the chosen functionality can be accessed using UNIX system calls.

White Box Integration

> If the source code of the tool or the libraries implementing the functionality have been supplied by the tool maintainer, the chosen functionality can directly be integrated on source code level.

Remote Box Integration

> Remote Box Integration is white box integration of a tool which is available on a remote host, i.e. not on the application host (see Figure 2-6). Within the encapsulation code interoperability services like RMI or CORBA are used to access the remote features. Note that in contrast to black box and white box integration, remote box integration is an ETI specific term.

Since the encapsulation layer is implemented in C++, other tool access methods can be made available, if necessary.

With respect to the four integration phases (conceptual modeling, encapsulation, HLL extension and HLL programming) only the encapsulation code depends on the kind of integration, i.e. currently black box, white box or remote box integration. The tasks performed in the other three phases are independent of it.

As an example, let us focus on the **aldebaran** feature that minimizes labelled transition systems with respect to the *tau*.a* bisimulation equivalence [Mil89] using the Paige/Tarjan algorithm [FM90, PT87]. This synthesis-compliant activity can be modeled as the activity `aldebaranMIN_STD_I` transforming an object of the abstract type AUTFile, representing the source labelled transition system, into another object of the abstract type AUTFile, representing the minimized one. This means that the activity can be represented by an entry of the form of

```
module (aldebaranMIN_STD_I, AUTFile, AUTFile)
```

within the repository file (see also Section 7.1.2).

In the following sections, we first document the encapsulation process of the black box integration in Section 7.5.3.1. Since the white and remote box integration process heavily depend on the API to access the tool's source code, libraries or the CORBA/RMI interface, there is no standard procedure for these kinds of integration. Following the black box encapsulation, we document the generic HLL extension and HLL programming phases in Section 7.5.3.2 and Section 7.5.3.3. The conceptual modeling is left out since it has already been documented in Section 7.1.

7.5.3.1. Black Box Integration

The *Black Box Integration Process* applies to tools that are only available in binary format and installed on the application host. Here, the tool functionality can only be accessed by executing an operating system command. The different tool features are addressed by command line options. During runtime, the tool execution may be *non-interactive* or *interactive*. In the first case the tool execution terminates without any interaction of the user. In the second case, the tool execution may stop at a certain point waiting for user input. After the user has provided the requested information, the execution of the tool continues.

Black Box Integration supports both kinds of execution modes, but with one restriction: if the tool execution is interactive, the information provided by the user may not influence the *type* of result delivered by the tool. This means, that the following situation may not be happen: depending on the information that is provided by the user during the runtime, the tool execution either results in an object of type T_1 or in an object of type T_2. It must be ensured the return type does not vary.

The rest of this section gives an overview of the encapsulation task performed during the integration process.

The Encapsulation Task

After the conceptual modeling, the next task in the integration process is the encapsulation. With respect to the black box integration process, we encapsulate the chosen tool feature into a static method of the unique C++ class wrapping all features of an integrated tool. Following the design guidelines documented in Section 7.2, the C++ method should have the same name as the activity, here `aldebaranMIN_STD_I`. Similar to the signature of the corresponding HLL function (see Section 7.5.3.2), the signature of the C++ method is standardized (see Example 7-17).

Example 7-17. The C++ Signature Specification of the `aldebaranMIN_STD_I` Method

```
static string aldebaranMIN_STD_I (const AUTFile& in,
                                   const AUTFile& out)
    throw (ETIException);
```

For a synthesis-compliant activity, the signature specification contains two arguments, one representing the input object and one representing the output object.

In general the signature of the C++ encapsulation method is parametric in three dimensions:

1. the *return value* gives access to the diagnostic information (see also Figure 6-16). The information, like the amount of memory which has been consumpted, is generated by the tool during its execution.
2. the *parameter list* handles the input and output data.
3. the *exception* is used to indicate that an unrecoverable error during the tool execution has occurred.

> **Note:** Some readers may find it more natural to deliver the computational result of the activity execution as return value of the method invocation. We did not follow this approach since
>
> - only synthesis compliant activities are guaranteed to return a single object. There might be activities whose execution may result in more than one output. This situation could of course be handled via a composite object, but this design seemed unnatural from our perspective.
> - conceptually, input and output objects belong together. For this they are grouped within the parameter list of the method signature. We did not want to mix input data and diagnostic information within the same portion, i.e. the parameter list, of the method signature.

In the context of black box integration, the following procedure is implemented by the encapsulating C++ method:

1. The operating system command which gives access to the chosen tool feature is built.
2. The command is executed. Within this step the output generated by the executed tool is collected and made accessible within the C++ method.
3. It is checked, if the tool has been executed successfully. If not, an `ETIException` is thrown giving the reason for the failed execution.
4. The output of the tool is analyzed to obtain the computational result. Depending on characteristics of the chosen tool, the computational result may not be contained in the tool output, since it may e.g. be written into a file. In this case, this step is obsolete.
5. The diagnostic result, which is part of the output generated in step 2, is returned.

Example 7-18. The Implementation of the `aldebaranMIN_STD_I` Method

The `aldebaranMIN_STD_I` method encapsulates the corresponding **aldebaran** feature. It can be accessed by the following system call:

aldebaran -std -imin in.aut > out.aut,

where *`in.aut`* is the name of the file storing the labelled transition system which should be minimized and *`out.aut`* is the name of the file into which the result of the minimization is written.

Following the procedure presented above, the `aldebaranMIN_STD_I` method is implemented as follows:

```
static string aldebaranMIN_STD_I (const AUTFile& in,
                                  const AUTFile& out)
    throw (ETIException)
```

```
{
    // 1. the system call is built
    string system_call = "/bin/bash -c \'aldebaran -std -imin  "
        + in.getFileName () + " > " + out.getFileName () + "\'";

    // 2. the command is executed and tool output is returned
    string tool_output;
    cout << "Executing: " << system_call << " ... " << flush;
    int ret = SystemUtils::execCommand (system_call, tool_output);
    cout << " finished (" << ret << ")!!" << endl;

    // 3. it is checked, if the tool has been executed successfully
    if (ret != 0) {
        // an error occurred
        throw ETIException ("CADPOperations::aldebaranMIN_STD_I:\n"
                            + tool_output);
    }

    // 4. no tool output analysis is necessary, since
    //    computational result is written into a file


    // 5. diagnostic output is returned
    return tool_output;
}
```

7.5.3.2. HLL Extension

After the tool’s functionality has been encapsulated into the C++ class it must be made available as HLL function. This is done by writing a METAFrame adapter specification and generating the corresponding METAFrame module (see Section 6.1.2.1).

> **Note:** In contrast to the encapsulation task, the HLL extension is independent of the type of the integration process, i.e. black box, white box or remote box integration, used to encapsulate the chosen tool feature.

There is currently no tool support to generate the adapter specification automatically (see also Table 7-1). It must be provided by the integrator. In the following we will present the generic structure of an adapter specification registering tool functionality to the interpreter. The complete format is documented in [Hol-b].

The adapter specification comprises

- the declaration part and
- the implementation part.

7.5.3.2.1. The Adapter's Declaration Part

The declaration part of the adapter specification (see Example 7-19) contains

the *require block*

which imports all HLL types which are needed to implement the HLL functions exported by this module.

the *header* block

which always remains empty in this scenario.

Example 7-19. The `CADP` Adapter's Declaration Part

```
%require ETIResult Int ETIFile AUTFile AUTGraph BCGFile LOTOSFile
         EXPFile MCLFile SEQFile XTLFile

%header
{
    // %header block can be left out, if empty!
}

%-----------------------------------------------------------------%
```

7.5.3.2.2. The Adapter's Implementation Part

The implementation part of the adapter specification provides three blocks:

1. the header block,
2. the init block and
3. the wrapper block.

As shown in Example 7-20, the *header block* contains only a C++ `#include` statement which imports the header file containing the C++ encapsulation class, here `CADP.hh`.

In contrast to the METAFrame modules exporting a type, the *init block* of a METAFrame module providing tool functionality in general remains empty.

Example 7-20. The header and init block of the `CADP` Adapter's Implementation Part

```
%header
{
    #include "CADP.hh"
}

%init
{
    // %init block can be left out, if empty!
}
```

The *wrapper block* provides the implementation of the exported HLL functions on the basis of the encapsulation class. Each HLL function has a standardized signature. Whereas the return value of the function is always of type ETIResult, the number and types of the parameters depend on the parameters of the corresponding C++ method. A synthesis-compliant activity taking an object of type T_1 as input and delivering an object of type T_2 is represented by an HLL function taking two arguments: One of the *HLL* type T_1 representing the input type and one of the *HLL* type T_2 representing the output type.

> **Tip:** The abstract type and the associated HLL type should have the same name for convenience.

Following this design, the `aldebaranMIN_STD_I` HLL function exported by the `CADP` module takes two AUTFile HLL objects as arguments and returns an ETIResult HLL object. The arguments represent the source and the minimized labeled transition system. Example 7-21 shows the specification of the function's signature within the METAFrame module adapter specification.

Example 7-21. The HLL Signature Specification of the `aldebaranMIN_STD_I` Function

```
%function aldebaranMIN_STD_I (ref AUTFile: in,
                              ref AUTFile: out): ETIResult
```

The implementation of the function wrapper does not provide any *real* functionality. It implements some error handling and delegates the computation to the corresponding method of the C++ encapsulation class. As shown in Example 7-22 the wrapper implements the following procedure:

1. The function's return value is initialized **(1)**.

2. It is checked, whether the passed HLL objects are properly initialized. If not, the HLL function's return value is a wrapped `ETIException` object **(2)**.
3. The computation is delegated to the corresponding method of the C++ encapsulation class **(3)**.
4. It is checked, whether the C++ method has terminated successfully. In this case, the function's return value wraps a `ETIInfo` object encapsulating the diagnostic tool output **(4)**. Otherwise the `ETIException` object thrown by the encapsulation method is wrapped into an HLL ETIResult object **(5)**.

Example 7-22. The `aldebaranMIN_STD_I` Wrapper

```
%function aldebaranMIN_STD_I (ref AUTFile: aut_file,
                              ref AUTFile: min_file): ETIResult
{
    // return value is initialized
    RETURN.value = NULL;                                                   (1)

    // proper initialization of the passed HLL objects is checked.
    if (aut_file.value == NULL || min_file.value == NULL) {
        RETURN.value
            = new ETIException ("CADP.aldebaranMIN_STD_I: "
                                "uninitialized AUTFile object");           (2)
    } else {

        // computation is delegated to the encapsulation class.
        try {
            string tool_output
                = CADPOperations::aldebaranMIN_STD_I (*aut_file.value,
                                                      *min_file.value); (3)
            RETURN.value = new ETIInfo (tool_output);                      (4)
        } catch (const ETIException& e) {
            RETURN.value = const_cast<ETIException&> (e).clone ();(5)
        }
    }
}
```

7.5.3.3. HLL Programming

The result of this task are two HLL code fragments defining the stand-alone and the tool-coordination execution behavior of the activity. The heart of both source code fragments is the HLL function encapsulating the chosen tool functionality (see **(2)** in Example 7-23 and **(2)** in Example 7-24). The remaining HLL commands are only concerned with collecting the input data and handling the output data of this function call.

7.5.3.3.1. Tool-Coordination Execution Mode

The code fragment provided for the tool-coordination mode is run when a coordination sequence is executed by the tool management application. Example 7-23 shows the tool-coordination code of the activity `aldebaranMIN_STD_I`.

Example 7-23. The tool-coordination Code of the `aldebaranMIN_STD_I` Activity

```
var AUTFile: min_file;                                         (1)
exec_result := CADP.aldebaranMIN_STD_I (aut_file, min_file);(2)
aut_file := min_file;                                          (3)
```

For activities whose input type does not equal ETINone, the value of the input object is determined by the value of the output object of the previously executed activity (see Figure 6-17). To ensure that the single code fragments of the activities can be combined to work properly the output object of one activity must be passed as input object to the subsequent activity (see Figure 6-17). On code level, the value passing is implemented by using one global variable to store all objects of a specific type. The canonical identifier of this variable is obtained by the name of the corresponding type where all capital characters are transformed to lowercase ones and words within the identifier are separated by the underscore character '_', e.g. the variable with identifier "aut_file" is used to store the `AUTFile` objects. Since the generated coordination sequences are type-correct, i.e. the input type of an activity is the same as the output type of the predecessor activity, the implementation of data passing is well defined (see Figure 6-17). This process may introduce multiple declarations of the same variable which are simply considered redundant by the HLL interpreter.

> **Note:** Depending on the implementation of the tool feature associated to an activity which has the same input and output type (e.g. activity `aldebaranMIN_STD_I`), using the same variable identifier for the input and output object might be problematic. Within this special scenario, an auxiliary variable has to be provided which stores the result of the activity execution temporarily (see the variable `min_file` in Example 7-23).

In general, the HLL fragment specifying the activity's tool-coordination constituent consists of the following steps:

1. A variable which will contain the output data of the activity is declared.

   Depending on the input/output behavior of the activity

   - declare an auxiliary variable (see **(1)** of Example 7-23), if the input and output type of the activity are the same,
   - otherwise, declare a variable having the canonical identifier.

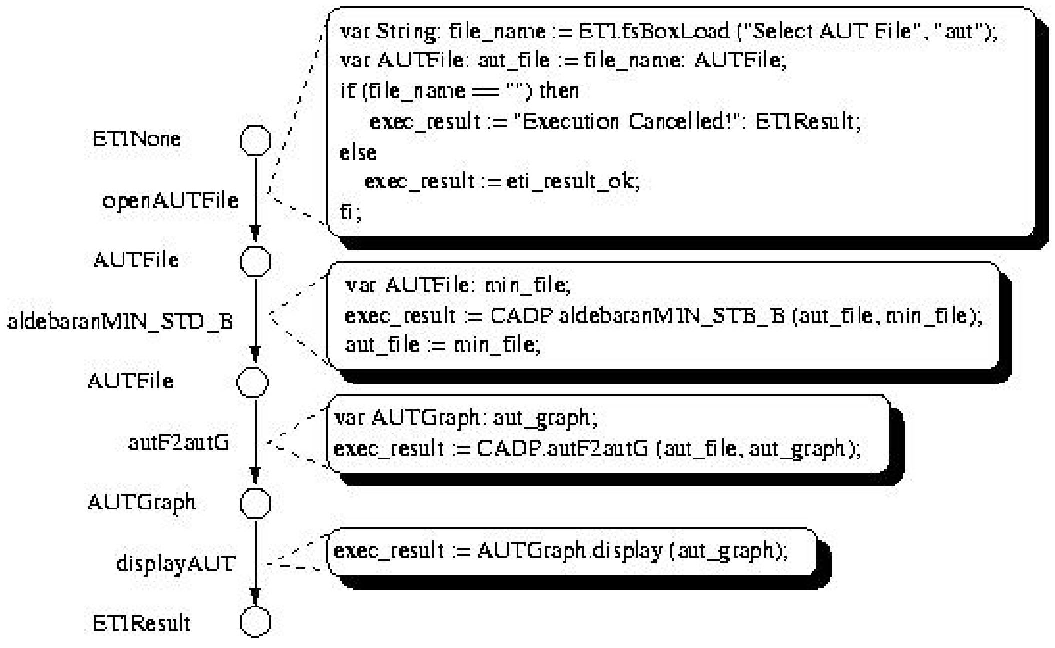
Note that this step is obsolete, if the output type of the chosen activity is ETINone. In this case the corresponding variable exists by default (see e.g `displayAUT` activity in Figure 7-2).

2. Execute the activity by calling the HLL function defined by the activity's implementation constituent (see **(2)** of Example 7-23).
3. Only if the input and output type of the activity are the same, the value of the auxiliary variable has to be assigned to the unique variable associated to the output type (see **(3)** of Example 7-23).

If the input type of the chosen activity is ETINone, it is always the first activity of a coordination sequence. Thus, the input object is of course not provided by a predecessing activity. In this case the code which initializes the input object has to be contained in the activity's tool-coordination code. With respect to the activity `openAUTFile` shown in Figure 7-2, the input object is created using a file whose location is requested from the user. This feature is provided by the `fsBoxLoad` function offered by the `ETI` module.

Figure 7-2 shows an example of a coordination sequence and the HLL code fragments associated to the activities.

Figure 7-2. A typical Coordination Sequence and the associated HLL Code



```
var String: file_name := ETI.fsBoxLoad ("Select AUT File", "aut");
var AUTFile: aut_file := file_name: AUTFile;
if (file_name == "") then
   exec_result := "Execution Cancelled!": ETIResult;
else
   exec_result := eti_result_ok;
fi;
```

```
var AUTFile: min_file;
exec_result := CADP.aldebaranMIN_STB_B (aut_file, min_file);
aut_file := min_file;
```

```
var AUTGraph: aut_graph;
exec_result := CADP.autF2autG (aut_file, aut_graph);
```

```
exec_result := AUTGraph.display (aut_graph);
```

As shown in Figure 7-2 the code fragment associated to an activity is only responsible to declare a variable storing the output object of the activity. The type-correctness of the generated coordination sequence ensures, that there exists already a variable (declared by the previously executed activity) for the input object of the currently executed activity.

7.5.3.3.2. Stand-Alone Execution Mode

In contrast to the tool-coordination mode, where the parameters required for the activity execution are automatically handled by the runtime environment, the parameter handling within the stand-alone code must be explicitly programmed. This means that the value of the input object must be requested from the user, and the value of the output object (if not of type ETIResult) must be handled, too. In most cases the value of the output object is stored to a file.

Example 7-24 shows the stand-alone HLL code of the activity `aldebaranMIN_STD_I`. Here, first the name of the file storing the graph to be minimized is requested from the user (see **(1)**). If the file selection has not been canceled by the user, the HLL function wrapping the chosen tool functionality is run in **(2)**. After the HLL function has successfully been executed, the user is asked to provide the name of the file to which the minimized graph should be stored (see **(3)**). Finally, the value of the minimized graph is stored into the specified file (see **(4)**), if the file selection process has not been canceled by the user.

Example 7-24. The stand-alone Code of the `aldebaranMIN_STD_I` Activity

```
var String: file_name := ETI.fsBoxLoad ("Select AUT file", "aut"); (1)
var AUTFile: aut_file := file_fame: AUTFile;
var AUTFile: min_file;

if (file_fame == "") then
  exec_result := "Execution Canceled!": ETIResult;
else
  exec_result := CADP.aldebaranMIN_STD_I (aut_file, min_file);     (2)
  if (ETIResult.ok (exec_result)) then
    file_name := ETI.fsBoxSave ("Select AUT file","aut");          (3)
    if(file_fame == "") then
      exec_result := "Execution Canceled!": ETIResult;
    else
      exec_result := AUTFile.saveAs (min_file,file_name);          (4)
    fi;
  fi;
fi;
```

7.6. Integration Effort

This section summarizes the advanced integration process by talking about the effort which is required to integrate a tool into the ETI platform. As documented in Section 7.4, the ETI light integration is not a time consuming process. We plan to provide tool support which fully automizes this procedure.

In the following two sections we will present the *relative* effort which is needed to integrate the two main entities of an ETI tool repository:

1. types (Section 7.6.1) and
2. activities (Section 7.6.2).

These two sections focus on the effort which is required to provide the integration code, i.e. the encapsulation classes and the adapter specifications.

Section 7.6.3 then gives *absolute* numbers which resulted from the experiences gathered during the integration of the tools into the STTT repository.

7.6.1. Types

Since the whole *file-integration* process is supported by utility software, which is able to generate the required integration code (see also Section 7.3.1), new file types can very easily be integrated.

More integration effort is needed when a *graph type* is to be added. Since parts of the encapsulation code and the adapter specification can be automatically generated (see Table 7-1), most effort is required to provide scanners and parsers which are required

1. to read and write the node, edge and graph information which are accessible via the `parse` and `unparse` methods of the label classes (see Section 7.5.2.1.1, Section 7.5.2.1.2, and Section 7.5.2.1.3).
2. to load and store graph objects using the file format classes (see Section 7.5.2.1.4).

With respect to data-type integration, the *integration of graphs systems* is most expensive. It requires the encapsulation of the component type (see Section A.2.1.1) as well as the implementation of the graphs system class which maintains the components (see Section A.2.1.2). Since component-type integration is essentially graph integration, the effort of the integrating a system component type is comparable to the graph-integration effort. Once the component type is available, the graphs-system integration effort is mainly influenced by the time which is needed to implement the parsers and scanners which read and write the global system information.

7.6.2. Activities

The integration effort, in particular the encapsulation effort (see Section 7.5.3), of an activity depends on the API which is available to access the chosen tool feature. As already documented in Section 7.5.3, we distinguish

- black box integration
- white box integration, and
- remote box integration.

Black box integration is by far the easiest way to integrate a tool into the ETI platform. This is due to the fact, that tools integrated this way work on files as input and output types, and the provided API is very simple. Single tool features can then be addressed by passing command line options to a tool invocation.

Integrating a tool in *white box manner* is in general more expensive than black box integration. This is due to the fact that using an API to access the tool's features is more complex than the system-call based feature-access.

The encapsulation of *remotely available tools* requires two steps:

1. providing a means to remotely access the chosen tool features, e.g. by defining and then implementing a corresponding CORBA IDL, and
2. accessing the classes implementing the CORBA IDL within the tool-specific encapsulation class (see Section 7.5.3.1).

Since the complexity of Step 2 is comparable to the white box integration process, remote box integration is most expensive, if the remote access facility is not already available.

7.6.3. Summary

As can be seen very easily from the facts presented in the previous sections, the black-box integration of a tool which takes a file as input and delivers a file as output is the easiest. Except for the fact that we had to provide the graph and graphs system types, most of the activities available in the STTT repository are conformant to this behavior. Table 7-2 shows absolute numbers which have been measured during the STTT tool-integration process. Here, the integration has been performed by a developer which had a good knowledge of the concepts and design of the ETI platform.

Table 7-2.

Repository Entity	Integration Effort
AUTFile	1/4 day
AUTGraph	1 day
`aldebaranMIN_STD_I`	1/4 day
HYFile	1/4 day

Repository Entity	**Integration Effort**
HytechGraph	3 days
HytechSystem	2 day
`hytechVerify`	1/4 day

III. Building reliable Web Applications

Chapter 8. Introduction

Whereas Part II of this thesis covers the ToolZone software which is used to access the tool repository hosted by an ETI site, we are now going into the details of the environment which has been used and is still in use to develop the *ETI Community Online Service* (www.eti-service.org). This personalizable Web application organizes the people being involved in the development and extension of the platform as well as in the hosting of an ETI site. It offers beside others access to discussion groups, mailing lists, platform documentation and source code.

Starting as a distributed document database, the Web has evolved into an architecture of new generation software systems called Web applications. Even the profile of Web applications has changed dramatically during the last years. In contrast to simple CGI (Common Gateway Interface) [CGI] scripts, like a visitor counter on a Web page, today's Web applications are complex (distributed) software systems. They range from shops and auctions to highly available online brokering services. From the implemented features' point of view there is currently no difference to normal applications, except for the fact that the graphical user interface (GUI) is in most cases implemented by HTML pages, rendered by a Web browser, instead of a powerful GUI library like Motif [Bra92] or Tk/Tcl [Wel97]. From the hardware and software architectural point of view aspects like high availability, load balancing, security, etc. become particularly important.

In consequence the development of a Web application is not a one man's task any longer. In most cases building a Web application is at least as complex as building a standard application. A team of people with different skills is necessary to realize the application. System engineers design the global architecture, Web designers build the application's GUI, software designers and programmers provide the software components implementing the features, and system administrators are responsible for setting up a highly available server farm running the application. The team has to be well coordinated to deploy the application on time and to maintain the software system once it is set up. This goal can only be achieved, if the system is built using a well organized software development environment. From the technical point of view things like coding guidelines, tool support, and a good Web-application architecture play an important role. On the other hand, human aspects are very important too. [DL99] gives a nice introduction to this part of project management.

This part of this thesis goes into the details of the software development environment introduced in Section 2.2. It starts with an overview of the process and the corresponding roles in Chapter 9. The rest of this part is organized along the core phases of software development: analysis (Chapter 10), modeling (Chapter 11), design (Chapter 12), implementation (Chapter 13) and integration test (Chapter 14).

Chapter 9. Process Overview

This chapter gives an overview of the development environment for reliable Web applications introduced in Section 2.2. In particular it introduces the environment specific roles. The development environment comprises a

- a role-based software development process,
- a five-layers architecture of a Web-application, and
- the Service Definition Environment of the METAFrame project as workflow-design tool.

Figure 9-1 shows an overview of the development process as UML activity diagram [BRJ98, FS99, Con00, UML]. It presents the tasks which are performed after the analysis and the modeling phases which are documented in Chapter 10 and Chapter 11, respectively. Note that Figure 9-1 shows an idealized version of the process. As in every incremental process the analysis and modeling phases as well as the tasks shown in this figure may be performed repetitively, each time enhancing the system under construction.

The process defines several roles which identify the tasks which have to be performed during the software development:

The System Engineer:

> The system engineer analyses the problem which should be addressed by the Web application. In cooperation with the customer he defines beside others the external behavior of the system, the environment in which the application should be integrated, and performance and security demands. His activity is mostly restricted to the analysis and modeling phases.

The OO Specialist:

> The OO specialist designs, implements and tests the business objects, i.e. he is responsible for the implementation of the business object layer. In contrast to the SIB integrator, he has major skills in object oriented analysis, design and implementation.

The SIB Integrator:

> The SIB integrator implements the physical view of the SIBs on top of the business classes. He needs only minor Java programming skills to perform his tasks, because he is guided by the platform rules.

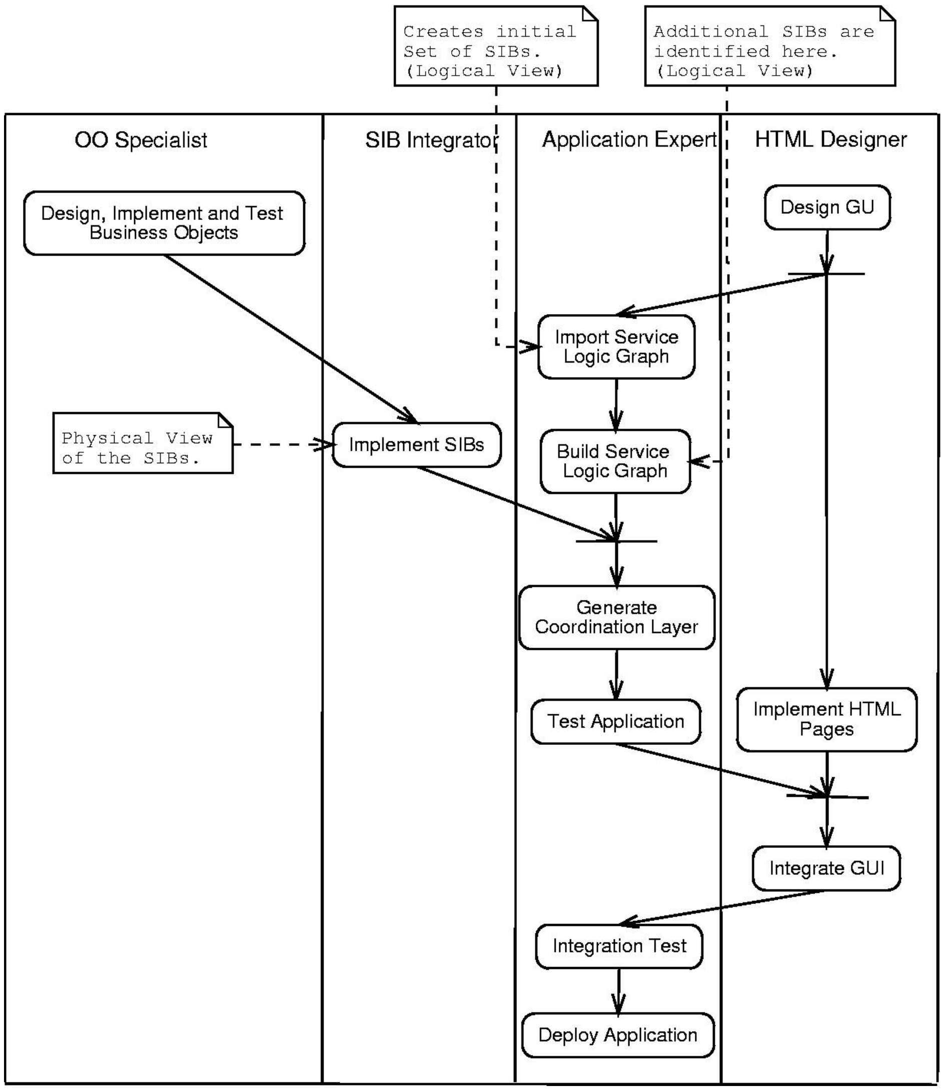
The Application Expert:

> The application expert builds the coordination layer of the system using the SIBs provided by the SIB integrator. Supported by the Service Definition Environment documented in Section 12.2 and Section 13.1 he never gets in touch with Java programming or system administration tasks.

The HTML Designer:

> The HTML designer is responsible for the presentation layer of the Web application. He provides the HTML pages which constitute the GUI of the Web application under construction.

Figure 9-1. The Process Overview


Creates initial Set of SIBs. (Logical View)
Additional SIBs are identified here. (Logical View)
OO Specialist
SIB Integrator
Application Expert
HTML Designer
Design, Implement and Test Business Objects
Design GU
Import Service Logic Graph
Physical View of the SIBs.
Implement SIBs
Build Service Logic Graph
Generate Coordination Layer
Test Application
Implement HTML Pages
Integrate GUI
Integration Test
Deploy Application

The rest of this part is organized along the five core phases of a software development process. We start with the analysis phase (Chapter 10) and the modeling phase (Chapter 11). Here, we do not present a detailed description of the tasks which have to be performed, since they are not in the focus of the process description. Instead we propose an organization of the requirements document and a model of the graphical user interface of the Web application which are suitable with respect to the task which are performed during the design and implementation phases.

After that, the design phase (Chapter 12) and the implementation phase (Chapter 13) are covered. With respect to the presented process, these development phases are the most important ones. Within the two sections we focus on the use of the Service Definition Environment during the design and the implementation of the system. This means that we focus on the tasks done by the application expert and SIB integrator. As already mentioned, the presented process is parametric with respect to the implementation of the business classes and the HTML pages. Consequently, the OO specialists and the HTML designers may use their own tools and processes to deliver their contributions to the project, i.e. the libraries implementing the business classes and the HTML pages.

Finally, the Chapter 14 will give some information on the integration test which is performed before the Web application is made publicly available.

At the end of every chapter we summarize the tasks which have to be performed by the people being involved in the construction of the Web application. This is done by presenting a table which covers

- the people,
- their tasks and
- the artifacts

of the corresponding software development phase. The tasks presented in the design, implementation and integration test summaries are also documented in Figure 9-1.

Within this chapter we use a simple shop application as example, instead of the complex ETI Community Online Service. This decision was taken, since this kind of Web application is well-known to every reader and therefore does not require any further explanation.

Chapter 10. Analysis

The development of every application starts with an analysis phase. Here a vision statement documenting the scope of the application, and the non-functional (e.g. hardware requirements, collaboration with third-party software) and functional requirements of the system to be developed are identified and specified in the requirements document. This document describes the external behavior of the system, i.e. it specifies *what* the system should do but *not how* it should perform its tasks. The main part is an overview of the features and their user-level description in form of scenarios which document the interaction of the user with the software system. Additionally, the business processes in which the system is embedded or which should be supported by the software are documented.

The requirements document is the central document of every application, since it is the basis for

- the project plan,
- the project's cost estimation,
- the legal contract,
- the user manual and
- the system test document.

Feature List of a simple Shop Application

The following feature descriptions are taken from a simple shop application. The associated feature identifiers are used to track the requirements during the whole software development process. These identifiers start with the string *NFR* or *FR* depending whether the chosen feature is a non-functional or functional requirement of the system. After that an identifier denoting the *feature group* follows. Feature groups are the means to group application features. Here, *PT* denotes the requirements on the platform on which the Web application will be provided and *CF* means that the features belong to the customer feature group. Finally, the three digit part of the feature identifier references the feature within the feature group.

Web Server (Feature ID: NFR-PT-001)

In the first phase of the project it is planned to deploy the Web application on a Windows NT server running the Microsoft Internet Information server. Later the deployment platform will be changed to a server farm consisting of a cluster of 4 SUN Servers running Solaris 7 and Borland's AppServer [AppServer] as Web application server. The delivered Web application must run on each of the chosen platforms.

Personal Home Page (Feature ID: FR-CF-001)

> As soon as the customer has logged-in successfully, he will see his personal home page. On this page, at least the customer's name and three product advertisements should be shown.
>
> Starting at this page, the customer should be able to change the data he provided during the registration (see feature FR-CF-002) and to get the list of his favored products (see feature FR-CF-003).

Change Profile (Feature ID: FR-CF-002)

> The Change Profile functionality shows an HTML form, where the customer can modify his personal data. At the end of the form a Submit button should be presented. When the customer selects this button, the provided data will be validated. If the validation is successful, the customer data in the database will be changed accordingly. Otherwise an error message will be presented to the customer giving information on occurred problem.

Favored Products List (Feature ID: FR-CF-003)

> The list of favored products shows a list of the products the customer ever bought. This list should be sorted by the totally ordered quantity and be restricted to ten products. Additionally, three product advertisements should be shown on this page.

Handling User Roles

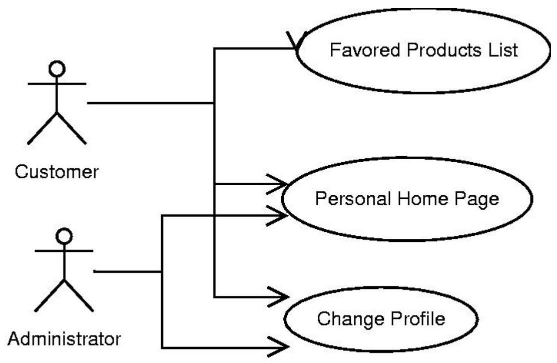
In a complex Web application, there are different users having different permissions which control the access to the data and functionality provided by the Web application. Permissions can be grouped to roles which then can be associated to users. Depending on the role of the user who accesses a feature, the feature may have then different characteristics. In the following we present the specification of the *Change Profile* feature which distinguishes two roles: a customer and an administrator:

Change Profile (Feature ID: FR-CF-002)

> The Change Profile functionality shows an HTML form, where the customer can modify his personal data. In contrast to that, an administrator may change the profile of any user registered to the shop application. At the end of the form a Submit button should be presented. When the user (customer or administrator) selects this button, the provided data will be validated. If the validation is successful, the customer data in the database will be changed accordingly. Otherwise an error message will be presented to the user giving information on occurred problem.

Within a UML use case diagram, a user role is represented by an actor. An association between an actor and a use case specifies, that a user having the role represented by the actor may access the feature modeled by the corresponding use case. Figure 10-1 shows the UML use case diagram of the shop application.

Figure 10-1. The Use Case Diagram of the Shop Application



Summarizing, the requirements document may be organized as follows:

1. the vision statement,
2. the non-functional requirements and
3. the functional requirements.

Within the chapters documenting the non-functional and functional requirements, there is one section dedicated to each feature group. Each section then contains a paragraph for the features belonging to the corresponding feature group. A use case diagram like the one shown in Figure 10-1 at the beginning of the feature-group sections is useful to give an overview of the features contained in this group and the roles interacting with the documented features.

Table 10-1. Analysis Summary

People	Task	Artifacts
System Engineer and Customer	analysis of the software system to be built	requirements document

Chapter 11. Modeling

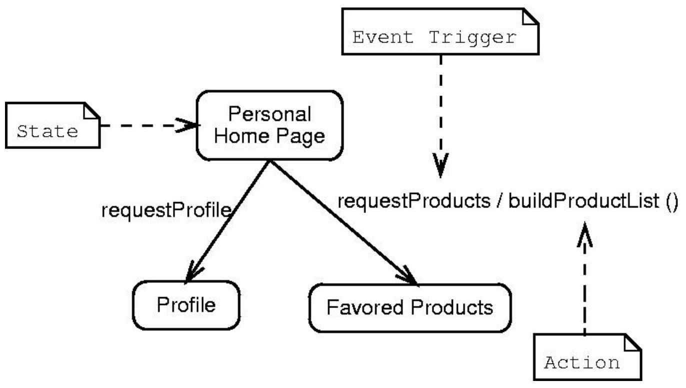
The modeling phase formalizes the requirements document into a domain-level model of the software under construction. Beside other tasks, the conceptual business classes, their responsibilities and collaborations are identified using standard object-oriented methods like the one documented in [Shl88]. Additionally, the internal processes associated to the identified features are analyzed and documented.

The modeling phase in particular results in the model of the application's graphical user interface. This model is represented as a UML statechart diagram which documents

- the *conceptual HTML pages* focusing on what information should be presented to the user instead of how this information is layouted,
- the *navigation options* and if required
- entry points to functionality which influences the structure of a dynamic HTML page.

Figure 11-1 shows a fragment of the GUI statechart diagram associated to the shop application introduced in Chapter 10.

Figure 11-1. The GUI Statechart Diagram of the Shop Application



Every state of the diagram represents one HTML page of the Web application which is presented to the user. A transition models a possible navigation option from one page to another. There are two kinds of navigation options. *Static links* are just pointers to other HTML pages. In contrast to that, *dynamic links* model that a computation has to be performed before the requested HTML page is shown. In this case, the content of the requested HTML page depends on the result of the computation.

Within the GUI statechart diagram, the distinction whether a transition represents a static or dynamic link is made by the label of the transition. If the transition models a dynamic link between Web pages, this transition contains an action which represents the computation to be executed during the state transition (e.g. `buildProductList` in Figure 11-1). Within the design phase, the actions will be mapped to nodes within the service logic graph, thus linking the GUI and the features offered by the application. Additionally, every (static and dynamic) transition contains an event trigger representing one of the navigation options within the start page of the transition (e.g. `requestProducts` in Figure 11-1).

The GUI model in form of the UML statechart diagram only documents the structure of the application's GUI. It does not specify the information which should be displayed in each of the states (screens). Thus the GUI description must be enriched accordingly, resulting in the *GUI specification* which in addition provides the information *what* should be presented to the user in each GUI state. The look and feel of the Web pages is not covered. The GUI specification is passed to the HTML designers who build the application's GUI with respect to this specification extending it by the customer-preferred look and feel.

Table 11-1. Analysis Summary

People	Task	Artifacts
System Engineer and Customer	domain modeling	conceptual business classes as well as their responsibilities and collaborations
System Engineer and Customer	domain modeling	business-process description
System Engineer and Customer	GUI modeling	GUI specification

Chapter 12. Design

The design phase refines the model of the application with respect to the environment in which the application should be deployed. Here, aspects like performance, security, the chosen Web application architecture, influence the model. All aspects which must be considered during this transition should be fixed in the non-functional requirements part of the requirements document (see Chapter 10).

The following three major tasks can now be performed independently (see also Table 12-1):

- Designing the business classes providing the base functionality. This task is performed by the OO specialist using standard UML modeling tools like MagicDraw [MagicDraw] or Rational Rose [Rose].
- Building a static prototype of the application's GUI according to the GUI specification created in the modeling phase. The prototype of the GUI does not provide any functionality. Dynamic information are exemplarily hard-coded into the Web pages. It is the basis for an initial discussion with the customer on the aimed GUI if the product.
- Building the service logic graph which defines the application's coordination layer. The GUI statechart diagram created in the modeling phase serves as starting point for this task.

Table 12-1. Design Summary

People	Task	Artifacts
HTML Designer	designs GUI layer by Web-page programming	static GUI prototype
OO Specialist	object-oriented design	model of the business classes
Application Expert	imports the GUI model	initial service logic graph
Application Expert	SLG design	model of the application's coordination layer

In the following paragraphs we go into the details on how the Service Definition Environment is used to model the application's coordination layer. For this, we need detailed information on the component model of the Service Definition Environment, i.e. the Service Independent Building Blocks (SIBs). This will be provided by Section 12.1 which focuses on the SIB-definition aspect. Note that the other constituents of a SIB specification can be found in Section 2.2.2. Section 12.2 then covers how service logic graphs are built on the basis of the SIBs using the Service Definition Environment. In this section we focus on the validation features provided by the tool.

12.1. Service Independent Building Blocks

Service Independent Building Blocks (SIBs) are software components from which service logic graphs (see Section 12.2) are built. Like activities (see Section 2.1.1) they are one type of building blocks for coordinating workflows presented in this thesis. Whereas activities focus just on data transformation, SIBs specify the control flow of an application: A SIB execution returns a result which navigates the execution through the service logic graph to decide which SIB is subsequently executed.

An instance of a SIB can be configured using SIB parameters, which provide the means that the same SIB can be used in different contexts of the same service logic graph or even in different service logic graphs. In contrast to more complex component models like JavaBeans [JavaBeans], the SIB model is tailored to the skills of the application expert, the end-user of the Service Definition Environment. Similar to activities and types, the components of the coordination sequences (see Section 2.1.2), SIBs can be organized in taxonomies to ease their retrieval.

As already presented in Section 2.2.2.1, a SIB is defined by several constituents: SIB definition, SIB documentation, simulation code, local-check code and implementation code. The central aspect of the SIB specification is covered by the SIB definition, which beside others specifies the SIB identifier, the parameters and the set of return values. Example 12-1 shows the SIB definition of the `ShowPersonalHomePage` SIB. A SIB definition comprises

- the *SIB identifier* (see (1) in Example 12-1), which is beside other user to reference a SIB within a constraint definition (see Section 12.2.1).
- The *SIB class identifier* identifier is used to group SIBs together in classes (see (2) in Example 12-1). Typically SIBs which perform similar functions, or act on similar data, are grouped into the same class. Whereas SIBs are represented by leafs within the taxonomy DAG, intermediate nodes model sets of SIBs, called SIB classes.
- The *parameters* of a SIB are the key to the service independent nature of SIBs (see (3) in Example 12-1). They are used to customize an instance of a SIB within the service logic graph.
- *Exits* of SIBs represent different possible results a SIB execution may return (see (4) in Example 12-1). A SIB may have any number of these. If a SIB does not define any exits then the service logic graph execution will terminate at the corresponding node.

Example 12-1. The `ShowPersonalHomePage` SIB Definition

```
SIB     ShowPersonalHomePage    (1)
CLS     Interaction             (2)
PAR     template  STR 100 ""    (3)
PAR     in_state  STR 100 ""
PAR     out_state STR 100 ""
PAR     out       DIM  *  0
```

```
BR      requests[]out           (4)
```

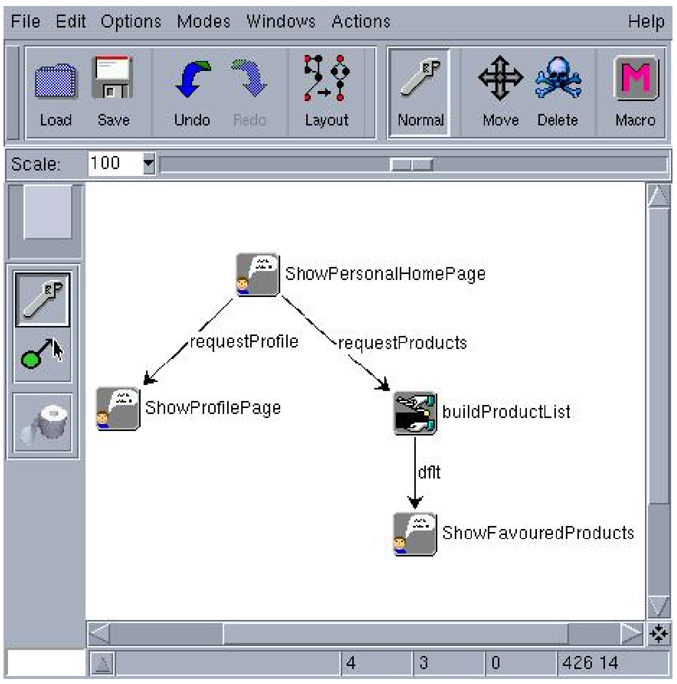
There are two SIB types: *Interaction SIBs* and *Coordination SIBs*. Interaction SIBs terminate the execution of the current request. They generate a Web page which is sent to the client by the Web server. Thereby, they give the user the means to interact with the application. Coordination SIBs control the business objects to perform the intended task. Within the service logic graph there will be an interaction SIB for every state contained in the GUI statechart diagram.

Note the difference between a SIB type and a SIB class. A SIB type is a conceptual notion, which splits up the chosen set of SIBs into two subsets, regardless of the SIB class. In contrast to that, SIB classes may be defined by a user. They are application specific. One SIB class can contain interaction SIBs as well as coordination SIBs.

12.2. Building the Service Logic Graph

Service logic graphs model Web applications at the level of the coordination layer (see Figure 2-8). Whereas coordination sequences (see Section 2.1.4.1) focus on data transformation and provide a trivial workflow model in form of sequential composition, service logic graphs are control flow centric. Data is modified as side effect of a SIB execution. This means that data is not explicitly modeled within a service logic graph.

Figure 12-1. The Service Logic Graph Editor



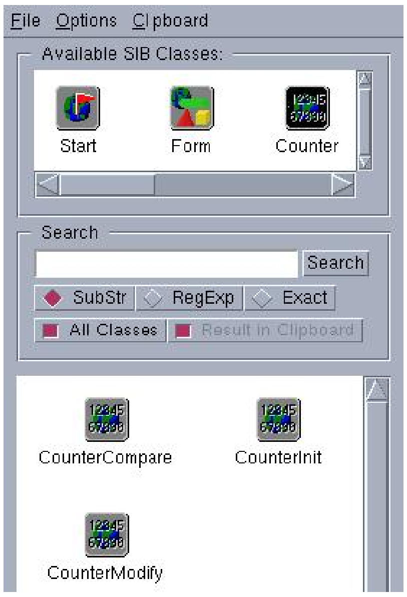
Service logic graphs are designed using the Service Logic Editor (see Figure 12-1) of the

Service Definition Environment. Here, the service designer can build a graph from a set of Service Independent Building Blocks (SIBs) (see Section 12.1) which can be accessed via the SIB palette. Figure 12-2 shows a fragment of the SIB palette which has been used for the shop service logic graph.

The Service Definition Environment focuses on the dynamic behavior: (complex) functionalities are graphically stuck together to yield flow graph-like structures embodying the application behavior in terms of control. During the design of the service logic graph the SIB set may be continuously expanded and modified when the action states are refined or new requirements are added to the application.

Within the presented development environment, the application expert does not build the initial service logic graph from scratch. Instead, he can automatically import the GUI statechart diagram (see Chapter 11) as service logic graph into the Service Definition Environment. Appendix B presents the algorithm that is used for this process. Since the Service Definition Environment also offers the functionality to store service logic graphs as UML statechart diagrams, the Service Definition Environment can tightly be integrated with UML modeling tools.

Figure 12-2. A simple SIB Palette



12.2.1. Validation Features

In analogy to the concepts documented in [SMCB96, SMCBRW96], the validation features are the key to reliable application design. Here, the user can check the consistency of the application on the level of the service logic graph at design time, i.e. *before* the coordination layer is generated and the application is tested. Consequently, a lot of errors are found in a very early phase of the software development process. This reduces the costs and the delivery time of the Web application.

It is important to note that the validation features work on the level of the application’s coordination layer. They are based on the SIB definition, the simulation code and the local-check code to perform their tasks. In particular, they do not consider the implementation code of the SIBs in form of the Java classes. This means that the validation can be performed even before the implementation of the SIBs is available (see also Section 2.2.2.3).

In the following we give an overview of the offered validation features:

1. Symbolic Execution (Section 12.2.1.1),
2. Local Checking (Section 12.2.1.2), and
3. Model Checking (Section 12.2.1.3).

12.2.1.1. Symbolic Execution

Service logic graphs can symbolically be executed using the tracer feature offered by the Service Definition Environment. The tracer validates the functional aspect of the service logic graph. Using this feature, the user gets a feeling whether the graph specifies the desired behavior at the coordination level. Since the tracer relies on the simulation code of the SIBs instead of its implementation, the application’s logic can be animated even before the implementation of the business classes and the SIBs is available. For each SIB, the simulation code can be provided as an HLL-code fragment (see Example 12-2).

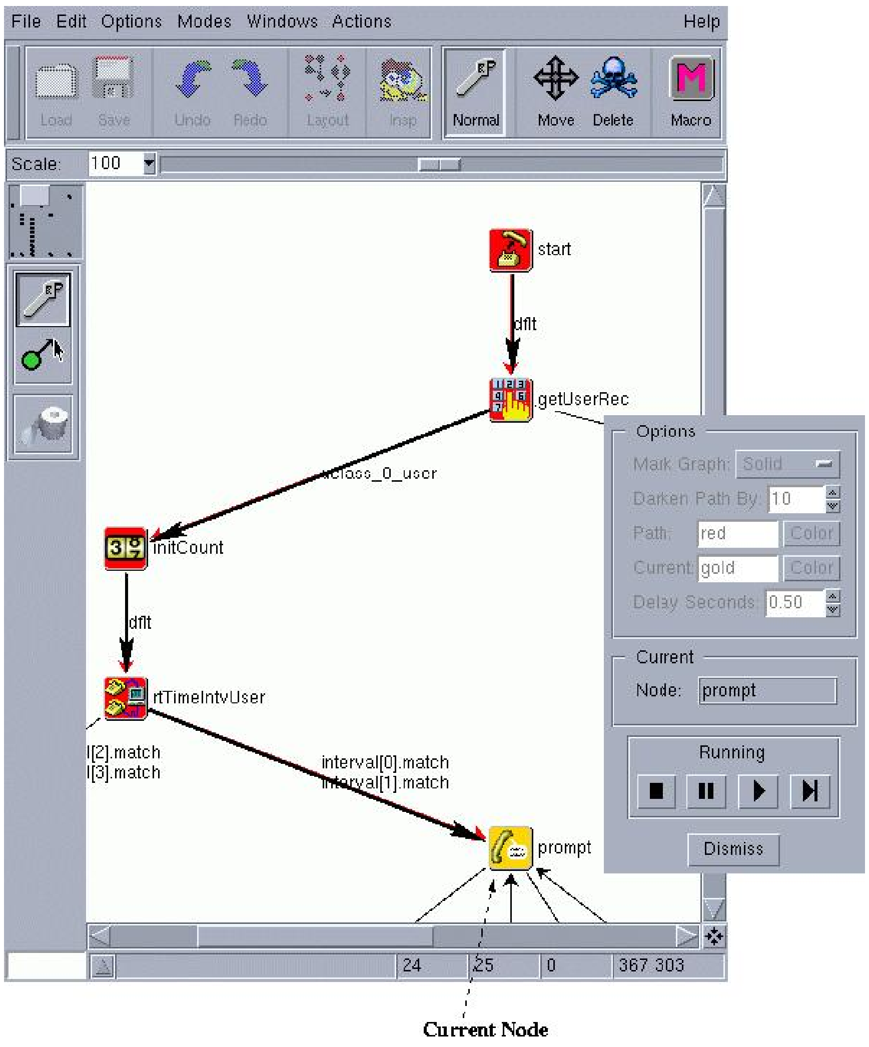
Example 12-2. A SIB Simulation-Code Fragment

```
(* read counter id and increment value *)
var String: cid := SD.getSibParameter (Tracer.current_node, "id");
var String: inc := SD.getSibParameter (Tracer.current_node, "inc");

(* execute the increment-command ’cid := cid + inc;’ *)
eval (cid + " := " + cid + " + " + inc + ";");

(* return dflt exit *)
Tracer.setBranch ("dflt");
```

Apart from the service logic design, the tracer is also helpful during the analysis. Here, it can be used for online validation of the requirements with the customer.

Figure 12-3 shows a snapshot of a tracer run. During the tracing, the current node is marked gold and the path which has already been executed is marked red. The user may control and configure the tracer using the Tracer window shown on the right of Figure 12-3. Note that in Figure 12-3 the currently executed path is marked by bold arrows.

Figure 12-3. A Tracer Run



Options
Mark Graph: Solid
Darken Path By: 10
Path: red
Current: gold
Delay Seconds: 0.50

Current
Node: prompt

12.2.1.2. Local Checking

The local checking facility searches for local errors. This means that it checks the consistency of the configuration of *single* SIBs within the service logic graph. No SIB interaction is considered. Besides others, the Service Definition Environment provides checks which ensure that

- every SIB parameter has a defined value,
- every non-optional SIB exit is assigned to an outgoing edge and that
- every edge has at least one assigned SIB exit.

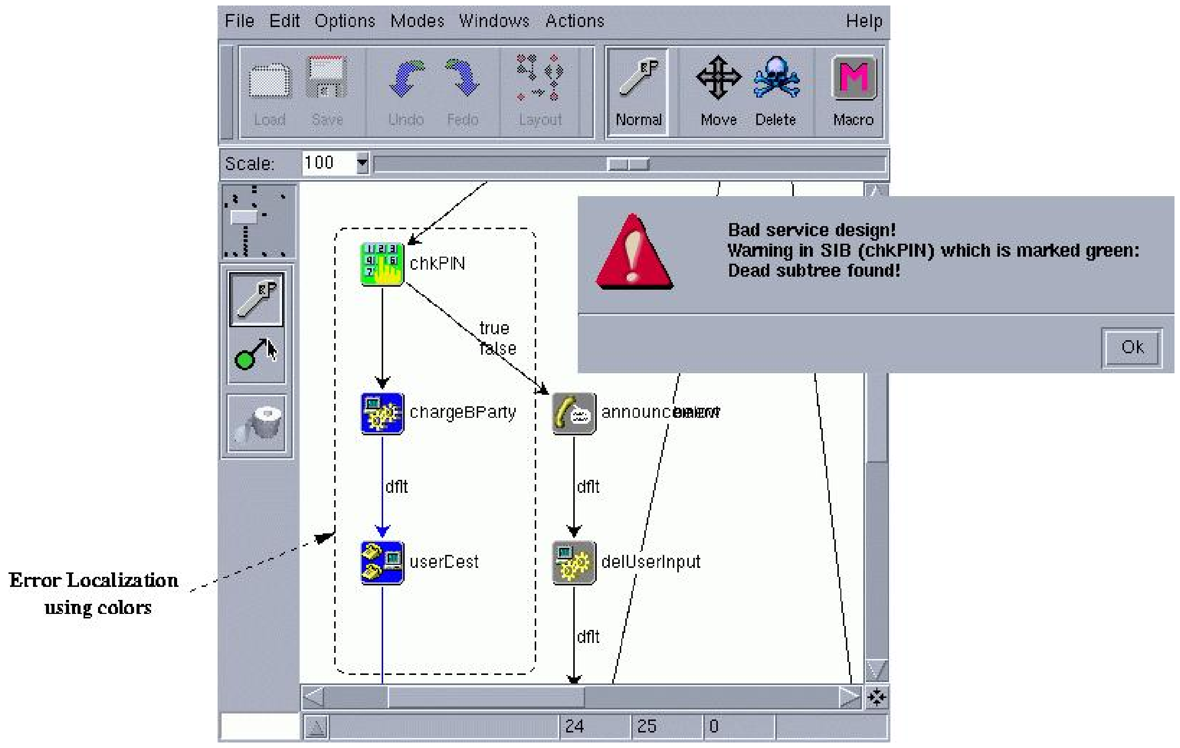
In contrast to other software development tools, the local checks are not hard-coded into the Service Definition Environment implementation, they can be programmed using the HLL. Example 12-3 shows a local check code fragment which tests, if the value of the SIB parameter `template` of an interaction SIB is set.

Example 12-3. A Local Check Code Example

```
var String: template
    := SD.getSibParameter (SDLocalCheck.local_check_node, "template");
if (empty (template))
then
    SDLocalCheck.localCheckError ("Parameter 'template' not set.");
fi;
```

For every error found in the service logic graph, the local checker shows a message box documenting the kind of the error. Faulty nodes are marked red or green, depending whether the found problem is an error or just a warning. Figure 12-4 shows a service logic graph where a dead subtree exists, which starts at node `chkPIN`: the edge starting at node `chkPIN` and ending at `chargeBParty` has no assigned SIB exit.

Figure 12-4. The Local Checker detects an Error


Bad service design!
Warning in SIB (chkPIN) which is marked green:
Dead subtree found!

12.2.1.3. Model Checking

In contrast to the local checker, the model checker verifies the *global* consistency of the service logic graph, i.e. it checks the interaction between the SIBs.

Model checking (see [VW86, SO97, MSS99, Cle99] for an overview) is a technique which (semi-) decides, whether a mathematical structure (called model), here a labelled transition system (LTS), satisfies a certain (temporal) constraint. An LTS is a directed graph where atomic information can be attached to the nodes and edges. Constraints are specified by formulas written in a certain logic like the modal mu-calculus [Koz82]. Via the formula temporal properties on the basis of the node and edge information can be specified. The formula can only be used to express properties which are concerned with the structure of the graph. Data are typically not taken into consideration or they are symbolically coded into the node information.

In the context of the Service Definition Environment, a service logic graph is interpreted as an LTS, where the node information encodes the SIB names and the values of the SIB parameters. Edge information encode SIB exits. Model checking is performed by a first-order extension of the algorithm presented in [SCKKM95] where constraints are specified in the logic presented in [Hof97] (see Example 12-4 as an example). This logic is a first-order extension of the modal mu-calculus, which even is sensitive to the values of the SIB parameters.

Example 12-4. A SLG Constraint

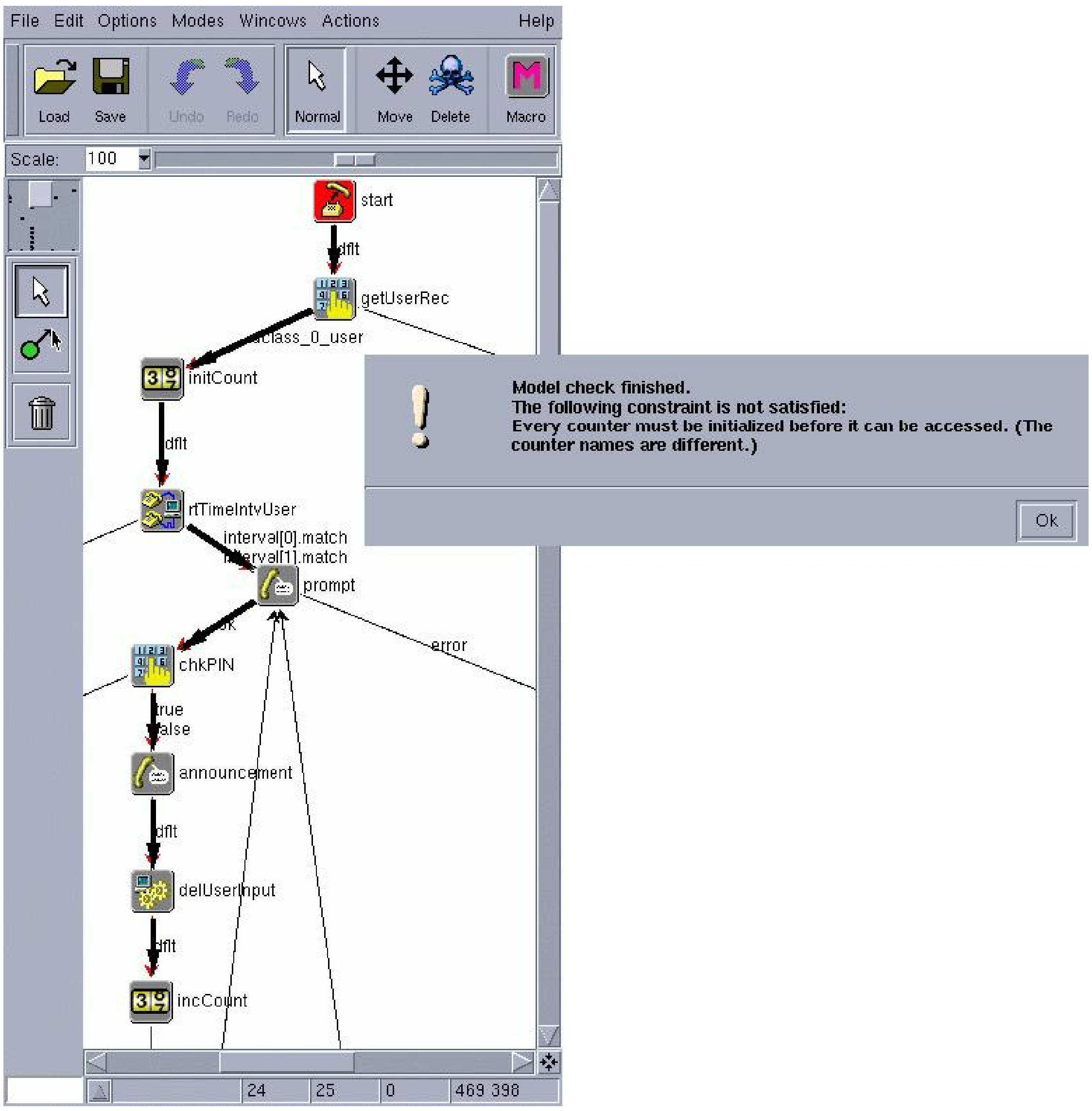
```
//: CounterCheck
// Every counter must be initialized before it can be accessed.
// (The counter names are different.)
constraint CounterCheck
{
   Forall X in model ('start =>
   AWU_F (~('incCount[ name == X ] | 'cmpCount[ name == X ]),
          'initCount[ name == X ]) )
}
```

Using the textual form shown in Example 12-4 or a formula editor, the user can specify constraints which e.g. ensure that

- specific parts of the Web application can only be accessed after a successful login,
- user information can only be accessed after the user record is fetched from the database,
- a user must pay before he leaves the shop application with the goods.

If the model checker finds an error, a message box will be shown which indicates the source of trouble (see Figure 12-5).

Figure 12-5. The Model Checker finds an Error



Model check finished.
The following constraint is not satisfied:
Every counter must be initialized before it can be accessed. (The counter names are different.)

In addition to the message box, the Service Definition Environment is able to generate error information at the level of the service logic graph. This is done by determining a sub-structure of the service logic graph which makes the chosen constraint unsatisfiable. This sub-structure is also called a *counter example*. Depending on the structure of the constraint, the counter example may be a path or a tree. If the counter example is a path, this path is automatically marked in red within the service logic graph (see [BMSY98]). Figure 12-5 shows an example of an error path within the service logic graph, where the red edges are highlighted by thicker arrows. If the counter example is a tree, the error information will be determined by the Service Definition Environment in interaction with the user. Whereas paths can be handled by the Service Definition Environment, a prototypical implementation of tree-based counter examples is currently being integrated (see [Yoo]).

Chapter 13. Implementation

The activities of the design phase deliver

- the model of the business classes,
- the model of the application’s coordination layer and
- the prototype of the application’s GUI.

During the implementation phase, the OO specialists, the SIB integrators, the application experts, and the HTML designers provide the implementation of the part of the Web application they are concerned with. The result of the implementation is the Web application which has been ordered by the customer. It has finally to be tested before it can be rolled out (see Chapter 14). Within this software-development phase, the OO specialists and the HTML designers may use their own tools and methods to provide the implementation for the business classes and to integrate the dynamic elements into the static HTML pages. In contrast to that

1. the SIB integrator provides the implementation of the SIBs used in the service logic graph on the basis of the business classes, and
2. the application expert provides the implementation of the coordination layer by compiling the service logic graph into a set of Java classes.

Table 13-1. Implementation Summary

People	Task	Artifacts
HTML Designer	integrates the GUI by adding dynamic elements to the static Web pages	implementation of the GUI layer
OO Specialist	implements the business classes	implementation of the business object layer
SIB Integrator	implements the SIBs and provides the corresponding skeleton pages	implementation of the SIBs
Application Expert	compiles the service logic graph	implementation of the coordination layer
Application Expert	packages the application	

The following paragraphs go into the details of the two tasks, i.e. the SIB implementation and the service-logic-graph compilation. We start with the presentation of the service-logic-graph compiler in Section 13.1 since this gives an overview on the generated components and their collaboration. After that, Section 13.2 goes into the details of SIB programming.

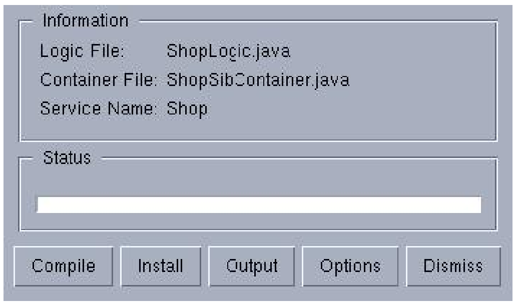
13.1. Compiling the Service Logic Graph

After the service logic graph has been designed using the Service Definition Environment, and the business classes and SIBs have been implemented, the implementation of the coordination layer of the Web application is automatically generated by the *Service-Logic-Graph Compiler*.

Note that the Service Definition Environment including its service-logic-graph model is in general independent of the application domain. It can be used to design and validate workflows appearing e.g. in the Web area or the telephony area. In contrast to that, the transformation of service logic graphs into code of a programming or scripting language is application-domain specific and depends on the chosen environment. This means that depending on the application domain service logic graphs may either be translated into C/C++, Java programs or Perl scripts. For this, the Service Definition Environment provides an application programming interface (API) which allows the integration of customized compilers.

In the Web-application domain, the Java-compiler can be accessed via the SD Compilation Inspector window shown in Figure 13-1.

Figure 13-1. The SD Compilation Inspector Window


Information
Logic File: ShopLogic.java
Container File: ShopSibContainer.java
Service Name: Shop
Status

The result of the compilation process are two Java classes which store

- the branching structure of the service logic graph and
- the parameterization of the SIBs contained in this graph.

The generated classes are derived from the classes `Logic` and `SIBContainer` which provide the methods to access the encapsulated service-logic-graph information:

The class `Logic`:

An object of this class encapsulates the branching structure of the service logic graph. It contains the public method

```
String getSuccSIB (String state, String exit)
```

which maintains the commitment relation, i.e. the information of the successor nodes on the basis of the current node and a SIB exit.

The class `SIBContainer`:

> The `SIBContainer` object maps occurrences of nodes contained in the service logic graph to instances of `SIB` objects represented by the nodes. This information is important since the SIB represented by a node can be configured via its parameters. The instance of the SIB object associated to the node then stores the values of the parameters of the SIB represented by this node.

Example 13-1 and Example 13-2 show the `Logic` and `SIBContainer` classes generated from the service logic graph shown in Figure 12-1.

Example 13-1. The generated `ShopLogic` Class

(1) Represents the edge starting at node ShowPersonalHomePage and ending at ShowProfilePage with branch requestProfile.

```
public class ShopLogic extends Logic
{
   public ShopLogic ()
   {
      super();
      addBranch ("ShowPersonalHomePage", "requestProfile", "ShowProfilePage");(1)
      addBranch ("buildProductList", "dflt", "ShowFavoredProducts");
      addBranch ("ShowPersonalHomePage", "requestProducts", "buildProductList");
   }
}
```

Example 13-2. The generated `ShopSIBContainer` Class

(1) Maps the node ShowFavoredProducts to the Java object of class `ShowInteractionState`. The values of the SIB parameters of the node are passed as constructor arguments.

(2) Sets the start node information.

```
public class ShopSIBContainer extends SIBContainer
{
   public ShopSIBContainer ()
   {
      super ();

      // fill map which links node names to SIB objects
      addSIB ("ShowFavoredProducts",
               new ShowInteractionState ("PersonalHomePage",
                                          1,
                                          "FavoredProducts",
                                          "product_template.html"));  (1)
      addSIB ("ShowProfilePage",
              new ShowInteractionState ("PersonalHomePage",
                                        1,
                                        "Profile",
                                        "profile_template.html"));
      addSIB ("ShowPersonalHomePage",
              new ShowInteractionState ("Start",
                                        2,
                                        "PersonalHomePage",
                                        "home_template.html"));
      addSIB ("buildProductList", new ExecuteAction ());

      // set the start node information of this service logic graph
      setStartSIB ("ShowPersonalHomePage");                         (2)
   }
}
```

The two classes are used by the *Service-Logic-Graph Interpreter*, which controls the execution of the application logic on their basis. The service-logic-graph interpreter, which is implemented as a Servlet [HC98, Servlet], is the mediator between the application's GUI and its logic in a model-view-controller (MVC) [GHJV95] fashion. The MVC design pattern enforces the conceptual separation between presentation (view), access logic (controller) and data (model). In our scenario the view is the application's GUI, the model is represented by the two generated Java classes and the generic controller is the service-logic-graph interpreter.

The next paragraphs give an overview of how instances of the interpreter and the generated `Logic` and `SIBContainer` classes interact. This is done by documenting how an incoming client connection requesting functionality offered by the Web application is processed. But before we go on with the details, we have to give some fundamental information on the HTTP protocol [HTTP] which is used to access the Web applications via the Internet.

13.1.1. HTTP Fundamentals

In general, the functionality offered by a Web application is made available via the Hypertext Transfer Protocol (HTTP) [HTTP]. To access the offered features, a client connects to the HTTP server providing the Web application via a Uniform Resource Locator (URL) (see Example 13-3 as an example). The URL is a string which contains

1. the protocol to connect the server (here HTTP),
2. the host name running the HTTP server,
3. an optional port number specifying the location of the HTTP server on the host (if the port number is omitted the port 80 is used by default) and
4. a string which encodes the requested functionality.

The functionality string in turn provides information on

1. the name of the Web application,
2. a reference to the functionality within this application and
3. and optional parameters.

Example 13-3. A typical Web-Application URL

The URL shown below defines an HTTP connection to the host `www.somedomain.com` on the default port 80. It request the feature `AddShoppingCartItem` of the Web application `MyShop`. The requested item is specified by the parameter `itemid`, whose value is 27 in this example.

```
http://www.somedomain.com/MyShop/AddShoppingCartItem?itemid=27
```

When a client connects to an HTTP server, the server analyses the functionality string and passes the functionality and options portion of this string to the encoded Web application for further processing. The Web application then performs the requested task and sends an HTML page presenting the result of the computation back to the client via the HTTP server.

In the presented scenario, the communication between the HTTP server and the Web application is handled via the Servlet Application Programming Interface (API).

13.1.2. Handling a Client Request

With respect to Web applications which have been build using the Service Definition Environment, client requests are processed as follows (see also Figure 13-2):

1. The requested URL is analyzed by the HTTP server.
2. If the request asks for functionality offered by the Web application the functionality and the options portion of the URL (see above) are passed by the HTTP server to the service-logic-graph-interpreter using the `doGet` or `doPost` method provided by the Servlet API.

   Note: The `doGet` method is used, when the incoming request is an HTTP-get request. The request will be forwarded using `doPost`, if it is of type HTTP-post.

3. The service-logic-graph interpreter then analyzes the functionality and the option portion (if any) of the URL. Technically, the functionality portion encodes either a node identifier contained in the service logic graph, or a node identifier and a SIB exit name.
4. Depending on whether the URL passed by the client to the service-logic-graph interpreter contains a node identifier and a SIB exit name or only a node identifier, the request is processed as follows:

If the request encapsulates a node identifier *and* a SIB exit name,

> the `Logic` object is asked by the interpreter to determine the successor node of the node encapsulated in the URL with respect to the passed the SIB exit. Then the execution of the associated SIB object is invoked by the interpreter using the `SIBContainer` object, which maps node identifiers to SIB objects.

If the URL encapsulates only a node identifier,

> the execution of the associated SIB object is directly invoked by the interpreter using the `SIBContainer` object. In this scenario, no interaction of the interpreter with the `Logic` object is required.

In both cases the SIB object is executed by calling the `exec` method which must be provided by each SIB object. This method is the generic entry point for the SIB-execution (see Section 13.2 for more details).

5. The result of the SIB's `exec` method invocation is a `String` object which represents a SIB-exit name. It is passed back to the service-logic-graph-interpreter for further processing.
6. Depending on the result of the `exec` method, the execution of the current request is terminated or continued with another SIB:

If the returned `String` object does not equal `null`,

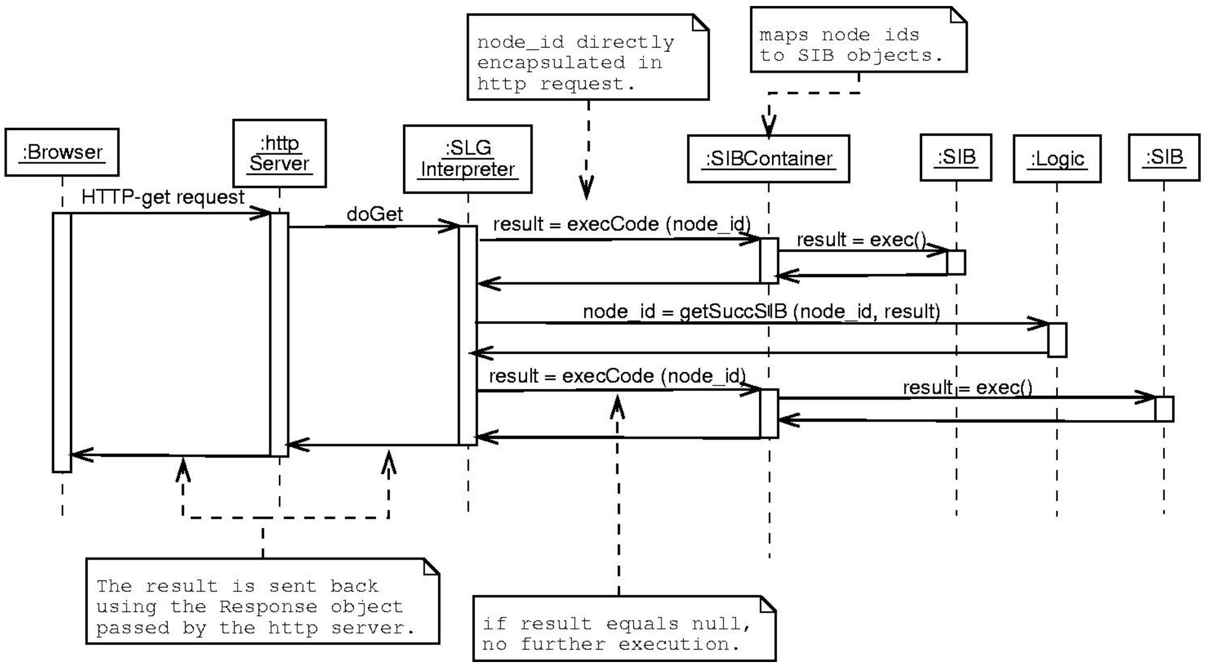
> the execution of the current request will be continued. For this, the service-logic-graph interpreter determines the next node to be executed via the `Logic` object. The associated SIB instance is then run by the `SIBContainer` again (see step 5).

If the result of the SIB execution equals `null`

> the service-logic-graph interpreter stops the execution of the current request. It sends the HTML page generated by the last executed SIB back to the client. For this the Web application uses the `Response` object contained in the Servlet API, which provides writable access to the current HTTP connection.

HTTP is a stateless protocol, this means that the same request delivers the same result each time it is executed. In consequence, we need a mechanism to store data which should be passed between two requests of the same client. This is done by a session object which serves as a container for data of this kind. Here, the Web server identifies the requesting client by an id passed as a cookie or encoded in the URL and associates the corresponding `Session` object to this request. The session object can then be accessed by the SIB object during the execution (see Section 13.2 for more details).

Figure 13-2. Execution of an HTTP-get Request asking for Web Application Functionality


node_id directly encapsulated in http request.
maps node ids to SIB objects.
:Browser
:http Server
:SLG Interpreter
:SIBContainer
:SIB
:Logic
:SIB
HTTP-get request
doGet
result = execCode (node_id)
result = exec()
node_id = getSuccSIB (node_id, result)
result = execCode (node_id)
result = exec()
The result is sent back using the Response object passed by the http server.
if result equals null, no further execution.

> **Note:** Some parameters passed as additional arguments to methods shown in Figure 13-2 are left out since they are not relevant to the presented scenario.

13.2. Implementing SIBs

A SIB is realized by a Java class implementing the `SIB` interface shown in Figure 13-3. Here, coordination SIBs are are built on the basis of the business classes identified during the software modeling (see Chapter 11). Interaction SIBs generate HTML pages which are sent to the client and rendered by the Web browser.

Figure 13-3. The `SIB` Interface

```
public interface SIB
{

    /**
     * Exec Method
     *
     * @param call_context Calling context of the SIB. Beside others, it
     *                     stores information on the session object which
     *                     is associated to this request.
     * @return Branch id that should be used for selecting the next SIB
     *         may be 'null'. That means the application waits for
     *         user interaction.
     * @exception ServletException Servlet Exceptions.
     * @exception IOException IO Exceptions.
     * @exception SIBExecException SIB Exceptions.
     */
    public abstract String exec (CallContext call_context)
        throws ServletException, IOException, SIBExecException;
}
```

As shown in Figure 13-3, each SIB implementation must provide the method `exec`, which is the generic entry point for the SIB execution (see also the “Strategy Pattern” documented in [GHJV95]). This method is invoked by the interpreter through the `SIBContainer` object each time the control flow of the application reaches a node representing an instance of this SIB (see Figure 13-2). The method invocation returns a SIB-exit identifier declared in the SIB definition (see Section 12.1), which is used by the service-logic-graph interpreter to determine the SIB object which should be executed next. The specification of the `exec` method defines, that the returned `String` object must be `null`, if the SIB which is implemented by the Java class is an interaction SIB. This tells the service-logic-graph interpreter to stop the execution of the current request. The interpreter then sends the Web page prepared by the corresponding interaction-SIB object back to the client’s browser and waits for the next user interaction (see Section 13.1.2).

The parameter *`CallContext`* of the `exec` method encapsulates the environment, in which the Web application and the SIB objects are executed. Via this object a SIB instance has beside others access to three data areas:

- data which is global to the Web container in which the Web application is running,
- data which is restricted to the Web application, like a customer database, and
- objects whose scope is limited to the current request. This local information is accessible via the `Session` object, which can be used to pass local data between SIB instances.

13.2.1. Storing the Values of SIB Parameters

SIB instances can be configured via SIB parameters within the Service Definition Environment. The values of the SIB parameters are hard coded into the generated Java classes by the service-logic-graph compiler. To store the values of the SIB parameters in the objects associated to the corresponding service-logic-graph node, the Java class implementing a SIB must provide a data attribute for each parameter specified in the corresponding SIB definition. The Java class presented in Example 13-4 implements the SIB `ShowPersonalHomePage` defined in Example 12-1. Here, the data attribute `template` is used to store the value of the SIB parameter `template`.

Note: Although it is not required that the data attribute of the Java class has the same identifier as the associated SIB parameter, choosing the same identifiers makes the application better maintainable.

To pass the values of the parameters from the service-logic-graph node to the associated SIB object, the compiler requires that the Java class offers a public constructor whose signature depends on the number and types of the SIB parameters. As shown in Example 13-4, the signature of the constructor provides an argument for each SIB parameter.

Example 13-4. Java Code Fragment of the SIB ShowPersonalHomePage

```
public class ShowPersonalHomePage
    implements SIB
{
    /**
     * Stores the value of the SIB parameter template of type STR.
     * This parameter holds a reference to the HTML page
     * representing the associated GUI state.
     */
    private String template;

    /**
     * Stores the value the SIB parameter in_state of type STR.
     * This parameter is used to hold information on the
     * object flows.
     */
    private String in_state;

    /**
     * Stores the value the SIB parameter out_state of type STR.
     * This parameter is used to hold information on the
     * object flows.
     */
    private String out_state;

    /**
     * Constructor
     *
     * @param in_state  the value of the SIB parameter in_state
     *                  set during the service logic design.
     * @param template  the value of the SIB parameter template
     *                  set during the service logic design.
     * @param out_dimension  the value of the SIB parameter
     *                       set during the service logic design.
     * @param out_state the value of the SIB parameter out_state
     *                  set during the service logic design.
     */
    public ShowPersonalHomePage (String in_state,
                             int out_dimension,
                             String out_state,
                             String template)
    {
        this.in_state = in_state;
        this.template = template;

        // the value of out_dimension will not be used in the
        // implementation of this SIB. It is only passed to
        // the constructor to make the signature of the
        // constructor conform to the SIB definition.

        this.out_state = out_state;
    }

                :
                :
}
```

13.2.2. Linking GUI and SIBs

In addition to the Java classes implementing the SIBs, the SIB integrator has to provide an *HTML Skeleton-Page* for each interaction SIB. This skeleton page presents the information and the navigation options which are shown in the associated state of the GUI statechart diagram. The information to be presented is defined in the GUI specification (see Chapter 11). Using the skeleton pages, the application can already be tested, before the real HTML pages are finished (see Figure 9-1). This is the basis for the independent development of the application's features and their GUI-presentation. Although the HTML skeleton pages already present all required information and offer the requested navigation options, they are typically very simple from the HTML-design point of view. The reasons for this are

- that the look and feel of the GUI does not matter and is often even not approved by the customer in this state of the application development.
- that we want to decouple the GUI look and feel from the application code
  - as key for the division of labor and
  - to provide different GUIs for the same application.

- that SIB integrators are not expected to be HTML experts.

To present the dynamic information within the resulting HTML page, i.e. to integrate GUI and application, each skeleton page contains scripting elements which extract data from the business objects maintained by the Web application. It is important to note, that the scripting elements only *present* the required information within the HTML page. In contrast to HTML-embedded scripting techniques for Web site development, no real functionality is implemented using the scripting fragments. The functionality of the application remains under full control of the service-logic-graph interpreter. Thus, the skeleton pages specify the interface (in form of scripting fragments and navigation options) between the application's functionality and its GUI. Currently, the Service Definition Environment uses the Velocity Template Engine of the Jakarta project [Velocity] for dynamic Web-page creation.

Example 13-5 shows a scripting fragment which iterates through a collection of user objects. The names of the users are presented as an unordered list within the HTML page.

Example 13-5. A Scripting Fragment of a Skeleton Page

```
<UL>
#foreach $user in $UserList
    <LI>$user.Name</LI>
#done
</UL>
```

During the GUI integration task of the software development process, the skeleton pages and the HTML pages of the static GUI prototype are integrated. The result of this activity is the real GUI of the application. The integration is essentially done by replacing the static parts of the HTML-prototype pages which should present dynamic content by the scripting elements of the skeleton pages.

Chapter 14. Integration Test

The integration test is the last task before the Web application is finally delivered to the customer. It is performed after the GUI integration task (see Figure 9-1) has been finished. The integration test complements the testing activities performed during the previous phases of the application development, like the testing of the business classes or the validation of the service logic graph. Therefore, it should focus on aspects not covered by the previous tasks of the software development. Consequently, beside others, functional testing on GUI level, performance issues, and the server farm configuration are of major interest.

Currently, there is no integration test support provided by the Service Definition Environment within the Web application framework today. But there is ongoing work which aims at the automated functional testing of Web applications (see Chapter 17).

With respect to the Web domain, the integration test must today be performed using testing tools like the Rational TeamTest suite [TeamTest], which beside others can be used to perform functional tests, or JMeter [JMeter], which tests the performance of the Web application.

IV. Conclusion and Perspectives

Chapter 15. Summary

In this thesis we have presented *Electronic Tool Integration (ETI) Project* which is intended to provide an infrastructure which supports the tool evaluation process by offering

1. the ETI Platform and
2. the ETI Community Online Service.

The ETI Platform

The *ETI Platform*, whose central ingredient is the *ToolZone Software*, is used to build a network of moderated Internet sites, called *ETI Sites*. These moderated Web sites offer continuous support during the four phases of the Web-based tool evaluation process:

1. searching for candidate tools,
2. reading the available documentation,
3. installing the software tool, and finally
4. experimenting with the tool.

The whole service is provided via the Internet in a

- secure,
- performant, and
- failsafe

manner.

In the current status of the project the

- Caesar Aldebaran Development Package [FGK96, CADP],
- the Uppaal tool environment [LPY97, Uppaal],
- the HyTech tool [HHW97, HyTech],
- Kronos [Yov97, Kronos],
- some spin features [Hol97a, spin], and
- the smv model checker [McM92, SMV]

are available via two ETI sites. These sites are hosted by

- the Chair of Programming Systems at the computer science department of the University of Dortmund and

- the department of Computing and Information Sciences at the Kansas State University.

The ETI Community Online Service

The *ETI Community Online Service* organizes the people who are involved in the

- maintenance,
- extension, and
- hosting

of the ETI platform. This personalizable Web application serves as a virtual meeting point which beside others offers online access to the source code and the documentation of the ETI platform. It enables team members to exchange information and discuss on ETI-related topics. In particular end users are invited to comment on the ETI platform and to propose change requests and future directions.

The construction and maintenance of the ETI Community Online Service is supported by a development environment comprising

- a role-based software development process,
- a Web-application architecture, and
- a workflow-design tool.

Although we have already put a lot of effort into the development of the ETI platform, the ETI Community Online Service and the Web development environment, there is still a lot of work to be done.

The Common Pattern

The thesis has illustrated our approach to component-based development of Internet-based applications by two variations, the ETI platform and the Web development environment. The characteristic

- strict division of labor and
- coordination-centric application development by non-programmers

is achieved by the following four-step procedure:

Development of Base Functionality by Software Experts:

> The coordination-centric development of a component-based Internet application starts with the implementation of the base functionality. Within the ETI project, the base functionality is provided either by third-party software tools or by Java libraries.

Component Development by Component Integrators:

> The base functionality is encapsulated into components which are adequate for the corresponding coordinating-workflow model. As component types we have presented ETI activities and types, and service independent building blocks. The components can be classified using predicates, which are organized in component-type specific taxonomies.

Coordination-centric Design of the Application Logic by Application Experts:

> On the basis of the components an application expert without programming skills, can specify the behavior of the application on his level of understanding. For this he can use HLL programs, coordination sequences and service logic graphs. During this task, he is supported by formal methods like program synthesis and model checking which guarantee the correctness of the application at the coordination layer.

Application Execution:

> Applications can be put into operation without any programming effort. Either they are interpreted (HLL programs and coordination sequences) or they are automatically compiled into native code like Java (service logic graphs). The applications are executed within a workflow-model specific runtime environment. In this context we have presented the ToolZone software which is able to run HLL programs and coordination sequences via the Internet. The code generated out of the service logic graphs is executed within a Java 2 Enterprise Edition compliant Web Container.

Future Directions

Beside some technical enhancements of the ETI platform covered in Chapter 16 the middle-term goals of the project are

- the integration of new tools and
- the involvement of additional research groups.

For this, we are in the course of setting up a network of excellence comprising several European research sites.

Chapter 17 illustrates a concept for automated functional testing of Web applications which applies the coordination-centric software development process in the field of test suite management. In addition, it adds testing facilities to the development environment presented in Part III. This concept is part of another Ph.D. thesis at the chair of Programming Systems and Compiler Construction of the University of Dortmund.

Chapter 18 presents an experiment with an extension of the service logic graph model which in combination with program synthesis allows even end users to modify the application at the coordination layer in a safe manner.

Chapter 16. ToolZone Enhancements

From the implementational point of view, the following enhancements of the ETI platform are currently planned:

Utility Tools

As e.g. mentioned in Section 7.6 we plan to offer more and advanced utility tools which simplify the integration process.

Firewall-compliant Communication

Currently, the communication between the client and the server side is mainly based on Java RMI. This results in a lot of trouble, when a firewall is located between the two communicating hosts, especially since the client-side software relies on *incoming* RMI calls. The problems can be solved, if the firewalls are configured properly (see [RMIFAQ]). With respect to the server side, this is not an issue, since it is the interest of the ETI site to provide this service. But in general, the ETI user has no influence on the configuration of the client-side firewall.

"XMLification" of Configuration and Data Files

Some formats of the tool management application's configuration and data files are still "old fashioned" with respect to the file formats. Therefore, we are working on defining XML DTDs for the taxonomy files, and for storing graph and graphs systems. This will enable us to use and integrate standard software and eases the data exchange between the ETI platform and third party applications.

Taxonomy Editor

In the current version of the ETI platform, the taxonomies can only be modified by editing a text file. This is of course not very user friendly. Therefore we are in the course of enhancing the taxonomy window which will enable the tool integrator to modify ETI's taxonomies via the ToolZone software.

The items only present the main enhancements scheduled within the platform's project plan. In addition to the modifications proposed by the ETI developers, every user is invited to propose future enhancements via the quality management system offered by the ETI Online Community Service.

Chapter 17. Automated functional Testing of Web Applications

The purpose of functional testing of the Web applications is to validate that the Web application is conformant with the functional requirements specified in the requirements document (see Chapter 10). Since the functional requirements describe the behavior of the system from the user's point of view, the testing is typically performed by interacting with the Web application via a Web browser.

This chapter presents first ideas of *automated* functional testing of Web applications. The work illustrated in this chapter is part of another Ph.D. thesis at the chair of Programming Systems and Compiler Construction of the University of Dortmund. It

- applies the coordination-centric software development process to design and execute test suites, and to manage the configuration of the system under test, and
- adds automated functional testing facilities to the Web development environment.

The test concept illustrated in the following two sections is based on the Integrated Test Environment (ITE) for Computer Telephony Integration (CTI) Applications [NMNSBI00, NMHNSBI01, NMHSBI01, NNHKGEH01], which has been successfully realized in cooperation with a group at Siemens Witten, Germany, and the METAFrame Technologies GmbH [MFTech]. The ITE supports the generation, execution, evaluation and management of system-level tests. In the rest of this chapter, we will focus on the test generation and test execution capabilities.

On the basis of the service logic graph, which models Web application under test at the coordination layer, and a set of test data, the environment supports

- the automated generation of test suites (Section 17.1), and
- their automated execution (Section 17.2).

17.1. Generating Test Suites

A *test suite* is a collection of *test graphs* which check the system under test (SUT) with respect to a certain test criterion, like code coverage. Here, test graphs are modeled as service logic graphs (see Example 17-1), which contain two kinds of *test blocks* (SIBs):

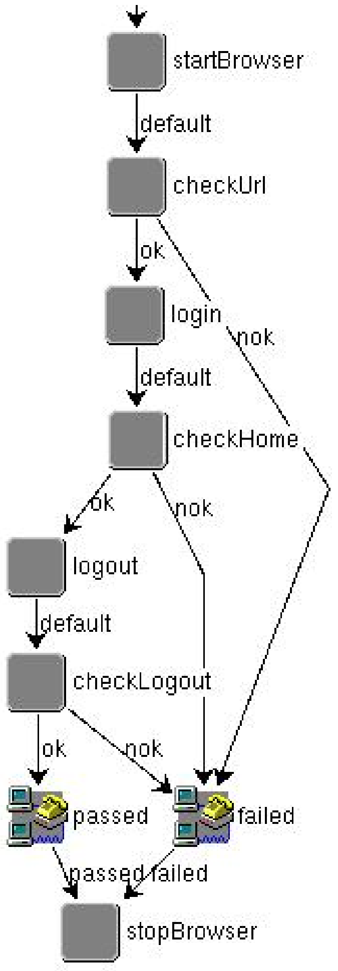
- *write blocks* send stimuli to the SUT and (see e.g. `login`, `logout` in Example 17-1)
- *read blocks* are able to get status information from the SUT (see e.g. `checkLogin`, `checkHome` in Example 17-1).

The read blocks are used to evaluate the system status after a test stimulus has been sent by a write block. For this they can be used to decide whether the execution of a test stimulus resulted in the expected system status, i.e. to decide whether the test passed or failed. This contrasts test sequences which are defined by a composition of test stimuli only.

Example 17-1. A simple Test Graph

This example shows a simple generated test graph. After some initialization tasks, like starting the Web browser on the client, the test sequence

1. logs in to the Web application,
2. checks that the home page of the user is shown,
3. logs out from the Web application and
4. checks whether the logout page is presented.

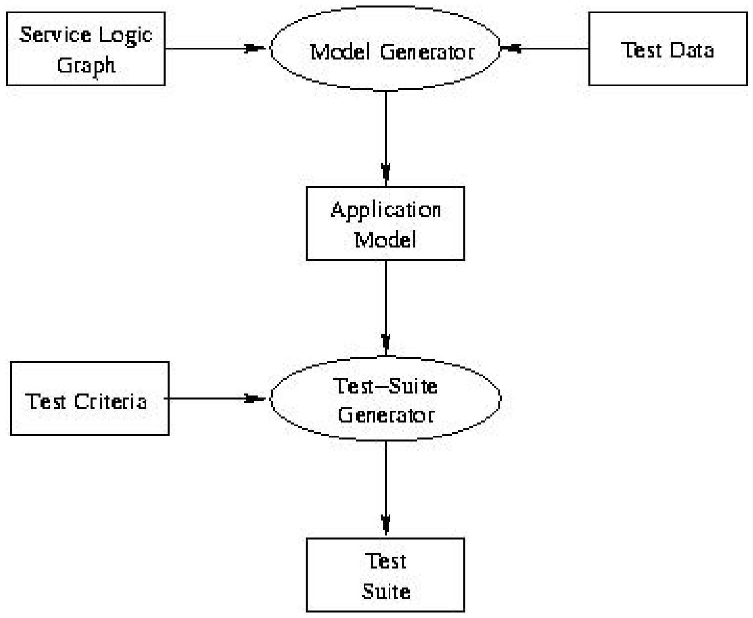


In general, the test graphs are built from application-independent test blocks which beside others are able to

- follow a link within an HTML page,
- input data into HTML forms and
- examine the title of an HTML page.

The test suite is generated in two steps (see Figure 17-1).

Figure 17-1. Automized Test-Suite Generation



First, a model of the application in form of a labelled transition system is generated. The model encodes

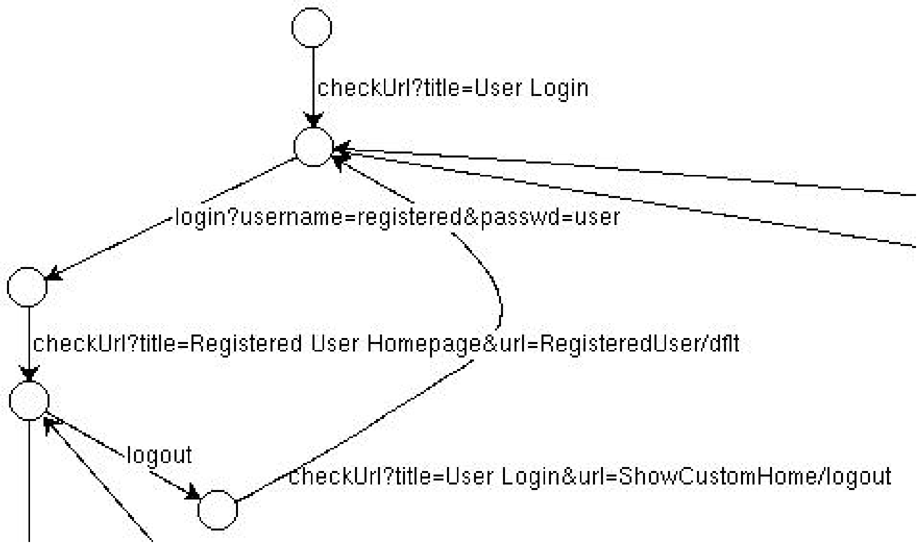
- the structure of the service logic graph of the Web application and
- a set of test data.

The test data is beside others

- extracted for the GUI specification (see Chapter 11) which provides the navigation capabilities of each HTML page and
- provided by the SIB integrator. For interaction SIBs (see Section 12.1) he can specify test data which defines input for HTML forms.

Figure 17-2 shows a fragment of an application model. As can be seen, the edge information encode actions like `checkLogin` or `login` as well as the data that is used to perform them (e.g. `title=User Login`).

Figure 17-2. A Test Model Fragment


checkUrl?title=User Login
login?username=registered&passwd=user
checkUrl?title=Registered User Homepage&url=RegisteredUser/dflt
logout
checkUrl?title=User Login&url=ShowCustomHome/logout

Using the application model the test-suite generator delivers the test suite. As already mentioned, the corresponding test graphs depend on test criteria. Currently the prototypical implementation supports the coverage of execution sequences as test criteria. This means that for each execution sequence specified by the service logic graph of the Web application under test, an appropriate test graph is generated.

17.2. Automated Test Execution

The test graphs are executed by the *Test Coordinator*, which is added to the system architecture as shown in Figure 17-3.

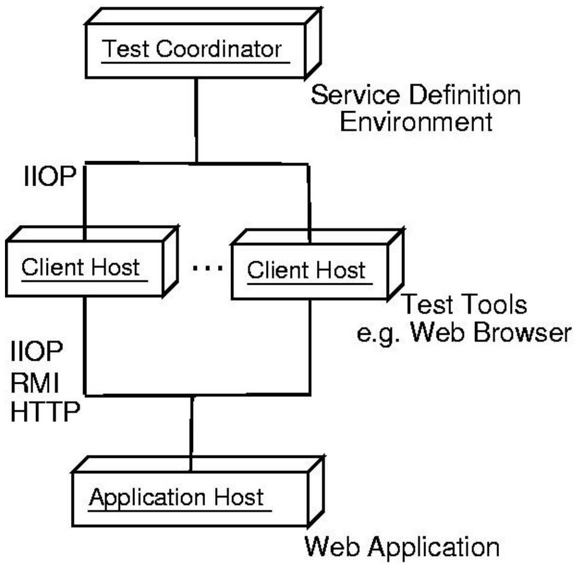
The test coordinator has access to the Web application at different layers (see also Figure 2-10). Via the presentation layer, it is able

- to control the application by following links shown in the HTML pages or by entering data into HTML forms, and
- to retrieve information contained in the HTML pages, like its title.

To control the GUI of the Web application, the test coordinator remotely operates the Web browser. This is realized via

- a test tool installed on the client, and
- a CORBA object which makes the test-tool features remotely available via the Internet Inter-ORB Protocol (IIOP).

Figure 17-3. The Test Setting



In the current implementation of the test setting, we use Rational Robot [Robot] as test tool on the client machine. Rational Robot is a product of Rational Software [Rational]. Beside others, it allows to control the GUI of an application via scripts.

The IIOP/RMI link to the business object layer of the application gives access to the Web-application's data. This connection can be used to analyze the effect of a GUI interaction in more detail. Note, that the link to the business object layer of the Web application is not necessarily required. It gives access to the state of the Web application which can be used for a more detailed analysis. In a non-intrusive scenario, the automated tests can also be performed without having the IIOP/RMI connection to the Web-application servers. Here, the application status can still be evaluated by analyzing the HTML pages presented as result of a test-stimulus execution.

As central software component, the test coordinator runs the Service Definition Environment. Each test graph contained in the generated test suite, is executed by the Service Definition Environment using the tracer feature (see Section 12.2.1). During the test run, test reports are built, which beside others provide information on

- the time when the test graph was executed,
- the environment in which the test graph was run,
- the information whether the test passed or failed and
- if the test failed, the reason why it did not complete successfully.

Chapter 18. Self-Adapting Web Applications

This chapter presents an extension of the Web development environment where even end users of Web applications are able to modify the implementation of the applications coordination layer in a safe manner. The proposed extension is part of an experiment undertaken in a two-terms course at the University of Dortmund (PG 312) by twelve students (see [ISPA98]). It proposes a new model for Web applications at the coordination layer, called *Agent Universe*, which in combination with program synthesis, similar to the one presented in Section 2.1.4.3, is used for the adaption procedure.

Although there are still a lot of questions which have not yet been answered (see Section 18.6) we want to present the results of the experiment as motivation for further research.

18.1. Motivation

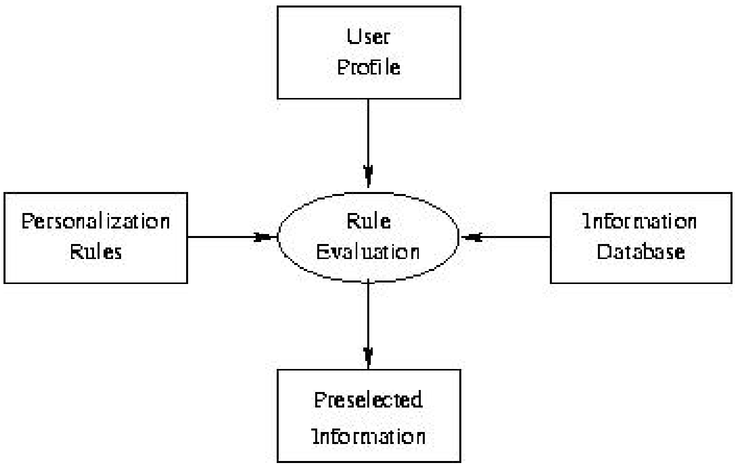
Since the Web is so popular today, a lot of people with different profiles and habits access the publicly available applications. This means that a successful Web application cannot be a static system. It must be personalizable by the user or even better self-adapting.

Today, locally personalizable Web applications (like [MyYahoo] or [MySun]) are state of the art. Here, the user may provide personal data which is in most cases used to preselect information from huge databases and to present them within an HTML page. But there are two major disadvantages concerning this kind of personalization. First, the application is only locally personalizable. This means that the personalization only effects a single Web page. There is no mechanism that changes the application logic. Second, if the habits or the preferences of the user change over the time, the user himself is responsible to inform the application. He can do this e.g. by updating his profile information manually.

In contrast, self-adapting Web applications observe the user during his interaction with the system. By this they gather important information about the behavior of the user in addition to the data provided by him manually. This information is then used to update the profile of the user and in consequence the behavior of the application accordingly. The modification process is done automatically, no interaction of the user is required.

There are two types of self-adaption. One which influences the application locally, and one which results in global changes. The local type has impact on a single Web page only. Here, slots contained in the HTML page are filled with profile-dependent information. Typically, product advertisements will appear. But in general, the concept is not restricted to this kind of information. It can even be used to present access to new functionality which seems to be in the user's interest. The user-specific information is determined on the basis of rules which specify the relation between user profiles and information which should appear in the slots of the HTML page presented to the user (see Figure 18-1). This type of self-adaption is beside others handled in [Wel00, Sch01]. In contrast to the local type of self-adaption, the global one changes the behavior of the application.

Figure 18-1. The local Personalization Process



This chapter introduces an extension of the environment presented in Chapter 9 which realizes global self-adaption by automatic modification of the application's coordination layer.

18.2. Fundamentals of Global Self-adaption

Global self-adaption is based on an extension of the feature notion introduced in Chapter 10. There, we exposed two types of features:

- *static features* whose behavior is the same for all users of the application and
- *dynamic features* whose behavior depends on the role of the user interacting with the Web application.

The second feature type is almost adequate with respect to globally self-adapting Web applications. Except for the fact, that the behavior of a *self-adapting feature* depends on further user attributes than the role, which is here only one aspect of the user's profile. To formalize the profile-dependent behavior of a self-adapting feature, we use *Micro Features*, which represent sub-functionalities of self-adapting features. The behavior of a self-adapting feature, which depends on the user profile and the system state, is then defined by a temporal specification on the basis of the micro features. Micro features can be compared to activities introduced in Section 2.1.1. Similar to activities they can be combined using the temporal logic presented in Section 5.2.2.1.

The relation between a user profile and the appropriate personalized feature-behavior is defined by means of *Personalization Rules*. They map user profiles and system states to temporal micro-feature specifications. The declaration of the micro features and the specification of the different feature characteristics are also part of the feature definition.

Within this chapter we use the “Personal Shop Tour” feature as an example. This feature of the Web-shop application introduced in Chapter 10 guides the customer through the shop instead of letting him search the “desired” goods on his own. The tour depends on the customer’s profile and on the time of the day at which the customer enters the store.

Personal Shop Tour (Feature ID: FR-CF-004)

The Web shop provides a personal tour to the customer, which can be started from his personal home page. Here, the customer is guided through the shop departments offering milk & yogurt, meat, alcoholic drinks and sweets. The tour depends on the customer’s profile and on the time of the day at which the customer enters the store. The personal shop tour provides the following micro features which are used within the personalization rules to define profile-dependent feature behavior:

- *EnterMilkAndYogurt*: Guides the customer to the milk & yogurt department.
- *EnterMeat*: Guides the customer to the meat department.
- *EnterAlcoholics*: Guides the customer to the alcoholic drinks department.
- *EnterSweets*: Guides the customer to the sweets department.

On the basis of the micro features, the following rules specify, that

- young customers should visit the milk & yogurt department and before they go to the sweets section (sub-formula `EnterMilkAndYogurt < EnterSweets` of rule 1). They should never have a chance to buy alcoholic drinks (sub-formula `G (~EnterAl-coholics)`).
- that male customers which are not young should be guided in the evening through the alcoholic drinks department and never be bored by milk & yogurt products.

```
[5] YoungCustomer                             =>
        (EnterMilkAndYogurt < EnterSweets) & G (~EnterAlcoholics).

[8] MaleCustomer & ~YoungCustomer & Evening =>
        F (EnterAlcoholics) & G (~EnterMilkAndYogurt).
```

Note that this is a very simple example, which is only meant to illustrate the presented idea.

The rest of this chapter is organized as follows. First, Section 18.3 introduces the extension of the model of the application’s coordination layer. Section 18.4 then presents the life cycle of a self-adapting Web application. Beside others, the life cycle contains the agent-adaption phase which modifies the behavior of the Web application. This phase is then illustrated in Section 18.5. Section 18.6 concludes this chapter.

18.3. The Agent Universe

In the Web-application environment introduced in Chapter 9, there is *one* implementation of the coordination layer which is valid for *all* users of the system. This is not true for globally self-adapting Web applications. Here, depending on his profile, each user may have another implementation of the coordination layer and thereby of the Web application.

To reflect the new concept, we extend the model of the application's coordination layer (the service logic graph) as follows. The new model defines

- a skeleton implementation of the coordination layer common to all users
- a portion which specifies profile-dependent features or feature characteristics.

The second aspect provides the parts in which the application may vary for different user profiles. The extension of the service logic graph model is achieved by distinguishing two types of edges. *Must edges* model the flow of control which must be present in every implementation of the coordination layer. In contrast, *may edges* model optional behavior. Mathematically, this model is covered by *Modal Transition Systems* [LT87, LT88, Lar89]. We call the extended model of the service logic graph *Agent Universe*.

Figure 18-2 shows a fragment of the agent universe of the shop example extended by the personal tour feature. Here, may edges are drawn using dashed lines. Note that all validation features of the Service Definition Environment covered in Section 12.2.1 can also handle modal-transition-systems based service logic graphs.

A user accesses the application via his *User Agent*. This subpart of the agent universe models the user-specific implementation of the coordination layer.

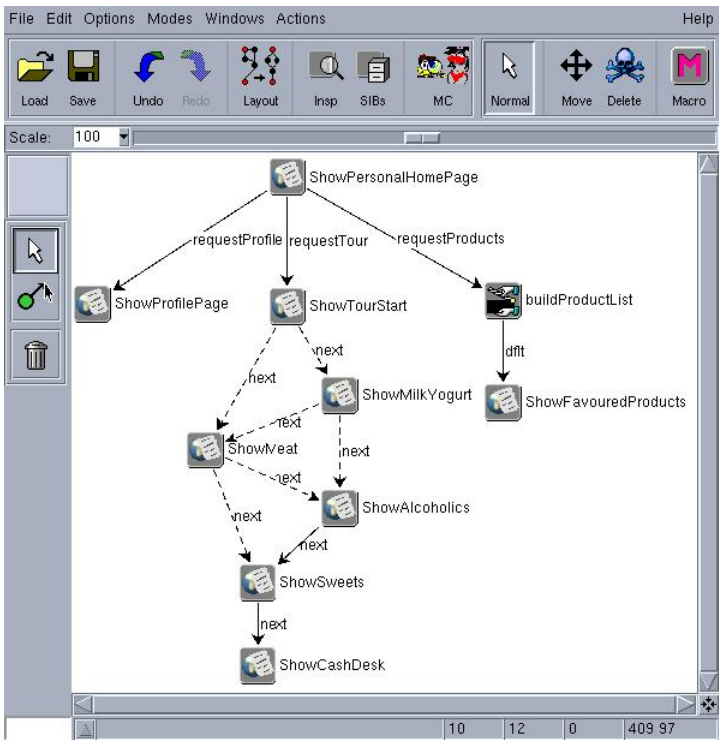
To be well-defined, the agent universe must satisfy certain properties, which ensure that the resulting user agents are deterministic graphs. Otherwise the generated Java code cannot be handled by the service logic graph interpreter introduced in Section 13.1. These properties can be checked using the validation features presented in Section 12.2:

- It is not allowed that the same SIB exit is assigned to
  - more than one must edge or
  - a must and a may edge

  starting at the same node. This is because the adaption process may result in a non-deterministic user agent, if one of the two situations occurs.

- The same SIB exit may be assigned to several may edges. Within a user agent, only one of these may edges may then be enabled (see the conflict resolution presented in Section 18.5.2).

Figure 18-2. A Fragment of the Agent Universe of the Shop Application



18.4. The Life Cycle

The life cycle of an application defines coarse grain phases which are traversed by the application during its life time. In general, we can identify phases like

- development
- deployment
- usage and
- maintenance.

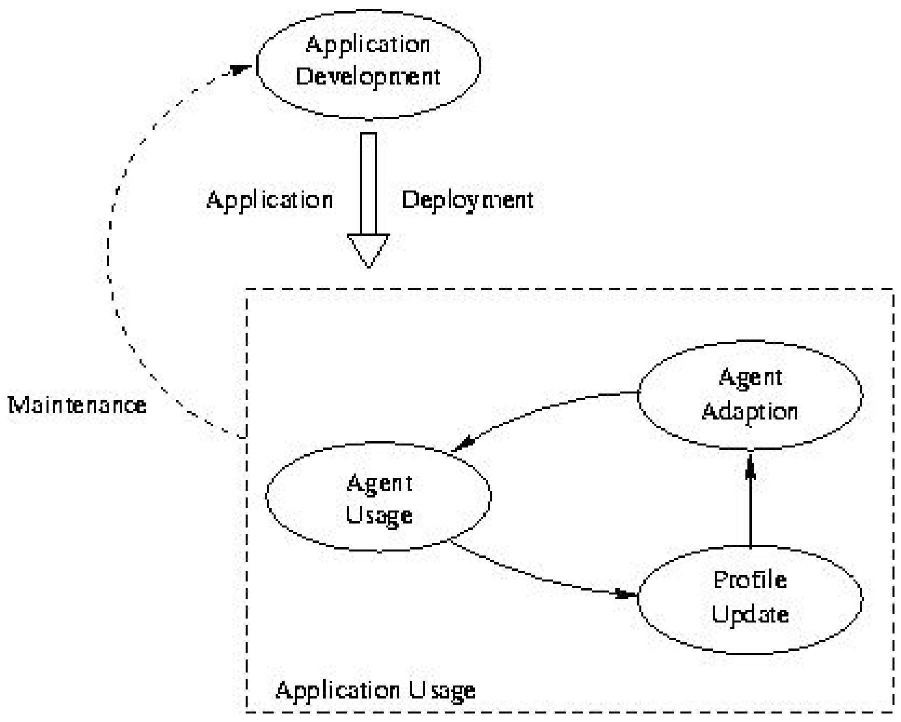
In the context of standard Web applications, only the development and the maintenance phases modify the behavior of the application. Here, the application's behavior is manually changed according to new requirements. In particular, the application usage does not effect the application's behavior.

This changes in the context of self-adapting Web applications. There, the application usage may result in more information about the user, which is then used to modify user's profile and in consequence the user agent accordingly (see also [BM99]). To model this fact within the life cycle of a Web application, the *Application Usage* phase is split up into three sub-phases (see also Figure 18-3):

- the *Agent Usage*
- the *Profile Update* and
- the *Agent Adaption* phase.

In these phases the behavior of the agent may be changed according to the new profile of the user. This means that the next time the user logs in, he uses the *same* application but is guided by a possibly *modified* agent.

Figure 18-3. The Life Cycle of a self-adapting Web Application



In the *Agent Usage* phase, which is limited to *one single* session, the user accesses the Web application via his agent. Thereby he is observed by the agent. By this, the agent collects information on the user's habits. As soon as he has logged-out (has terminated the current session), the new information is used to modify the user's profile accordingly in the *Profile Update* phase. Note that the application usage phase applies to *several subsequent* user sessions with the application performed by different users. The agent usage phase is limited to *one single* session of a specific user.

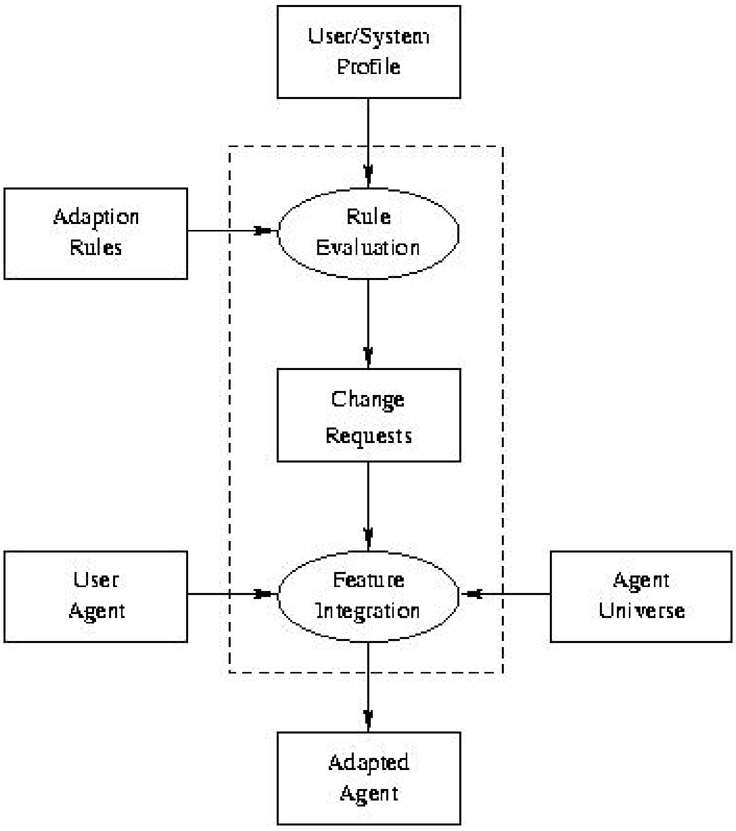
The *Agent Adaption* phase then changes the agent's behavior in accordance to the user's new profile on the basis of the personalization rules, if required. This is done, when the user logs-in the next time. Consequently, the user's view of the application may have been changed in a way that fits better his needs and the current system state. In contrast to the development and maintenance phase, the modifications performed during the agent adaption are done fully automatically. No intervention of a service logic designer or programmer is necessary.

In the next sections, we will document the agent adaption phase in detail.

18.5. The Agent Adaption Phase

The agent adaption process modifies the user agent according to the current user-profile and the actual system state. It uses the personalization rules which map user and system properties to change requests defined by the feature expressions. change requests of the user agent. The change requests are the basis for the subsequent, automatic reconfiguration of the user agent on the level of the application logic (see Figure 18-4).

Figure 18-4. The Agent Adaption Process



Concretely, the adaption process takes

- the *User and System Profile,*
- the *Personalization Rules,*
- the *User Agent*, and
- the *Agent Universe*

and delivers the *Adapted Agent* as output. The next time, the user logs-in the adapted agent is determined and then used to access the application.

> **Note:** The system profile contains information on the context and global aspects of the Web application. Here data like the current time of the day or the number of currently active user sessions are stored.

As shown in Figure 18-4, the agent adaption process is performed in two steps:

1. rule evaluation (see Section 18.5.1) and
2. feature integration (see Section 18.5.2).

The next two sections cover these two task of the agent adaption process.

18.5.1. Rule Evaluation

The first step of the agent adaption process determines change request which specify either

- new features or
- new characteristics of already available features.

The rule evaluation is performed by evaluating the personalization rules which map user and system profiles to change requests formalized by *feature expressions* (see Figure 18-5).

In general, the rules are of the form of

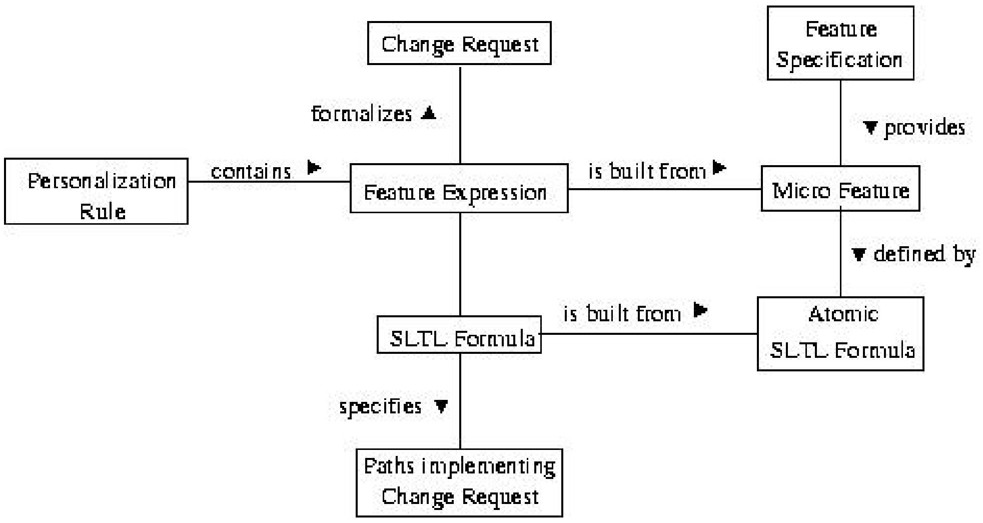
```
[weight] user/system expression => feature expression,
```

where

- `weight` is a natural number defining the weight (importance) of the rule,
- `user/system expression` is a Boolean expression built out of user/system properties and
- `feature expression` is a temporal expression based on micro features.

Each personalization rule models the fact that the feature or feature configuration defined by the right hand side of the rule is meant to be relevant for users matching the given user expression in the context of the defined system state.

Figure 18-5. The Conceptual Model of the Agent Adaption



For a specific user, the change requests are determined as follows: for every rule, it is checked if the current user/system profile matches the left hand side of the rule. If it matches, the preference of the feature expression, the right hand side of the rule, is increased by the weight of this rule.

> **Note:** Instead of summing up the weights of the rules to determine the preference of a feature expression, other strategies can be applied if neccessary.

After the rule evaluation has been finished, the agent adaption component “knows” the preference of every feature expression applicable to the chosen user in the current system state. Now, the agent adaption selects the feature expressions which should really be integrated into the user agent. This procedure uses a filter which selects the corresponding expressions on the basis of their preferences. Possible filter strategies are to select the 5 feature expressions having the highest preference or the integration of all feature expressions having at least preference 10. Of course, any other filter may be realized too. The result of the rule evaluation is afterwards processed by the feature integration.

18.5.2. Feature Integration

The feature integration adds new features to the user agent or changes the behavior of already existing ones. Here, the new features are specified by the change requests delivered by the rule evaluation (see Section 18.5.1). Whithin this step of the agent adaption process the implementation of the Web application's coordination layer is modified.

The feature integration is performed as follows:

1. For each change request which has been delivered by the rule evaluation the following steps are undertaken:

    1. *SLTL-Formula Transformation*: The feature expression is transformed into an SLTL formula specifying the implementation of the feature to be realized within the agent universe. This process combines the SLTL formulae associated to the micro features regarding the operators contained in the expression.
    2. *Path Generation*: A path which satisfies the SLTL formula is determined within the agent universe.

        In the current implementation, this path is always unique, since the synthesis component is configured to deliver one randomly selected shortest path (see also Section 2.1.4.3).
    3. *Path Integration*: If the path determined in step 2 is not empty, it is integrated into the current version of the user agent.

        Note: The path may be empty, if the SLTL formula is not satisfiable with respect to the agent universe, which means that no path within the agent universe can be found which implements the requested functionality.

2. Finally, the *Must Completion* determines the adapted agent.

Before we illustrate the steps of the feature integration in detail we introduce the data structure which is used by this procedure.

The feature integration uses a *temporal model* which represents the user agent in the different iterations of the adaption process. After all change requests have been integrated into this model, the adapted agent is determined by the must completion on the basis of this tempoal model (see Section 18.5.2.2).

Beside may and must edges, the temporal model also contains edges of two other types, which represent a certain status of a may edge:

- *Current edges* represent may edges which are available in the current version of the temporal model. They have been added by previous iterations of Step 1 of the feature integration process.

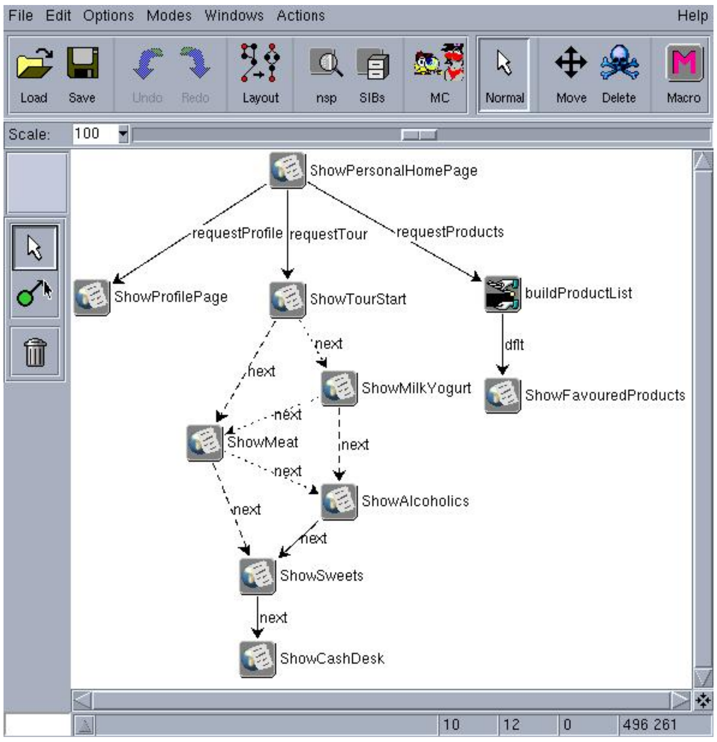
- *Proposed edges* illustrate may edges which are hit by the path representing the actual functionality to be integrated. They are added to the temporal model in Step 1.3 of the feature integration process (path integration).

In its initial state, the temporal model is equal to the agent universe where the may edges also contained in the current user agent are marked “current” (see Example 18-1).

Example 18-1. An initial temporal Model of a User Agent

This example shows the initial temporal model of a user agent where the current edges are drawn using dotted lines. Here, the personalized tour visits all departments in the following order:

1. milk & yogurt,
2. meat,
3. alcoholic drinks and
4. sweets.



The next sections illustrate the following steps of the feature-integration task:

- Section 18.5.2.1 handles the SLTL-formula transformation (Step 1.1),
- Section 18.5.2.2 covers the path integration (Step 1.3), and
- Section 18.5.2.3 illustrates the must completion (Step 2).

The path generation (Step 1.2) is not presented in this thesis, since it uses the algorithm published in [SMF93].

18.5.2.1. Building the SLTL Formula

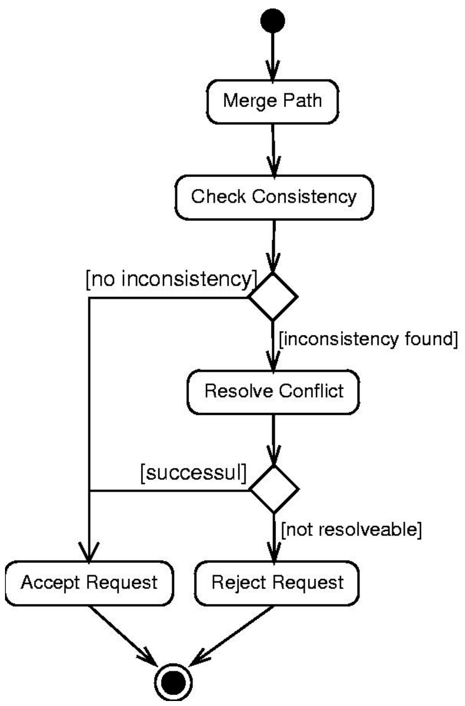
The link between a feature expression and the path within the agent universe is established via a Semantic Linear-time Temporal Logic (SLTL) formula [SMF93]. This formula is built from the SLTL formulas which have been used to define the micro features (see also Figure 18-5).

In general an SLTL formula specifies a set of paths within a labelled transition system like the agent universe. The main advantage of SLTL is that the paths are not specified on a detailed level. Instead of this they are described using temporal properties like precedence and eventuality of actions (here SIBs) and conditions (here SIB exits). By this, micro features and in consequence the feature expressions are only loosely coupled to their implementation within the agent universe. This allows certain modifications of the agent universe which do not affect the satisfiability of an SLTL formula and in consequence the availability of the associated micro feature.

18.5.2.2. Path Integration

This step of the feature integration adds a path delivered by the path generation (Step 1.2 of the feature integration procedure) into the temporal model of the user agent. An overview of the *Path Integration* procedure is shown in Figure 18-6 as UML activity diagram.

Figure 18-6. The Path Integration Task

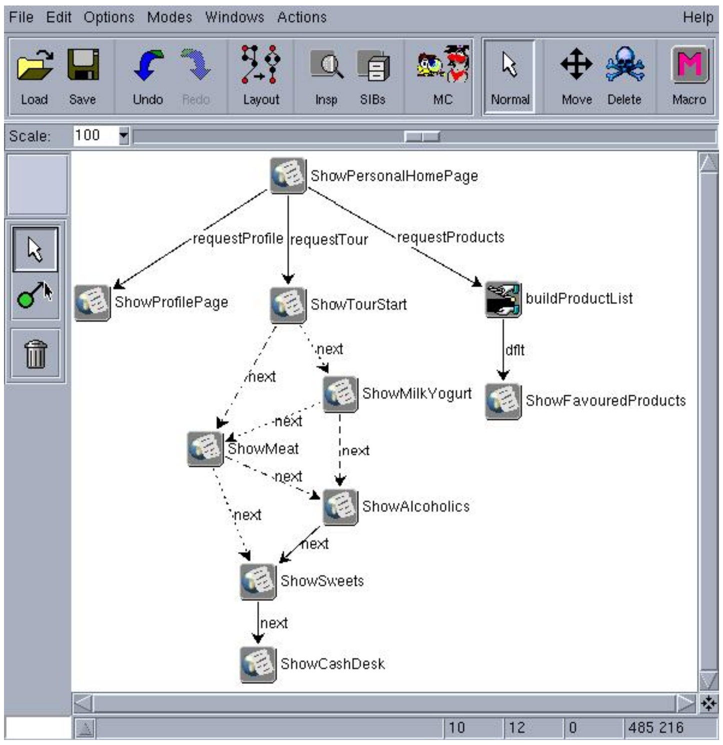


The path integration is the only task of the agent adaption process which takes the may edges contained in the agent universe into consideration.

“Merge Path” Activity

Before a path-integration procedure is performed within an iteration of the feature-integration process, the temporal model contains must, may and current edges representing the current status of the user agent. The path integration now adds proposed edges to this graph which illustrate the path implementing the chosen change request. This is done within the “merge path” activity shown in Figure 18-6. This path has been determined in Step 1.2 (Path Generation) of the feature-integration process. With respect to the shop example and the `F (EnterAlcoholics) & G (~EnterMilkAndYogurt)` functionality, the temporal model including the proposed (dashed and dotted) edges is shown in Figure 18-7. Here, the personal tour leads from the meat department directly to the alcoholic drinks and ends at the sweets section.

Figure 18-7. The temporal Model of the Shop including the new Feature



“Check Consistency” Activity

As next step, the temporal model is checked for inconsistencies. In general, an inconsistency results in a non-deterministic user agent. This means, that there exists at least one node which has two outgoing edges having the same SIB exit assigned. This situation of course cannot be handled by the service-logic-graph interpreter presented in Section 13.1. With respect to the temporal model, a conflict is found, if there exists a node which has one outgoing current and one outgoing proposed edge having the same SIB exit assigned. This means, that in the agent universe two may edges starting at the same node have the same SIB exit assigned.

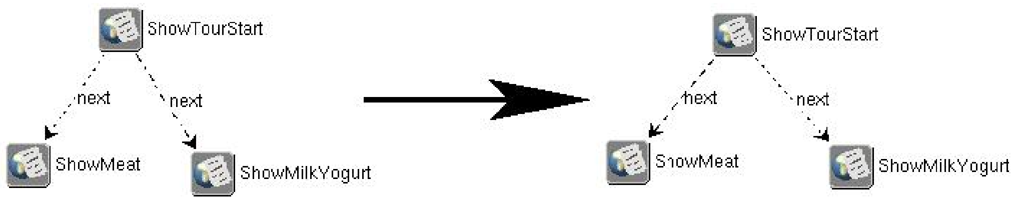
> **Note:** Due to the rules defining the consistency of the agent universe (see Section 18.3), this is the only inconsistent situation which can occur during the feature integration.

“Resolve Conflict” Activity

If a conflict is found, the currently processed functionality interacts with a feature already available in the temporal model. For this, the *conflict resolution* component tries to solve this problem. Currently, we can only handle one simple conflict. This is characterized by the fact that the non-determinism occurs at the first node of the path to be added. In this case we give the new feature higher priority which means that it is favored over the currently existing one. The conflict is resolved by marking the conflicting edge of the new requirement as “current” and the edge of the old requirement as “may” again (see Example 18-2). By this the old requirement is disabled in the adapted agent, since it will be deleted from the temporal model during the must completion step (see Section 18.5.2.3).

Example 18-2. Resolving a Conflict

In this example, the conflicting situation occurs at the node `ShowTourStart` (see Figure 18-7). As already mentioned, this conflict is resolved in a way that the new requirement (`next`-edge leading from the node `ShowTourStart` to `ShowMeat`) is favored over the old functionality (`next`-edge leading from the node `ShowTourStart` to `ShowMilkYogurt`).



If the conflicting situation cannot be handled, the requested requirement is rejected by resetting the proposed edges to “may”. In this case the application expert is notified by the software, since an unresolvable inconsistency can indicate an undetected problem within the agent universe and the personalization rules.

“Accept Request” Activity

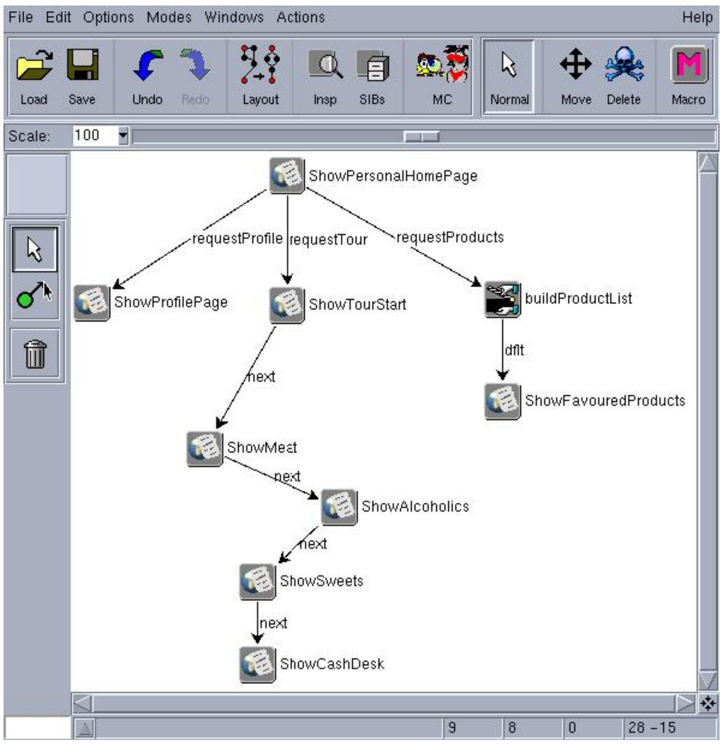
Finally, the path implementing the change request, if any, is merged into the temporal model by changing the status of the proposed edges to “current”. Consequently, after step three only must, may and current edges are contained in the temporal model again.

18.5.2.3. Must Completion

After all change requests have been processed, the must completion determines the adapted agent. This is done by computing the subgraph of the temporal model which starts at its unique start node and contains all nodes which are reachable via must and current edges. By this, the non-optional functionality is added via the must edges and the profile-dependent features are reached via the current ones. The parts of the agent universe which are not relevant for the current user profile (modeled by the may edges) will be removed in this step from the temporal model.

Figure 18-8 shows the adapted agent valid for an elder male customer using the application in the evening. He is guided from the meat department, entering the alcoholic drinks section and finally reaching the sweets' shelfs. Note that the customer must walk through the meat section since we explicitly specified in the formula, that we do not want him to be bored with milk & yogurt products (sub formula `~EnterMilkAndYogurt`) and there is no other way leading to the alcoholic beverages than passing the meat department.

Figure 18-8. The adapted Agent



18.6. Conclusion

This chapter has presented the results of an experiment with an extension of service logic graphs which in combination with program synthesis enables even end users to modify the behavior of a Web application in a safe manner. We want to point out that the illustrated ideas are only first steps towards a theoretical concept. In particular, the feature integration needs further thoughts. New features can currently only be modeled as paths. This is due to the fact that the synthesis component is limited to this data structure. Tree-based program synthesis as developed in [Yoo] would make the environment more flexible. This conceptual extension is in particular a promising way to a systematic integration of several requirements at the same time.

Chapter 19. Final Remarks

The Electronic Tool Integration (ETI) project offers

- the ETI platform, which is used to set up Web sites (called ETI Sites) providing services for Internet-based software tool experimentation, and
- an Internet-based infrastructure, which supports the people being involved in the development, extension and hosting of the platform.

Encouraged by the feedback of the ETI users, we believe that the network of moderated ETI Sites will bridge the gap between tool providers and people looking for software tools to solve specific problems. Each site of the network supports all four steps of the tool-evaluation process and provides a platform to tool providers which can be used to make their software tools available to a wide range of potential “customers” in an easy and secure way.

The development environment, in particular its formal methods based workflow design tool, which has been used to realize the infrastructure, has in the meantime turned into a commercial product. This shows that even non-scientists accept and benefit from the use of formal methods in “real-life” projects, if they are seamlessly integrated into their infrastructure at their level of understanding. The presented coarse-granular verification approach does of course not guarantee the total correctness of the software system, but it allows early error detection which improves the quality of the developed products, reduces the development time and results in a shorter time to market.

Appendix A. Design and Integration of Graphs Systems

This appendix covers design and integration of graphs systems. These aspect have not been covered by the main parts, since the handling of graphs systems in the context of the ETI platform is very similar to the handling of directed graphs (see Section 6.2.1.3 and Section 7.5.2).

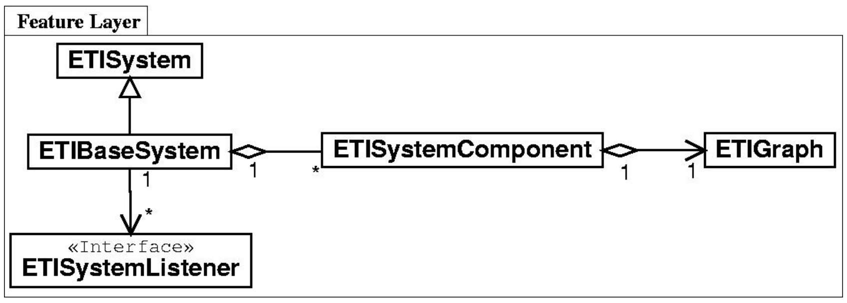
We start with the presentation of the graphs-system design in Section A.1. Section A.2 then goes into the details of graphs-system integration.

A.1. The Design of Graphs Systems

When modeling a system which should be verified, a single graph is often not expressive enough. Sometimes graphs systems build a more adequate model. Graphs systems are collections of directed graphs. They structure these graphs and provide some global information.

The ToolZone software implements a generic graphs-system type which is extended to reflect the needs of a specific graphs system (Section A.2 goes into the details of the extension process). Figure A-1 shows the structure of the generic graphs-system type as UML class diagram.

Figure A-1. Class Hierarchy of the generic Graphs-System Type



All information associated with a graphs-system object is accessible via the `ETIBaseSystem` class. Using the base class, the application has access to the graphs system's global information and to its components. Like the other generic types implemented by the ToolZone software, a graphs system type can be referenced by a string identifier. This identifier can be used to create a graphs system object of the chosen type using the management functionality provided by the class `ETISystem`.

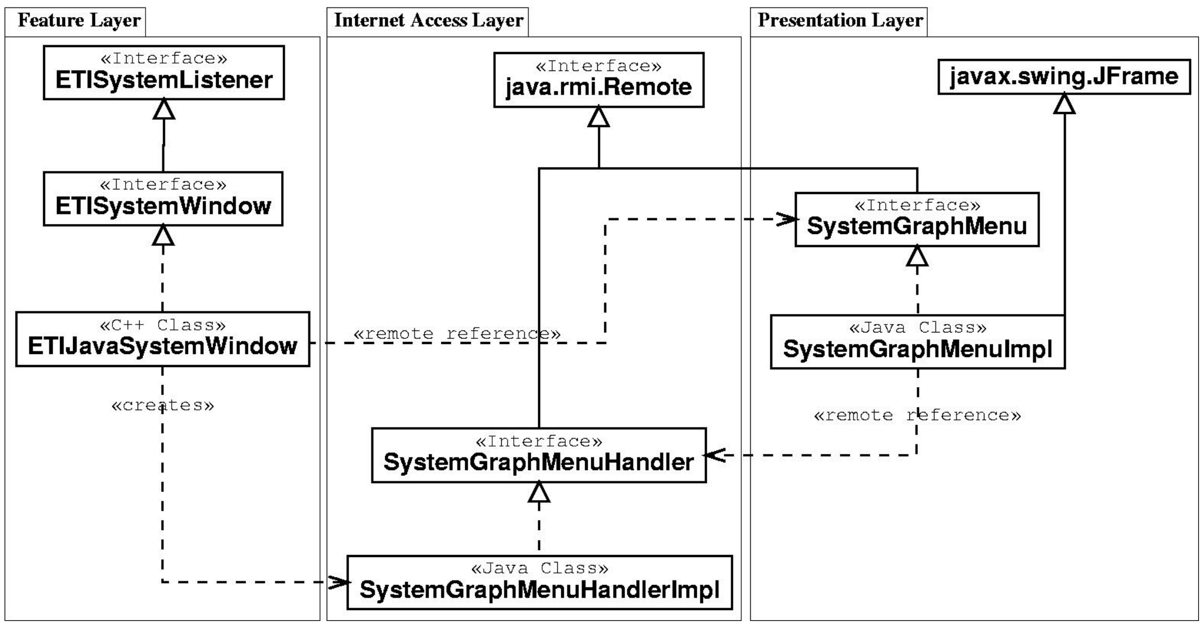
In addition to the components, an `ETIBaseSystem` object maintains a set of `ETISystemListener` (see observer pattern documented in [GHJV95]) which can be added or removed from a graphs system object. They get informed by the graphs system in the case that a component is removed or added, the global information has been changed, etc.

A.1.1. The Graphical User Interface

The GUI of a graphs system, i.e. the graphs-system editor window (see Figure 5-11), is implemented as an `ETISystemListener`. It reacts on changes within the displayed graphs-system object in order to represent it accordingly on the screen.

Figure A-2 shows all classes which contribute to the GUI of a graphs system. The corresponding objects are distributed over all the three ToolZone-software layers (see Figure 2-5). Within the feature layer, the main class to access the graphs system GUI is the class `ETIJavaSystemWindow`. To display a graphs-system object on the screen, the tool management application creates a new object of this type and invokes the `createGraphic` method on the graphs-system object to be shown on the screen.

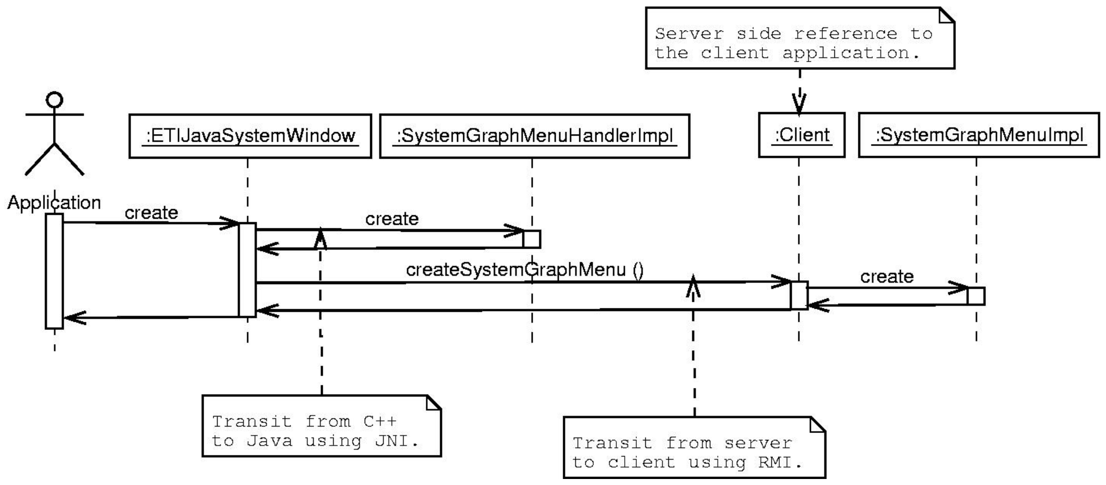
Figure A-2. The Graphs System Editor Window Class Hierarchy


Feature Layer
Internet Access Layer
Presentation Layer
«Interface» ETISystemListener
«Interface» ETISystemWindow
«C++ Class» ETIJavaSystemWindow
«creates»
«remote reference»
«Interface» java.rmi.Remote
«Interface» SystemGraphMenuHandler
«Java Class» SystemGraphMenuHandlerImpl
javax.swing.JFrame
«Interface» SystemGraphMenu
«Java Class» SystemGraphMenuImpl
«remote reference»

The creation of the `ETIJavaSystemWindow` by the tool management application, initiates the creation of the corresponding objects within the Internet access layer and the presentation layer (see Figure A-3).

When an `ETIJavaGraphWindow` object is created, first an object of type `SystemGraph-MenuHandlerImpl` is instantiated within the Internet access layer. This object is responsible for handling the communication between the client and the feature layer. It interacts with the tool management application via JNI and with the ToolZone client using Java RMI. After the menu handler object has been created, the client is requested to create a new window object of type `SystemGraphMenuImpl`. Here, a remote reference to the previously created menu handler object (interface `SystemGraphMenuHandler`) is passed to the client. This method invocation really displays the graphs system editor window on the client's screen. The `SystemGraphMenuImpl` class only provides the implementation of the menu bar and the area in which the component windows are drawn. The component windows are implemented as a special version of the graph windows which are documented in Section 6.2.1.3. As already emphasized in Section 6.1.1 and Section 6.2.1.3.1 this complex communication procedure has been implemented to deploy a secure and thin client application.

Figure A-3. Creating a remote Graphs System Editor Window


Server side reference to the client application.
Application
:ETIJavaSystemWindow
:SystemGraphMenuHandlerImpl
:Client
:SystemGraphMenuImpl
create
create
createSystemGraphMenu ()
create
Transit from C++ to Java using JNI.
Transit from server to client using RMI.

A.1.2. File Formats

Since we have already introduced the concept and the features of the file formats in Section 6.2.1.3.2, we go directly into the details of their design in the context of graphs systems.

In contrast to the graph types where the save and store functionality is controlled by the graph label and not in the graph facade class, the graphs system class provides this functionality itself. Thereby it makes use of file system classes derived from the class `SystemFileFormat`, which is derived from the class `PLFileFormat` extending it by graphs-system specific methods.

Figure A-4. Saving an `HYSystem` Object to a File

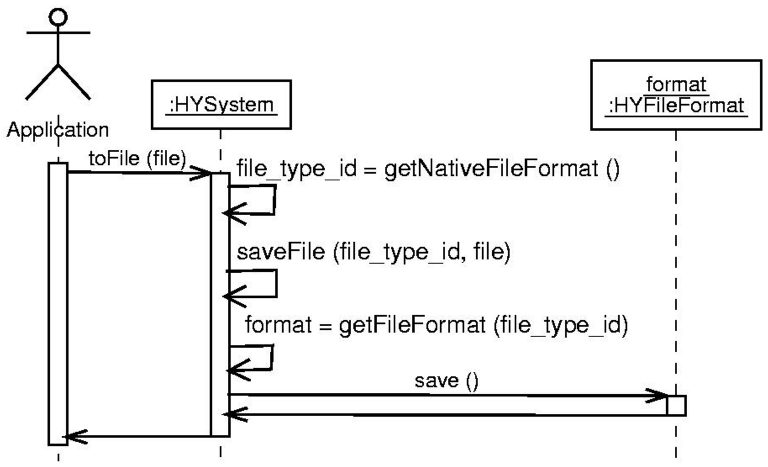

Application
:HYSystem
format
:HYFileFormat
toFile (file)
file_type_id = getNativeFileFormat ()
saveFile (file_type_id, file)
format = getFileFormat (file_type_id)
save ()

Figure A-4 shows a UML sequence diagram which documents a scenario where a `HYSystem` object (which represents a graphs system stored in the HyTech file format) is written into a file in the native file format (see also Section 6.2.1.3.2). The store procedure is completely implemented in the super class of the class `HYSystem` (class `ETIBaseSystem`). The only information which is provided by the `HYSystem` object is the string which specifies the file type identifier of the native file format. This identifier is obtained by invoking the `getNativeFileFormat` method on the `HYSystem` object.

A.2. Integrating a Graphs System Type

The ToolZone software provides an API which allows a user to integrate graphs systems. Similarly to integrating a new graph type, integrating a new graphs system type means "linking" a graphs system file format or library to the ETI infrastructure. After the integration process, graphs system objects of the chosen type can be graphically edited via ETI's graphs system editor (see Figure 5-11). In addition, ETI's graphs system library should serve as an Intermediate Representation Language for transforming graphs-system file formats into others. The first step in the process of integration is again the selection of a unique identifier, which will be used to prefix C++ class names, HLL types, METAFrame modules, etc.

In this section we will be integrating as an example the Hytech graphs system type which links the `hy`-file format, used to store real-time systems which can be analyzed by the **HyTech** tool (see Section 4.1.2), to the ToolZone software. In the example we focus on integration of the graphs system data type. Here, we assume that the file type HYFile which represents the file format to store Hytech graphs systems and the corresponding verification program are already available. In addition, the graph type HytechGraph representing a system component should be integrated, too. Since the HLL extension task is similar to the one documented in Section 7.5.2.2, we focus on the graphs system encapsulation.

Note, that except for the encapsulation of the component's graph type, the graphs-system integration is currently not supported by a utility program (see also Table 7-1).

A.2.1. Encapsulation

Let <*`ID`*> be the identifier of the chosen graphs system type, then the encapsulation requires four steps:

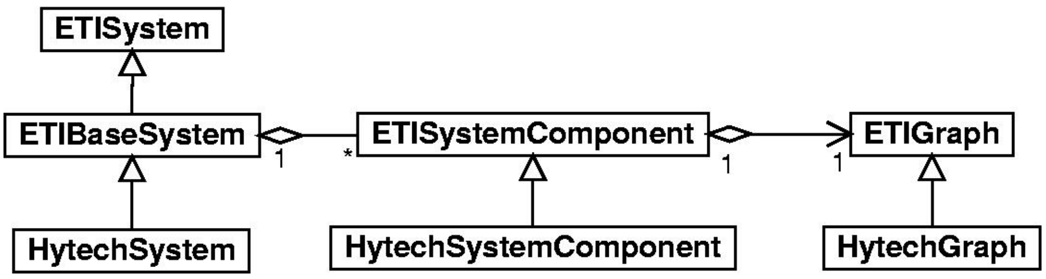
1. Encapsulate the graph type <*`ID`*>Graph which represents the type of the component graphs. This has already been documented in Section 7.5.2.1.
2. Implement the graphs system component class <*`ID`*>`SystemComponent` consisting of a component id and a component graph (Section A.2.1.1).

3. Implement the graphs system class <*ID*>`System` which manages the components, global graphs system data and provides graphs system type specific functionality (Section A.2.1.2).
4. Implement a <*ID*>`FileFormat` class which encapsulates the native file format. This process is similar to the one documented in Section 7.5.2.1.4.

> **Note:** Classes implementing a graphs system file format must be derived from `SystemFileFormat` instead of `PLFileFormat` which has a similar interface.

The ToolZone software API provides the two classes `ETISystemComponent` and `ETIBaseSystem` from which the component and graphs system classes must be derived. Figure A-5 shows a UML class diagram of the HyTech graphs system specific C++ classes and their link to the ToolZone software.

Figure A-5. The `HytechSystem` Class Hierarchy



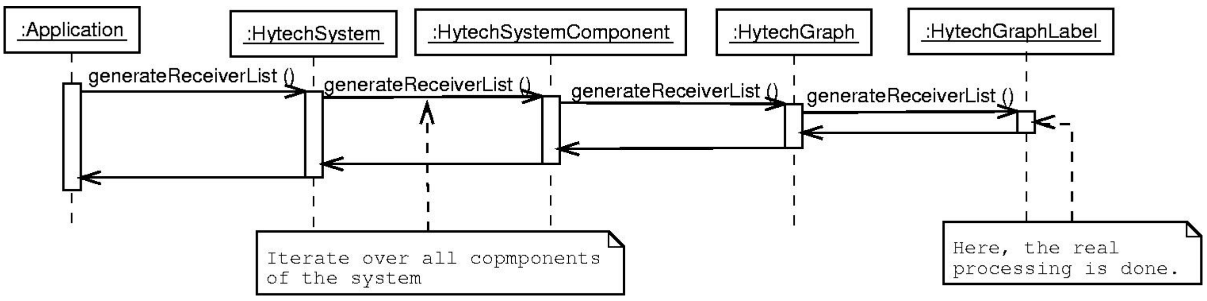
The remainder of this section is organized along the two (not already documented) steps which have to be performed to encapsulate a new graphs system type:

1. implementing the system component class (Section A.2.1.1) and
2. implementing the graphs system class (Section A.2.1.2).

A.2.1.1. Implementing the System Component Class

The next step in the encapsulation process is to implement a component class derived from `ETISystemComponent`. This class encapsulates the access to a component identifier which is unique in the whole system object and the component graph. In addition to this basic functionality provided by `ETISystemComponent` the component class gives access to methods and data that are specific to the chosen component type. The implementation of this specific set of functionality should in general not be provided by the component class itself. Instead it should be implemented in the graph label class associated to the component's graph class. This means, that an invocation of a component type specific method is just delegated by the component object to the associated component graph which again delegates this method invocation to the associated graph label object (see Figure A-6).

Figure A-6. Delegating Component Method Invocations



In addition to the methods implementing the component specific set of functionality, the component class must at least provide the following constructor:

`<ID>SystemComponent (<ID>Graph* g, const string& id)`

The associated component graph object and a string representing the component identifier are passed as arguments *`g`* and *`id`*, respectively. Since all component specific data should be encapsulated in the associated component graph label, a typical implementation of this constructor just invokes the corresponding constructor of the base class `ETISystemComponent` (see Example A-1).

Example A-1. The `HytechSystemComponent` Constructor

```
HytechSystemComponent::HytechSystemComponent (HytechGraph* g,
                                              const string& id)
    : ETISystemComponent (g, id)
{
}
```

The implementation of the destructor is in general not required, since the ETI design guidelines suggest to encapsulate the component specific data in the graph label associated to the component's graph object. The destructor's default implementation offered by the class `ETIBase-System` already destroys the associated graph object including its label.

In addition to the constructor which has to be provided by the chosen component class, one can optionally implement the method `virtual void clear (void)`. This method clears the data associated to the component class. The default implementation of this method just delegates the invocation to the associated graph object. In consequence, this method has only to be implemented, if the chosen component class provides data attributes other than the component identifier and the reference to the component graph, which are "owned" by the base class

`ETISystemComponent`. Since encapsulating data into the component class does not conform to the ETI design guidelines, there is in general no need to implement the `clear` method.

A.2.1.2. Implementing the Graphs System Class

Similarly to the graph design, there exists one facade class named <*`ID`*>`System` (see Figure A-5), which gives access to the whole graphs system. But in contrast to the process of integrating a graph type, this class is not generated but must be implemented by the integrator. This graphs system class must be derived from `ETIBaseSystem`, which provides the basic functionality giving access to the graphs system components. Beside the set of graphs system components, the facade class maintains global (component spawn) data, e.g. a set of clocks declared in a real time system. In addition to the data and functionality which are specific to the chosen graphs system type, this class must implement the following methods:

`virtual const char* getNativeFileFormat () const`

Similar to the method referenced in Section 7.5.2.1.3, this method returns the native file format identifier to which this graphs system type is linked to. Amongst others, this method is invoked in the graphs system's load and save methods to determine the native file format filter (see also Section 7.5.2.1.3).

`virtual void check (ETISystemInfo& info) throw (ETIException)`

Changing component or graphs system information may result in an inconsistent graphs system object. Therefore, the integrator must provide the method `check`, which checks the consistency of the graphs system information. Similar to the `check` method offered by the class `ETIGraphLabel` (see Section 7.5.2.1.3) the *`info`* argument can be used to return diagnostic information to the application.

`check` throws an `ETIException` if the check failed.

`virtual void parse (const string& data) throw (ETIParseException)`

The method `parse` initializes the global graphs system data from the passed string *`data`*. It is used to modify the data via the graphs system console. If the data cannot be initialized by the passed string object, this method throws an `ETIParseException` giving detailed information on the error that occurred.

If there is no global graphs system information which can be edited by the user, this method need not to be implemented. In this case the method `isInspectable` of the class `ETIBaseSystem` should be overwritten in a way that it returns `false`.

`virtual string unparse (void) const`

`unparse` returns a string representation of the global graphs system data. It is used to show the content of the graphs system information in the graphs system console window.

If there is no global graphs system information which can be edited by the user, this method need not to be implemented. In this case the method `isInspectable` of the class `ETIBaseSystem` should be overwritten in a way that it returns `false`.

`virtual ETISystemComponent* newComponent (void) const`

This method returns a new (initial) component object associated to the chosen graphs system type. Amongst others, it is used internally when quering the graphs system object for a new component via ETI's graphs system editor.

Example A-2 shows the implementation of the `newComponent` method used for the encapsulation of the Hytech graphs system type.

Example A-2. The `newComponent` Method of the Class `HytechSystem`

```
ETISystemComponent*
HytechSystem::newComponent (void) const
{
    HytechGraph* g = new HytechGraph ();
    return new HytechSystemComponent (g,"");
}
```

In addition to the mandatory methods, it is optional, but recommended, to overwrite the method `virtual void clear (void)` of the base class `ETIBaseSystem`. The base class implementation of this method removes only the components of the graphs system. In consequence, one has to provide a C++ code fragment which clears the global data specific to the chosen graphs system. Example A-3 shows that this method takes two steps in order to perform its task:

1. Clear all data that is specific to the chosen graphs system.
2. Invoke method `clear` of the base class `ETIBaseSystem`.

Example A-3. The `HytechSystem` `clear` Method

```
void
HytechSystem::clear (void)
{
    // clear global data which is
    // Hytech graphs system specific
    def_list.clear ();
    integrator_list.clear ();
    stopwatch_list.clear ();
    clock_list.clear ();
    analog_list.clear ();
    parameter_list.clear ();
    discrete_list.clear ();
    cmd_list.clear ();

    // call clear () of the base class
    ETIBaseSystem::clear ();
}
```

Appendix B. GUI Statechart Diagram Transformation

The transformation of GUI statechart diagrams into service logic graphs is the key to the tight integration of the Service Definition Environment and UML modeling tools. Here, GUI statechart diagrams which are built using UML modeling tools like Rational Rose [Rose] and MagicDraw [MagicDraw] are *automatically* converted into the service logic graph format and vice versa. The prototypical implementation of the presented algorithm can handle statechart diagrams which are available in the XML Metadata Interchange (XMI) format [XMI]. XMI is an Object Management Group (OMG) standard combining UML [BRJ98, FS99, UML] and XML [HM00, XML].

The following sections present an algorithm which transforms the state focussed model of the GUI, represented by a statechart diagram, into a control-flow oriented one, represented by a service logic graph. Conceptually, this transformation requires two steps:

1. Change the state-oriented view of the GUI into a control-flow oriented one. For this, the UML statechart diagram is transformed into an equivalent UML activity diagram (see Section B.1).
2. Change the representation of the control-flow oriented view into a format understandable by the Service Definition Environment. This step transforms the UML activity diagram into a service logic graph (see Section B.2).

After that, the initial service logic graph can be refined using the Service Definition Environment (see Section 12.2). Note that this conceptual two step transformation can be implemented in one phase for reasons of efficiency.

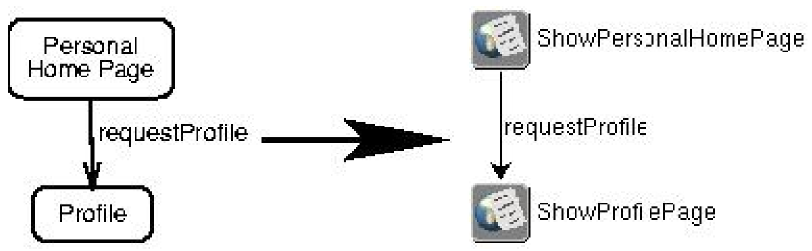
The basic idea of the two-step transformation is the following. Each state of the statechart diagram is represented as an interaction SIB in the service logic graph. Additionally, for each action associated to a transition, a SIB is added, which is later refined by a set of coordination SIBs during the service logic graph design (see Section 12.2).

Transitions contained in the GUI statechart diagram are handled as follows:

Transitions which do not contain an action:

For each transition which does not contain an action (i.e. it only provides an event trigger), an edge between the corresponding interaction SIBs is created in the service logic graph. Additionally a SIB exit having the name of the event trigger is assigned to this edge (see Figure B-1).

Figure B-1. Handling Transitions without Actions

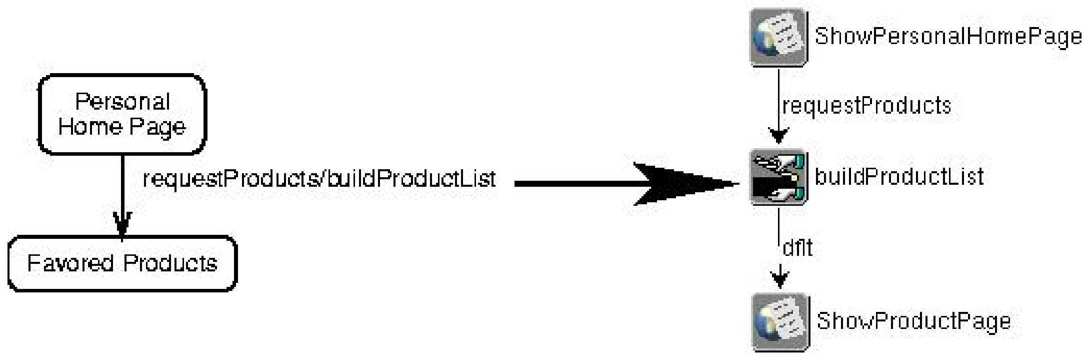


Transitions which contain an action:

If the chosen transition contains an action two corresponding edges are created in the service logic graph (see Figure B-2):

- One edge starts at the associated source SIB and ends at the SIB which has been created to represent the action. This edge gets a SIB exit having the name of the transition's event trigger assigned. Note that every transition contained in the GUI statechart diagram has at least an event trigger assigned.
- The other edge starts at the action SIB and ends at the interaction SIB, which is associated to the end state of the chosen transition. Here, the default SIB exit `dflt` is assigned to this edge.

Figure B-2. Handling Transitions with Actions



The next two sections go into the details of each of the two automatic transformation steps.

B.1. Transforming the GUI Statechart Diagram

UML statechart diagrams and activity diagrams model the same aspect of the system, but with a different focus (see [BRJ98]). Whereas statechart diagrams look at the potential states of the system and the transitions among them, activity diagrams emphasize the flow of control from activity to activity. This is the reason why the GUI of an application is often modeled as a statechart diagram. On the other hand, UML activity diagrams are more suitable to represent service logic graphs.

In the following, we present an algorithm, which transforms the state focused view of the GUI into a control flow oriented one. This is mainly done by representing a state of the GUI statechart diagram by a node in the activity diagram which represents the activity that builds the associated GUI state. For this, the presented algorithm splits up each state in the statechart diagram into a so-called *interaction state* and a branch in the activity diagram (see Figure B-3). The interaction state is later transformed into an interaction SIB which sends the HTML page representing the associated GUI state to the client.

For each action which is associated to a transition of the statechart diagram a so-called *action state* will be created in the activity diagram. Figure B-3 shows the activity diagram which is equivalent to the statechart diagram shown in Figure 11-1.

Formally, the activity diagram can be obtained from a GUI statechart diagram by performing the following steps in order:

1. For every state in the statechart diagram, a corresponding state (called *interaction state*) in the activity diagram is created.
2. For every state in the statechart diagram which has at least two outgoing transitions, a branch in the activity diagram is created in addition to a transition which starts at the interaction state and ends at the branch.
3. For every action associated to a transition in the statechart diagram, a corresponding state (called *action state*) is created in the activity diagram.
4. For every transition in the statechart diagram, the following transitions in the activity diagram are created.

If no action is associated to the transition,

> it starts at the branch associated to the transition’s source state and ends at the interaction state (created in step 1) associated to the transition’s target state.

If an action is associated to the transition,

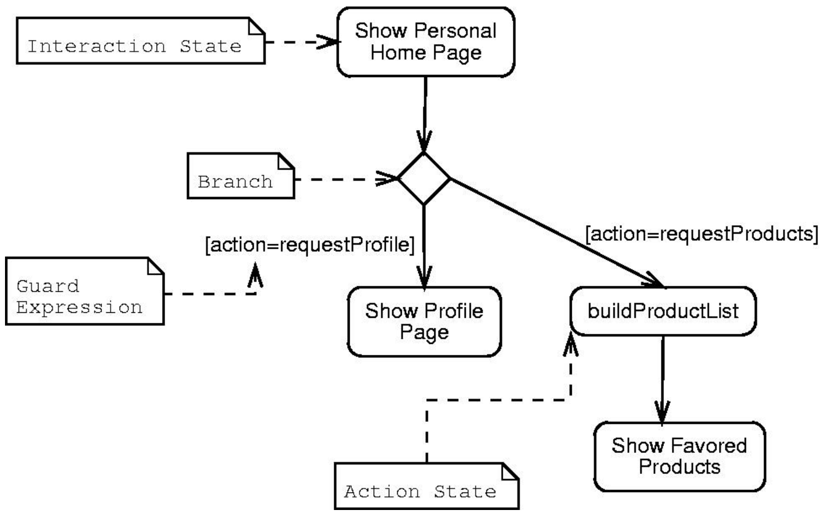
> it starts at the branch associated to the transition’s source state and ends at the action state (created in step 4) associated to the transition’s action. Additionally, a transition in the activity diagram is created which starts at the action state and ends at the interaction state associated to end node of the chosen statechart transition.

In any case, the guard expression of the transition is set to

`[action=’Transition Event Trigger’].`

Figure B-3 shows the activity diagram which is obtained from the statechart diagram shown in Figure 11-1.

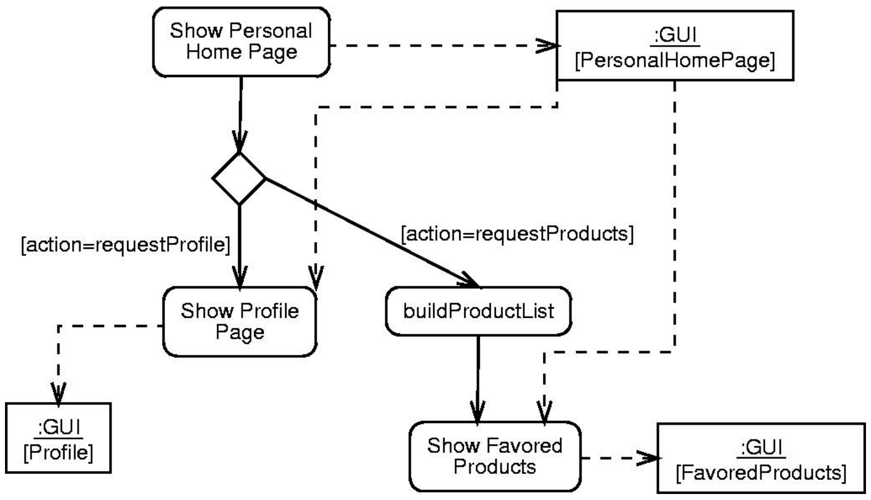
Figure B-3. A simple GUI Activity Diagram



Adding State Information to the Activity Diagram

Activity diagrams can be enriched by object flows to document the changes of the object's states during the execution of the activities.

Figure B-4. A UML Activity Diagram enriched by Object Flows



To capture the object flows, the state information of the GUI is modeled by so-called *object states* (represented by rectangulars) within the activity diagram. Figure B-4 shows the activity diagram presented in Figure B-3 enriched by the objects flows. Here, each object state represents one possible state of the application’s GUI. The object flows are then modeled by dependency relations represented as dashed arrows in the following way:

A dashed arrow which starts at the object state *`in`* and ends at an interaction state documents the GUI is in state *`in`* *before* the activity is executed. On the other hand, a dashed arrow which starts at an interaction state and ends at the object state *`out`*, models that *after* the execution of the activity the GUI is in state *`out`*.

> **Note:** Dashed arrows representing object flows will never start or end at action states of the activity diagram.

Formally, the activity diagram is enriched by the objects flows as defined by the following procedure:

1. For every state in the statechart diagram an *object state* is added to the activity diagram which represents the corresponding status of the GUI object.
2. The object flows are then specified by the following procedure:

   For every transition in the statechart diagram starting at state *`s1`* and ending at state *`s2`* two dashed arrows representing the object flows are added to the activity diagram:

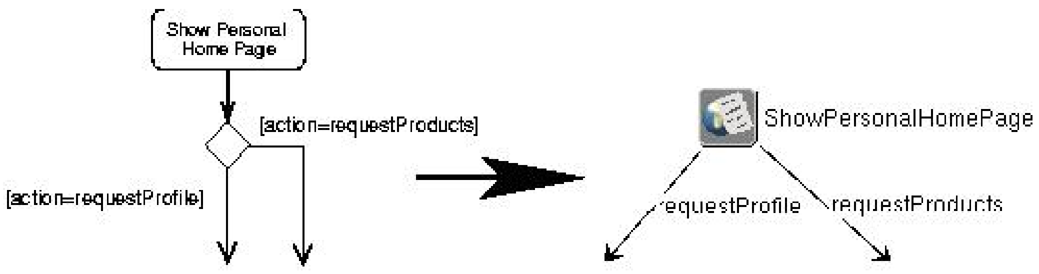
   1. one dashed arrow starts at the interaction state representing the state *`s1`* (created in step 1 of the previously documented transformation) and ends at the object state which represents *`s1`*, and
   2. the other dashed arrow starts at the object state representing the state *`s1`* and ends at the interaction state which represents the state *`s2`* (created in step 1 of the previously documented transformation).

The modeling of the object flow in the activity diagram is not necessary for the transformation of the statechart diagram into the service logic graph. But this information is important, when the statechart diagram should later be re-generated out of the service logic graph again.

B.2. Transforming the Activity Diagram

Once the activity diagram is obtained from the statechart diagram, the generation of the associated service logic graph is straightforward. Conceptually, the presented algorithm combines an interaction state and its associated branch (if there is any) to an interaction SIB. The corresponding exits of the SIB are determined by the event trigger part of the branch’s transitions (see Figure B-5). Each action state is represented by a SIB of the class `Action` having one exit named `dflt` (default).

Figure B-5. Transforming an Action State into a SIB



The object flow of the activity diagram is stored in the parameters of the interaction SIBs.

The transformation of the activity diagram into the service logic graph is performed by the following steps:

1. For every *interaction state* with name `n` in the activity diagram, a node with name `n` is added to the service logic graph. The SIB which is represented by these nodes is defined as follows:

```
SIB     ShowInteractionState
CLS     Interaction
PAR     template  STR 100 ""
PAR     in_state  STR 100 ""
PAR     out_state STR 100 ""
PAR     out       DIM  *  0
BR      requests[]out
```

   After the activity diagram has been imported into the Service Definition Environment the generated SIB definition can be modified to reflect the application-specific aspects. Here, the SIB parameters `in_state` and `out_state` should never be modified. This is because they store automatically generated information on the object flow contained in the activity diagram. In detail, the string parameter `in_state` contains the name of the state the GUI is in *before* the SIB is executed, and the parameter `out_state` holds the name of the state the GUI is in *after* the execution of this SIB. This information is used to re-generate the GUI statechart diagram out of the service logic graph if requested.

   The parameter `template` is generated for convenience purposes only. It stores a reference to the HTML page which is associated to the interaction state.

   The SIB exit definition needs some more explanation. It is split up into two parts. The SIB parameter `out` stores the number of possible SIB exits. It is of type `DIM *`, which defines the size of a dynamically-sized array. The second part of the definition is realized by the `BR` line. This declares an array of SIB exits whose size is determined by the value of the SIB parameter `out`. This means that if the value of `out` equals 10, the SIB has 10 exits with identifiers *`request[0]out`* to *`request[9]out`*.

When adding the edges to the service logic graph in step 3, each of the SIB exits represents one event trigger of an edge contained in the activity diagram.

2. For every *action state* with name *`n`* in the activity diagram, a node with name *`n`* is added to the service logic graph. The represented SIB is defined as follows:

```
SIB     ExecuteAction
CLS     Action
BR      dflt
```

3. For every transition in the activity diagram starting at the state (or at the branch which is associated to the state) *`s1`* and ending at state *`s2`*, an edge is added to the service logic graph which starts at node *`s1`* and ends at node *`s2`*.

If the source node of the edge is associated to an interaction state of the activity diagram:

> the name of the SIB exit associated to the edge is set to the guard-expression's event-trigger identifier of the corresponding edge in the activity diagram. The exit's identifier will be arbitrarily chosen.
>
> Note the difference between the identifier of a SIB exit and the associated name. Each SIB definition specifies a set of SIB-exit *identifiers*. They can be attached to edges which start a node representing this SIB. The *name* of the SIB exit is representation of the SIB-exit within the GUI. By default, the name of a SIB exit equals its identifier. For reasons of readability, this default name can be changed. Here, the renaming is done automatically by the transformation algorithm. The algorithm changes the default name of the SIB exit (e.g. `request[0]out`) to the event trigger identifier (e.g. `requestProduct`) of the chosen transition.

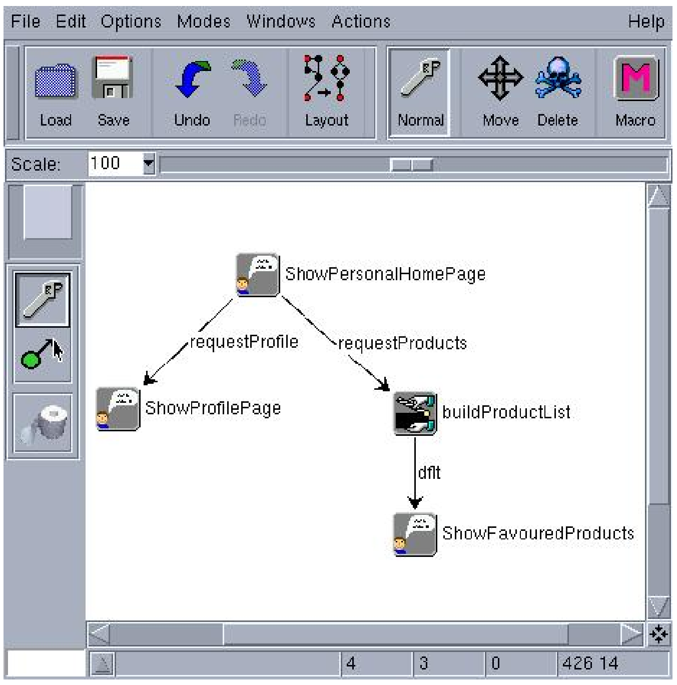
If the source node of the edge represents an action state:

> the SIB exit's identifier and the name associated to the edge are set to `dflt`.

Figure B-6 shows the service logic graph which corresponds to the activity diagram shown in Figure B-3. It is obtained by the algorithm presented above.

After the initial service logic graph has automatically been imported into the Service Definition Environment, it is refined by the application expert to model the full logic of the Web application, i.e. the application's coordination layer. Starting with the imported service logic graph, the application expert refines (not replaces) the SIBs representing the actions of the GUI statechart diagram by coordination SIBs implementing the required functionality.

Figure B-6. The Shop Service Logic Graph



Glossary

Activity

Atomic functional entity of the ETI tool repository, which represents a single functionality of an integrated tool.

Activity Taxonomy

Classification of the activities contained in the tool repository (see also Taxonomy).

Adaption Rule

Rule which matches user profiles to features offered by a personalized Web application.

Agent

A user specific view on the implementation of a Web application's coordination layer.

Agent Universe

Specification of the Web application's coordination layer which comprises the definition of features which should be available to every user and features which should only be available to users having a specific profile.

API

Application Programming Interface. Set of classes and functions which can be used to access the functionality of a software library.

Applet

An applet is a program written in the Java programming language that can be included in an HTML page.

AWT

Abstract Window Toolkit. Java class library which can be used to build graphical user interfaces.

Black Box Integration

Kind of tool integration which applies to software tools being available only in binary format and which run locally on the application host.

Business Class

Software component which represents things of the chosen application domain, like a shopping cart within a Web-shop application.

CADP

Caesar/Aldebaran Development Package. A software engineering toolbox for protocols and distributed systems.

CGI

Common Gateway Interface. An agreement between HTTP server implementors about how to integrate gateway scripts and programs to a legacy information system.

Computational Result

The data which is the result of the activity execution (see also diagnostic result).

Cookie

Persistent information which is written by the browser on the client's hard disk. Mainly used to store data between two HTTP requests.

Coordination Formula

SLTL-based formula which specifies a set of coordination sequences.

Coordination Sequence

A coordination sequence is a sequential program build on the basis of synthesis-compliant activities. It is generated by ETI's synthesis component out of an abstract description called loose specification.

Coordination Universe

Data structure which represents all possible combinations of the synthesis-compliant activities contained in the tool repository.

CORBA

Common Object Request Broker Architecture. A vendor-independent architecture and infrastructure that computer applications use to work together over networks.

CTI

Computer Telephony Integration. Scenario in which software programs interact with telephony switches.

DAG

Directed Acyclic Graph.

DBMS

Database Management System. Software system which is used to maintain databases.

Diagnostic Result

The result of the activity execution which comprises the diagnostic information generated by the software tool during its execution (see also computational result).

EJB

Enterprise Java Beans. Server-side component architecture being part of the J2EE standard.

Encapsulation Code

Part of a METAFrame module which wraps a third-party data type or tool feature into a set of C++ classes.

ETAPS

European Joint Conferences on Theory and Practice of Software.

ETI Community Online Service

The virtual meeting point where people involved in the development, extension and hosting of the ETI platform may discuss and exchange information.

ETI Site

A moderated Web site which uses the ETI platform to offer access to a repository of software tools.

Facade Class

Design pattern which is used to hide the complexity of a software library by one single class.

Graphs System

A collection of directed graphs plus some “global” information. The interpretation of the collection and the global information is application specific.

GUI

Graphical User Interface.

HLL

High Level Language. Procedural coordination language used to program the METAFrame Interpreter.

HTML

Hypertext Markup Language. Language used to write documents which can be viewed using a Web browser.

HTTP

Hyper-Text Transfer Protocol. Internet protocol used to access Web pages and applications.

HTTP Server

Software component which makes Web pages and applications accessible via HTTP.

HyTech

A tool for the analysis of embedded systems.

IIOP

Internet Inter-ORB Protocol. Internet protocol which is used for the communication between CORBA objects.

ISO

International Organization for Standardization. A worldwide federation of national standards bodies from 140 countries.

J2EE

Java 2 Enterprise Edition. Software architecture promoted by SUN Microsystems to build enterprise applications.

Java Byte Code

Machine-independent representation of Java programs. It can be executed via a Java Virtual Machine process.

Java Web Start

Java Web Start allows a user to simply deploy Web applications via the Internet. Java Web Start enabled applications can be started by clicking on an appropriate link on a Web page.

JDBC

Java Database Connectivity. API that lets you access virtually any tabular data source from the Java programming language.

JDK

Java Development Kit. Set of tools to program, document and run Java applications.

JNI

Java Native Interface. API which allows the integration of Java applications and applications written in other programming languages like C++,

JVM

Java Virtual Machine. Interpreter which is able to execute Java byte code.

Kronos

A tool for checking properties of real-time systems modeled as collections of timed automata.

Loose Specification

Language used to specify coordination sequences.

LOTOS

A formal description language based on temporal ordering of observational behavior defined by the ISO.

LTS

Labelled Transition System. Directed graph, where atomic information can be associated to nodes and edges.

METAFrame Module

Software component exporting HLL types or functions. These components can at runtime be imported to the METAFrame Interpreter to extend the basic programming language by new data types and functions.

Micro Feature

Part of a personalized-feature definition of a Web application. Micro features are used to define profile dependent characteristics of a personalized feature.

Modal Transition System

A labelled transition system where the edge set is split up into may and must edges.

Model Checking

Model checking is a technique which (semi-) decides, whether a mathematical structure (called model), e.g. an LTS, satisfies a certain (temporal) constraint.

Model-View-Controller Pattern

Design pattern which is used to strictly separate data and its presentation.

Module Adapter

Part of a METAFrame module which wraps the functionality offered by the encapsulation code into HLL types and functions.

Motif

Software library which can be used to realize the graphical user interface of an application.

MP3

MPEG Audio Layer 3. Coding of audio data for digital storage media at up to about 1,5 Mbit/s.

MPEG-2

Moving Pictures Expert Group. A working group of ISO/IEC in charge of the development of standards for coded representation of digital audio and video.

mu-calculus

Logic which can be used to specify temporal constraints on LTSs.

OMG

Object Management Group. The OMG is an open-membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications.

PDF

Portable Document Format. The open de facto standard for electronic document distribution.

Proxy Pattern

A proxy is a placeholder for another object to control access to it.

Remote Box Integration

Kind of tool integration which applies to software tools being installed on a machine which is not the application host.

RMI

The Java infrastructure which is used to realize application communication via a network.

Service Definition Environment

The Service Definition Environment is a tool which allows the graphical construction and validation of coordinating workflows on the basis of a component library.

Servlet

Java Servlet technology provides Web developers with a simple, consistent mechanism for extending the functionality of a Web server and for accessing existing business systems.

SIB

Service Independent Building Block. Basic software component which is used to build coordinating workflows with the help of the Service Definition Environment. SIBs can be classified either in interaction or coordination SIBs. Whereas interaction SIBs generate HTML pages which represent the result of a user interaction, coordination SIBs perform internal (not visible by the user) tasks.

SLG

Service Logic Graph. Directed graph which models the Web application at the coordination layer. Service Logic Graphs can be edited using the Service Definition Environment.

SLTL

Semantic Linear-time Temporal Logic. Temporal linear-time logic which can be used to loosely specify a set of paths within an LTS. The associated path set is determined by the algorithm documented in [SMF93].

SSI

Server Side Includes. Relatively non-complex commands that can be easily inserted into HTML pages to execute CGI programs, insert files, insert a date and time stamp and more.

SSL

Secure Socket Layer. The industry-standard method for protecting Web communications.

Strategy Pattern

The strategy pattern is used to provide abstract access to an algorithm. As an example, the comparison method used by a sorting procedure can be designed using this pattern. In this scenario, the strategy pattern allows the implementation of the sorting algorithm independent of the actually used comparison method.

Synthesis Solution Graph

The result of the synthesis process which represents all coordination sequences satisfying the given loose specification and being conformant to the provided synthesis strategy.

Synthesis Strategy

A synthesis strategy is a filter which is used to specify a subset of the coordination sequences satisfying a given loose specification. This subset is then returned as result of the synthesis process. Using the synthesis strategy, an end user can select the coordination sequences on the basis of their length.

Taxonomy

DAG-based classifications of activities and types available in the ETI tool repository.

ToolZone Software

The central client/server application of the ETI platform, which gives Internet-based, secure, performant and failsafe access to an ETI tool repository.

Type Taxonomy

Classification of the types contained in the tool repository (see also Taxonomy).

UML

Unified Modeling Language. Graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software intensive system.

UPPAAL

An integrated tool environment for modeling, validation and verification of real-time system modeled as networks of timed automata.

URL

Uniform Resource Locator.

Velocity

A Java-based template engine. It permits Web page designers to reference methods defined in Java code.

WAR

Web Archive Files. The J2EE standard for packaging the components realizing a Web application.

White Box Integration

Kind of tool integration which applies to software tools which can directly be accessed via an API.

XMI

XML Metadata Interchange. The OMG standard combining UML and XML.

XML

Extensible Markup Language. XML is an open standard of the World Wide Web Consortium (W3C) designed as a data format for structured document interchange on the web.

X Protocol

Protocol of the X Window System which is beside others used to display the GUI of an application on a remote device.

Print Bibliography

[BB87] T. Bolognesi and E. Brinksma, *Introduction to the ISO specification language LOTOS, Computer Networks and ISDN Systems*, 14, 25-59, 1987.

[BBCD97] M. von der Beeck, V. Braun, A. Claßen, A. Dannecker, C. Friedrich, D. Koschützki, T. Margaria, F. Schreiber, and B. Steffen, *Graphs in METAFrame: The Unifying Power of Polymorphism*, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), *Lecture Notes in Computer Science (LNCS)*, 1217, 112-129, Springer-Verlag, Heidelberg, 1997.

[BFKM97] M. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier, *Protocol Verification with the Aldebaran toolset*, *International Journal on Software Tools for Technology Transfer*, 1, 1+2, 166-183, Springer-Verlag, 1997.

[BK96] J. Bergstra and P. Klint, *The ToolBus coordination architecture*, Coordination Languages and Models (COORDINATION '96), *Lecture Notes in Computer Science (LNCS)*, 1061, 75-88, Springer-Verlag, Heidelberg, 1996.

[BKMS99] V. Braun, J. Kreileder, T. Margaria, and B. Steffen, *The ETI Online Service in Action*, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), *Lecture Notes in Computer Science (LNCS)*, 15791, 439-443, Springer-Verlag, Heidelberg, 1996.

[BM99] V. Braun and T. Margaria, *Neue Internet-Organisation zur Strukturierung und Organisation von Inhalten und Abläufen*, Online'99, Congress "Web Computing, Java, CORBA & DCOM", Symposium "WebCommerce: Intelligente Java-Anwendungen für Electronic Commerce", 3-89077-197-1, 1999.

[BMSB97] V. Braun, T. Margaria, B. Steffen, and F. K. Bruhns, *Service Definition for Intelligent Networks: Experiences in a Leading-edge Technological Project Based on Constraint Techniques*, Int. Conf. on Practical Application of Constraint Technology (PACT'97), The Practical Application Company, 1997.

[BMSY98] V. Braun, T. Margaria, B. Steffen, and H. Yoo, *Automatic Error Location for IN Service Definition*, 2nd Int. Workshop on Advanced Intelligent Networks (AIN'97), *Lecture Notes in Computer Science (LNCS)*, 1385, 222-237, Springer-Verlag, Heidelberg, 1998.

[BMSYR97] V. Braun, T. Margaria, B. Steffen, H. Yoo, and T. Rychly, *Safe Service Customization*, IEEE Computer Society Workshop on Intelligent Networks, IEEE Computer Society Press, 1997.

[BMW97] V. Braun, T. Margaria, and C. Weise, *Integrating tools in the ETI platform, International Journal on Software Tools for Technology Transfer*, 1, 1+2, 166-183, Springer-Verlag, 1997.

[Bra92] M. Brain, *Motif Programming: The Essentials...and More*, Digital Press, 1992.

[Bra99] V. Braun, *The Electronic Tool Integration Platform: Integration Guide*, Technical Documentation of the ETI Project, 1999.

[Bra99-a] V. Braun, *The Electronic Tool Integration Platform: ToolZone Client User Guide*, Technical Documentation of the ETI Project, 1999.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.

[CDHLPRZ00] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng, *Bandera : Extracting Finite-state Models from Java Source Code, Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[CES83] E. Clarke, E. Emerson, and A. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach*, Tenth Annual ACM Symposium on Principles of Programming Languages (POPL'83), 117-126, ACM Press, 1983.

[CES86] E. Clarke, E. Emerson, and A. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications, Transactions on Programming Languages and Systems*, 8, 2, 244-263, ACM Press, 1986.

[CGM93] N. Carriero, D. Gelernter, and T. Mattson, *Experience with the Linda Coordination Language and its Environment, Technical Report 958*, Yale University Department of Computer Science, 1983.

[Cla97] A. Claßen, *Component Integration in METAFrame*, Ph.D. Thesis, University of Passau, 1997.

[Cle99] R. Cleaveland, *Pragmatics of model checking: an STTT special section, International Journal on Software Tools for Technology Transfer*, 2, 3, 208-218, Springer-Verlag, 1999.

[Con00] J. Conallen, *Building Web Applications with UML*, Addison-Wesley, 2000.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen, *The Concurrency Workbench: a semantics-based verification tool for finite state systems, ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15, 1, 36-72, ACM Press, 1993.

[CSMB97] A. Claßen, B. Steffen, T. Margaria, and V. Braun, *Tool Coordination in METAFrame*, MIP-9707, Fakultät für Mathematik und Informatik, University of Passau, 1997.

[CW98] M. Campione and K. Walrath, *The Java Tutorial: Object-Oriented Programming for the Internet*, Second Edition, Addison Wesley, 1998.

[Dan98] A Dannecker, *Entwurf und Implementierung eines Hypertext-Systems in METAFrame*, Diploma Thesis, University of Passau, 1998.

[DCNB00] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham, *The PROSPER Toolkit*, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000), *Lecture Notes in Computer Science (LNCS)*, 1785, 78-92, Springer-Verlag, Heidelberg, 2000.

[DGSZ94] G. Dinkhoff, V. Gruhn, A. Saalmann, and M. Zielonka, *Business Process Modeling in the Workflow Management Environment LEU*, 13th International Conference on the Entity-Relationship Approach, *Lecture Notes in Computer Science (LNCS)*, 881, Springer-Verlag, Heidelberg, 1994.

[DL99] T. Demarco and T. Lister, *Peopleware: Productive Projects and Teams*, Second Edition, Dorset House, 1999.

[Dow98] T. Downing, *Java RMI: Remote Method Invocation*, IDG Books Worldwide Inc., 1998.

[Eme90] E. Emerson, *Temporal and modal logic*, *Handbook of theoretical computer science*, Edited by J. van Leeuwen, Elsevier, 1990.

[FGK96] J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu, *Cadp: A protocol validation and verification toolbox*, International Conference on Computer Aided Verification (CAV'96), *Lecture Notes in Computer Science (LNCS)*, 1102, Springer-Verlag, Heidelberg, 1996.

[FI00] D. Fields and M. Icolb, *Web Development with JavaServer Pages*, Manning, 2000.

[Fla98] D. Flanagan, *JavaScript*, O'Reilly, 1998.

[FM90] J.-C. Fernandez and Laurent. Mounier, *Verifying Bisimulations "On the Fly"*, International Conference on Formal Description Techniques (FORTE'90), North Holland, 1990.

[FS99] M. Fowler and K. Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Second Edition, Addison Wesley, 1999.

[Gar96] H. Garavel, *An overview of the Eucalyptus Toolbox*, Proc. of the COST 247 International Workshop on Applied Formal Methods in System Design, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[GM93] M. Gordon and T. Melham, *Introduction into HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.

[Gor98] R. Gordon, *Essential JNI: Java Native Interface*, Prentice Hall, 1998.

[HC98] J. Hunter and W. Crawford, *Java Servlet Programming*, O'Reilly, 1998.

[HHW97] T. A. Henzinger, P. Ho, and H. Wong-Toi, *HyTech: a model checker for hybrid systems*, *International Journal on Software Tools for Technology Transfer*, 1, 1+2, 110-122, Springer-Verlag, 1997.

[HM00] E. R. Harold and W. S. Means, *XML in a Nutshell*, O'Reilly, 2000.

[Hof97] J. Hofmann, *Program Dependent Abstract Interpretation*, Diploma Thesis, University of Passau, 1997.

[Hol97a] G. Holzmann, *The Model Checker Spin*, *IEEE Transactions on Software Engineering*, 23, 5, 279-295, 1997.

[Hol97b] A. Holzmann, *Der METAFrame-Interpreter: Entwicklung und Implementierung eines dynamischen Modulkonzeptes*, Diploma Thesis, University of Passau, 1997.

[Hol-a] A. Holzmann, *The High-Level_language: Programming Language of the METAFrame Interpreter*, Technical Documentation of the METAFrame Project.

[Hol-b] A. Holzmann, *Writing Adapter Specifications*, Technical Documentation of the METAFrame Project.

[ISPA98] *Internet Services with Profile based Adaptation*, PG 312, University of Dortmund, 1998.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

[KO96] P. Klint and P. Oliver, *The TOOLBUS Coordination Architecture: A Demonstration*, Fifth International Conference on Algebraic Methodology and Software Technology (AMAST'96), *Lecture Notes in Computer Science (LNCS)*, 1101, 575-578, Springer-Verlag, Heidelberg, 1996.

[Koz82] D. Kozen, *Results on the propositional mu-calculus*, International Colloquium on Automata, Languages and Programming (ICALP'82), *Lecture Notes in Computer Science (LNCS)*, 140, 348-359, Springer-Verlag, Heidelberg, 1982.

[Lar89] K. G. Larsen, *Modal Specifications*, Proc. of Automatic Verification Methods for Finite State Systems, *Lecture Notes in Computer Science (LNCS)*, 407, 232-2461, Springer-Verlag, Heidelberg, 1989.

[LOTOS] *ISO/IEC. Lotos: A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour* , Internal Standard 8807, 1992.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi, *Uppaal in a nutshell*, *International Journal on Software Tools for Technology Transfer*, 1, 1+2, 134-152, Springer-Verlag, 1997.

[LT87] K. G. Larsen and B. Thomsen, *Compositional Proofs by Partial Specification of Processes*, R87-20, Aalborg University Center, 1987.

[LT88] K. G. Larsen and B. Thomsen, *A Modal Process Logic*, Symposium on Logic in Computer Science (LICS'88), 203-210, IEEE Computer Society Press, 1988.

[Lut01] M. Lutz, *Programming Python*, O'Reilly & Associates, 2001.

[MB98] T. Margaria and V. Braun, *Formal Methods and Visualization: A Fruitful Symbiosis*, Int. Workshop on Visual Issues for Formal Methods (VISUAL'98) as Satellite Workshop of TACAS'98, *Lecture Notes in Computer Science (LNCS)*, 1385, 190-207, Springer-Verlag, Heidelberg, 1998.

[MBK97] T. Margaria, V. Braun, and J. Kreileder, *Interacting with ETI: a user session*, *International Journal on Software Tools for Technology Transfer*, 1, 1+2, 134-152, Springer-Verlag, 1997.

[MBS98] T. Margaria, V. Braun, and B. Steffen, *The ETI Online Service: Concepts and Design*, 4. Fachkongress "Smalltalk und Java in Industrie und Ausbilding, 1998.

[MBS01] T. Margaria, V. Braun, and B. Steffen, *Coarse Granular Model Checking in Practice*, SPIN Workshop 2001, Satellite to ICSE 2001, *Lecture Notes in Computer Science (LNCS)*, 2017, Springer-Verlag, Heidelberg, 2001.

[McM92] K.. L. McMillan, *Symbolic model checking - an approach to the state explosion problem*, PhD thesis, SCS, Carnegie Mellon University, 1992.

[Mil89] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[Mon00] R. Monson-Haefel, *Enterprise JavaBeans*, Second Edition, O'Reilly, 2000.

[MS97] T. Margaria and B. Steffen, *Coarse-grain Component Based Software Development: The METAFrame Approach*, 4. Fachkongress "Smalltalk und Java in Industrie und Ausbildung" (STJA'98), 29-34, 3-00-001828-X, 1997.

[MSS99] M. Müller-Olm, D. Schmidt, and B.. Steffen, *Model-Checking: A Tutorial Introduction*, Static Analysis Symposium (SAS'99), Edited by A. Cortesi Edited by and G. File, *Lecture Notes in Computer Science (LNCS)*, 1694, 330-354, Springer-Verlag, Heidelberg, 1999.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised),* MIT Press, 1997.

[NMHNSBI01] O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, and H.-D. Ide, *Library-based Design and Consistency Checks of System-level Industrial Test Cases*, Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2001), *Lecture Notes in Computer Science (LNCS)*, 2029, 233-248, Springer-Verlag, Heidelberg, 2001.

[NMHSBI01] O. Niese, T. Margaria, A. Hagerer, B. Steffen, G. Brune, and H. Ide, *Automated Regression Testing of CTI-Systems*, IEEE European Test Workshop (ETW 2001), 2001.

[NMNSBI00] O. Niese, T. Margaria, M. Nagelmann, B. Steffen, G. Brune, and H.-D. Ide, *An open Environment for Automated Integrated Testing*, 4th Int. Conf. on Software and Internet Quality Week Europe (QWE'00), November 2000.

[NNHKGEH01] O. Niese, M. Nagelmann, A. Hagerer, K. Kolodziejczyk-Strunck, W. Goerigk, A. Erochok, and B. Hammelmann, *Demonstration of an Automated Integrated Testing Environment for CTI Systems*, Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2001), *Lecture Notes in Computer Science (LNCS)*, 2029, 249-252, Springer-Verlag, Heidelberg, 2001.

[Nye95] A. Nye, *X Protocol Reference Manual : Volume Zero for Xii, Release 6: Definitive Guide to X Windows, Vol 0*, O'Reilly & Associates, 1995.

[PT87] R. Paige and R. Tarjan, *Three partition refinement algorithms, SIAM Journal of Computing*, 16, 6, 973-989, 1987.

[Res00] Edited by E. Rescorla, *SSL and TLS: Designing and Building Secure Systems*, Addison-Wesley, 2000.

[Rog97] D. Rogerson, *Inside COM*, Microsoft Press, 1997.

[SBFMMS98] A. Sicheneder, A. Bender, E. Fuchs, R. Mandl, M. Mendler, and B. Sick, *Tool-supported Software Design and Program Execution for Signal Processing Applications Using Modular Software Components,* International Workshop on Software Tools for Technology Transfer (STTT'98), Edited by T. Margaria Edited by and B. Steffen, *BRICS Notes Series NS-98-4*, 61-70, 1998.

[SC93] B. Steffen and R. Cleaveland, *A Linear-Time Model-Checking Algorithm for the Alternation-Free Mu-Calculus, International Journal on Formal Methods in System Design*, 1, 1, 1993.

[Sch01] S. Schäfer, *Auswertung regelbasierter Personalisierungsbeschreibungen für E-Commerce-Systeme*, Diploma Thesis, University of Dortmund, 2001.

[SCH99] M. Schumacher, F. Chantemargue, and B. Hirsbrunner, *The STL++ Coordination Language: A Base for Implementing Distributed Multi-agent Applications*, Third Int. Conference on Coordination Models and Languages (COORDINATION'99), *Lecture Notes in Computer Science (LNCS)*, 1594, 399-414, Springer-Verlag, Heidelberg, 1999.

[SCKKM95] B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria, *The Fixpoint Analysis Machine*, 6th International Conference on Concurrency Theory (CONCUR'95), Edited by J. Lee Edited by and S. Smolka, *Lecture Notes in Computer Science (LNCS)*, 962, 72-87, Springer-Verlag, Heidelberg, 1995.

[Shl88] S. Shlaer, *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1988.

[Sie00] J. Siegel, *CORBA 3 Fundamentals and Programming*, 2nd Edition, John Wiley & Sons, Inc., 2000.

[SM99] B. Steffen and T. Margaria, *METAFrame in Practice: Design of Intelligent Network Services*, *Lecture Notes in Computer Science (LNCS)*, 1710, 390-415, Springer-Verlag, Heidelberg, 1999.

[SMB97] B. Steffen, T. Margaria, and V. Braun, *The Electronic Tool Integration platform: concepts and design*, *International Journal on Software Tools for Technology Transfer*, 1, 1+2, 9-30, Springer-Verlag, 1997.

[SMB98] B. Steffen, T. Margaria, and V. Braun, *The Electronic Tool Integration Platform*, Int. Workshop on Software Tools for Technology Transfer, Satellite to ICALP'98, Edited by T.. Margaria Edited by and B. Steffen, *BRICS Notes Series*, NS-98-4, 53-54, 1998.

[SMBK97] B. Steffen, T. Margaria, V. Braun, and N. Kalt, *Hierarchical Service Definition*, *Annual Review of Communication*, 847-856, Int. Engineering Consortium (IEC), 1997.

[SMC96] B. Steffen, T. Margaria, and Andreas Claßen, *Heterogeneous analysis and verification of distributed systems*, *SOFTWARE: Concept and Tools*, 17, 1, 13-25, Springer-Verlag, 1996.

[SMCB96] B. Steffen, T. Margaria, A. Claßen, and V. Braun, *Incremental Formalization: A Key to Industrial Success*, *SOFTWARE: Concepts and Tools*, 17, 2, 78-91, Springer-Verlag, 1996.

[SMCB96a] B. Steffen, T. Margaria, A. Claßen, and V. Braun, *The METAFrame'95 Environment*, International Conference on Computer Aided Verification (CAV'96), *Lecture Notes in Computer Science (LNCS)*, 1102, 450-453, Springer-Verlag, Heidelberg, 1996.

[SMCB96b] B. Steffen, T. Margaria, A. Claßen, and V. Braun, *Incremental Formalization*, Int. Conference on Algebraic Methodology and Software Technology (AMAST'96), *Lecture Notes in Computer Science (LNCS)*, 1101, 608-611, Springer-Verlag, Heidelberg, 1996.

[SMCBNR96] B. Steffen, T. Margaria, A. Claßen, V. Braun, R. Nisius, and M. Reitenspieß, *A Constraint-Oriented Service Creation Environment*, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96), *Lecture Notes in Computer Science (LNCS)*, 1055, 418-421, Springer-Verlag, Heidelberg, 1996.

[SMCBR96a] B. Steffen, T. Margaria, A. Claßen, V. Braun, and M. Reitenspieß, *A Constraint-Oriented Service Creation Environment*, Int. Conf. on Practical Application of Constraint Technology (PACT'96), The Practical Application Company, 1996.

[SMCBR96b] B. Steffen, T. Margaria, A. Claßen, V. Braun, and M. Reitenspieß, *An Environment for the Creation of Intelligent Network Services*, *Intelligent Networks: IN/AIN Technologies, Operations, Services, and Applications - A Comprehensive Report*, 287-300, International Engineering Consortium (IEC), 1996.

[SMCBRW96] B. Steffen, T. Margaria, A. Claßen, V. Braun, M. Reitenspieß, and H. Wendler, *Service Creation: Formal Verification and Abstract Views.*, 4th Int. Conf. on Intelligent Networks (ICIN'96), 96-101, 1996.

[SMF93] B. Steffen, T. Margaria, and Burkard Freitag, *Module Configuration by Minimal Model Construction*, MIP-9313, Fakultät für Mathematik und Informatik, University of Passau, 1993.

[SO97] B. Steffen and E. R. Olderog, *Formale Semantik und Programmverifikation*, *Informatik-Handbuch*, Edited by P. Rechenberg Edited by and G. Pomberger, Carl Hanser Verlag, 1997.

[Str97] B. Stroustrup, *The C++ Programming Language*, Third Edition, Addison Wesley, 1997.

[Tha99] T. Thai, *Learning DCOM*, O'Reilly, 1999.

[TL01] T. Thai and H. Lam, *.NET Framework Essentials*, O'Reilly, 2001.

[VW86] M. Vardi and P. Wolper, *An automata-theoretic approach to automatic program verification*, Symposium on Logic in Computer Science (LICS'86), IEEE Computer Society Press, 1986.

[WC99] K. Walrath and M. Campione, *The JFC Swing Tutorial: A Guide to Constructing GUIs*, Second Edition, Addison Wesley, 1999.

[WCO00] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl*, 3rd Edition, O'Reilly & Associates, 2000.

[Wel97] B. Welsch, *Practical Programming in Tcl an Tk*, Second Edition, Prentice Hall, 1997.

[Wel00] T. Wellhausen, *Regelbasierte Personalisierung von E-Commerce-Systemen*, Diploma Thesis, University of Darmstadt, 2000.

[WHCHF99] S. White, M. Hapner, R. Cattell, G. Hamilton, and M. Fisher, *JDBC API Tutorial and Reference: Universal Data Access for the Java 2 Platform*, Second Edition, Addison Wesley, 1999.

[Yoo] H. Yoo, *title to be announced*, Ph.D. Thesis, University of Dortmund, to appear.

[Yov97] S. Yovine, *Kronos: a verification tool for real-time systems, International Journal on Software Tools for Technology Transfer*, 1, 1+2, 123-133, Springer-Verlag, 1997.

Web Bibliography

The referenced URLs have been collected in the year 2001. But since the Web is evolving every day, there might be links providing better information on the aspects or even links which may even not exist any longer by the time you read this document.

[Apache] *The Apache Project*, www.apache.org/.

[Applets] *Applets*, java.sun.com/applets/.

[AppServer] *AppServer: e-Infrastructure for e-Business*, www.borland.com/appserver/.

[Bandera] *Bandera: Software Model Construction for Finite-state Verification*, www.cis.ksu.edu/santos/bandera/.

[CADP] *The Caesar/Aldebaran Development Package*, www.inrialpes.fr/vasy/pub/cadp.html.

[CGI] *CGI: Common Gateway Interface*, www.w3.org/CGI/.

[ColdFusion] *Cold Fusion, a cross-platform Web Application Server*, www.allaire.com/products/coldfusion/.

[COM] *Component Object Model-based technologies*, www.microsoft.com/com/.

[Coordination2002] *COORDINATION 2002: Fifth International Conference on Coordination Models and Languages*, www-users.cs.york.ac.uk/~wood/Coord02/Coordination2002.html.

[CORBA] *CORBA Basics*, www.omg.org/gettingstarted/corbafaq.htm.

[CSSA] *comp.simulation software archive*, www.nmsr.labmed.umn.edu/~michael/dbase/comp-simulation.html.

[DownloadCOM] *CNET download.com*, www.download.com.

[Dreamweaver] *Macromedia Dreamweaver*, www.macromedia.com/software/dreamweaver/.

[EJB] *Enterprise JavaBeans technology: The Industry-Backed Server-Side Component Architecture*, java.sun.com/products/ejb/.

[EUCALYPTUS] *A Guided Tour of EUCALYPTUS* , www.inrialpes.fr/vasy/cadp/tutorial/.

[Flash] *Macromedia Flash 5*, www.macromedia.com/software/flash/.

[FME] *Formal Methods Europe*, www.fmeurope.org.

[FrontPage] *FrontPage Home*, www.microsoft.com/frontpage/.

[HTML] *Hypertext Markup Language Home Page*, www.w3.org/MarkUp/.

[HTTP] *HTTP - Hypertext Transfer Protocol*, www.w3.org/Protocols/.

[HyTech] *HyTech: The HYbrid TECHnology Tool*, www-cad.EECS.Berkeley.EDU/~tah/HyTech/.

[IBMebusiness] , www.ibm.com/software/ebusiness/.

[ISO] *International Organization for Standardization*, www.iso.ch.

[Jakarta] *Home Page of the Jakarta Project*, jakarta.apache.org.

[J2EE] *Java 2 Platform Enterprise Edition*, java.sun.com/j2ee/.

[Java] *The Source for Java Technology*, java.sun.com.

[JavaAPI] *Java 2 SDK, Standard Edition Documentation*, java.sun.com/products/jdk/1.2/docs/.

[JavaBeans] *JavaBeans: The only Component Architecture for Java Technology*, java.sun.com/products/javabeans/.

[JavaCodeConv] *Code Conventions for the Java Programming Language*, java.sun.com/docs/codeconv/.

[Javadoc] *Javadoc 1.2*, java.sun.com/products/jdk/1.2/docs/tooldocs/javadoc/.

[JavaPlugIn] *The Java Plug-in*, java.sun.com/products/plugin/.

[JavaPorts] *Java Platform Ports*, java.sun.com/cgi-bin/java-ports.cgi.

[JavaPress] *Java Press Release 4/17/95*, java.sun.com/pr/1995/04/pr950417-01.html.

[JavaWebStart] *Java Web Start*, java.sun.com/products/javawebstart/.

[JBuilder] *JBuilder: Pure Java Visual Development*, www.borland.com/jbuilder/.

[JDBC] *Java Database Connectivity*, java.sun.com/products/jdbc/.

[JDC] *The Java Developer Connection*, developer.java.sun.com/developer/.

[JMeter] *A Java desktop application designed to load test functional behavior and measure performance*, java.apache.org.

[JNI] *Java Native Interface*, java.sun.com/products//jdk/1.2/docs/guide/jni/.

[JSP] *Java Server Pages*, java.sun.com/products/jsp/.

[Kronos] *Kronos, a tool to verify complex real-time systems.*, www-verimag.imag.fr/TEMPORISE/kronos/.

[Linda] *Yale Linda Group*, www.cs.yale.edu/Linda/linda.html.

[MagicDraw] *MagicDraw, a visual UML modeling and CASE tool.*, www.magicdraw.com.

[MF] *The METAFrame Project*, sunshine.cs.uni-dortmund.de/projects/METAFrame/.

[MFTech] *METAFrame Technologies Home Page*, www.metaframe.de.

[mokassin] , Mokassin Home Pagewww.informatik.uni-bremen.de/grp/mokassin/.

[MPEG] *The MPEG Home Page*, www.cselt.it/mpeg/.

[MSNetServers] *Microsoft Servers Home Page*, www.microsoft.com/servers/.

[MySun] *My Sun*, mysun.sun.com.

[NET] *Microsoft .NET*, www.microsoft.com/net/".

[MyYahoo] *My Yahoo*, my.yahoo.com.

[PLGraph] *The PLGraph Library*, sunshine.cs.uni-dortmund.de/projects/METAFrame/plgraph/.

[PNTD] *The Petri Nets Tool Database*, www.daimi.aau.dk/PetriNets/tools/db.html.

[Prosper] *Proof and Specification Assisted Design Environments*, www.dcs.gla.ac.uk/prosper/.

[Rational] *Rational Software: the e-development company*, www.rational.com.

[RMI] *Java Remote Method Invocation (RMI)*, java.sun.com/products/jdk/rmi/.

[RMIFAQ] *Frequently Asked Questions on RMI and Object Serialization*, java.sun.com/ products/jdk/1.2/docs/guide/rmi/faq.html.

[Robot] *Rational Robot: Reduce testing time and effort*, www.rational.com/products/robot/.

[Rose] *Rational Rose Product Information*, www.rational.com/products/rose/.

[Sally] , lrs.fmi.uni-passau.de/projekte/sally.html.

[SanFrancisco] *IBM SanFrancisco: Overview*, www.ibm.com/software/ad/sanfrancisco/.

[SanFranciscoAD] *IBM SanFrancisco: Application Development Environment*, www.ibm.com/software/ad/sanfrancisco/app_overview.html.

[Servlet] *Java Servlet API*, java.sun.com/products/servlet/.

[SMV] *SMV guided tour*, www.cs.cmu.edu/~modelcheck/tour.html.

[spin] *On-the-fly, LTL Model Checking with SPIN*, netlib.bell-labs.com/netlib/spin/whatispin.html.

[SSI] *TDavid's Server Side Includes (SSI) Tutorial*, www.tdscripts.com/ssi.html.

[SSL] *SSL 3.0 Specification*, home.netscape.com/eng/ssl3/.

[STLPP] *STL++: A Coordination Language for Autonomy-based Multi-Agent Systems*, www-iiuf.unifr.ch/~schumach/publications/rep-march98/.

[STTT] *International Journal on Software Tools for Technology Transfer*, sttt.cs.uni-dortmund.de.

[SUN] *SUN Microsystems*, www.sun.com.

[Swing] *The Swing Connection*, java.sun.com/products/jfc/tsc/.

[TeamTest] *Rational TeamTest Product Information*, www.rational.com/products/teamtest/.

[Together] *ControlCenter Product Information*, www.togethersoft.com/products/controlcenter/.

[tomcat] *Tomcat, the official Reference Implementation for the Java Servlet and JavaServer Pages technologies.*, jakarta.apache.org/tomcat/.

[ToolBus] *The ToolBus Application Architecture*, www.cwi.nl/projects/MetaEnv/toolbus/.

[TZUser] *The ToolZone Client User Guide*, http://www.eti-service.org/software/documentation.html.

[UML] *UML Resource Page of the OMG*, www.omg.org/uml/.

[UMW] *UM Weather: Software Library*, cirrus.sprl.umich.edu/wxnet/software.html.

[Uppaal] *UPPAAL, an integrated tool environment for modeling, validation and verification of real-time systems.*, www.docs.uu.se/docs/rtmv/uppaal/.

[vajava] *Visual Age for Java: Overview*, www.ibm.com/software/ad/vajava/.

[Velocity] *Velocity, a Java-based template engine*, jakarta.apache.org/velocity/.

[VisualStudio] *Microsoft Visual Studio Home Page*, msdn.microsoft.com/vstudio/.

[WaterBeans] *WaterBeans: A Custom Component Model and Framework*, www.sei.cmu.edu/cbs/cbse2000/papers/23/23.html.

[WBC] *WebSpehre Business Components: Overview*, www.ibm.com/software/webservers/components/.

[WBCC] *Component Composer: Overview*, www.ibm.com/software/webservers/components/composer/.

[WebLogic] *BEA WebLogic Server*, www.bea.com/products/weblogic/server/.

[WebMacro] *WebMacro Home Page*, www.webmacro.org.

[WebSphere] *IBM WebSphere Server Platform*, www.ibm.com/websphere".

[XEmacs] *XEmacs, a highly customizable open source text editor*, www.xemacs.org.

[XMI] *XML Metadata Interchange*, www-4.ibm.com/software/ad/library/standards/xmi.html.

[XML] *CORBA, XML And XMI Resource Page of the OMG*, www.omg.org/xml/.

[XWindows] *Technical X Window System and Motif WWW Sites*, www.rahul.net/kenton/xsites.html.