AUGUSTA

A reuse-oriented Software Engineering Environment for the Development of Ada-Applications

Elke Hochmüller, Roland T. Mittermeir
Institut für Informatik
Universität Klagenfurt
Universitätsstr. 65
A-9020 Klagenfurt
AUSTRIA
e-mail: elke@ifi.uni-klu.ac.at

Abstract

Increasingly complex demands on functionality and quality of software systems along with the lack of qualified staff require a fundamental change in the software development process. A shift from personnel intensive individual software development to capital intensive industrial software production must take place. This could only be achieved by utilizing tools supporting the development process and by planned design and production of generally applicable and reusable components.

The *AUGUSTA* system (**A**da **U**nits **G**eneralization **U**tility and **S**ystems **T**ailoring **A**ssistant) concentrates on the aspect of reusability. It allows the instantiation of programs from generic components and the compostion of complete application systems based on an equally generic application structure. Furthermore, the *AUGUSTA* approach postulates a special process model including a particular concept for user roles according to their experience and tasks in the software construction process.

1. Motivation

The way software is developed has undergone substantial changes within the last decades. Increased size and complexity of software needed and the long lifetime of software systems, with myrads of modifications made to such large systems led to systems which are almost impossible to maintain on the level of undocumented code only. New programming techniques (e.g. stuctured programming, module concept, information hiding), new design methods (e.g. Jackson System Development, Structured Design), new management methods (e.g. partitioning the development process in life cycle phases), and tools came into use - the integration of these techniques, methods and tools directed to a new paradigm, *software engineering*.

But in spite of the improvements made, a set of crucial problems is still plaguing todays software development. Examples thereof are:

- The cost of software is constantly increasing.
- The demand for complex software systems exceeds the supply of well educated software engineers.
- Software is too often delivered too late.
- Software maintenance is tedious and error prone, and its cost is very high.
- Only modest increases in software productivity could be materialized in recent years.

Software engineering methods and tools improved this situation far less than expected. Furthermore, case studies at a large software developer have demonstrated that within their originally created code, only 40 to 60 per cent where inherently original and specific. The rest existed in more or less the same form in more than one application [Horo89]. Hence, an improvement in software productivity by means of software reuse in combination with adequate automation of the software development process seems to be achievable and desirable.

The concept of software bases [Mitt87] presents a proposal how to achieve reusability on the components level in taking benefit from (exploiting) similarity between components and between applications. This concept has been refined and tailored to a particular language domain in *AUGUSTA* (**A**da **U**nits **G**eneralization **U**tility and **S**ystems **T**ailoring **A**ssistant) [Hoch92]. *AUGUSTA* was designed to be a reuse-oriented software engineering environment for Ada-programming. Our work thus focusses on a symbiosis between the concept of software reuse and the utilization of a particular software engineering environment.

This paper shows, how the software base concept has been refined and extended in developing the *AUGUSTA*-environment. The next section gives a short overview of the *AUGUSTA* approach. Afterwards, the architecture of the *AUGUSTA* environment will be described. We finally conclude with a summary and an outline of the state of the project.

2. Concept behind AUGUSTA

2.1. Introduction

Before introducing *AUGUSTA*, we present a brief introduction to the ideas behind the concept of a software base [Mitt87]. Its main aim is to reduce software development costs by speeding up the processes of development and maintenance in benefitting from reuse and by involving end-users in the development and maintenance of software systems.

Database Analogy

As end-users cannot be expected to dispose of professional programming skills, they should be supported in their software development effort by a powerful tool, the so-called *Software Base Management System* (*SBMS*). The term *SBMS* expresses an analogy to the area of databases; programs should be collected in a software base like data in a data base, the SBMS is intended to fulfil the same function as a DBMS fulfils in the database field. From this point of view, an application system can be regarded as a combination of programs stored in a software base.

Program Classification

In order to store and retrieve the programs administered by the SBMS a classification of these software components must be possible. For this purpose, two dimensions of classification are proposed: On one hand programs can be classified according the function (*task*) they fulfil (e.g. input, calculation, output), on the other hand programs can be related to a particular *application* (e.g. health insurance, liability insurance, comprehensive insurance). Similar tasks can be summarized within a special *task category*, while similar applications form a particular *application category*. The representation of this classification strategy for programs within a software base can be achieved by a so-called *program classification matrix* (Fig. 1). In this matrix, P_{ij} represents a program which fulfils the function j within application i. Since an individual application will consist only of few particular functions, usually the program classification matrix will be a sparse matrix.

$$\begin{array}{c|c|ccccc}
 & & \multicolumn{5}{c}{\text{TASK-CATEGORY}} \\
 & & TC_1 & TC_2 & TC_3 & .. & TC_m \\
\hline
 & A_1 & P_{11} & P_{12} & P_{13} & .. & P_{1m} \\
 & A_2 & P_{21} & P_{22} & P_{23} & .. & P_{2m} \\
\text{APPLICATION} & A_3 & P_{31} & P_{32} & P_{33} & .. & P_{3m} \\
 & .. & .. & .. & .. & P_{ij} & .. \\
 & A_n & P_{n1} & P_{n2} & P_{n3} & .. & P_{nm}
\end{array}$$

Fig. 1: Program Classification Matrix

2.2. Generalization and Program Development

For reuse purposes one of the following two conditions should be valid for as many programs as possible:

$$(1) \qquad \exists\, s,t\colon P_{sj}=P_{tj}$$

$$(2) \qquad \exists\, PG_j\colon \forall\, s\colon P_{sj} \text{ IS-A } PG_j$$

Condition (1) implies that the same program can be used in more than one application. This is the ideal case, but not necessarily the most usual one, as many applications may use functions of the same task category requiring a specific adaption of the program to the particular application purpose. Condition (2) states that there exists a code skeleton which is a program generalization (PG) of all programs within a task category.

The *AUGUSTA Software Base* aims at this last condition where each task category has it's own program skeleton (the so-called *program generic*). This skeleton is not yet executable and has to be refined according to application-specific requirements in order to receive the programs of the program classification matrix. Each task category is provided with a set of *specialization rules* which play a twofold role. First of all they have to be obeyed during the refinement process, secondly they represent stubs to be refined by *specializations* concerning *algorithm*, *data* and *interfaces*. A schematic view of the task refinement is represented in Fig. 2.

TASK-CATEGORIES

		TC_1	TC_2	...	TC_m
		PG_1 SR_1 $\{IT_{1.}\}$	PG_2 SR_2 $\{IT_{2.}\}$	...	PG_m SR_m $\{IT_{m.}\}$
APPL.	A_1	S_{11}	S_{12}	...	S_{1m}
	A_2	S_{21}	S_{22}	...	S_{2m}
	..	...	...	...	...
	A_n	S_{n1}	S_{n2}	...	S_{nm}

Fig.2: Program Classification Matrix with Task Refinement

Types of Specialization

The concept of software bases [Mitt87] proposes three main types of specializations: algorithm, data and interfaces. *AUGUSTA* [Hoch92] gives each of these types of specialization a set of specific interpretations and extends it even to specializations on the level of types.

For specializations concerning the *algorithm* the *AUGUSTA* approach suggests the substitution of the so-called *procedure-stubs* by complete procedures. Formal parameters can either be completely defined during the development of the program generic or instantiated during the refinement process. In the latter case, specialization rules have to be defined at the task category level. Furthermore, stubs for procedure or function calls and the appropriate actual parameters are possible, too.

In extension to the software base concept, *AUGUSTA* is not limited to *constant values* when dealing with *data specializations*, since specializations of *type definitions*, *type identifiers* and *variable identifiers* are also supported. Identifier specializations should help programmers to place appropriate names within the application context.

Interface specializations concern the execution order of programs as well as the description of data to be exchanged by the programs concerned. This type of specialization is of special nature as programs of different task categories have to be regarded jointly. Programs may consist of several procedures and functions which could be called by other programs. Provided this general case, one could imagine that the administration of interface information, including refinement guidelines, represents a complex problem. As this is not intended to be the central point of our work we propose that each task category communicates with the external world only through one particular procedure, the so-called *interface procedure*. The formal parameters of this special procedure grant the data exchange between programs of different task categories. Additional to the essential declarations, the interface procedure is the main element of the program generic. Further procedures are either available in their code representation or as stubs.

For better administration of interface information, the software base concept distinguishes two levels of integrity constraints - task category and program - and proposes a matrix structure at each level: the *task interface matrix* (*TIM*) at the task category (TC) level and the *program interface matrix* (*PIM*) at the program level. The entries of these structures represent information whether direct connections between pairs of task categories (TIM) and pairs of programs (PIM) have to, might, or must not be provided. While these two structures are the only interface representations suggested by the software base concept, we learned during our work on *AUGUSTA* that they are not sufficient. Hence, we propose at each level of integrity constraints an additional list structure: the *task interface attention list* (*TIAL*) at TC level and the *program interface attention list* (*PIAL*) at program level. These structures represent strong constraints, while TIM and PIM are regarded as week constraints which have to be obeyed by TIAL and PIAL. The latter two lists contain for each task category and program the identifiers of task categories and programs to be necessarily executed before; hence these list structures guarantee that the input data is available before calling any particular task or program.

2.3. Generalization and Application Development

As enterprises will usually need several similar applications, the generalization concept can be applied at the level of application, too. Hence the software base approach postulates for each application category a so-called *application lattice* which can be completed to similar applications by replacing *composition stubs* with programs according to particular *composition rules*. These composition rules refer to programs to be candidates for composition stub substitution. Fig. 3 shows in a simple manner a possible application lattice with some stubs. The shape of the stubs depends on the connected compositon rule, e.g.

CS_1 and CS_4 are examples for stubs with the same composition rules and therefore the same choice of possible programs to be plugged in, whereas the different shapes of stub CS_2 and CS_3 indicate that they need to be instantiated with programs drawn from different task categories. The *AUGUSTA* approach distinguishes two different kinds of composition stubs - *mandatory* and *optional* stubs.

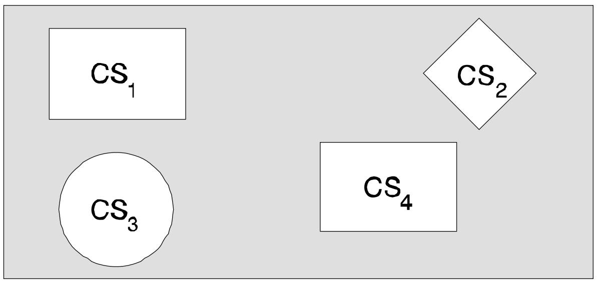

Fig.3: Application Lattice with Composition Stubs

Mandatory stubs have to be refined by choosing one particular program meeting the requirements of the appropriate composition rule which can be of one of the following three types:

(a) The composition rule contains all possible programs gained from the interface specialization rules at program level (PIM, PIAL).

(b) *Selection List*: The composition rule contains only some programs out of those which would in principle be possible according to (a).

(c) *Exclusion List*: This list contains some programs which meet the interface specialization rules at program level (a) but must not be used in the particular case. The actual composition rule containing the candidate programs will be generated automatically using the exclusion list as input.

Optional stubs may be refined but don't have to. It can be decided at composition time whether it is necessary to plug in an appropriate program or whether the stub can be simply removed without any substitution. This kind of stubs can occur within a sequence as well as connected with selections. Furthermore, optional stubs provide the possibility to determine the flow of control during application composition by choosing alternative ways of generation.

2.4. Role Concept

Similar to databases, both the software base and the *AUGUSTA* approach provide for different user roles. Hence, the following five classes of users are supported by the SBMS in developing software systems (Fig. 4):

- *Software Base Administrator (SBA) at task category (TC) level*: All task categories are characterized by the same meta-structure, the so-called *task category type*. Completely defined task categories are called *task category instances*. The Software Base Administrator is responsible for the definition of these task category instances. He must specify one program generic and the appropriate specialization rules concerning data, algorithm and interfaces (TIM, TIAL) for each TC instance. A special language - *Task Definition Language* (*TDL*) - consisting of Ada constructs and additional elements for the representation of specialization stubs supports the SBA in defining the TC instances.
- *Programmer*: At the *program level*, the programmer integrates a given task category instance to one or more programs by refining the given program generic according to the given specialization rules. During interface specialization he defines entries for PIM and PIAL obeying the information contained in TIM and TIAL. The usage of the *Specialization Definition Language* (*SDL*) assures that the program generic cannot be changed by the programmer at this level any more.
- *Software Base Administrator at the application category (AC) level*: The SBA defines the application lattice consisting of particular symbols representing programs, stubs and control stucture elements (sequence, selection, iteration). The language provided at this level is called *Application Definition Language* (*ADL*).
- *Application Expert*: At *application level* the application expert composes the application system using a particular *Application Manipulation Language* (*AML*). On one hand he influences the system structure by his decisions in case of optional stubs, on the other hand he determines the contents of the application system by replacing stubs with special programs.
- *Application User*: The application user processes his data by the composed system during the performance of his duty.

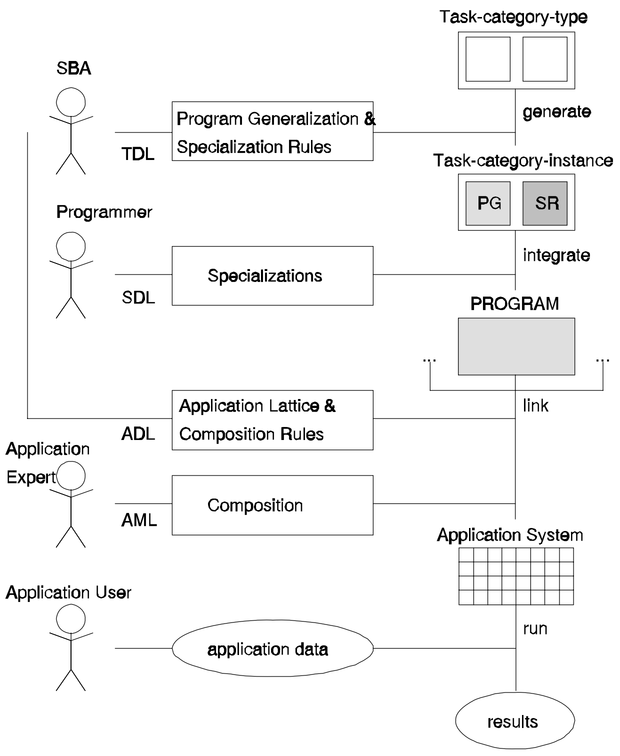

Fig. 4: Schematic View of User Roles and their Activities

2.5. Integrity Constraints

The software base concept as well as the *AUGUSTA* approach provide a lot of integrity constraints defined either within the SBMS itself or by different types of users. The model of integrity constraints suggested by *AUGUSTA* introduces various layers of constraints according to the role concept previously explained. A schematic view of this model is represented in Fig. 5 expressing the principle that constraints specified in an inner layer have to be obeyed in outer ones.

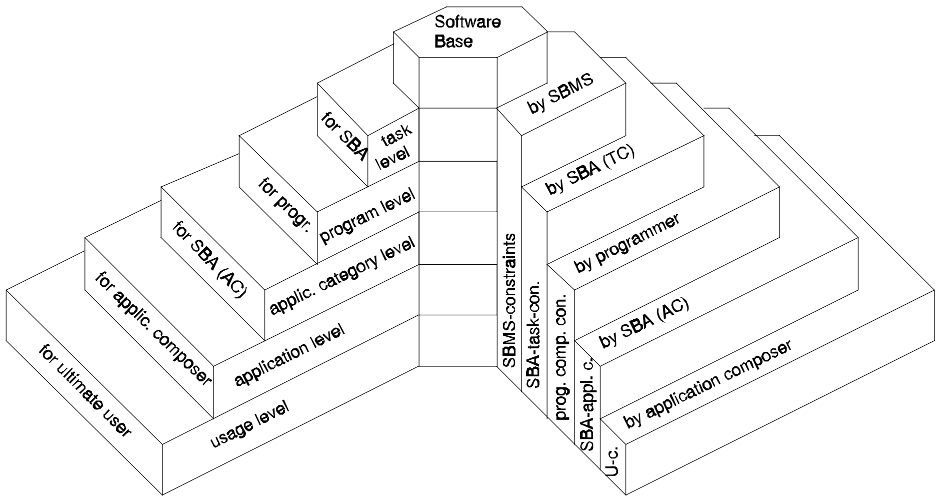

Fig. 5: Layers of Integrity Constraints

The following list contains examples for the most important constraints provided by *AUGUSTA*:

- Constraints implemented within the AUGUSTA-SBMS:
  * for SBA at TC level:
    - Each task category instance consists of one and only one program generic (but two files for definition and body package respectively are possible) and associated specialization rules.
    - Each program generic has one and only one interface procedure; interface rules refer to this special procedure only.
    - Only types of specialization rules provided by the AUGUSTA-SBMS can be used.
    - For each stub a default specialization must be defined.
    - Only completely defined TC instances can be released for refinement.
  * for program level:
    - Each program must be a specialization of an existing program generic.
    - Only completely specified programs can be released for application composition.
  * for SBA at AC level:
    - Only types of composition stubs provided by the AUGUSTA-SBMS can be used.
    - Mandatory stubs must contain the identification of a task category.
    - Composition stubs can only contain those task categories which have their input data already generated by tasks located at earlier positions in the application lattice (sequence of task categories is important).
    - Only completely defined application lattices can be released for application composition.
  * for application level:
    - The given composition rules must be obeyed during application composition.
    - The sequence of programs must correspond to the interface constraints.
    - Only completely defined application systems can be released for final usage.

- Constraints defined by SBA at TC level:
  * for program level:
    - During the refinement of each program the specialization rules defined by SBA must be obeyed (TIM, TIAL).
  * for SBA at AC level:
    - Interface specialization rules must be obeyed during application lattice development (TIM, TIAL).
- Constraints defined by programmer at program level:
  * for SBA at AC level:
    - Interface specializations must be obeyed by the SBA during the development of the application lattice and the associated composition rules (PIM, PIAL).
  * for application level:
    - If neither selection lists nor exclusion lists exist, the application expert can choose programs according the interface specializations (PIM,PIAL).
- Constraints defined by SBA at AC level:
  * for application level:
    - In case of mandatory stubs one out of the candidate programs has to be selected for substitution.
    - In case of optional stubs the flow of control of the application can be influenced.
  * for usage level:
    - The frame of functionality for all possible applications is determined by the structure of the application lattice.
- Constraints defined by application expert at application level:
  * for usage level:
    - Access constraints on data for specific applications have to be obeyed during application execution.

3. System Architecture of AUGUSTA

This section deals with the architecture of AUGUSTA as a software engineering environment. First of all a global view of the architecture will be presented and after a short outline regarding the concrete architecture the components of *AUGUSTA* will be described.

3.1. Global System Architecture

The overall architecture of *AUGUSTA* is represented in Fig. 6. It corresponds in principle to the structure of the ECMA/NIST-"toaster"-model for integrated CASE environments ([Norm92], [Chen92]). The fundamental three layers of the *AUGUSTA* environment are the *AUGUSTA-User-Interface*, the *AUGUSTA-Software-Base* and the set of specific *AUGUSTA*-tools. The latter serve for base administration, TC management, program management, AC management, and application management. A brief description of these tools is given in section 3.3.

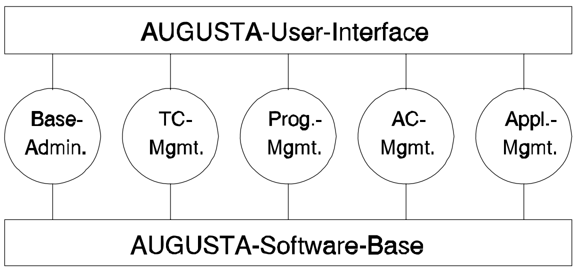

Fig. 6: Global System Architecture

The *AUGUSTA-User-Interface* provides as a homogeneous look-and-feel by using uniform graphical elements in order to ease handling and to have only very tiny differences between the individual tools in user interaction. This can be achieved by the same menu structure for all user groups as well as by using standard elements provided by OSF/Motif. The only difference exists in the various selection possibilities according to special user roles.

The *AUGUSTA-Software-Base* serves as a uniform repository for all documents generated and manipulated using the five *AUGUSTA-Tools*. According to various user roles and integrity constraints the tools have only restricted access to different software components.

3.2. Concrete System Architecture

The data exchange within the *AUGUSTA* environment is realized using a *client-server model* (Fig. 7). The environment is partitioned into two main components - the *server* for the administration of persistent data and the *clients* for user communication.

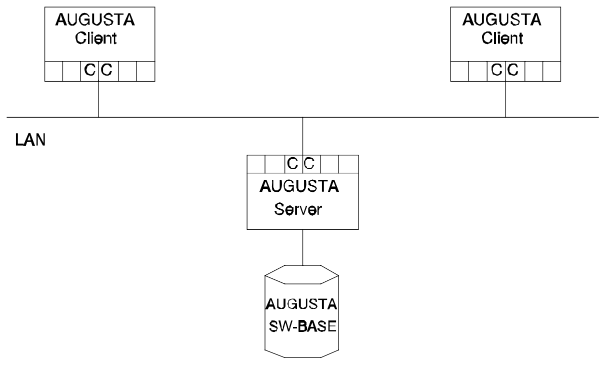

Fig. 7: Client-Server Architecture of AUGUSTA

In our case the *AUGUSTA-SBMS* performs the function of the *server* by administrating the *AUGUSTA-Software-Base*. Each access to the contents of the software base can only take place using the *AUGUSTA-SBMS*.

The *clients* are implemented as window-based environments provided by particular tools according to the different user roles.

The *communicational components* (CC in Fig. 7) are necessary to bridge differences between different languages (and platforms) used for the server (implemented in Ada) and the clients (implemented in C). They serve for conversion of data structures from one language into a particular standard data format before data transmission and vice versa after data transmission.

The clients communicate with the server via request-response-channels. The requests of the clients are atomic and independent of each other. They are serially sent to the server via a uniform communication interface. The server processes the requests according to the order of arrival.

3.3. AUGUSTA-Tools

The *AUGUSTA* environment consists of five tools supporting three different users - the SBA (base administration, TC-, and AC-management), the programmer (program management), and the application expert (application management). It will be described in this section. Each particular component of these tools can be assigned either to the *AUGUSTA-User-Interface* or to the *AUGUSTA-SBMS*.

Base Administration Support Tool

The SBA is supported in those administrative tasks which would conform to analog tasks to be performed by a data base administrator by the *Base Administration Support Tool* (Fig. 8). The functionality of this tool comprises the administration of user privileges, the conducting of various statistics, and the global deleting of special contents of the *AUGUSTA-Software-Base.* The only component of the *AUGUSTA-User-Interface* is the Base Administration Editor, while the *AUGUSTA-SBMS* consists of the Base Administration Storage Unit.

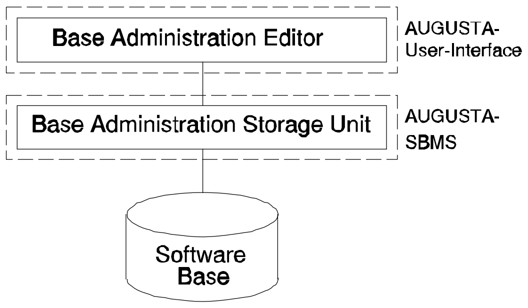

Fig. 8: Base Administration Support Tool

Task Category Management Tool

The *Task Category Management Tool* (Fig. 9) supports software base specific functions. It helps the SBA on the task category level. The *AUGUSTA-User-Interface* consists of four editors (TC, program, specialization rule, task interface); the corresponding storage units, the specialization rule filter (checks for each stub whether a specialization rule and a default specialization were defined), and the default specialization integrator (generates a default program) belong to the *AUGUSTA-SBMS*.

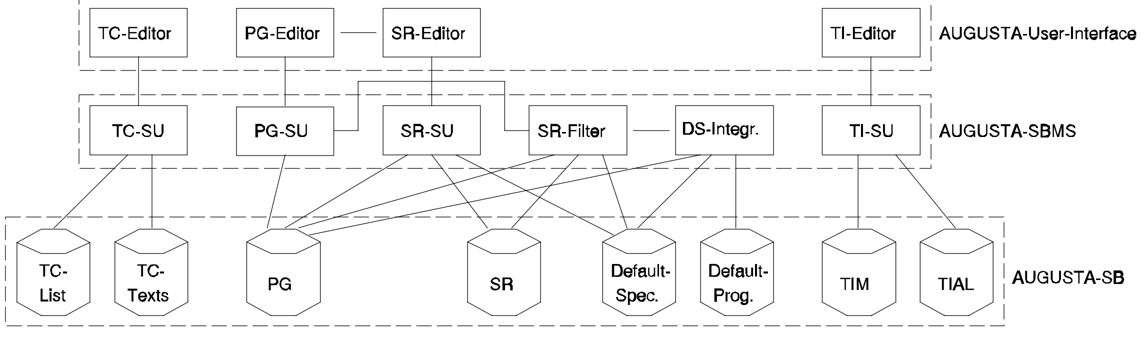

Fig. 9: Task Category Management Tool

Program Management Tool

The *Program Management Tool* (Fig. 10) supports the programmers in program specialization. The *AUGUSTA-User-Interface* is represented by the program refinement editor and the program interface editor, respectively. The task category viewer (provides the program interface editor with informations about the task category), the specialization storage unit, the specialization integrator (generates the refined program), and the task/program interface management (administrates interface information according to given constraints) form the *AUGUSTA-SBMS*.

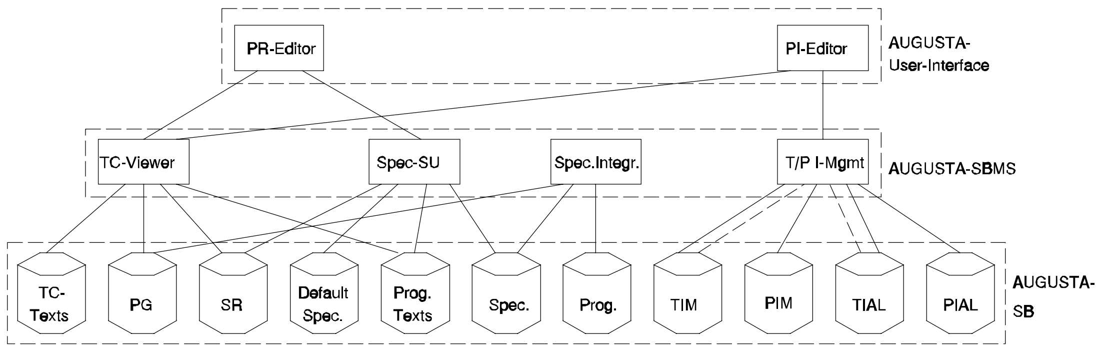

Fig. 10: Program Management Tool

Application Category Management Tool

At the AC level the SBA will be supported by the *Application Category Management Tool* (Fig. 11). The *AUGUSTA-User-Interface* consists of three editors (application category, application lattice, composition rule), while the *AUGUSTA-SBMS* is formed by the corresponding storage units (application category, application lattice), a task-category/program viewer, and a sophisticated composition rule management (is not only a storage unit, but also a checker of stubs with respect to TIM and TIAL as well as a generator of selection lists according to PIM and PIAL).

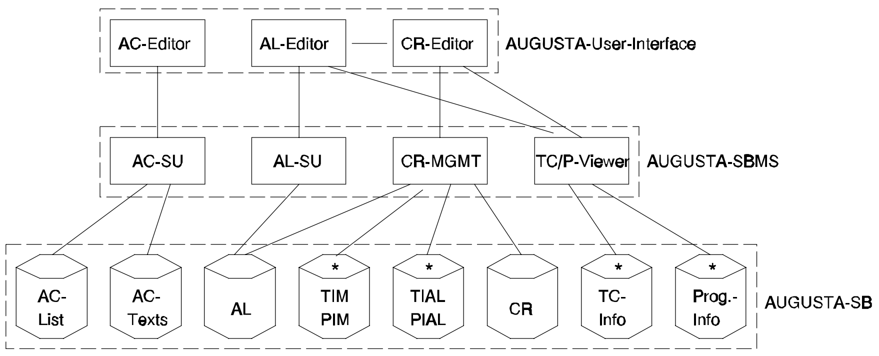

Fig. 11: Application Category Management Tool

Application Management Tool

The *Application Management Tool* supports the application expert in composing systems. The only component of the *AUGUSTA-User-Interface* is the composition editor which allows as input a textual description of the application, selection of programs, specification of names for parameters and literals, and path decisions in case of alternative ways of generation. Furthermore, the representation of the application using a special script language [Buch91] is possible. The *AUGUSTA-SBMS* consists of the composition storage unit, two viewers (task-category/program, application-category/application) providing information about the contents of the software base, the application integrator (generates the *application execution control script* to be stored in the software base), and the software integration platform (generates the final application system) [Buch91].

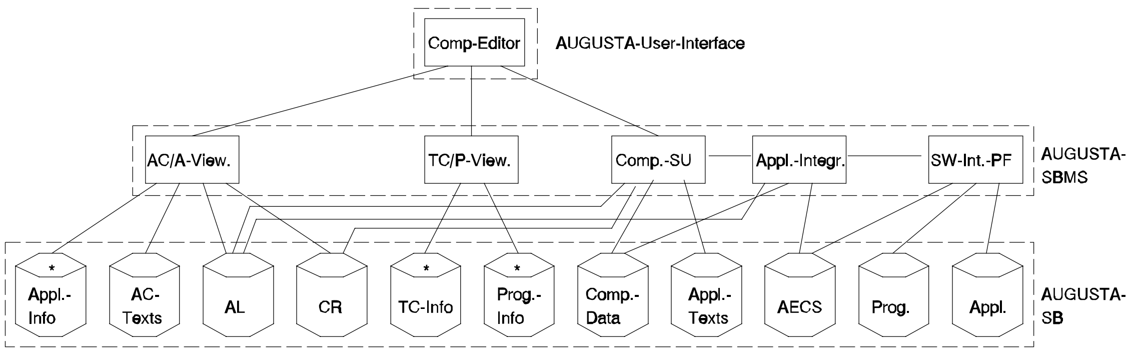

Fig. 12: Application Management Tool

4. Summary

This paper presented an environment (*AUGUSTA*) for supporting reuse of Ada-code. It organizes program components in a database like manner and allows for composing application systems via special queries against this base. Without demanding object-orientedness, it takes advantage from generalization hierarchies among program-components and among applications.

The current implementation of *AUGUSTA* runs in a UNIX-workstation environment. It comprises the functionality as described in this paper, except for the base administration support tool and parts of the graphical user interface.

5. References

[Buch91] F. Buchhäusl: "Die Realisierung eines Software-Integrationssystemes für ein Software Base Management System", Technical Report, Institut f. Informatik, Universität Klagenfurt, Klagenfurt, Sept. 1991

[Chen92] M. Chen, R.J. Norman: "A Framework for Integrated CASE", IEEE Software, Vol. 9, No. 2, March 1992, pp. 18-22

[Hoch92] E. Hochmüller: "AUGUSTA - eine reuse-orientierte Software-Entwicklungsumgebung für die Erstellung von Ada-Applikationen", Ph.D.-Thesis, Vienna, May 1992

[Horo89] E. Horowitz, J.B. Munson: "An Expansive View of Reusable Software", in T.J. Biggerstaff, A.J. Perlis (eds.): "Software Reusability", Vol. 1, 1989, pp. 19-41

[Mitt87] R.T. Mittermeir, M. Oppitz: "Software Bases for the Flexible Composition of Application Systems", IEEE Transactions on Software Engineering, Vol. 13, No. 4, April 1987, pp. 440-460

[Norm92] R.J. Norman, M. Chen: "Working together to integrate CASE", Guest Editor's Introduction to IEEE Software, Vol. 9, No. 2, March 1992, pp. 12-16