PROCESS SUPPORT FOR SOFTWARE REUSE WITH *AUGUSTA*

Elke Hochmüller
Institut für Informatik
Universität Klagenfurt
Klagenfurt
AUSTRIA

ABSTRACT

Increasingly complex demands on functionality and quality of software systems and higher application dynamics 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 composition 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.

This paper explains how the *AUGUSTA* system would support software development based on component integration, prototyping and reuse. It demonstrates the approach using an example from the domain of electric power plants.

MOTIVATION

The way software is developed has undergone substantial changes within the last decades. Increased size and complexity of software with the myriads of modifications made during their long lifetime led to systems which are very hard to maintain. New programming techniques (e.g. structured programming, module concept, information hiding), new design methods (e.g. Jackson System Development, Structured Design, Object Oriented Design), new management methods (e.g. partitioning the development process into life cycle phases), and tools came into use - the integration of these techniques, methods and tools constitutes the current basis of *software engineering*.

But in spite of the improvements made, a set of crucial problems is still plaguing today's 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.
- Too often, software is 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 were inherently original and specific. The rest existed in more or less the same form in more than one application (Horowitz, 1989). Hence, an improvement in software productivity by incorporating reuse into the process of software development in combination with adequate automation of this development process seems to be achievable and desirable.

The scope of reuse clearly is not limited to the individual software engineer building and using his personal library. Only if reuse is an integral part of the whole enterprise, its organization, and its process structure, all of its benefits will accrue (Tracz, 1990, Dusink, 1992). Reuse specific process models are proposed by Caldiera and Basili (1991), Prieto-Diaz (1991), and Hochmüller and Mittermeir (1993).

However, implementing such changes is often hampered by already installed software engineering environments. Most

existing CASE tools are built in the tradition and along the paradigm of a phased, top-down oriented waterfall life cycle model. Notably after a development phase has been completed, it is difficult to step back and apply changes. Likewise, it is difficult to look forward for preparing design units for the future integration of reusable components.

AUGUSTA (**A**da **U**nits **G**eneralization **U**tility and **S**ystems **T**ailoring **A**ssistant) (Hochmüller, 1992) was designed to be a reuse-oriented software engineering environment for the development of Ada-applications aiming at an improvement in software productivity by proposing and supporting a particular software production process tailored to software reuse.

The next section gives a short introduction to the *AUGUSTA* approach. Afterwards, the reuse-oriented process model supported by *AUGUSTA* will be described. Finally, the concepts of the *AUGUSTA* approach will be illustrated by using a technical example.

CONCEPT BEHIND *AUGUSTA*

Introduction

Before introducing *AUGUSTA*, the ideas behind the concept upon which it was built, the notion of a software base (Mittermeir and Oppitz, 1987), will be described briefly. The main aim of this concept 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; in a software base, programs should be collected like data in a data base. The SBMS is intended to fulfil the same function as a DBMS fulfills 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 made. For this purpose, two classifying dimensions 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. temperature control, fluid level control, rotation speed control) they serve. Similar tasks can be grouped together into a special *task category*, while similar applications form a particular *application category*. Conceptually, classification of programs on these two dimensions yields a so-called *program classification matrix* (Figure 1). In this matrix, P_{ij} represents a program which fulfills the function j within application i. Usually the program classification matrix will be a sparse matrix since individual applications will consist only of few particular functions.

TASK-CATEGORY

APPLICATION		TC_1	TC_2	TC_3	..	TC_m
	A_1	P_{11}	P_{12}	P_{13}	..	P_{1m}
	A_2	P_{21}	P_{22}	P_{23}	..	P_{2m}
	A_3	P_{31}	P_{32}	P_{33}	..	P_{3m}
	..	..	..	..	P_{ij}	..
	A_n	P_{n1}	P_{n2}	P_{n3}	..	P_{nm}

FIGURE 1: PROGRAM CLASSIFICATION MATRIX

Generalization and Program Development

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

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

(2) $\exists$ PG_j: $\forall$ s: P_{sj} 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 Figure 2.

Types of Specialization. The concept of software bases (Mittermeir and Oppitz, 1987) proposes three main types of specializations: algorithm, data and interfaces. *AUGUSTA* (Hochmüller, 1992) gives each of these types of specialization a set of specific interpretations and extends it even to specializations on the level of types.

Algorithmic specializations are realized in *AUGUSTA* by 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.

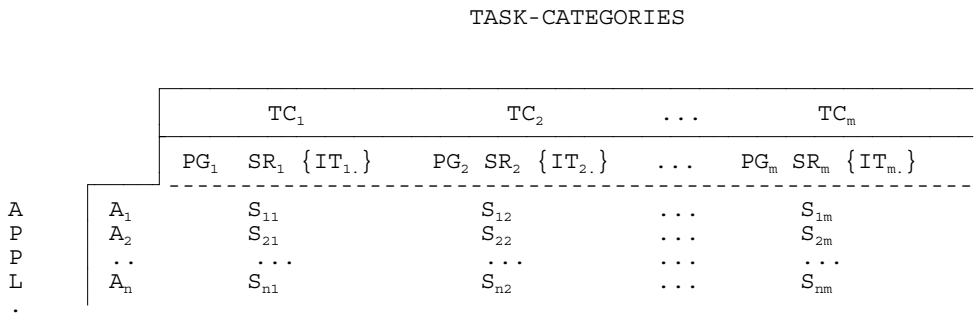

FIGURE 2: PROGRAM CLASSIFICATION MATRIX WITH TASK REFINEMENT

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 considered 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 this work it is suggested 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. These two structures are the only interface representations suggested by the software base concept, but during the work on *AUGUSTA* they turned out to be not sufficient. Hence, at each level of integrity constraints an additional list structure is proposed: 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 weak 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; thus, these list structures guarantee that the input data is available before calling any particular task or program.

Generalization and Application Development

As enterprises will often 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 various 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.

Figure 3 shows in a simple manner a possible application lattice with some stubs. The shape of the stubs depends on the connected composition 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.

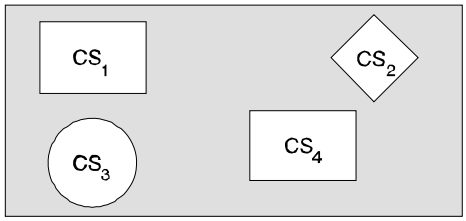

FIGURE 3: APPLICATION LATTICE WITH COMPOSITION STUBS

The *AUGUSTA* approach distinguishes two different kinds of composition stubs - *mandatory* and *optional* stubs.

Mandatory stubs have to be refined by choosing one particular program, meeting the requirements of the appropriate composition rule. Composition rules can be of one of the

following three types:

(a) *Complete List:* 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 of the candidate programs of those which would on 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 sequences as well as in relation to selections. Furthermore, optional stubs provide the possibility to determine the flow of control during application composition by choosing alternative ways of generation.

Comparison with object-oriented Concepts

Since generalization as a prerequisite for inheritance represents an essential part of object-orientation, the necessity of comparing the *AUGUSTA* approach with object-oriented concepts becomes obvious. Pure object-orientation allows as many levels of class hierarchies as you like and also permits unlimited overwriting if only interfaces are preserved. Certainly, the generalization concept of *AUGUSTA* allows only single-stage hierarchies in both Program and Application Development, but yet a clear distinction between reusable generic and rather application specific software pieces takes place such that specialization and composition rules can be posed in order to guide as well as restrict the software development process. Beyond that, *AUGUSTA* offers a particular process model which will be described in the next section.

Hence, as a result of the comparison parallels to the differences between third and fourth generation languages come into mind: the concept of object-orientation is rather generally applicable whereas the *AUGUSTA* approach is more powerful within a special domain.

THE *AUGUSTA* PROCESS MODEL

The core of the *AUGUSTA* process model consists of a particular role concept which is tightly integrated with integrity constraints grouped in layers corresponding to the roles. Before going into the details of the process model, two important strategies for the incorporation of reuse into the software development process will be briefly introduced.

Reuse Strategies

Development for Reuse deals with the construction of new software components for later reuse. Unlike traditional good software engineering technology (e.g. structured analysis/ structured design, structured programming) not only the development of easily maintainable software satisfies the notion of development for reuse, but specially anticipation of future requirements and independence of each single design component are characteristics of development for reuse. Thus, on one hand software components must be as generic as possible to guarantee repeated reuse by instantiation, and on the other hand they must be independent concerning their immediate environment in order to be applicable in as many contexts as possible.

During *Development with Reuse* the components produced in a development for reuse process are used again within various software projects. In case that existing components do not exactly meet the required specification adequate adaptations must take place.

These reuse strategies should not be considered as alternatives, most powerful reuse potential can only be achieved by benefitting from their interrelationship. Thus, this correlation is also evident within the role concept suggested by the *AUGUSTA* approach which will be discussed next.

The *AUGUSTA* Role Concept

In order to take into account the structure of the contents of the software base the development process proposed by *AUGUSTA* is divided into several subprocesses which support the division of labour between members possessing different user roles.

Hence, the *AUGUSTA* SBMS supports the following five classes of users (Figure 4):

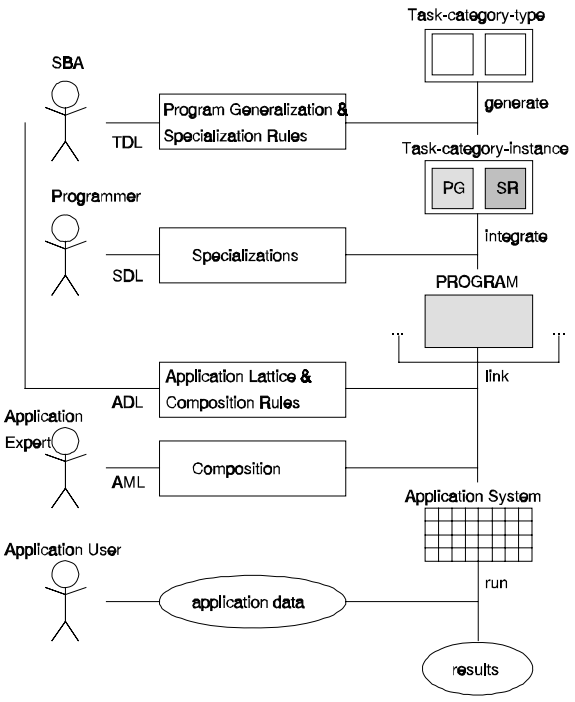

FIGURE 4: SCHEMATIC VIEW OF USER ROLES AND THEIR ACTIVITIES

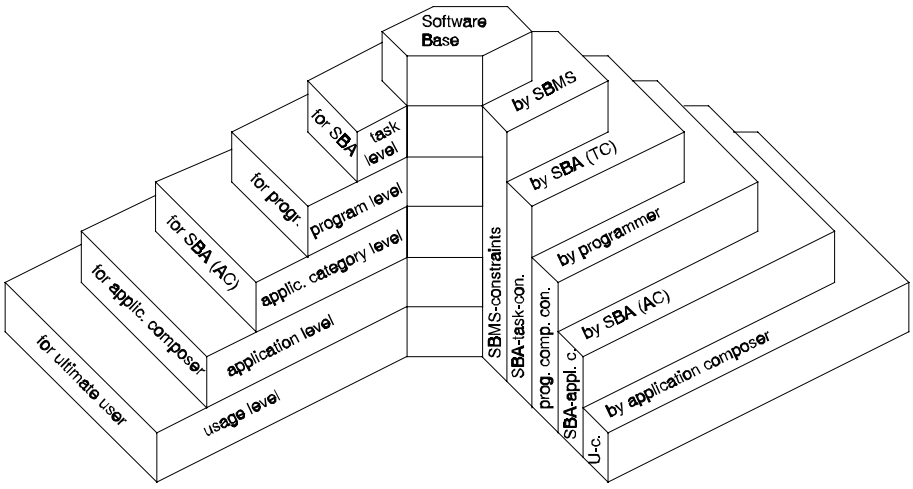

FIGURE 5: LAYERS OF INTEGRITY CONSTRAINTS

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.

The tasks of the SBA at TC level are the only ones which doubtless belong to development *for* reuse.

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.

Regarding the reuse strategies mentioned before, the function of the programmer is twofold: on the one hand he is involved in development *with* reuse by instantiating the generics provided by the SBA, on the other hand he develops *for* reuse by constructing programs for later usage.

Software Base Administrator at the application category (AC) level. The SBA defines the application lattice consisting of particular symbols representing programs, stubs and control structure elements (sequence, selection, iteration). The language provided at this level is called *Application Definition Language* (*ADL*).

Similar to the equivalent tasks at TC level the SBA also constructs generic structures at the AC level, hence he develops *for* reuse. Nevertheless, at the same time the SBA is also involved in development *with* reuse by reusing components (i.e. task categories, programs, interface structures) produced during earlier stages of the development process.

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.

Classifying the function of the application expert according reuse strategies, it is obvious that the related work is integrated within development with reuse.

Application User. The application user runs the generated system. As this kind of user is not directly involved in the development process in the strict sense, there will be no relationship to any of the reuse strategies discussed before.

Integrity Constraints

In order to achieve a smooth process course 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 Figure 5 expressing the principle that constraints specified at an inner layer have to be obeyed by agents working on outer layers.

The *AUGUSTA* Environment and its Application

The overall architecture of *AUGUSTA* is represented in Figure 6. It corresponds in principle to the structure of the ECMA/ NIST-"toaster"-model for integrated CASE environments (Norman and Chen, 1992, Chen, 1992). The fundamental three layers of the *AUGUSTA* environment are the *AUGUSTA-User-Interface*, the *AUGUSTA-Software-Base* and the set of special *AUGUSTA*-tools tailored to the specific user roles. Hence, these tools support the SBA in base administration, TC management, and AC management, the programmer in program management, and the application expert in application management.

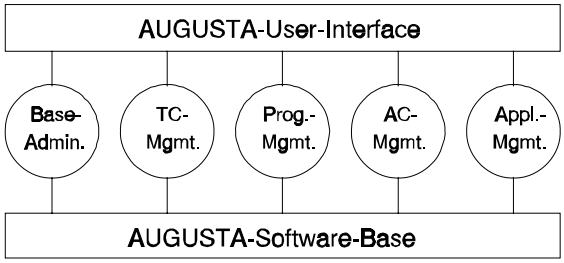

FIGURE 6: GLOBAL SYSTEM ARCHITECTURE

The *AUGUSTA-User-Interface* provides 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.

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

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 Figure 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 the other way round after data transmission, respectively.

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.

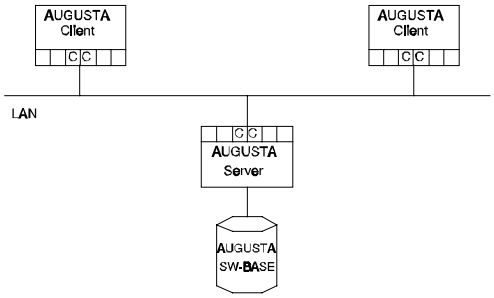

FIGURE 7: CLIENT-SERVER ARCHITECTURE OF *AUGUSTA*

Let us take now a closer look at the application of the *AUGUSTA* environment concerning prototyping support, multi-user and teamwork support, and required user profiles.

Prototyping Support. Besides life cycle support from the phase of detailed design, *AUGUSTA* also supports prototyping, particularly at the application level. Thus, the application expert may create and test applications in a straightforward manner. In doing so, the application expert is supported by a software integration platform (Buchhäusl, 1991) which serves for the generation of the application system out of the composition decisions. Prerequisite for rapid prototyping at the application level is the existence of a variety of candidate programs which can be chosen for application composition. Hence, the efficiency of the AUGUSTA SBMS depends on the contents of the Software Base, and, therefore, on the competence of its users.

Another possibility for prototyping exists at the task level by supporting the generation of defaults for program stubs. Thus, the SBA can test new generics by instantiating default specializations, too.

Multi-User and Teamwork Support. *AUGUSTA* was designed for multi user development. Furthermore, the built-in process model enforces teamwork by supporting different categories of users (SBA, Programmer, Application Expert). Each of these categories can consist of more persons. A client-server architecture in combination with a simple transaction concept protects against inconsistent modifications. Integrity can be guaranteed by automatic checks regarding constraints expressed within the layered integrity model of *AUGUSTA*.

Every user disposes of a private workspace where programs and applications can be developed locally. However, all software components are visible for all users because potential reuse should not be artificially inhibited.

Required User Profiles. In order to benefit as much as possible from the *AUGUSTA* role concept, the following technical skills are suggested for members of the different user categories:

The Software Base Administrator should dispose of sound software engineering experience. Furthermore, practice in Ada programming is required for the task category management whereas domain knowledge is necessary for the application category management.

The Programmer activities require practice in Ada programming in combination with experience in structured programming.

The Application Expert is expected to have expert knowledge within the application domain. Additionally, he should be able to read data and control flow diagrams.

A DEMONSTRATIVE EXAMPLE

The following example should illustrate the usage of the *AUGUSTA* approach within the domain of a process control system of an electric power plant.

Considering a process control system, we can identify several possible application categories, e.g. measurement, statistical reporting, plant maintenance, and specific feedback control systems.

In the context of a feedback control system (FCS) iterative control and adjustment actions take place. These actions concern the actual values of a given measurable property where future values are to be influenced according to particular set points.

Boiler temperature control in steam power plants, reservoir level regulation and turbine rotation speed control in hydro-electric power plants are just some of the possible applications of a FCS. Each of these applications could be considered an instance of a generic application lattice. A very simplified form of this application lattice is represented in Figure 8.

The system starts with an input task measuring the actual value, identifying the appropriate set point as well as terminating the system on demand. An additional task of adjustment will be necessary, if the actual input value requires further treatment (e.g. aggregation, filtering) in order to yield the definite value which finally is of interest for control. Soon after availability of this accurate value a threshold value control has to take place. If the threshold value is reached or even exceeded the system has to react with an alert. Afterwards the actual regulation takes place. The kind of regulation behaviour is application specific because it depends on the particular property to be controlled. In our case we identify three possible actions: proportional control action (P), proportional plus integral control action (PI), and proportional plus integral plus derivative control action (PID). The comparison of the actual value and the set point as well as the calculation of the required action take place within these three task categories whereas the regulation action itself will be performed within the subsequent output task.

Within the application lattice of Figure 8 the stubs for `Input`, `TVC` (threshold value control), and `Output` are mandatory stubs. Additionally, the `Input` stub represents an example for loop control stubs expecting a boolean return value such that the number of actual loops can be decided at runtime. `Adjust` (adjustment) is an example of an optional stub. During application composition, the application expert can decide whether to refine this task category by choosing a program out of the possible candidate programs or to eliminate the stub without any substitution. `Alert` illustrates the case where a particular program was already chosen during lattice creation. This program will be component of all applications within the FCS application category.

AUGUSTA supports two different types of selection stubs, they differ in the time at which the decision is made in favour of a specific choice. The mandatory selection stub `TVC` is an example for the classical case of selections where the path to be executed will be chosen at runtime. The other kind of selection stubs allows for higher flexibility during application composition by enabling the application expert to chose out of several alternative paths for application generation. In our example the actual type of control action (`P`, `PI`, `PID`) used for each particular feedback control system depends on the decision of the application expert during system composition.

Given the already mentioned three example applications, a part of the program classification matrix for this application category could look like as shown in Figure 9 where column identifiers correspond to task categories and row identifiers characterize applications. The programs within each task category have a common origin but can slightly differ according to various aspects. Thus, the set points identified within `input` tasks could either be constant or adjustable values. Programs of each of the three `regulation behaviour` task categories could use different constants or result in different behaviour dependent on the kind of limit violation. `Output` tasks can trigger different actions or require different output formats.

SUMMARY

This paper presented a concept and 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. Furthermore, a specific process model is supported in order to take advantage of reuse by integrative system development with increased end-user involvement.

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.

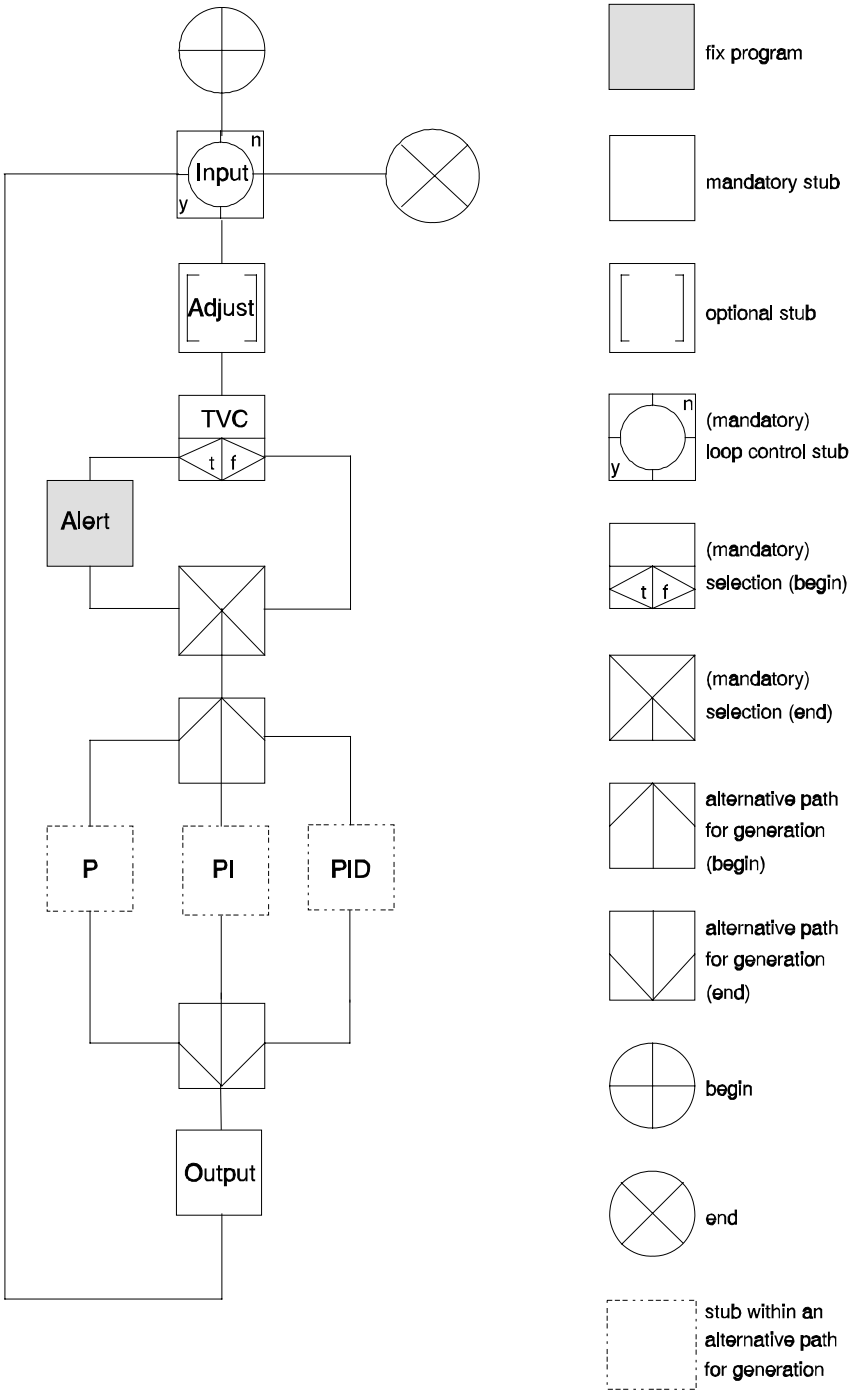

FIGURE 8: APPLICATION LATTICE 'FEEDBACK CONTROL SYSTEM'

	TASK CATEGORY							
APPLICATION	input	adjustment	threshold value control	alert	regulation behaviour P	regulation behaviour PI	regulation behaviour PID	output
boiler temperature control	constant set point monosensory polling	noise reduction	immediate threshold control	prescribed alert procedure	-----	boiler temperature	-----	burner adjustment
reservoir level regulation	adjustable set point multisensory polling	noise reduction & multisensory average	interval threshold control	prescribed alert procedure	reservoir level	-----	-----	weir flap setting
turbine rotation speed control	constant set point interrupt driven sensor	-----	immediate threshold control	prescribed alert procedure	-----	-----	rotation speed	guide vane setting

FIGURE 9: PROGRAM CLASSIFICATION MATRIX 'FEEDBACK CONTROL SYSTEM'

ACKNOWLEDGEMENT

Many thanks to Prof. Mittermeir for his fruitful comments on former versions of this paper. Additionally, the author wishes to thank Mr. Erich Buchacher and Mr. Michael Dobrovnik for their technical contributions as interviewees with respect to the demonstrative example.

The work on this paper was performed while the author was on leave at Österreichische Draukraftwerke AG, which lead her to pick out the demonstrative example from the domain of electric power-supply companies.

REFERENCES

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

Caldiera, G., and Basili, V.R., 1991, "Identifying and Qualifying Reusable Software Components", *IEEE Computer*, Vol. 24, No. 2, pp. 61-70.

Chen, M., and Norman, R.J., 1992, "A Framework for Integrated CASE", *IEEE Software*, Vol. 9, No. 2, pp. 18-22.

Dusink, E.M., 1992, "Reuse is not done in a vacuum", *Proceedings, 5th Annual Workshop on Institutionalizing Software Reuse*, Palo Alto.

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

Hochmüller, E., and Mittermeir, R.T., 1993, "Rahmenbedingungen für erfolgreiches Software Reuse", *Proceedings, Der Wiener IT-Kongreß 1993 "Informations- und Kommunikationstechnologie für das neue Europa"*, ADV, Vienna, pp. 269-284.

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

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

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

Prieto-Diaz, R., 1991, "Making Software Reuse Work: An Implementation Model", ACM SIGSOFT, *Software Engineering Notes*, Vol. 16, No. 3, pp. 61-68.

Tracz, W., 1990, "Where Does Reuse Start?", ACM SIGSOFT, *Software Engineering Notes*, Vol. 15, No. 2, pp. 42-46.