An Overview of The Totem Software Engineering Meta-Environment

Arndt von Staa[1]
arndt@inf.puc-rio.br
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
22453-900 Rio de Janeiro, RJ
Brazil

Donald D. Cowan
dcowan@csg.uwaterloo.ca
Computer Science Department
University of Waterloo
Waterloo, Ont.
Canada
N2L - 3G1

PUC-Rio.InfMCC 35/95

November 1995

ABSTRACT

This report presents an overview of Totem's functionality. Totem is a computer aided software engineering meta-environment. It may be used to build software development processes, representation languages and tools, as well as to execute according to these tools and control execution according to these processes.

Totem operates on a net of workstations, each containing an environment instance. Each of these environment instances provides tools to support some of the activities of a specific software development and maintenance process. The collection of environment instances supports virtually all activities of a given development process.

Totem stores fine grained objects in a distributed repository. The semantics of this repository as well as the definition of user interfaces, representation languages and processes are kept in a definition base. Definition bases are derived from an environment base which contains all facts about supported methods, tools and processes. The environment base is used by the environment builder to create and maintain processes, representation languages and repository definitions.

One of the basic aims of Totem is to compose code from high level specifications. For this purpose Totem has its own internal programming language. This language is used to construct specific tools, such as editors, code composers, representation transformers and representation validators. The programming language allows, also, to build the definitions of all representation languages available.

Keywords: definition base, meta-environment, repository, representation language, software engineering environment, software development process, software quality, software quality assurance.

---

[1] Supported by: CNPq - 201183/93-5

RESUMO

Este relatório apresenta uma visão geral da funcionalidade do meta-ambiente de engenharia de software Totem. Este meta-ambiente pode ser utilizado para projetar e implementar processos de desenvolvimento de software, linguagens de representação e ferramentas de apoio ao desenvolvimento de software, onde estes elementos formam um todo coerente e integrado dirigido pelo processo de desenvolvimento.

Totem opera em uma rede de estações de trabalho, cada qual contendo uma instância do ambiente. Cada instância provê um subconjunto dos processos, linguagens de representação e ferramentas necessários para a execução das tarefas planejadas a serem realizadas na estação de trabalho. O conjunto de todas as instâncias de ambiente suportam virtualmente todas as atividades do desenvolvimento, da manutenção e da gerência do desenvolvimento de software.

Totem armazena objetos de pequena granularidade em um repositório distribuído. A semântica deste repositório, a definição das interfaces de usuário específicas para uma instância de ambiente, a definição do processo de desenvolvimento, a definição das linguagens de representação e a definição das ferramentas disponíveis é mantida em uma base de definição. A base de definição é criada a partir de uma base de ambiente. Esta contém a definição dos diversos processos, linguagens de representação e ferramentas disponíveis no meta-ambiente. A criação de novos componentes de ambientes se faz nesta base de ambiente.

Um dos objetivos de Totem é permitir a composição e a manutenção de código a partir de especificações de alto nível. Para atingir esta meta, Totem dispõe de uma linguagem de programação interna. Esta linguagem permite a criação de ferramentas, linguagens e processos. Através desta ferramenta são configurados os editores, validadores, transformadores, geradores de documentação, compositores de código e a interface de usuário necessária para operar Totem.

Palavras chave: bases de definição, meta-ambientes, repositórios, linguagens de representação, ambientes de engenharia de software, processos de desenvolvimento de software, qualidade de software, garantia de qualidade.

Table of contents

1. Introduction

The technology and process surrounding software development present a moving target to both developers and organizations involved in producing software; constant change and adaptation are key issues. As stated in the CASE'90 Report [CASE90]:

> Among the greatest challenges lying ahead is the need for tighter integration among tools in a manner that supports openness to a variety of methods, notations, processes, tools, and platforms. Understanding the software development process and getting developers to use software engineering techniques correctly and consistently will remain a problem, especially in the face of evolving technology. A more disciplined approach to software process engineering employing metrics, feedback and continuous improvements is needed. Management will have to take a long-range economic perspective, modify objectives and incentives, and incorporate principles of Total Quality Management to ensure the ultimate success of CASE.

A major reason for the difficulty of developing and maintaining software is its inherent complexity [Brooks86]. Software can be viewed as a collection of documents such as executable computer program code, initial data bases, source code, engineering documents, user documents, training documents, help files and executable tutorials. All of these widely different documents are interdependent and must be consistent one with the other.

In addition to supporting openness in the face of constant change, software engineering environments must conform to the software class being developed. For example, the environment required to develop a large command and control system significantly differs from the environment required to develop a commercial application. Thus, software environments must not only adapt to changing methods, processes and tools, they must also adapt to the target system classes developed.

Creation of a software engineering meta-environment that would be a collection of tools for producing environments for specific methods, processes and classes of software could provide a solution to many of the problems just described. In addition, since software development is still not a well-defined discipline, such a meta-environment could support experimentation with the software process. Ideally the time required to learn and use the meta-environment effectively should be small in order to encourage its use in both production and to support experimentation.

Totem is a software engineering meta-environment directed toward these and other goals, that can be used to instantiate software engineering environments for development and maintenance of target software systems. A software engineering environment produced using Totem is strongly computer assisted in that it will contain tools that interact using a repository and support most of the development, maintenance and management operations. Many of the supported operations will occur without operator intervention. As a consequence of using a repository, creating or changing some fact in a document will automatically propagate that change to all other related documents including diagrams, code segments, indexes and cross-references. In such an environment inconsistencies are unlikely to occur and accumulate. Because of these and other properties we expect that Totem will reduce the cost and time to build an effective and efficient software engineering environment. Consequently, we will be able to study in more depth the pragmatic aspects of software development, adapt tools to increase their effectiveness, as well as build environments acceptable to industry.

1.1 Totem's goals

The set of goals can be partitioned into two major groups. The first addresses goals of the Totem meta-environment itself. The second addresses goals of the development process used to develop and maintain Totem. In this section we will state the goals of the environment.

Goal 1: Totem shall support the creation and evolution of computer aided software engineering environments.

Building working and useable environments, or better, populating them with reliable tools and processes, is a major effort. This effort is virtually endless, since environments must adapt to the kind of software being developed. This is so since software development and maintenance steps correspond to specific projects, even if no formal planning is performed. Also when software is composed from standard components, each system is the result of a specific project. Thus, environments must be adapted to each system being developed and later on maintained.

Ideally environments should be composed from a set of already existing tools and allow adding new tools whenever necessary. The set of tools itself should be adapted and fine tuned in order to fit the needs of a specific project, or to encompass new acquired definitions about software development techniques and processes.

Software development environments contain several components which must be adjusted one to the other and which aim at the development and maintenance of software. Optimizing some of these components not necessarily optimizes the environment. It is our experience that it is not sufficient to populate an environment with an *integrated* set of tools, i.e. tools which obey a well-defined interface. It is necessary that these tools *interact*, that is they must inter-operate in such a way that the effect of one tool propagates to other tools and the corresponding result propagates back to the initial tool. On the other hand, as stated by Lehman and Belady in [BL85], we know that system evolution is an intrinsic property of software and, hence, cannot be avoided even with the best possible development processes.

Goal 2: Environments generated with the aid of Totem shall be capable of economically producing high quality software.

A fundamental goal of any environment is to produce faster software of better quality, otherwise nothing would be gained when compared to unaided development. It is our aim that Totem produces software compliant with the ISO 9000 series of standards. It is also our aim that development processes based on the use of Totem should qualify for acknowledged standards, such as level 5 of the CMU/SEI Process Capability Maturity Model [PCCW93]. Note that this does not mean that with the use of Totem these standards will be assuredly be satisfied. It does mean, however, that Totem must not create obstacles to conforming to such standards, and indeed Totem should make it easier to achieve conformance.

Goal 3: Totem shall act as a laboratory to perform applied research in software engineering.

It has been recognized that software development processes must change dramatically, if development and maintenance costs are to be reduced and if, simultaneously, the quality of software is to be increased [Brooks86]. Thus, tools must be built to fit smoothly with these development processes and not the other way round. In other words, trying to build an environment from a set of tools will not result in significative changes in cost and quality. However, defining a process and building tools which fit this process is expected to have a great impact on productivity.

The development process depends on the nature of the target system to be built, on the tools and techniques which may be used to build them, and on the technical background of people involved in developing software. Thus, similar to conventional software systems [BL85], development processes interact with themselves in the sense that changing the development environment to better suit a given development process will also entail changes to this very development process. Consequently, completely specifying software engineering environments which provide adequate support to some development processes is a difficult if not impossible task. However, *evolving* by continual enhancement to a well established and mature process and its supporting environment seems to be feasible. But, in order to adequately evolve, controlled (measured) experiments must be made[Basili95], some of them certainly will lead to blind alleys. Obviously these experiments must not take too much time and they must not require a large amount of resources.

In [Fenton93] Fenton states that evidence of the effectiveness of already proposed solutions is often unrealistically assessed and is frequently based more on guesswork than on proven facts. In

[Fenton94] he then defines a basis to conduct software measurement. Clearly, experiments performed in laboratory contexts, although necessary, do not provide sufficient feedback, since the peopleware found in laboratories is quite different from that found in industry. Thus, it is important that experiments using proposed solutions are made not only in a research laboratory set up, but also in industrial contexts. However, to achieve this, tools and processes must be used and assessed in industrial contexts and, hence, must be of industrial quality.

New representation languages are being created almost every day. Seldom, though, their effectiveness is thoroughly assessed. Most often they are adopted on the base of belief. On the other hand, performing controlled experiments, changing tools as a result of the findings of these experiments are far too costly and time consuming to be of practical value. We expect that with the aid of Totem we will be able to significantly reduce this cost, at least in a laboratory setting.

Goal 4: Totem shall support reverse engineering and re-engineering of existing systems.

Few totally new systems are being developed. Most of the systems are evolved from already existing systems, most of which are already inconsistent with existing specification and design documents. These systems should be internalized into the computer aided environment. In order to achieve this there must be tools and processes which allow the reverse engineering. These tools should also allow selective re-engineering in order to improve or evolve the refurbished systems. A significative part of the existing code is not touched when evolving reverse engineered systems. Hence code generation from the repositories should be capable of reproducing this code exactly as it was before the reversal. In other words, we do not aim simply at transforming legacy code into better structured code possibly written in a new different, we aim at absorbing legacy code into the environment allowing it to be maintained in a reasonable way once absorbed.

Goal 5: Totem shall provide means for the effective reuse at high levels of abstraction.

Reuse is one of the most promising ways to increase productivity by orders of magnitude and, simultaneously, increasing quality. However, to achieve this increase of productivity, reuse should be performed at high levels of abstraction and not just at the detailed design level. Furthermore, reuse should be performed by reusing components as they are without any changes. Copying existing elements and then changing them, although a form of reuse, duplicates the amount of code or design to be maintained. A better form, though harder to achieve, is to reuse by referencing directly to the reused element without any change whatsoever.

One interesting form of reuse which is becoming more widespread is achieved over computer networks, in particular by means of ftp and WWW services. Here it may be interesting to establish links between repositories over networks. Interesting problems arise when considering evolution, version and configuration control.

1.2 Goals of the Totem project

In this section we will state the goals of the software engineering environment used to develop Totem.

Goal 6: The software process to be used to develop Totem shall provide a consistent set of metrics.

A basic characteristic of process based development is measurement, and process assessment and improvement based on these measurements. Metrics of past projects should be used to direct the planning of new projects. Metrics should also be capable of providing clues to detect process, representation and tool deficiencies. By acting on the causes of such deficiencies one expects to improve the processes [Humphrey 89, Humphrey95, Basili95]. Thus a reliable project history data base must be set up. Using this data base to compare metrics of new projects, it is possible to assess the impact of changes in processes and the use of new tools. It is then of fundamental importance to collect project data in order to assess the impact of Totem once it becomes available. We will establish and populate the project history data base while developing Totem itself.

Goal 7: The software process to be used to develop Totem shall evolve to a ISO 9000 compliant environment.

One of Totem's goals is to establish an ISO 9000 compliant software development environment. In order to achieve this, a solid experience in working in ISO 9000 compliant environments must be achieved. This experience will be used to assess and review Totem's implementation. Furthermore, while developing Totem, data will be gathered which will be used to review and redirect the goals of Totem itself whenever necessary in order to achieve this goal. Although we aim at becoming ISO 9000 compliant, we will not attempt at obtaining a ISO 9000 compliant certificate.

Goal 8: The process shall result in a pre-industrial version of Totem satisfying the goals stated in the previous section.

By a pre-industrial version we understand a version which is reliable and satisfies high engineering standards, but which not necessarily satisfies high marketing standards or expectations. In other words, it is not our intention to produce and maintain a marketable software product.

1.3 Background

Totem is an evolution of Talisman [Staa93]. The following are some of Talisman's features:

- it is an integrated CASE tool supporting several activities of the software development process, such as: requirements specification, system specification, system design, modular design and implementation, work break down based planning, document generation.
- all tools interact via a repository. This repository is a special purpose data base. It stores project objects of small granularity. Coarse granularity objects – *representations* –, such as source code files, diagrams, and user documents, are built whenever they are needed. Representations are built by composing and formatting a selection of small project objects into these large objects.
- the repository provides a basic set of data base functions, such as crash recovery and access optimization.
- it uses an internal programming language, the *form programming language*. This language directs how to compose textual representations from the contents, graphical or not, of the repository. This language is used, among others, to define how to compose source code from the repository's contents, how to define representation verifiers capable of generating quality control reports, how to implement representation transformers used to partially convert from one representation to another.
- it allows building code generators. Among these generators are form programs capable of generating data base applications from extended data models. Some of these generators are now being used in several corporations to develop and maintain business applications.
- it allows building code composers (linearizers). These are form programs capable of building sequential code from the contents of the repository. They have been used to develop Talisman itself. Presently all of the modules which comprise the Talisman project are composed from Talisman repositories.
- the set of editors, i.e. text editor, structure editor and diagram editor, are all definition-base-driven meta-editors. This allows for the addition and evolution of representation languages without having to change the code of the meta-environment itself.
- the set of editors provide a hyper-document like navigation capability.
- it is a meta-environment prototype. Existing representation languages can be dynamically adapted. New representation languages can be added.

- Talisman was originally designed to run on a simple PC, which it does quite well. In general the response of the system, even on a PC, is sufficiently fast in order not to annoy developers while browsing or updating repositories.

The experience gained with the development and use of Talisman led to the proposal and development of Totem. Thus, in a sense, Talisman is a prototype of Totem.

1.4 Overview of this report

In Section 2, we will provide an abridged functional overview of Totem and introduce some of the terminology used throughout this report. We briefly describe how Totem is used to create and maintain processes, representation languages and tools. How environment instances are built by assembling selected processes, representation languages and tools into specific environments, which are then partitioned into several environment instances. We also provide a brief overview of the organization of the repository.

In Section 3, we will state some of the requirements which should be satisfied by software engineering environments assembled with the Totem's support. We will examine solely the services provided by these software engineering environments, without examining how to satisfy these requirements. We will be using a narrative style instead of the requirement list style frequently found in requirements documents.

In Section 4, we describe Totem's high-level architecture. This architecture is process, representation language and tool independent. Data contained in the definition and process base define the specific behavior required in order to implement an environment instance. This organization allows the construction of virtually any environment, without needing to reprogram any of Totem's components.

In Section 5, we will briefly describe how we are planning to implement Totem. Similar to building compilers, Totem will be built in several steps using a bootstrapping approach. That is, components of the Totem system will be built in such a way that the result of a given development step provides tools to build the components of the next step. In this way, components will be systematically evaluated in practical use right after they are made available. We expect that in this way Totem will quickly converge to a reliable and useful system. In fact we are following recommendations by Brooks [Brooks86], in the sense that systems should be systematically grown. Furthermore we are following a spiral model of development [Boehm], in the sense that the result of each step can be viewed as a prototype to be assessed and which will direct the development of the next step. Different from the spiral model, we expect that the Totem system will be ready when completing the last cycle and, hence, will not require a full conventional development process in order to build the final accepted system.

Finally, in Section 6, we present a brief wrap up of this report.

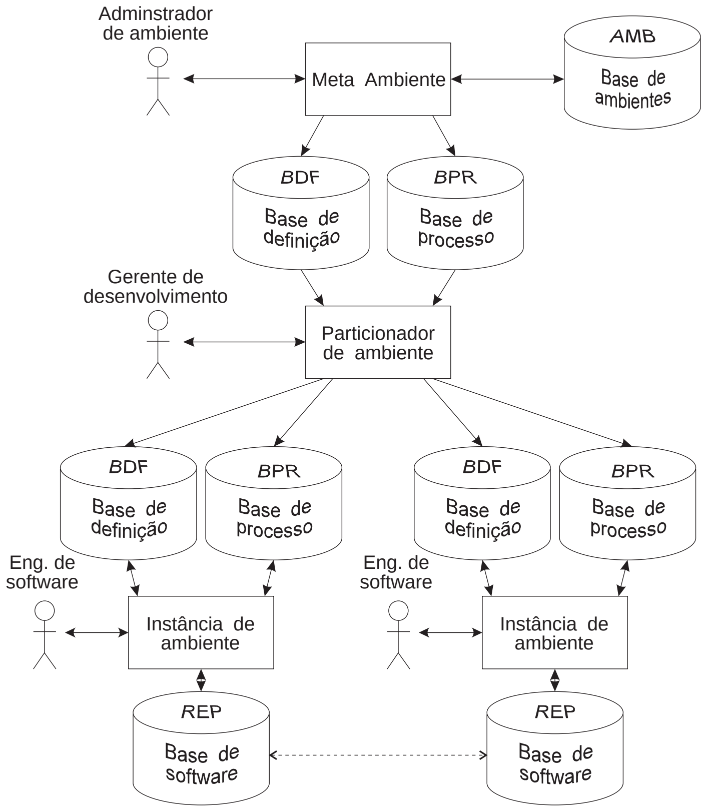

Figure 1. Flow within the meta environment structure

2. Functional overview

In this section, we provide an abridged functional overview of Totem and introduce some of the terminology used throughout this report. We briefly describe how Totem is used to create and maintain processes, representation languages and tools. How environment instances are built by assembling selected processes, representation languages and tools into specific environments, which are then partitioned into several environment instances. We also provide a brief overview of the organization of the repository.

Figure 1 depicts Totem's overall organization. Using the Totem meta-environment, the *environment builder* defines processes, representation languages and tools. All these items are kept in the *environment base*. This is a repository of all facts concerning environment components. In order to create and maintain the environment base, the software builder uses the *meta-environment representation languages*, and *meta-environment tools*. The definitions of these representation languages and tools are kept in the *meta-definition base*. The definition of the process used to build and maintain the environment base is kept in the *meta-process base*.

Once several processes, representation languages and tools are available, the environment builder selects from these items and assembles a *specific environment*. A specific software engineering environment supports the development of a class of software systems. Several different specific environments may be assembled. For example, one might want to create a specific environment to develop information systems, and another one to develop embedded control systems. The criteria used to select a specific environment from the environment base are typically: quality requirements of the systems to be developed, software development standards to be obeyed, and class of software systems to be developed. Each specific environment is defined by means of a definition base and a process base. The definition base describes all the tools (verification and validation tools, measurement tools, transformers, code composers) and representation languages (aspect, user interfaces, syntax and semantics) to be used. The process base defines the complete software process to be used. This process is bound to the tools and representation languages contained in the definition base.

Once a specific environment is defined, the *project manager* partitions this environment into *environment instances*. Each instance resides in a personal workstation and contains:

1. A specific *definition base* which defines the representation languages and tools to be used in this workstation.
2. A specific *process base* which defines the sub-process to be followed while working on this workstation.
3. A local *repository* containing all facts about a target system gathered or modified while working on this workstation. It may also contain copies of facts generated by means of checkout operations.
4. A local *metrics base*. The metrics base contains all measurement data automatically or manually gathered and recorded while enacting parts of the process. The local metrics base may be integrated with the environment metrics base. The environment metrics base contains measurement data about all projects developed. It contains thus the accumulated knowledge about past and in progress developments. It may also contain measurement and assessment data from other projects.

Using an environment instance the *software engineer* creates and maintains facts about *target systems*. These facts are kept in a *local repository*. Once accepted, these local repositories may be checked-in to public repositories. When developing a new work-product, the local repository starts empty. When maintaining an already existing work-product, the local repository may be populated by means of a check-out operation, or a software base (described later in this section) may become temporarily part of the local repository and, thus becoming unavailable for other environment instances. Another way of integrating the contents of repositories is achieved by means of links between objects contained in different software bases. In this case no check-in operation will be needed. These two mechanisms allow the integration of all facts about target systems developed or re-engineered with the support of the environment.

The integrated set of repositories constitutes the *corporate repository* which contains all facts about all target systems being developed, maintained or reverse-engineered with the support of the environment. In addition to the facts about target systems, repositories contain also feedback information such as target system assessments, findings reports related to representations, change requests, and defect reports.

Totem's repository is composed of several *software bases* (databases) and *files*. The software bases store objects, where each object contains several attributes. When considering Totem's software bases from an object oriented point of view, Totem's objects are abstract objects in the sense that they are composed of several attributes which in fact are concrete objects. Attributes are fine grained objects, they may be, among others, strings, plain texts, marked up texts, names, records (in the Pascal sense), file names and relations. File names allow the access and manipulation of coarse grained objects such as source and object code files and documentation files. Relations establish links between objects, where these links are grouped in accordance to their semantics. For example, an activity (object) may be *decomposed* into other activities (objects of the activity object class), an activity may also *precede* other activities, and an activity may *belong* to a project. Relations may link objects belonging to different databases. The schema of the repository is kept in the definition base. Specific representations are assembled from several attributes by means of *form programs*. The form programs are also stored in the definition base.

3. Environment requirements

In this section, we will state some of the requirements which should be satisfied by software engineering environments assembled with the Totem's support. We will examine solely the services provided by these software engineering environments, without examining how to satisfy these requirements. We will be using a narrative style instead of the requirement list style frequently found in requirements documents.

3.1 Abstract interface

Software engineering environments are systems in their own right. They must be specified, designed, developed and improved. They must be populated with tools and with development processes. Environment users must be trained in the proper use of these tools and processes. The environment's characteristics and performance must be continually evaluated and upgraded [PCCW93]. Software engineering environments must be maintained, i.e. evolved, adapted and corrected. They must be very reliable and continually available, since they are the basis for the development and maintenance of all target systems used by some enterprise. Finally, environments must be long lived, since they must provide at least maintenance support to all target systems which have been developed with their aid. A consequence of this longevity is the need to continually assure upwards compatibility, i.e. whenever the environment is changed for some reason, all repositories must be adapted in order to conform to the new environment version, where this adaptation should not provoke a change in the target systems themselves.

A software engineering environment is used simultaneously by several people, i.e. *environment users*. These users cooperate while developing several *target systems*, see figure 2. Totem assumes that each environment user has access to a workstation. These workstations are inter-connected by some local area network.

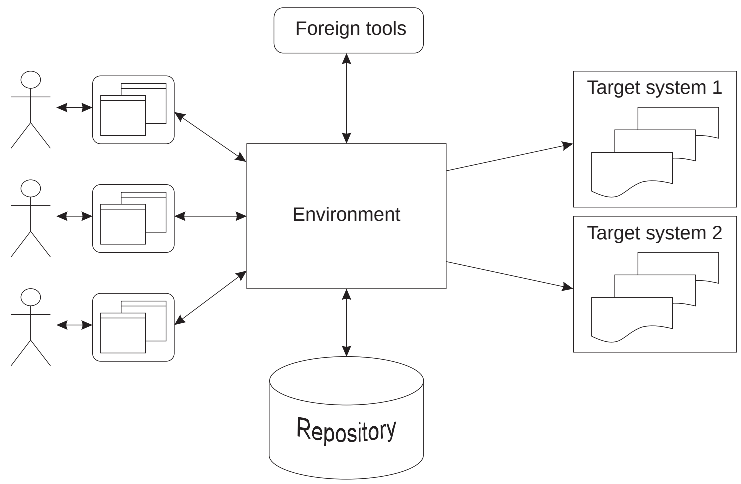

Figure 2. Abstract interfaces of a software engineering environment

The development of target systems is performed by populating and maintaining a repository. On the environment administrator's discretion, repositories may contain all facts about all target systems of the enterprise, or may contain facts about one or few of these target systems.

Environments are assembled from a selection of the processes, representation languages and tools contained in the meta-environment base. But these components may interact with tools developed by third parties. An example of such a tool is a compiler. One of Totem's aims is to provide means to create interoperating interfaces with existing tools. In particular, one of the major aims of Totem is to support development and maintenance right down to code and documentation. Thus, environment instances are capable of exporting source code to compilers, and source text to formatters.

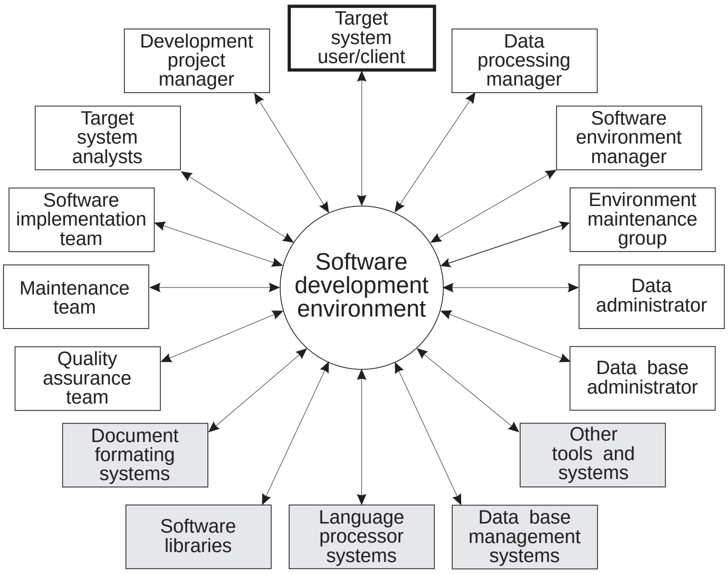

Figure 3. Typical environment user roles

3.2 Environment interface

In this section we will describe the interfaces and the means used to establish interfaces of environment instances.

3.2.1 Software engineering environment users

Totem supports a wide variety of *environment user roles*. In figure 3 we depict some of these user roles. Environment user roles may be enacted by people – *environment users* – trained or not in software development, or may be performed by tools (shaded boxes). Notice that a same person may enact several of these roles. In contrast, a given role may be enacted by several individuals. Since the roles depicted are fairly well known, we will not describe them in more depth here.

Environments must provide adequate support for the enactment of each user role. The goal of every environment user role is to contribute in a defined way to the development and maintenance of one or more target systems. Thus the environment must provide an adequate set of tools, representation languages and processes for each of the user roles.

User roles are interdependent. For example, database standards established by the database administrator must be strictly followed by at least the development teams, the maintenance teams and the quality assurance groups. Thus software environments must provide efficient and effective means to support communication among the different environment users. Much of this communication relies on some form of documentation, or *representations*. Even when informally discussing aspects of some target system, this discussion is usually performed while having several documents present. In order to establish communication among environment users, Totem allows the assembly of several different representations from the facts contained in the repository.

Environments depend on the software class being developed. Thus software environments must not only adapt to specific environment users, they must also adapt to the *target system classes* developed with their support. Consequently, it is virtually impossible to identify the set of all possible user roles, since new or slightly changed user roles could be required in order to develop new classes of software. The environment must then be adaptable to a variety of user roles and to persons enacting these roles, where the set of roles is not necessarily known when building the meta-environment. They should be known when assembling a specific environment with the aid of the meta-environment.

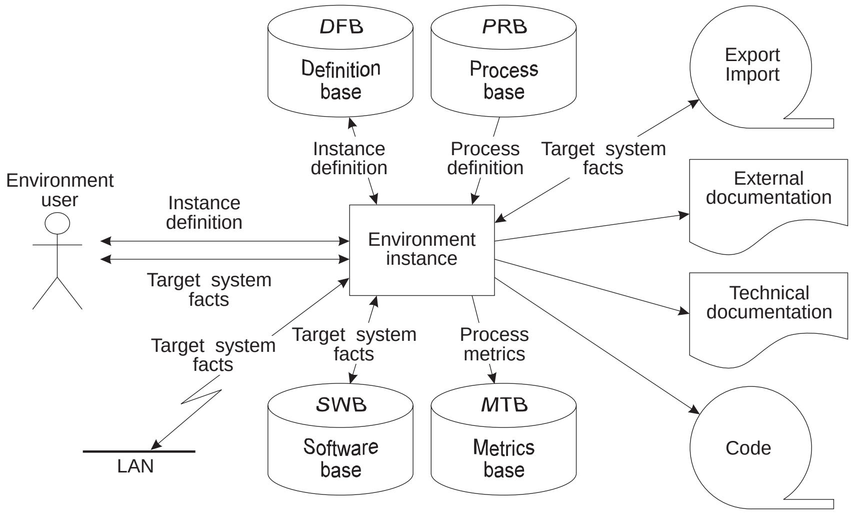

Figure 4. Interfaces of an environment instance

Given the difficulty of defining user roles a priori, Totem does not impose a set of predefined user roles. Rather it allows to create and improve roles, processes, representation languages and tools assuring a harmonious evolution of environments.

3.2.2 Environment instance interfaces

In this section we will describe the interfaces of an environment instance. As seen in section 2, Totem defines three layers:

- the *meta-environment* layer. In this layer processes, representation languages and tools are defined and improved.
- the *specific environment* layer. In this layer a selected set of the meta-components are assembled into a specific environment.
- the *environment instance* layer. In this layer a specific environment is partitioned into several environment instances.

An environment instance is a partition of the specific environment used to develop one or more components of a target system. Totem requires that each environment instance be defined to provide support to a given *development activity* or set of activities. An environment instance is assembled from a selection of the processes, representation languages and tools defined in the meta-environment. Environment instances are specifically tailored to satisfy the needs of each environment user (person). Each environment user performs one or more environment user roles. Figure 4 shows the typical interfaces of an environment instance.

Every environment instance is used by a single *environment user*. This person explores, adds, changes and deletes *target system facts* about the system being developed. These facts are kept in the local repository. Each instance allows some tools to be used and, consequently, only certain classes of target system facts may be changed. These access permissions are contained in the *definition base*. The definition and sequencing of activities, the definition of tools to be used while performing each activity, the definition of the work-products of each activity, and the quality requirements of the work-products are all kept in the *process base*.

Totem provides two forms of creating and maintaining definition and process bases. The first form follows the description presented in section 2. This first form requires updating the environment base and, consequently, can only be performed by the environment builder and the project manager. As seen in section 2, in order to transfer changes at the meta-level to the instance level, special tools must be used in order to assemble and partition the definition and process bases. These tools provide version and configuration control of the environment itself. This version control allows the evolution of the environment to have a minimal effect on the ongoing work. Totem allows work to progress using an old version of the environment, where the result of this work can be made to conform to a new version of the environment at discrete points, typically when stable points have been reached.

The second form of changing the definition and process bases allows the environment user to tune the environment instance to his particular needs. To achieve this, the environment user creates and modifies *instance definitions*. Changes of this sort are restricted to changes and additions to a limited set of form programs. These changes may no modify representation language schemas, or even software base schemas. They may only create or improve some of the tools.

While the environment instance is being used, several *process metrics* may be gathered, such as number of objects created, changed and deleted, usage of tools and error counts while using tools. Other metrics are extracted from the repository, such as number of objects per object class, existence and size of a selected class of attributes. All these metrics can be used to identify possible improvements of the processes and tools used in the environment instance.

Environment instances interact by means of some network. Typically, public repositories are shared over the network. Totem provides services which allow the disconnected use of an environment instance. This allows environment users to work even if they are not linked to the network. Such a situation occurs for example when environment users are allowed to work at home. This allows also to work with wide area networks where response time is not sufficient to assure acceptable performance of the tools.

3.2.3 Composition of environment instances

A specific software engineering environment is composed of several environment instances. See Figure 5. Each of these environment instances is used by some user and fits this user's needs. Each of the environment instances has its own definition and process bases defining its specific behavior. However, the set of all environment instances to be used while developing a given target system does not have to be always fully present. In other words, environment instances may be added, improved, or removed during the development process as a function of the current development stage. Adding, improving, and removing environment instances corresponds to defining or updating definition and process bases during the enactment of the development process.

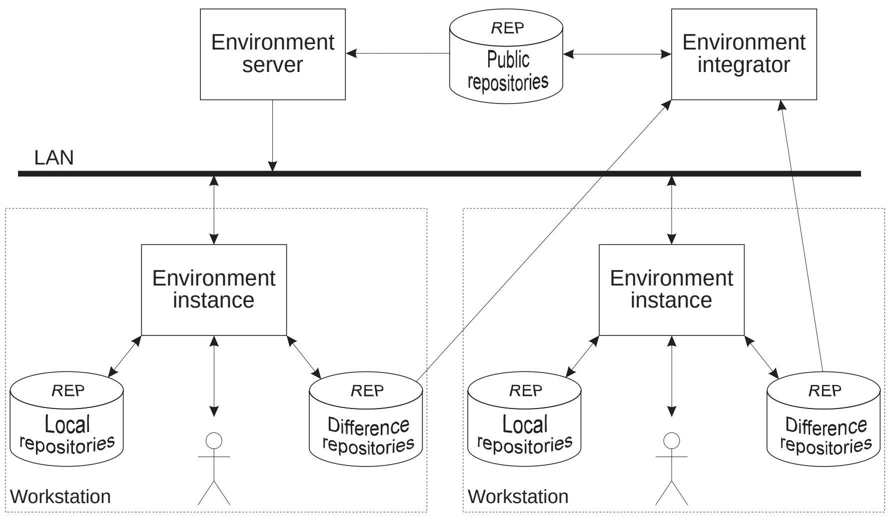

Figure 5. Composing an environment from several instances

Totem implements repositories as a distributed set of inter-linked software bases. Not always all team members work in a same place though. It is even possible that for some reason there is no communication link between different environment instances. Still development must be able to proceed. Totem allows work to proceed even if instances are temporarily disconnected, the observed effect being "attribute not accessible" messages placed wherever a non-accessible attribute should be displayed.

When editing a representation, public objects may have to be updated. For example, when establishing a reuse link, the reused object must be updated in order to identify all places where it is used. To allow this kind of updating, Totem keeps all changes to public objects in differential repositories. At given times these differential repositories must be integrated, i.e. checked-in, to the corresponding public repository. We will discuss this in more detail in a later section.

3.2.4 Representation languages

Every representation is written in some *representation language*. Each representation language defines:

1. *Aspect*, involving things such as shapes of graphic elements, style sheets, fonts, and margins of textual elements.
2. *User interface,* defining how to create or modify representations written in the language.
3. *Syntax*, defining object classes which can be manipulated and how objects of these classes may be combined.
4. *Semantics*, defining the meaning of syntactical constructs.
5. *Construction process*, defining how a specific representation can be built from a starting object using the content of the repository. This definition uses form programs kept in the definition base.

Totem groups representation languages into:

1. representation language families, and
2. representation language classes.

A *representation language family* defines a collection of strongly interrelated representation languages. For example, the **DataFlow** representation language family, contains among others the **DataFlowDiagram**, the **ProcessSpecification**, the **DataStoreSpecification** and the **DataStructureOrganization** representation languages.

Each *representation language class* is edited by a *meta-editor*. Each of these meta-editors provides a set of operations to the environment user. Each meta-editor handles all representation languages of the corresponding class. For example, the diagram meta-editor handles all representation languages based on diagrams; the structure meta-editor handles all representation languages using hierarchical structures; the form meta-editor handles all textual representation languages. Thus, all representation languages which are diagrams are edited by the same diagram meta-editor and correspond to the diagram class. This meta-editor is instantiated for a specific representation language by means of a environment user defined form program.

Each representation language family contains several *object classes*, see figure 6. Examples of such object classes are: "modules", "classes", "functions", etc. *Objects* of these classes will be created when writing a representation using this language. Examples of objects of the class "Function" are: "Create list" and "Destroy list". Objects in these classes may be interrelated, i.e. cross-referenced. For example, a "class" refers to all "functions" which it contains, and a "function" to all "classes" in which it is contained. Hence, the class "List handler" references the functions "Create list" and "Destroy list".

Each object class defines several *attributes* which its objects may contain. For example: an "Entity" class has an "Informal description attribute", a "Structural integrity specification attribute", and a "Declaration attribute". In some target system, an object of the "Entity" class, say "Student", could have the "Informal description attribute" filled out but not the "Structural integrity attribute". In Totem, these attributes are grouped into several *attribute categories*, such as "names", "strings", "plain text fragments"[2], "marked up text fragments", "source code text fragments", and "records". Each at-

[2] a text fragment is in fact a text of unbounded size.

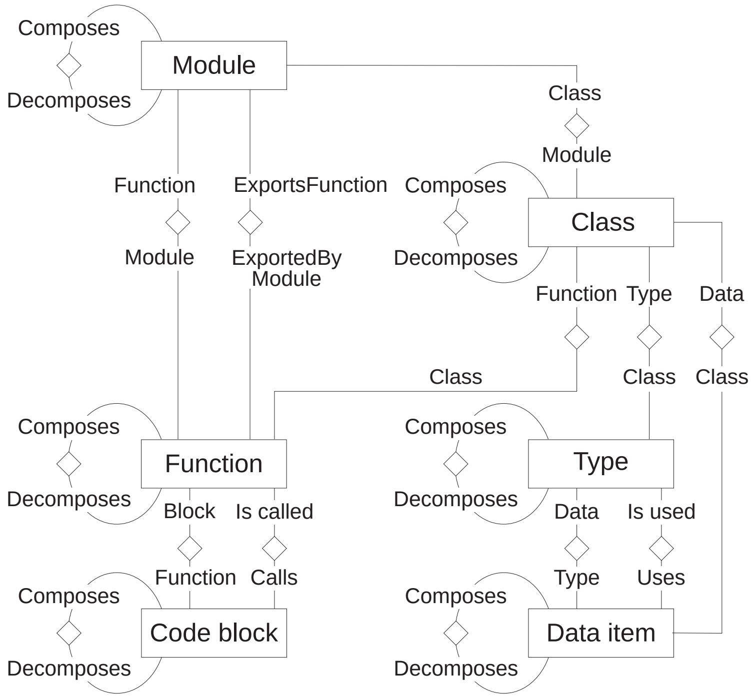

Figure 6. Example of a representation language family schema

tribute category defines its own set of repository access functions; however, all attribute categories provide a same set of access functions.

Each object class defines several *relations* to other object classes. Interrelated objects may belong to a same or to different representation languages. At the representation language level, relations define permissions to establish relationships between objects of the related object classes. There may be several relations between the same two object classes. Each of these relations has a well defined semantics. For example, consider a "work break down structure". Here an activity "decomposes" into several other activities, but it also "depends on the completion" of other activities. Relations may require specific information for each of its relationships. For example, consider a "function related to class" relation. Here it may be necessary to discriminate whether this function is "public" or "private" with regard to this class. However, the same function may be related to some other class, where this other class defines the visibility in a different way. Thus visibility is not a property of the function itself, but is a property of each of the "function related to class" relationships.

The object classes and the relations of a given representation language are defined by means of a schema such as shown in Figure 6. In this schema, entities correspond to object classes and labeled edges correspond to relations between object classes. Labeled edges are paired precisely, identifying each inverse relation. The representation model language is a representation language in its own right. Hence it can be defined as a representation language contained in the repertoire of the representation languages supported by the meta-environment.

The example shown in Figure 6 defines the basic composition of a module design and implementation representation language, geared towards object oriented implementation. For convenience reasons, Totem permits each representation language to be defined by its own diagram. Relations between object classes of different representation languages are defined in an other diagram. In fact, each of these diagrams defines a view of the definition base. The union of all these diagrams corresponds to the complete model of the definition base.

Valid attributes and other properties of the objects of each object class are defined in a data dictionary and are associated to the object class data models. The attributes are stored in the repository, however their existence is defined in the definition base.

```
BeginForm void "Edit module"( void )
   Title "Name of this module: " ;
   NoNewLine ;
   Name ;
   If Exists( [ Text ChangeRequestText ] )
   Then
      Title "Change requests" ;
      Text ChangeRequestText ;
   Else
   EndIf ;
   If Exists( [ Text ReportText ] )
   Then
      Title "Findings report" ;
      Text ReportText ;
   Else
   EndIf ;
   Title "Description of this module" ;
   Text TxtDescr ;
   Title "Functions exported by this module" ;
   Relation ExportsFunction ;
   Title "Functions contained by this module" ;
   Relation Functions ;
EndForm ;
```

Figure 7. Example of a specification form

In order to create and maintain attributes and relations, a specification form must be defined for each object class of the representation language. Each of these specification forms, when applied to an object of the corresponding object class, builds the specification representation of this object. Similarly, different tools are defined by means of form programs. Thus, for each of the object classes of a representation language, there must be several form programs, one for each of the actions which may be triggered by the environment user. Examples of such actions are *CreateObject*, *EditObjectSpecification*, *DeleteObject*, *ValidateObject*, *TransformObject(ToParm*) and *ShowDiagram*. Figures 7 and 8 illustrate a specification form and a small fragment of a validation tool.

3.2.5 Representations

When developing software, several documents – *representations* – are created, read, inspected and maintained. Examples of representations are: source code files, executable code files, requirement

```
BeginForm void "Validate module"( void )
   If Not Exists( [ Relation Function ] )
   Then
      Title "Module does not contain any function" ;
   Else
   End ;
   If Not Exists( [ Relation ExportsFunction ] )
   Then
      Title "Module does not export any function" ;
   Else
   End ;
   ForAll ExportsFunction Do
      If Not Exists( [ Text TxtPreCond ] )
      Then
         Title "Function does not define a pre-condition" ;
         Name ;
      Else
      End ;
   End ;
EndForm
```

Figure 8. Example of a validation program

specification documents, design documents, help files, target system user documents, data element specifications, function pre- and post-conditions, diagrams, and function body structures. Welsh and Han [Welsh94] even depart from the premise that software development is a document-based process. These composite concrete representations usually consist of a number of basic representations such as a paragraph, a fragment[3] of program code, a diagram, a specification and the relationships among them. The basic representations are written in some language such as English, entity relationship diagrams, Z, or C++. Representations appear in some machine readable form, on paper or on screens, and a basic goal of a representation is to serve as a means of communication between the different environment users, regardless of whether they are persons or software systems.

Representations may vary from the large and complex as exemplified by a user manual to a very small and simple one such as a data element specification. Coarse grained project objects or complex representations are usually composed from a large quantity of facts, whereas simple representations, or small grained project objects, contain a few facts. Totem stores small grained objects in the form of object attributes in its repositories. In order to display a representation, i.e. a coarse grained object, form programs access the repository reading all attributes which compose the representation. Totem reconstructs representations whenever they are accessed. In Totem, a representation is defined by following tuple:

$< RLF , RLC , RS , OBJ >$ where:

RLF identifies the representation language family to be used.

RLC identifies the representation language class to be used.

RS identifies the rule set to be used. There is always one default rule set. However, there may be more than one rule set, leaving to the environment user the choice of which rule set to use.

OBJ identifies the starting object from which the representation will be built.

As an example consider the editing of a data flow diagram and its corresponding data dictionary. In this case the representation language family is **DataFlow**. When editing a data flow diagram the language class is **Diagram**. When editing data dictionary specifications the language class is **Form**. When editing a data flow diagram using bubble charts, the rule set selected by the environment builder may be **BubbleChart**. The rule set is defined by means of a collection of form programs, containing forms for each user action and object class defined in the representation language.

Assume now that the user wants to see the data flow diagram corresponding to an object ***A-DFD***. Let us assume that ***A-DFD*** is a process which is specified by means of a data flow diagram. In order to display the diagram the user triggers the action *ShowDiagram* with object ***A-DFD*** selected. Usually the context identifies the language family **DataFlow**, when not, the user must decide which language family is to be used. The language class is **Diagram**, as a consequence of the action triggered by the user. Both language family and class are used to select the rule set. Usually there will be just one rule set. If there are other rule sets available, the user is prompted for a choice. Let us assume that the rule set is **BubbleChart**. From this rule set, the rule **ProcessDiagram** is selected. Now the repository will be accessed retrieving the facts related to the diagram ***A-DFD***. These facts are stored in the workspace and, afterwards, displayed in an appropriate window. A *form program* defines the contents and format of this presentation. By means of hotkeys or binding operations the selections can be skipped.

If now the user chooses to see and edit the specification of some process ***A-PR*** contained in the displayed diagram, he points to the object in the diagram and triggers the action *ShowSpecification*. Now the identifiers **DataFlow** and **Form** are used to select the rule set **BubbleSpec** and from this, to select the rule **ProcessSpecification**. Using this rule the repository will be accessed retrieving all facts related to the specification of ***A-PR***. These facts are stored in a workspace too, different form the one

[3] code text fragments are for example: a conditional expression used in an if statement, a list of data declarators, a list of executable code lines and function header declarations

containing the diagram, and then are displayed in an appropriate window. Another form program defines the contents and format of this representation. The user can now edit this specification. Once he closes the form window or saves the changes, all changes recorded in the workspace are stored in the repository and a *RepositoryModified* message is issued. This message triggers the refreshing of all open windows.

Totem does not keep representations in its repositories, instead whenever necessary it reconstructs representations from some starting object applying to it a rule set belonging to a representation language class and family. This architecture assures that representations always exactly reflect the content of the repository.

Representations may be composed of several other representations, each of which is written in a different representation language. For example, a typical system design representation contains explanatory text, data models, functional models, user interface models and data dictionaries. In Totem, all representations, composite or not, are built by means of form programs. These programs are developed by the environment builder and are stored in the environment base. Later on they are compiled and stored in a definition base. Totem defines special kinds of relations where each relationship is a tuple in the form <*O*, *RLFC, RS*>, where *O* is a target object contained in the repository and *RLFC* is a language class of a given representation language family and *RS* is a rule set. When displaying the composite representation, each tuple will generate the representation build from object *O* and written in representation language *RLFC* according to the rule set *RS*. These tuples may be generated by means of form programs.

Representations contain *cross references* linking facts in one representation to facts in another. For example, the definition of a data structure should be cross referenced to all representations which make use of this data structure. In general these cross references define some semantic interdependency. Building these cross references by hand is error prone (omissions occur frequently) and tedious. Thus some assistance should be provided to at least partially generate the set of cross-references by automatic means.

Totem does not interrelate representations, instead it interrelates objects. Given a target object contained in a relation and, if desired the target language family class and rule set, the target representation will be composed. The composition is defined by means of programs written in Totem's internal language. This architecture supports a wide range of navigation facilities using explicit links, i.e. relations, or implicit links, i.e. some rule used to select or find a target representation. Since the construction rule applies to all documents regardless of their size and complexity, this architecture allows to navigate also between quite complex documents. Finally, when editing relations, all navigation operations immediately adapt to the new set of relationships.

Totem creates or updates relations as a response to a user action, an explicit editing or an assignment in some program. For example, when a user creates a data flow diagram for some process instance contained in another diagram, several relations are created, as for instance the binding of the process instance to the newly created diagram and the binding of the corresponding process specification to this diagram. When editing a form, a user may insert new elements in a relation either creating a new object or selecting from already existing objects. Finally, when the user activates a transformation, this transformation may create or update several relations.

The *Show* operation together with the identified parameters and relations permit hyper-text-like navigation through the contents of the repository [Bigelow88, Conklin87]. Thus, in any given representation belonging to any specific language class, it is possible to select some displayed object name and ask to show another representation for which this object is the starting object. Any representation which is shown may be edited, unless editing is explicitly prevented.

Different from conventional hyper-text systems though [Conklin87], in Totem links and frames have types, where these types convey some semantic information. For example, when examining a specific requirement (a fact) in a requirements definition (a frame), one might want to see how this requirement influenced the system specification. Thus, one may want to see all the entities (facts) in the set of data models (frames) which reify this requirement. In an other occasion we might want to see all user interfaces (frames) which reify this same requirement. The desired semantics is selected by the representation language family and class.

Cross referenced representations must be *consistent*. That is, the information content of one representation must not conflict with the information content of the other representation linked by a cross-reference. In order to mechanically reduce the chance of inconsistencies Totem uses transformers and validators. Given the facts of an antecedent representation, a transformer creates or updates facts of a corresponding consequent representation. Given the facts of some representation, a validator verifies whether they are consistent with respect to other facts contained in the repository. Any discrepancy will be noted and stored together with the other attributes of the offending object. We will examine transformers and validators later.

Representations are not generated just for the purpose of documenting aspects of the target system. They are an *active part* of the development process, guiding this very process. One of the goals of Totem is to compose code from specification and design documents. Thus in Totem, representations are generated and maintained as part of the process which culminates with an operational target system. For example, when using a state transition diagram to specify some user interface, one wants to be able to systematically transform this diagram into code structure skeletons. After this transformation, these skeletons are filled with code fragments. Then the filled skeletons are linearized (flattened) into compilable sequential code. When performing changes in the diagram, all derived elements which are not related to the changed portion remain as they are and still allow to correctly compose the corresponding source code.

Totem users may perform following operations on representations:

1- *addition*: This occurs when a representation is written from scratch, when an existing representation is enlarged, when a generated skeleton of a representation is filled with information, or when explicitly cross referencing several existing representations.

2- *extraction*: This occurs when a representation is presented to the user, when a hard-copy is generated, or when a file is generated to be used by some foreign tool. In order to be a proper extraction, the user must understand what he is reading in such a way that the acquired knowledge corresponds to the writer's intent. Thus representation languages must be well designed so that they convey the writer's intent in an adequate way.

3- *modification*: This occurs when a representation is updated. Typically, a modification of an already accepted fact leads to a new version of this fact. However, modifications of facts which have already been modified and not yet accepted, will not lead to a new version of this fact.

4- *propagation*: This occurs when a consequent representation is generated or changed in conformance to a given antecedent representation. A consequent representation is a representation which, in accordance to the normal flow of work, is to be created after the antecedent representation.

5- *reflection*: This occurs when an antecedent representation is built or adapted to conform to a given consequent representation. Reflections are typically performed during reverse engineering activities. Propagations and reflections are *transformations*. The first operates in the normal direction in which development proceeds. The other operates in the inverse direction.

6- *search*: This occurs when the environment user examines the repository in order to find a set of representations or objects satisfying some search criteria.

7- *reuse*: This occurs when the environment user incorporates an already existing representation as a consequent of some representation being built or modified. Totem promotes reuse through linking representations to reused representations.

8- *composition*: This occurs when several representations, elementary or not, are combined to form a composite representation. Similarly, *decomposition* occurs when disassembling an elementary representation into several other elementary representations. In Totem, compositions are defined by special classes of relations.

3.2.6 Reuse

Reuse is one of the most effective ways to reduce development effort. Totem provides two mechanisms for sharing facts:

1. representation languages share object classes.
2. representations are built sharing facts while navigating over relations.

The first mechanism allows different representation languages to share object classes. Consequently objects of these shared classes will appear in different representations each of which composed with one of these languages. For example, assume that we are specifying a system using entity relationship diagrams (ERD) and data flow diagrams (DFD). Both the ERD representation language and the DFD language share the class **DataStructure**. Thus a given entity, defined in the ERD language, may contain data structures which in turn are contained in several data flows and data stores, defined in the DFD language.

The second reuse mechanism is based on relations between objects. The related objects may belong to object classes which are used in different representation languages. Such relations permit form programs to navigate to objects belonging to any object classes even those which do not belong to the set of object classes of the initial representation language. Continuing the same example, it is possible to establish a relation between a data store in a DFD and an entity in a ERD. Now when building the specification of the data store, it is possible to navigate to the ERD and extract relevant information to be displayed in the data store specification.

These two mechanisms, together with the ability to share repositories and partially locking the contents of repositories against change, provide means to reuse standard specifications and designs. For example, a corporation could have defined a single corporate data schema. This data schema can now be used to build several different applications. Since the data schema is protected, no application developer can accidentally or willfully change it. Furthermore, if the data administrator changes the data schema, all applications bound to it will be notified.

Different representations may have non empty intersections. This redundancy is a positive property since it increases the ability to control consistency among different representations, and it also makes it easier for readers to understand a complex system. This redundancy is also necessary since different readers want to see the same information from different points of view. For example, the programmer of a function wants to see its source code. The user of this function wants to see the interface specification, the corresponding definition code, and some examples of its use. The user of an application containing this function wants to see help information relative to its specific operational aspects.

In Totem, representations share facts as a direct consequence of the way representations are build from the contents of the repository. The repository stores fine-grained objects such as symbolic names, strings, text fragments and relation lists. Thus, when a fact is changed in a representation, all other representations sharing this fact are automatically changed, in the sense that they will now be assembled using the fact's new value. Sharing facts between representations eliminates repetitive writing effort, since different representations can be generated from a repository containing these facts. For example, consider the development of a function library. There are at least five documents, the source code, the interface definition code (definition modules), the test data files, the help files, and the hard copy user manual. All these documents share a large amount of information. Generating all these documents from a common base virtually eliminates the inconsistencies which could build up whenever some maintenance is performed. In order create and maintain interrelated representations, Totem:

- allows facts to be acquired and updated in any of the representations,
- provides validators to control and propagate changes and
- provides fine-grained version and configuration control.

Obviously Totem supports also the traditional copy and paste kind of reuse. One interesting form of copy and paste can be achieved by means of links together with version control. In this reuse mode, a link is established between an object and some other which is the root of a more elaborate structure. Changing parts in this structure will generate new versions of the changed attributes. When navigating over the links the specific version will be defined. Thus, while composing a representation, only those attributes will be selected which are version correct with respect to the original object. This mechanism allows to reconstruct a representation which is a slight change of another one, by keeping in common all attributes and differentiating between them by means of a version tag.

3.2.7 Integration with foreign tools

In Figure 2, we have shown the interfaces of a software engineering environment. This environment supports cooperative software development of several target systems. In such an environment each environment user interacts with the environment using software tools available on a personal work station. Most of these tools are built using the meta-environment. Other tools, *foreign tools*, are provided by third parties. Typical examples of foreign tools are: compilers, linkers, database management systems, text formatters, possibly even specialized representation language processors.

All tools built with the meta-environment interact with each other using the repository. Thus, the repository establishes an interface standard between all internal tools. It is unreasonable though to expect that all foreign tools will adhere to a specific set of standards, in particular to the environment instance interface standard.

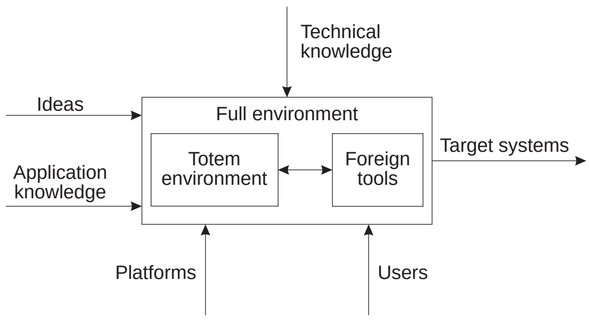

Figure 9. Composition and interfaces of a complete software engineering environment.

Figure 9 illustrates the organization of a full environment composed of a Totem environment interoperating with foreign tools.

Foreign tools may interface with the environment by means of sequential files. These files typically adhere to a syntax dictated by the foreign tool. Examples of exported files are source code files directed towards some language processor, and target system source documentation files directed towards some text formatter. Totem provides lexical and syntax analysis tools in order to help building interactive interfaces. These interfaces are implemented by means of form programs and may be incorporated in environment instances. As an example consider the case of cross code generation and compiling. Here the environment is used on a given development platform to produce code which will run on some different target platform. Typically the language processors and test support tools will reside on the target platform. In many cases it is cumbersome to continually transfer files between different platforms. Due to this, programmers will tend to compile and test programs and, possibly change the program, in the target platform. However, any changes to the generated programs made within the target platform will make the program inconsistent with the repository. Thus, unless there is some mechanism to recover changes made outside of the environment and insert them back into the repository, the user may feel that the environment hampers effective development.

Foreign tools may also interact by means of *import/export files*. These files are sequential files having a well defined structure in accordance to a standard established by Totem. They may be used to directly update the contents of the repository. This file format could be used by foreign tools to directly update repositories. It may also be used to interchange target system data among disconnected environment instances.

Since Totem will work in a windowing environment much of the integration between tools will be achieved through the communication between windows. For example several of the common services,

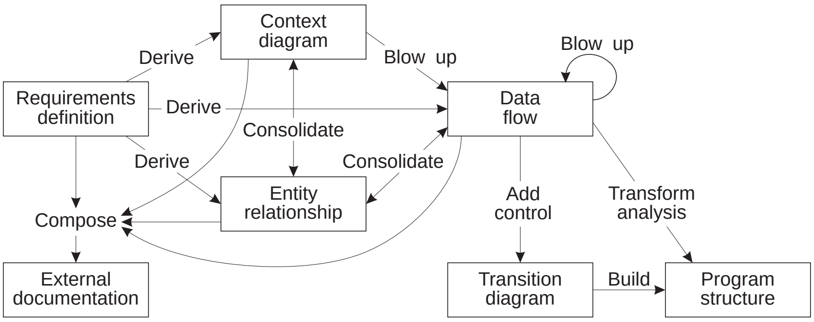

Figure 10. A partial language transformation graph

such as electronic mail, will be performed by available tools. Totem will provide interfaces allowing integration with such tools. For example, copy and paste or export and import over the electronic mail system would be one of these interfaces.

3.3 Software development process

3.3.1 Software development activities

In Totem, developing a system corresponds to *populating* and *updating* the repository. The order in which the repository is populated is development process independent. Of course, when following a well-defined development process, the amount of possible inconsistencies among representations will be small and can be removed soon after a representation has been built or changed. Consequently fewer inconsistencies will be detected when performing quality control, less corrective effort will be necessary in order to assure consistency among a collection of representations. Furthermore, fewer defects will remain in the target system.

The set of available representation languages can be organized as a labeled directed graph. See Figure 10. In this graph representation languages are the nodes and edges indicate the existence of a transformation rule from a *preceding representation language* to a *succeeding representation language*. The label of the edge identifies the transformation to be performed. This transformation is applied when transforming a representation written in the preceding language into another representation written in the succeeding language. Edges may be hyper-edges, indicating that the transformation uses or produces representations in several representation languages, for example the “Compose” transformation in figure 10.

The *language transformation graph* defines a development meta-process. In particular it defines the pre- and post-conditions, i.e. representations in a given representation language, which specify each transformation. There may be more than one transformation from a given representation language to another one. Any specific process using these representation languages and transformations must obey the constraints established by the language transformation graph. Given a language transformation graph, a specific development process schema can be derived. This process schema states the set of technical activities and defines the enactment order and control conditions. For each of the activities the input and output representation languages are defined and the activity is bound to a set of transformations consistent with these representation languages. While developing software, activities are performed which are not directly mapped onto transformations, for example management and quality control activities. These activities must be added to the process schema, leading to a complete process definition. This process may now be instantiated, resulting in a specific development plan. While enacting, performance data can be gathered and stored in the metrics base, thus provide information for future plans.

In one extreme, a transformation would be fully automatic — a *total transformation*. In this case there is a program or a set of rules which performs the transformation. For example, a compiler is a total transformation. In the other extreme, transformations might require intensive human participation — a *manual transformation*. In this case there is only some description telling people how to perform the transformation. For example, when transforming textual requirement definitions into system specifications, usually a manual transformation is performed. Most transformations fall somewhere in between these two extremes — a *partial transformation*. In these cases, part of the transformation can be done automatically, while the rest requires human intervention.

When transforming a representation written in some language to another representation, several cross references among these representations are established. These cross references define the information dependencies of facts contained in the two representations. Clearly, the closer a partial transformation approaches a total transformation, the more these cross-references will be established automatically.

Only total transformations are not affected by maintenance. The other transformations are affected, because the derived representation is modified by the environment user after having been generated by the transformation. In Totem, this problem is reduced to point-wise propagation. This is due to the use of very small-grained objects which are strongly interrelated. Thus, when modifying a given representation, all existing relations to objects contained in a derived representation are kept, except, of course, for those affected by the modification. When the derived representation is built its contents are mostly unchanged. In general only parts directly related to the changed portion of the origin representation must be modified.

Several *process definition languages* have been proposed. In [Derniame92], several papers describe different process definition languages. All these languages can be viewed as some form of representation language [Osterweil87]. Totem can be configured in such a way as to provide support to a process definition language. Typically such a language defines the set of required representations and acceptance conditions needed in order to initiate some task. Examples of acceptance conditions are "accepted", "contains errors" and "in progress". Each task defines the tools which may be used, the acceptance criteria of the resulting representations, the input representation with their acceptance conditions, and the resulting representations. The input and output representations are typically documents built from several small representations. This reduces the amount of detail to be provided when defining input and output conditions. It also assures that all documents are completely defined in the repository.

Process enactment can be controlled by keeping information about the *enactment state* in the repository. An enactment state is defined in terms of existing representations and their acceptance conditions. Access to tools and to portions of the repository will be denied if, according to the plan, the current enactment state does not allow this activity to be performed by this environment user.

3.3.2 Quality assurance

After creating or modifying a representation, its quality must be controlled. Whether performed by tools or done manually, quality control will generate a *findings text*. This text contains error and warning messages relative to the controlled representation. In Totem, findings texts are associated with objects. Thus, when controlling a representation, findings will be associated to the objects contained in this representation. The findings text of the representation corresponds to the set of findings texts of the objects contained in this representation.

Several causes may lead to defects found in a representation. Eventually, every defective representation must be modified in order to eliminate all causes of error messages and most of the causes of warning messages. In Figure 11, we show the set of actions and their interfaces which are performed when developing representations.

When developing and when controlling the quality of a consequent representation, it is possible that defects are uncovered in some of its antecedent representations. For example, one may find that a method is missing when implementing a class. It is unreasonable to enforce a zero specification defect rule, such as "all implementations must be strictly consistent with their original specifications", since all sorts of defects may have occurred when writing the specification. Enforcing that specifications be corrected immediately whenever a defect is uncovered, may hamper the development progress in some projects. On the other hand, letting consistency assurance go unenforced is a big risk in other projects. Finally, letting a specification be changed by the person implementing from it is certainly a

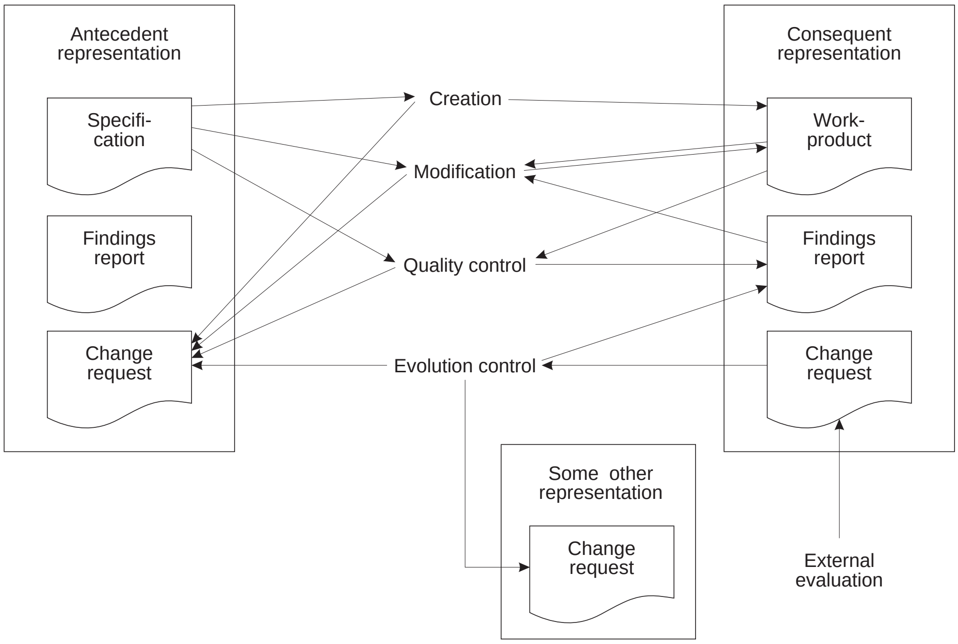

Figure 11. Representation development activities

source of trouble. Thus, due to the lack of uniformity when treating inconsistencies between specifications and their consequent representations, Totem provides quality enforcement policies which are adjustable to different projects and progress states in these projects.

In Totem, antecedent and consequent representations may be under development at the same time. For example, when developing data flows, processes and diagrams may be specified in a concurrent way. In a data flow, the diagram specifies the interface of each process. However, when detailing the specification of a process, one may find missing or excessive data in the interface of this process. These defects are removed by changing the data flow diagram. Thus it is unreasonable to expect that all antecedent specifications are necessarily accepted versions of some specification.

Totem allows facts — attributes of an object — to be *frozen*. Once a fact is frozen it may only be changed if it is unfrozen. Freeze and unfreeze are configuration control operations available to the project manager. If a defect report is directed to a frozen attribute it will be kept as a change request, otherwise it will be kept as a findings report. In order to achieve this, Totem allows the existence of several findings texts and change request texts associated with an object.

Changing an accepted specification may impact several simultaneously ongoing activities. Thus, specification changes may have to be postponed until some better opportunity arises, even if these changes are due to corrections. Thus, defects found in antecedent representations may have to be kept separate from conventional findings reports. These error messages will be kept as *change requests* of the antecedent representation. Change requests are also associated to objects. The set of change requests which affect a given representation can be established navigating through the representation and listing all change requests found while visiting a given object. While a given activity is being performed, or when getting more insight about the problem to be solved, problems or suggestions may be uncovered. All of these lead to external change requests, which are added to the change request lists related to objects contained in some representation.

Creating and updating representations may take considerable time. During this time, obviously, the representation will contain several defects. Continually applying quality control tends to be more of a nuisance than an effective aid, since defects will be reported for parts of the representation which the environment user knows are wrong. Thus, quality control should be applied only at the environment user's discretion. On the other hand, the environment user may find it helpful to obtain findings reports on incomplete representations. These reports will help him to complete and correct representations he is developing. In order to contain little noise, the quality control rules should be adaptable to the level of completion the environment user selects. However, full acceptance will only be granted if the representation satisfies quality levels defined in the development process.

Totem defines 3 levels of validation rules performed by tools:

- The first level contains rules which are applied before committing the result of an individual editing action to the workspace. Rules of this level may validate statements such as "data stores may not be linked to other data stores".
- The second level contains rules which are applied before committing the workspace to the repository. Rules of this level may validate statements such as "text fragment must satisfy syntax X".
- The third level contains rules which are applied to part or all of the content of the repository. Rules of this level may validate statements such as "an entity must define at least one key data item". Rules belonging to the third level generate finding reports and change requests.

The first two levels are handled while editing. They assure a minimal degree of semantic integrity of the repository content. This degree depends on the control rules. Lax rules lead to few faults detected while editing and a large volume of possible faults contained in the repository. Strict rules lead to a large amount of faults detected while editing, and a smaller amount of faults contained in the repository. However, validation rules at the editor level cannot be as strict as desired, since rules may require facts which are not necessarily available when their evaluation is triggered.

The third level examines strictly the contents of the repository. Part of the repository will be explored in accordance to some verification and validation rules. All defects will generate reports which will become change requests if the corresponding fact is already frozen, otherwise they will become findings reports.

In addition to verification and validation performed by tools, representations will undergo inspections performed by humans. The recording of the defects and suspicions found during inspections follows exactly the same rules as the recording of defects found by means of tools. Inspections may be aided by tools, for example check lists. However, the finding is reported by human action. To simplify reporting, Totem permits the linking of the report generators with inspection support tools.

The rigor of assured quality may evolve in time as the development progresses from a very abstract description of the target system to a more concrete description. For example, when designing a module, the functions which compose this module are specified. In the beginning, only the input and output data must be specified for each of these functions. Later on, decisions are made determining which of the data items are passed by parameters, which are global, and what their code names are. Finally, a code body is built for each of these functions. Obviously, each of these three states are at different levels of completion and must satisfy different quality rules.

At given times, pending change requests will be analyzed. This analysis, *evolution control*, will select some change requests to be implemented. This corresponds to moving messages contained in the change request to the corresponding findings report. Since change requests may have originated outside of the development group, they may have been attached to the wrong object. Evolution control will move these improperly filed requests to other objects. Once a change request has become an entry in the findings report of some object, representations containing this object will be modified while performing some evolution task.

Once a modified representation is accepted, all of its consequent representations must have their quality checked again. This quality control may generate new findings reports and change requests for several of the objects contained in this representation. In this way, modifications of some representation are propagated to all of its consequent representations, from these to their consequent representations, and so on. While propagating changes to other representations, defects in already accepted representations may be uncovered. These defects will lead to change requests. Typically, when changing something in one representation, other representations must be changed too in order to recover consistency. Thus a change provokes a *change ripple* which traverses back and forth the set of representations, possibly repeatedly returning to a same representation. Eventually this ripple must ebb out, that is, all representations must be mutually consistent, otherwise a stable set of representations cannot be established, which in turn means that no stable system can be developed. That does not mean that all of the causes leading to change requests and to findings reports have completely been eliminated. It may be acceptable for a given target system that pending change requests are kept for several of its components.

Totem's development paradigm assumes that representations cannot be correctly written from the onset. Rather they must *converge to correctness*, where correctness is defined by means of quality requirements and evaluation of the representation. Defects uncovered by these rules stimulate feedback leading to changes in the representation set. Obviously, if representations are always written in order to conform to the defined quality control rules, little effort will be spent processing change ripples. It should be noted that many quality control operations correspond to statistical filters. That is, only part of the defects contained in a representation will be uncovered by the quality control mechanisms. Thus, considering two representations, one containing many defects and the other few, after correcting these representations, the former will still contain more defects than the latter.

Once a representation has been found to be of an acceptable quality, a new *version* of may be established. In Totem, a new permanent version is created only after the representation is accepted. While the change is being developed, a temporary version will be under development. After accepting a temporary version a permanent version is built from it. When building a permanent version, only modified attributes will be stored. Later on, when accessing a given version of an attribute, the derivation path of this specific version is used in order to retrieve the most recent permanent version of this attribute.

As mentioned earlier, the facts corresponding to the accepted version will be frozen. Since an object may belong to different representations, where these representations correspond to successive development phases, it is possible in Totem that an object contains frozen facts and non frozen facts. When defining a representation language, the facts which are germane to this language are identified. These facts will be frozen when a representation corresponding to this language is accepted, or checked in. They will be unfrozen when this representation is checked out.

When a changing a representation, other representations may become inconsistent with this new version, although they are consistent with some older version of it. In other words, for each representation A related to a representation B, *version correctness* must be stated. This means that the relations between both representations must be specialized, indicating which version subset of representation A may be properly combined with a given version subset of representation B. This is done by *configuration control*. In Totem, the version information is an inextricable part of any given representation. Hence, version and configuration control cannot be performed adequately by foreign tools, they must be part of Totem's repository management system.

We may conclude then that a representation is composed of three parts:

1. the *representation proper*, containing the versioned facts of the target system. These facts may be shown to the environment user by means of representations written in some representation language.
2. the *findings report*, stating all defects found among the facts of this representation. The findings are stored as attributes of objects which are accessed by the representation.
3. the *change requests*, stating all defects found among frozen facts used by this representation, or identified by external observations.

3.3.3 Application generators and transformers

A transformer propagates or reflects information contained in some representation to some other related representation. Transformers are defined in terms of the representation languages used by the involved representations. For a given pair of representation languages there may be several transformers, of which one will be selected. See figure 10.

Application generators are a special class of transformers. Application generators have been proposed as specification driven prototype generators [Balzer81]. Once a prototype has been accepted, the target system is developed from the corresponding accepted specification. Application generators produce source code or tagged text based on the contents of the repository. Typically, an application generator produces applications directly from specifications containing little, if at all, implementation information. For example, application generators could be used to generate code skeletons directly from state transition diagrams. Associating a few source code fragments to states, transitions and labels (processes), it is possible to generate full operational graphical user interface [Hübscher95]. Changing the diagram may entail a significant change to the user observed interface, in general requiring almost no changes to the generated code.

Application generators eliminate several development steps. Usually a fair amount of source code is generated from few specifications. However, application generators are of pragmatical value only if they generate source code which needs not to be manually changed or adapted after having been generated. Otherwise, whenever maintenance is performed, all these changes must be performed again. In addition to being tedious, this is also error prone.

Application composers are special kinds of application generators. Typically, they organize the contents of the repository producing adequate input to some foreign tool. For example, consider a structure editor which is capable of editing a structure diagram and linking text fragments to this diagram. An application composer would traverse the structure and output the source code text fragments in the appropriate order required by a compiler.

Transformers must be capable of operating with several selected versions. For example, when changing the schema of a database, the database has to be regenerated. In order to perform the regeneration, some database management systems write the database contents out to a sequential file, and later read in part of this file, thus rebuilding the data base. The definition of the file to be written is based on the definition of the previous version of the data model. The definition of the file to be read depends on the previous and on the current data models. The elapsed time between starting the development of the new version and accepting it may be quite long. Not only this, during the development of the new version, partially populated databases may be used in order to test the new or upgraded application, where this application may have been generated from the new schema. It is necessary then that application generators be capable of composing code which takes into account the differences between two versions.

3.3.4 Reverse engineering and re-engineering

Most organizations already own and use a large amount of software. Independently of how this software has been built and independently of specific application user satisfaction, this software is usually a valuable asset of the organization and cannot simply be removed or substituted by another without causing major disruption.

Considering this background, it follows immediately that from a pragmatic point of view, any computer aided environment will succeed only if it is capable of absorbing already existing code, designs and specifications, independently of how they have been built. Thus pragmatic environments must be capable of supporting *reverse engineering*.

In its most simple form reverse engineering attempts at recreating engineering documentation from already existing code. For example, a code composer generates code from a repository using structure diagrams decorated with code fragments. These structure diagrams correspond essentially to parse trees. What the composer does is to flatten this parse tree, generating sequential source code. Obviously this process can be reversed. Given a program:

1. reorganize it removing all structure obscuring gotos,
2. parse it building the parse tree
3. create the repository so that it reflects this parse tree.

Composing the program from this parse tree yields exactly the same program. However, several of the names of objects contained in the repository will have been generated and thus lack semantic meaning. Furthermore, most of the specifications will be missing or not up to date.

Once the program is contained in the repository, the environment user changes the contents of the repository, assuring that it corresponds to an acceptable design. In so doing, the environment user may wish not only to change names and add specifications, but also modify the program's structure, cleaning up badly structured code. This is a form of *re-engineering*. As a result the repository will contain a well organized and documented program structure, from which an enhanced target system code can be generated. Guedes [Guedes93] describes a tool-supported process which recreates the repository containing the module design from source code. This re-engineering process can be performed selectively on some portions of the whole program. The portions which have not been affected by re-engineering will generate exactly the same code, with the same quality as contained in the original target system. The portions which have undergone re-engineering, will perform differently and, hopefully, better than the original target system.

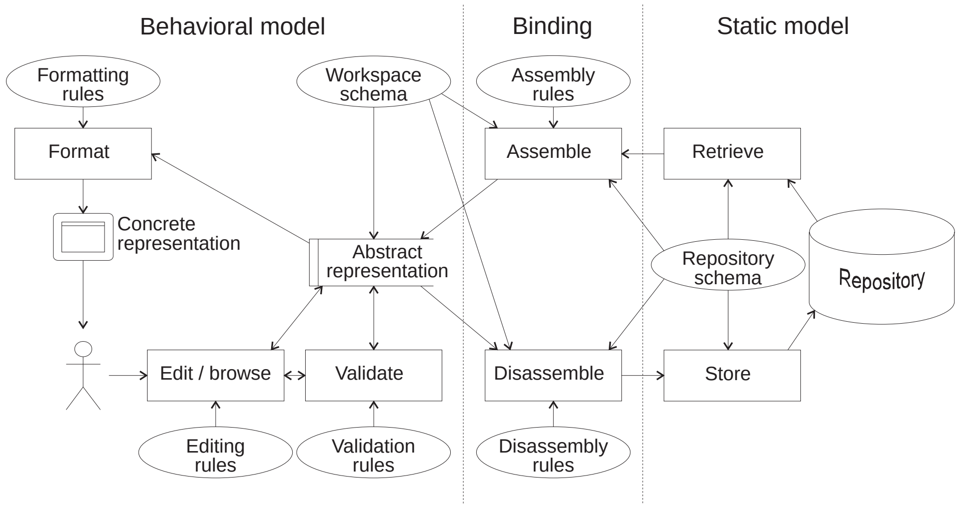

Figure 12. Meta-editor architecture

4. The architecture of Totem

In this section, we will describe Totem's high-level architecture. This architecture is process, representation language and tool independent. Data contained in the definition and process base define the specific behavior required in order to implement an environment instance. This organization allows the construction of virtually any environment, without needing to reprogram any of Totem's components.

4.1 Meta-editors

Each concrete representation is edited by some specialized editor. This editor is specialized with respect to the representation language class, such as diagram or specification text, and with respect to specific representation language families, such as data flow diagrams and entity relationship diagrams. Totem provides meta-editors for each of the representation language classes. All these meta-editors use the same generic architecture shown in Figure 12.

The generic architecture consists of a *behavioral model*, a *static model* and a *binding* between them. This architecture is close to that of abstract data views [CL95]. The behavioral model contains both concrete and abstract representations. The user views a concrete representation which is produced from an abstract representation through a set of formatting rules. During editing or validation the user interacts with the concrete representation while changing or validating the underlying abstract representation. In a formatted text editor, the abstract representation might consist of text fragments, code or specifications tagged with a language such as the Standard Generalized Markup Language — SGML [Goldfarb90] —, which are then formatted into a concrete representation using a formatter driven by some style sheet.

The repository schema provides a definition of the organization of the data elements in the repository. The data elements are stored in or retrieved from the repository using this schema. The binding assembles the chunks of data retrieved from the repository into abstract representations of the documents to be presented to the user, or disassembles these abstract documents for subsequent storage. As will be seen later, the repository schema is composed of two parts. One part defines the storage schema and is fixed. The storage schema is used by the repository accessing transactions. The other part defines the semantics of the repository contents. This schema is variable and is kept in the definition base. The semantic schema is used by the form programs to navigate between objects and to access objects and their attributes.

For each language class a specific set of program components has to be developed. The *formatter* displays the contents of the abstract representation on a viewport of the users terminal, displaying thus a window of the concrete representation. The same formatter is capable also to print concrete representations. The *editor* receives user commands and modifies accordingly the contents of the abstract representation. The *validator* verifies the correctness of the changes made on the abstract representation. As already seen, some of the user actions will be validated before changing the contents of the abstract representation and others will be validated before committing the abstract representation to the repository. These two validation operations are performed by the validator component. In this way the abstract representation acts as a data gathering device, where these data will form a repository updating transaction when the abstract representation is closed or saved.

There is a need to translate between the organization of the repository and the concrete representations with which the environment user interacts. The *assemble* and the *disassemble* components translate the organization of the data contained in the repository to the organization required by the abstract representation and back.

These five components are meta-components and provide means to edit and browse all representation languages of a given representation language class. Using an internal language — the *form programming language* —, the environment builder instantiates these meta-components in order to edit representations written in a specific representation language. As described in section 2, the environment builder defines representation languages creating and maintaining the environment base. A selection of this environment base is used to generate a specific environment, which then is partitioned into one or more environment instances. While generating and partitioning the specific environment, the representation language definitions are converted into a set of rules. These rules appear as ellipses in Figure 12. The formatting, editing and validation rules connect the abstract representation to the concrete representation manipulated by the user, whereas the assembly and disassembly rules define how to convert the contents of the repository to the abstract representation and back. There is a set of rules for every representation language and the set of all rules of all representation languages in use is kept in the definition base. In order to make a representation language belonging to a supported language class available, one needs only to write the corresponding form program. Experience with Talisman has shown that this can be accomplished in a few days and that it is worthwhile to invest in building or specializing tools for a given project.

Figure 13 illustrates how the different meta-editors interact. As already mentioned, the set of meta-editors is open-ended. However, each time a new meta-editor is added the programs which compose Totem must be changed and possibly the storage schemas of the repository and of the definition base will have to be changed too.

At any given moment, several different viewpoints may be active, each editing a given representation. The workspace base is a specialized database which contains all abstract representations currently being edited plus data needed to control the work-station.

As seen in Section *3.2.4 Representation languages*, Totem groups representation languages into *language families*. A language family defines how to compose, format and edit representations belonging to several distinct but interrelated language classes. For example, a data flow language uses a diagram language class, a structure language class and a text language class. The diagram class language is used to display and edit data flow diagrams. The structure language class is used to display the diagram decomposition hierarchy, or to display and edit data structures and data flow structures. Finally the text language class is used to display and edit the specifications of the various elements contained in the data dictionary. All these representations may be displayed using different concrete representations, depending on the standards and notation in use.

The dictionary editor is a specialized processor capable of browsing all object class directories. It triggers actions on a selection of objects. As discussed in section *3.2.5 Representations*, representations are always reconstructed from a starting object, a representation class identification and a specific rule set. According to the selected action, the dictionary editor selects a specific rule set and applies a specific meta-editor to the set of selected objects. For example, the dictionary editor may be used to generate code within the structured programming language family. In this case, the text meta-

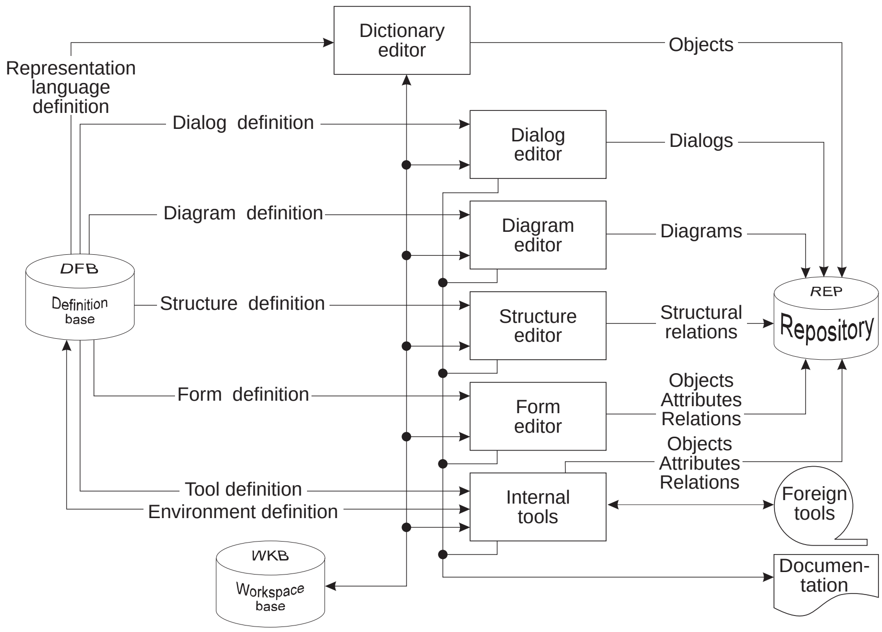

Figure 13. The structure and components of Totem

editor is selected, the code generating rule set corresponding to the structured programming language family is chosen, and the code is generated for each of the selected objects. If the target of the editor is a viewport, the user may edit all text fragments of the generated code which have been fetched from the repository. Otherwise, if the target of the editor is a file, the code will be written out onto this file and may then be submitted to a compiler.

Figure 13 shows some of the language classes supported. In the sequel we will briefly describe these meta-editors. The meta-editors shown are: a dialog language for dialog window representations, a diagram language for graphical representations, a structure language for hierarchical representations such as programs or documents, and a form language for fields containing text, strings and numeric values. These meta-editors are specialized for specific representation languages by obtaining information from the repository, the definition base and the workspace base. The repository contains the facts needed to build an abstract representation. The definition base contains the rules needed to edit the representation. Finally, the workspace base contains the abstract representation organized in a such a way as to optimize editing and browsing.

The abstract representation is typically a markup or tagged syntax tree of the representation. This syntax tree contains tags identifying attributes brought in from the repository, tags defining access rights and conventional structure tags. The first two groups of tags define the structure of the representation at the repository attribute level, whereas the third group of tags defines the internal structure of an elementary fact (typically a text fragment, or a coordinate on a diagram) contained in the repository. The representation language is used to define how to format and display the abstract representation. This is typically achieved by means of a style sheet, where this style sheet is part of the representation language.

Representation languages define many tools, including tools to validate the contents of the repository, transform representations into other representations even if these belong to a different representation language, compose hard-copy documents, and compose code.

Each representation language operates on some level of abstraction such as the system, module or code level and establishes an interpretation for its assembled collection of repository facts. For example, the interpretation could encompass activities such as data modeling, functional modeling, user interface modeling, or test data definition.

4.2 The Repository

In Totem all development activities interact with the repository. Developing a system corresponds to populating and/or updating this repository. Whenever a representation is required, the facts to be displayed are retrieved from the repository, composed in order to build the abstract representation shown in Figure 12 and then displayed. Whenever required data is not found, the form program can provide default data or appropriate error messages, allowing the environment user to correct representational defects. Thus, the use of the repository is independent of the order in which the repository is populated during the development process. This is a desirable feature since it assures that the environment does not impose a fixed development process, allowing environment builders to tailor development processes to specific needs.

The repository contains all facts about the target systems being developed and may also contain standard facts which have a wider scope. For example, corporate data models and dictionaries are often defined in order to assure integration between different target systems, possibly being developed in different environments.

Repositories may be acquired from third parties. For example, a corporation providing some interactive service for other corporations, could distribute repositories containing the interface characteristics of this service. Finally, the repository and tools should encourage generalized reuse. In other words, it is desired that the different target systems share large quantities of facts.

Reuse is a very effective way to reduce development effort and should be based on sharing facts in the repository among the different representations. For example, when specifying a system using data models, data flow diagrams and state transition diagrams, each representation can reference facts in the other ones. More specifically, a data store in a data flow may correspond to a database in a data schema, a transition in a transition diagram may be associated with a process in a data flow diagram, a data schema entity may contain data structures which in turn are also associated with several data flows. Hence, a representation corresponds to a view of the set of all facts contained in a given repository.

The repository contains *objects*. Objects may be partitioned into sub-objects, i.e. *instances*. Objects and instances belong respectively to *object classes* and to *instance classes*. Only one sub-object level is supported. Deeper structures can be achieved using recursive relations, such as *composition* and *decomposition* relations. In fact, an instance should be viewed as a named sub-domain of a given object's attributes. For example, a diagram — which is an object — contains several elements — which are instances.

Each object may have several attributes. Each attribute belongs to an *attribute category,* for example “name”, “text fragment”, “string”, “source code fragment”, and “marked up text fragment”. Notice that attributes can be quite large. Within each category, a virtually unrestricted number of attributes can be defined. Totem's repository objects are in fact abstract elements composed of several attributes. From a strict object oriented view, attributes correspond to objects and attribute categories correspond to classes.

Each object or instance class can be related to several other objects or instances. Relations are always m to n. Relations are treated the same way as attributes. In other words, a relation R between objects of classes A and B is physically implemented as a pair of list attributes R' and R''. Thus, if an object $a \in A$ relates to an object $b \in B$, R' is an attribute of a and contains all objects of class B to which a relates, and R'' is an attribute of b and contains all objects of class A to which b relates. Relations may also relate to a value object. That is, a relation R between object classes A and B binds to a value object class C. If $<a\ R'\ b>$ is in R then there is a $c \in C$ such that $<a\ R'\ b>$ is bound to c. This allows for example, establishing relations between “classes” and “functions”, where each individual link determines whether it defines a public or a private “function” of that “class”. Relations may satisfy several additional properties, such as allowing multiple references to a same object and ordering of the set of references.

Totem's repository is a multi-base repository, where the individual repositories may be distributed over a net-work. See Figure 14. Objects in one individual repository may relate to remote objects

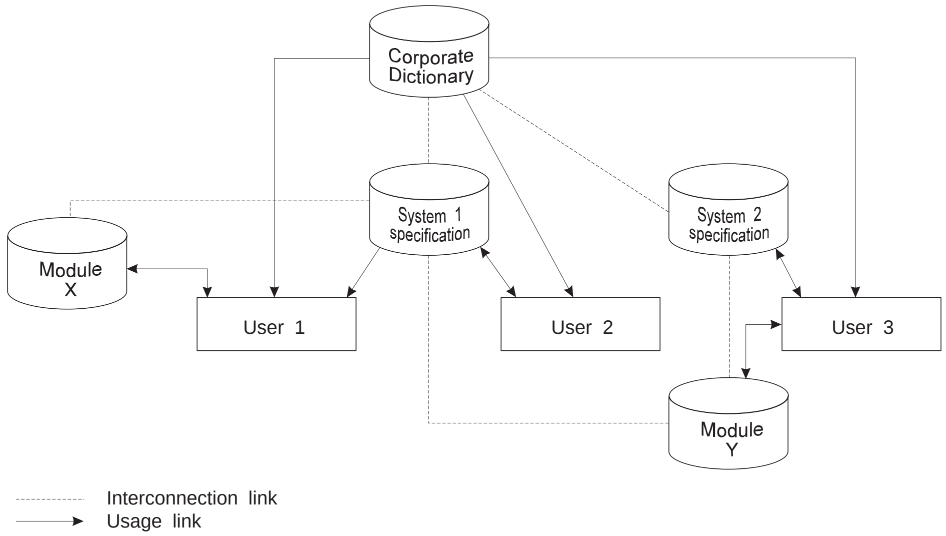

Figure 14. Logical partitioning of repositories

contained in another individual repository. The partitioning of the overall repository into individual repositories is arbitrary. There are object movement operations which move all attributes of an object between repositories. This multi repository structure allows establishing logical (e.g. financial and production) and abstraction-level partitions (e.g. requirements specification and design).

Several different versions of a same attribute may exist. However, the version derivation history is global for each individual repository. Some relations define both the target object and the version of that object. These relations can be used to define configurations. For example, in a versioned context, a given version of a module is composed of specific versions of its component functions. Since a relation is an attribute, the configuration of the module is itself versioned. Thus, the composition of the module in one version may be quite different from that in another version. Each individual repository has its own version history. This version history is not related to the target system version history, that is, a target system version is built from a configuration of attributes possibly contained in several different individual repositories.

Each versioned attribute, whether data or relation has its own access key. This assures that the semantic data schema of the repository can change, i.e. new object classes can be added, new attributes for any object class may be added, and new relations between object classes can be added, without affecting the data model of the implemented repository. Consequently, evolution of the environment does not require restructuring of the repositories.

As illustrated in Figure 5, every environment instance interacts with public and private repositories. Public repositories are read only. Private repositories may be selectively updated since some objects may be locked and hence cannot be updated. For example, when developing a target system which will use a specific library, the symbolic interface definition of this library could be included in the repository. Obviously these definitions may not be changed, otherwise it would be impossible to assure correct interfacing between target systems code and the pre-compiled library. When a public repository needs to be updated, it may either be disconnected and made private to the user performing the update, or a differential repository may be created. There is no mechanism for blocking elements contained in a public repository. The difference repository contains all changes made to the public repository by a given environment instance. Whenever the changes are accepted, the difference repository can be integrated with the public repository. This integration creates a new version for each of the modified attributes. If several instances simultaneously update the same public repository, conflicts may arise. Totem's integration tool provides mechanisms to help users to resolve these conflicts by negotiation.

The storage schema of the repository depends on the classes of representation languages which the meta-environment is capable of supporting. Thus, if a new representation language class is identified, possibly the storage schema must be adapted in order to facilitate composition and decomposition of representations written in languages of this new class. It is expected that this kind of evolution is not frequent.

The semantic schema of the repository depends on the definition of the environment contained in the definition base. An evolution of the environment may yield the semantic schema inconsistent with the existing contents of the repository.

Tools are provided to reconstruct the repository if its storage structure is corrupted. These tools are capable also to reconstruct the repository if its semantic schema is inconsistent with respect to the definition base. Data corruption and inconsistency may occur due to system malfunction, or due to changes to the definition base. Both storage and semantic schemas contain selected redundancy which allows crash recovery with a minimal amount of information loss. For example, if a disk is partially destroyed, the repository can be reconstructed, losing little more data than that written in the destroyed area.

4.3 The Definition Base

The definition base is partitioned into two sub-bases: the *root definition base* and the *instance definition base*. The root definition base defines the storage schema of all data bases used — the repository, the definition base itself and the workspace base —, it defines also the form programming language and several other items required to control Totem's basic functions. The root definition base is the same for all environment instances. Evolution of the root definition base entails a careful restructuring of existing repositories in order to conform to the new structure. Specialized tools must be provided to perform this restructuring.

The instance definition base defines the semantic schema of the repository and contains the set of rules of all representation languages available in the corresponding environment instance. Environments, that is instance definition bases, may not be evolved simultaneously with their use. They evolve in a way similar to a compiled program, i.e. change source code, compile and then use the new compiled code. However, the evolution of the environment instance base does not affect the storage schemas of the repositories and workspace bases. They may affect though the semantic schema of the repositories. As already mentioned, Totem provides tools to restructure the repository in order to assure its conformance to the specific environment. These tools are definition base driven and need not be reprogrammed when the instance definition base evolves.

Similar to the repository, the definition base is composed of objects, sub-objects, attribute categories, attributes and attribute versions. Each attribute of the definition base defines properties of the repository or of the environment instance itself. The organization of the definition base optimizes the runtime behavior of the environment. As already mentioned, the conversion from the environment base to the definition base is achieved while generating the specific environment and partitioning it into several environment instances.

When using the meta-environment several components must be defined. Once these components are known the development process and tools may be defined. All these definitions will be written using the environment definition representation language. More specifically, the environment definition language is used to:

- define the set of representation language families.
- define the set of representation languages and bind them to one or more families.
- define the composition of each representation language. These definitions typically define:
    - the object classes which compose the representation language,
    - the relations among objects of each class, both within the language and with object classes of other languages,
    - the attributes of the object classes.
- define the contents, formats, graphical aspects and editing rules of specification representations for each of the object classes.
- define quality evaluation rules.
- define transformation rules.

- define information interchange mechanisms with foreign tools and systems. For example, define how to generate code from the contents of the repository.
- define the development processes.
- configure environment instances.

4.4 The process base

The process base contains the definition of the development process and plans to be followed. Processes are defined in terms of pre-conditions and post-conditions. A pre-condition defines the required information, the representations which convey this information and the quality requirements these representations must satisfy. Similarly the post-condition defines the representations produced and the quality levels which will be assured. Each process defines also the tools to be applied while enacting the process. Finally, each process defines the metrics which will be gathered during its enactment.

In parallel with the development processes also management processes are defined, such as version control, configuration management, quality assurance, and process improvement.

Once a process has been defined it must be instantiated yielding a plan to develop or evolve a given software. The environment base must provide information, for example past development histories and statistics, in order to allow the project leader to allocate resources and establish time estimates. While a process is being enacted, several measurements can be performed automatically. The corresponding metrics are kept in the metrics base.

Both process and corresponding plans are kept in the process base. They link to tools in the definition base, to people or workstations who will enact each process, and to objects in the repository. The latter are needed in order to allow reconstructing the representations whenever they are needed. For example, instead of storing source code files, these could be reconstructed whenever the make program needs to recompile them.

5. Building the meta-environment

In this section we will briefly describe how we are planning to implement Totem. Similar to building compilers, Totem will be built in several steps using a bootstrapping approach. That is, components of the Totem system will be built in such a way that the result of a given development step provides tools to build the components of the next step. In this way, components will be systematically evaluated in practical use right after they are made available. We expect that in this way Totem will quickly converge to a reliable and useful system. In fact we are following recommendations by Brooks [Brooks 86], in the sense that systems should be systematically grown. Furthermore we are following a spiral model of development [Boehm], in the sense that the result of each step can be viewed as a prototype to be assessed and which will direct the development of the next step. Different from the spiral model, we expect that the Totem system will be ready when completing the last cycle and, hence, will not require a full conventional development process in order to build the final accepted system.

During the first step, part of the programs which comprise the Totem system will be developed, the root definition base will be built and a minimal instance definition base will be bootstrapped, leading to establishing an *initial meta-environment*. During this step specifications and programs will all be developed using Talisman. Several specialized Talisman tools will be developed for this purpose. While bootstrapping the meta-environment, several tables are used. See Figure 15. These tables are created and updated using a conventional editor. They define the contents of the root definition base as well as the contents of the initial instance definition base.

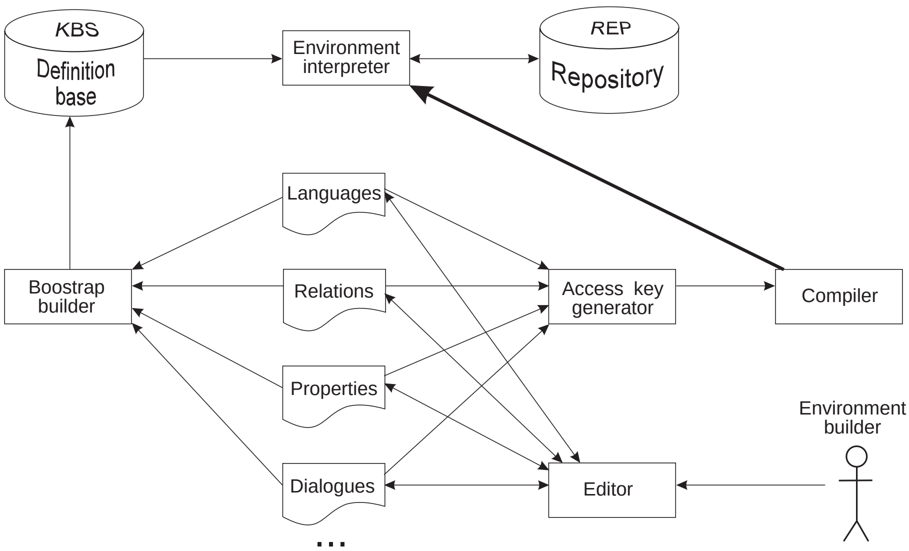

Figure 15. Bootstrap generation of the root definition base

During the second step, this initial meta-environment will be used to populate the environment base with development processes, representation languages and tools for the purpose of creating and maintaining environment bases themselves. The first representation language to be developed is the environment definition language. A specific environment and an environment instance will be derived from the environment base developed using the initial meta-environment. The result of this step is a *primary meta-environment*, which will be used to build environments as well as to maintain itself. We can now generate meta-environments of increasing power. See Figure 16. This provides means to evolve the meta-environment without having to reconstruct the root definition base every time from scratch. During this step existing components of Totem will be assessed and improved. As a result we expect a meta-environment capable of efficiently and effectively creating and maintaining process definitions, representation language definitions and tool definitions. We expect also that the tools needed to build an specific environment and to partition it into several environment instances are operating in a proper way. Finally, we expect that, at least in a small scale, the system effectively sup-

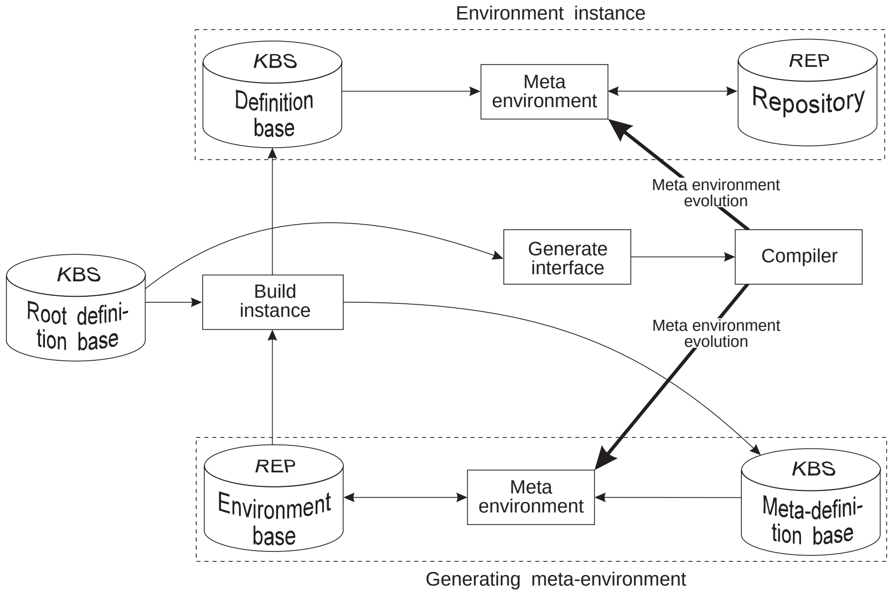

Figure 16. Self-maintaining meta-environment

ports cooperative work. Thus the result of this step can be viewed as a practical demonstration of the adequacy of Totem's basic concepts.

During the third step, the primary meta-environment will be used to create and maintain an elementary process, a minimal set of representation languages and tools. All these components are kept in an environment base and are built with the tools developed during the second step. This environment base will then be used to create a specific environment and to derive an *elementary environment*. This elementary environment aims specifically at supporting the development of Totem itself.

During the fourth step, all existing specifications, designs, tables and programs which comprise the existing Totem system will be imported into the elementary environment repository. This action corresponds to a reverse engineering of existing components, where these components have been built with the aid of Talisman. Once all existing components have been imported into Totem's repositories, they will be re-engineered in order to satisfy Totem's quality criteria and increased functionality. During this step the elementary environment will be assessed, as well as the reverse engineering and re-engineering methods used. As a result we expect an efficient and effective environment to be used to develop Totem itself, as well as a set of guiding principles to help in performing reverse engineering and re-engineering from systems developed with the aid of Talisman.

During the fifth step, we will complete the development of a first full version of Totem, using the elementary environment. During this step we will be assessing and improving the use of Totem to develop software. As a result of this step we expect a full version of Totem to be available and operating. We expect also that, at least from a laboratory point of view, it proves to be an efficient, effective and adequate environment to develop high quality software. Thus the result of this step can be viewed as a practical demonstration of the adequacy of the Totem meta-environment and of the environment instances built with it.

In the sixth step, we will start developing processes, representation languages and tools to be used during the development of other systems. The first of these components will be developed in a controlled environment in order to gather information to assess and improve Totem as an instrument to build industrial software systems. This activity never ends, in the sense that new representation languages may be added, or existing ones may be changed. Furthermore, Totem will need to be evolved giving raise to new components, repository structures, development processes, representation languages and tools.

6. Epilogue

In this report, we presented an overview of the functionality of the Totem meta-environment. Development of Totem is in progress. A prototype, Talisman, already exists. This prototype has allowed us to assess Totem's requirements as well as design issues.

Totem is being designed and implemented using Talisman. For the purpose of developing Totem several new tools have been added to the Talisman environment using its meta-environment capabilities. These tools adhere to the requirements of a defined development process [PCCW93]. Although this process is not fully integrated with the tools, the existing set up is providing valuable feedback for the design of Totem itself. In this way we are not only developing Totem, but are also using this very development effort to assess the stated requirements for tools and processes which are ultimately to be satisfied by Totem.

Acknowledgements

Several people indirectly contributed to the writing or reviewed early versions of this report. Our special thanks go to Prof. Daniel Berry who performed a thorough revision of an early version of this report.

References

[Balzer81] Balzer, R.; "Transformational Implementation: An Example"; *IEEE Transactions on Software Engineering* SE-7(1); 1981; pp 3-14

[Bigelow88] Bigelow, J.; "Hypertext and CASE"; *IEEE Software*; 1988; pp 23-27

[BL85] Belady, L.A.; Lehman, M.M.; *Program Evolution: Processes of Software Change*; Academic Press; 1985

[Brooks86] Brooks, F.P.; "No Silver Bullet - Essence and Accidents of Software Engineering"; *IEEE Computer*; 1986; pp 10-19

[CASE90] 1991 CASE Consulting Group; "CASE'90 REPORT: Results of the Fourth International Workshop on Computer Aided Software Engineering"; in *[CASE92]*; 1992; pp 289-326

[CASE92] Forte, G.; Madhavji, N.H.; Müller, H.A. eds.; *Proceedings of the Fifth International Workshop on Computer Aided Software Engineering*; Montreal; July 1992

[CL95] Cowan, D.D.; Lucena, C.J.P.; *Abstract Data Views*; 1995

[Conklin87] ConkLin, J.; Hypertext: "An introduction and survey"; *IEEE Computer* vol 20 no 9; 1987; pp 17-41

[Derniame92] Derniame, J.C.; *Software Process Technology, Proceedings of the Second European Workshop;* Lecture Notes in Computer Science 635; Berlin; Springer; 1992

[Fenton93] Fenton, N.; "How Effective are Software Engineering Methods?"; *Journal of Systems and Software* vol 22; 1993; pp 141-146

[Fenton94] Fenton, N.; "Software Measurement: A Necessary Scientific Basis"; *IEEE Transactions on Software Engineering* vol. 20 no. 3; 1994; pp 199-206

[Goldfarb90] Goldfarb, C.F. ed.; *The SGML Handbook*; Clarendon Press, Oxford, England; 1990

[Guedes93] Guedes, L.C.; Staa, A.v.; "Um Processo de Reengenharia Econômico e Eficaz"; *VII Simpósio Brasileiro de Engenharia de Software*; Rio de Janeiro, PUC-Rio; 1993; in Portuguese

[Hübsher95] Hübscher, P.J.E.; Sant'Anna, M.; Staa, A.v.; *Ferramentas Talisman para o desenvolvimento de programas Windows, utilizando C++ e MFC*; to appear; in Portuguese

[Humphrey90] Humphrey, W.S.; *Managing the Software Process*; Addison-Wesley, 1990

[Osterweil87] Osterweil, L.; "Software Processes are Software too"; *Proceedings of the 9th International Conference on Software Engineering*; 1987; pp 2-13

[PCCW93] Paulk, M.C.; Curtis, B.; Chrissis, M.B.; Weber, C.W.; *Capability Maturity Model for Software, version 1.1*; Carnegie Mellon University, Software Engineering Institute; Technical report CMU/SEI-93-TR-24, 1993

[Staa93] Staa, A.v.; *Ambiente de Engenharia de Software TALISMAN, Manual do Usuário*; Rio de Janeiro; Staa Informática; 1993; in Portuguese

[Welsh94] Welsh, J.; Han, J.; "Software Documents: Concepts and Tools"; *Software Concepts and Tools*; Vol. 1 no. 1, 1994