A PROPOSAL FOR AN INTEGRATED PROGRAMMING ENVIRONMENT FOR CHILL

T. Mehner, R. Tobiasch, J.F.H. Winkler

Siemens AG Munich
Federal Republic of Germany

ABSTRACT

For the efficient construction of large programs for telecom applications the software engineer must be supported by an integrated set of tools. Such a set of tools is called an integrated programming environment. In this paper a proposal for such an environment for the construction of large CHILL programs is presented. We concentrate on those phases of the software lifecycle in which programs are manipulated and deal especially with the requirements for large program families with revisions and variants that are typical for the telecom field.

1 INTRODUCTION

When developing, producing, and maintaining stored program controlled digital exchange systems the application of software engineering methods becomes more and more important, since these systems are characterized by some strong requirements concerning the system's availability, reliability, lifetime etc. When establishing the process of developing, producing, and maintaining such systems these requirements must be taken into consideration.

The proposed integrated programming environment (CHEKS) provides a tool set supporting this process. CHEKS stands for "CHILL Entwicklungs- und Korrektursystem", the (German) name of the proposed environment.

The main objectives of CHEKS are oriented towards

- the high quality of telecommunication software
- the consistent evolution of this software during its entire life time
- the enhancement of productivity.

The high degree of quality can be guaranteed by the validation facilities provided by CHEKS. The components' consistency, which is essential to the evolutionary approach, is controlled by the configuration management component based on the program family concept. The enhancement of productivity is achieved by providing automated process execution and by checking access rights to reduce inconsistencies and handling errors. Furthermore some functional redundancy will be reduced by re-using results once computed within the development process.

With respect to the objectives both the integration and the synergism of the tool set's components are essential.

The process, on which CHEKS is based, follows Boehm's waterfall-process-model [Boe 76]. Despite the fact that it does not have the theoretical base, the conceptional unity, and integrating framework that are essential to the achievement of process coherence over the entire life cycle of a program, it is widely accepted and well established in the software community. CHEKS mainly supports

- the design stage using the modular programming facilities and the concepts of separate and independent compilation provided by CHILL
- the implementation stage using facilities for programming in the small provided by CHILL
- the integration stage - which is in some sense reverse to the design stage - by using the structure information provided by the specification parts of a program accessible via the configuration description introduced below (see fig. 1, fig.2).

2 FAMILIES OF CHILL-PROGRAMS

Programs for telecommunication applications have the following important characteristics:

ch1) they are very large (10^5-10^6 LOC)

ch2) they are in operation for a long period of time (up to 30 years)

ch3) they are not simply single programs but rather groups of related programs, because every telephone exchange is normally uniquely tailored to its environment.

A typical example of such a program is the control program of a computerized telephone exchange. We call such a control program in this paper an "application program system" (APS) [Sie 81].

The programming language and the programming environment used for the construction of such an APS must have such properties that allow the construction of programs having the characteristics mentioned above.

In this section we deal with the characterization of program families and with the language constructs of CHILL which facilitate the construction of program families.

Since the APS are very large (ch1) they must be modularized in order to be manageable. This modularization must allow for the structuring of the whole program into manageable program units (PU) with well defined interfaces between those program units [Win 84].
The program units of CHILL are the module and the region.

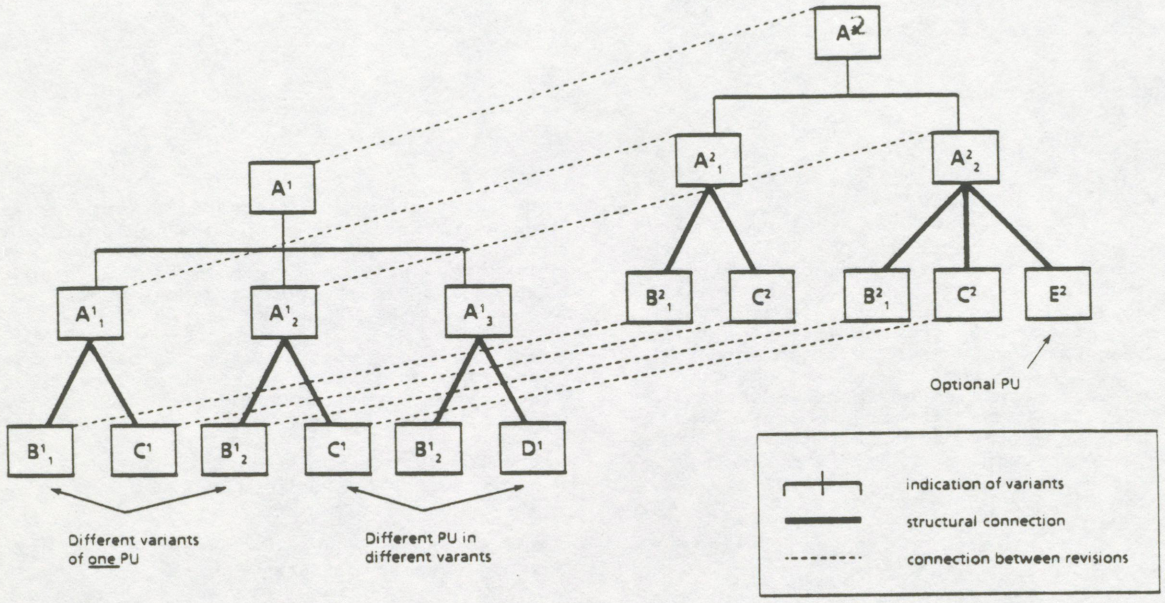

Fig 1. Revision oriented view of a PFRV

Since the APS are in operation for a long time they will evolve in time leading to different versions of the whole APS or of some program units.
In the following we distinguish two kinds of versions : the revisions and the variants.

Revisons evolve in time during further development and maintenance of a PU [Tic 82].

The main reasons for such revisions are

- new functional requirements;
- correction of errors;
- changes in the environment (espec. increasing the capacity of an exchange).

The subsequent revisions evolve in time but several of them may exist at one point in time because different exchanges may be equipped with different revisions of the APS.

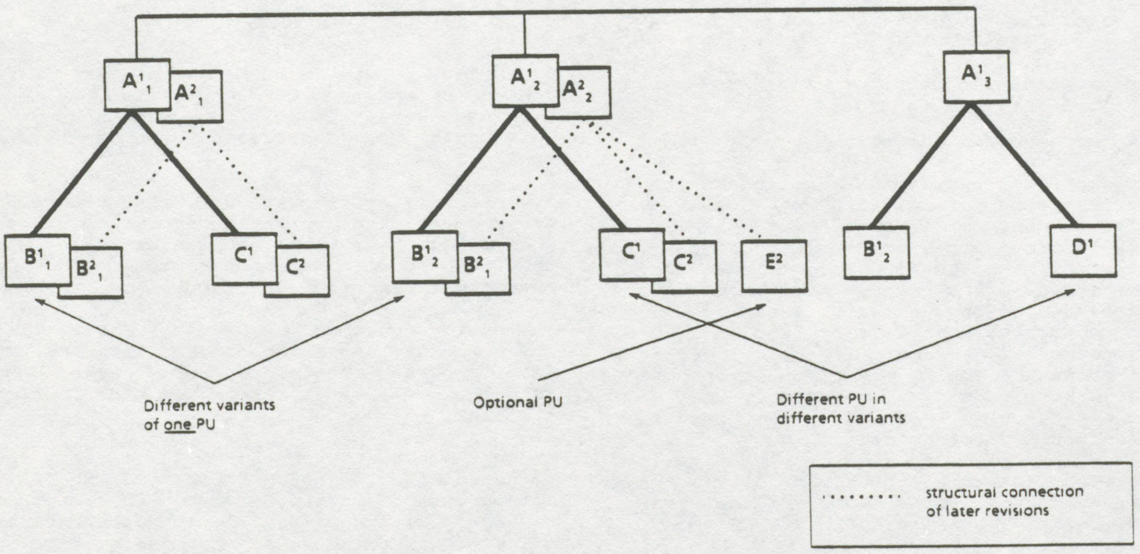

Fig 2. Variant oriented view of a PFRV

Exchanges located in different places usually differ in their functionality, capacity etc. especially if they are located in different countries. These differences lead to different variants of the APS.

We call a related set of programs with such revisions and variants a program family with revisions and variants (PFRV) [Win 77]. The typical structure of such a PFRV ist depicted in figures 1 and 2.
According to the two sorts of PUs the whole family can be organized in two different ways: revision oriented and variant oriented. The first of these possibilities is depicted in Fig.1 whereas Fig.2 shows the same family using the variant oriented view. This family consists of two revisions A^1 and A^2, where A^1 has three variants (A^1_1, A^1_2, A^1_3) and A^2 has only two variants (A^2_1, A^2_2).
In both figures the upper index indicates the revision and the lower index the variant of a PU.

Fig.1 and 2 show three typical situations which give rise to program variants :

- different alternatives for one PU : B^1_1 and B^1_2
- replacement of one PU by another : C^1 and D^1
- addition of a further PU : E^2

3 ENVIRONMENT AND STRUCTURE OF CHEKS

3.1 THE ENVIRONMENT OF CHEKS

CHEKS is designed to be an open system, since all internal interfaces are common to all components. CHEKS in the current stage represents a "MAPSE-class" functionality [DoD 80].

CHEKS does not cover all phases of the software life cycle. Especially both requirement phase and sections of the design phase are not yet supported. Nevertheless the gap between a SDL-tool, which matches these phases, and CHEKS will be bridged over depending on SDL's status and progress.

Beyond the development process of software itself the aspects of validation and documentation are essential to the software engineering discipline. The validation part of CHEKS currently contains

- the checking of consistency and completeness of the program family's design
- the analysis of syntactic and semantic correctness of both the design and the implementation of the program family's components.

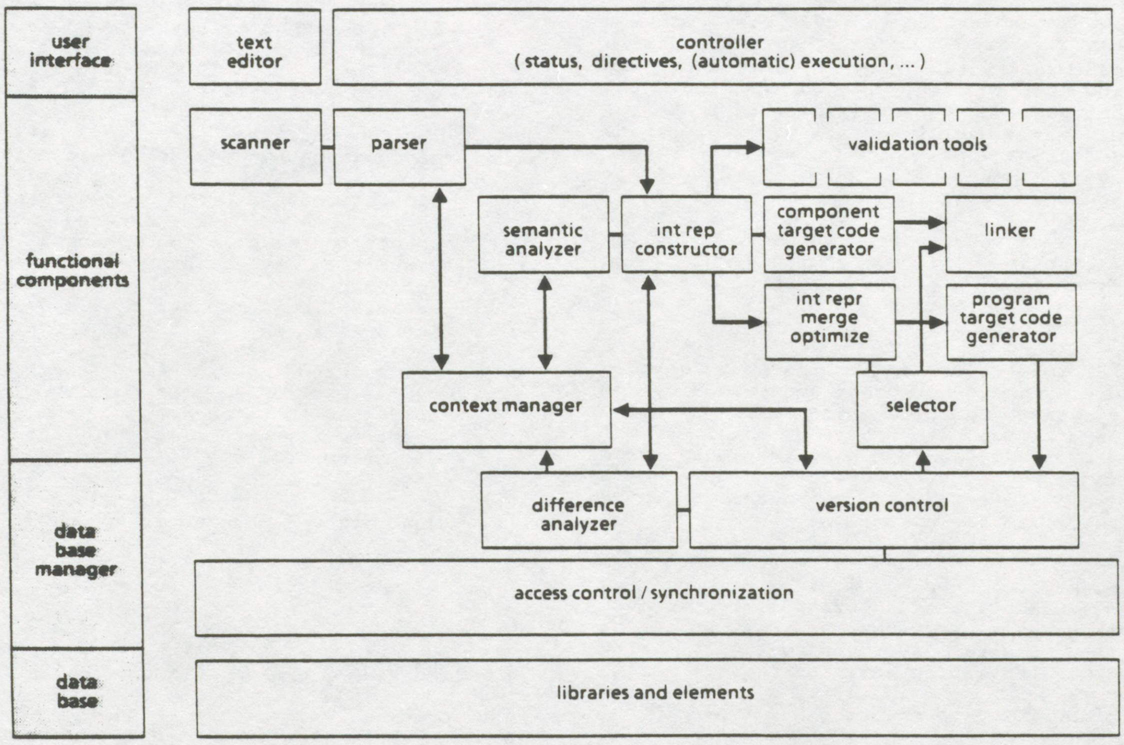

Fig 3. Coarse Structure of CHEKS

An integrated test bench is yet lacking the validation part. Documentation system facilities based on the description of program units will be applicable in connection with the configuration management component of CHEKS.

3.2 THE STRUCTURE OF CHEKS

The logical structure of CHEKS is oriented both towards the user's view and the functional view. With respect to the user there are four different layers characterizing the coarse structure of CHEKS:

(1) CHEKS' user interface

It provides facilities to identify users and attaching access rights and authorizations to them. Beyond this the user interface establishes common procedures to initiate automatic execution of complex activities, e.g. compiling sequences of program units, program production etc., and to get CHEKS' status information.

(2) CHEKS' functional components

This layer consists of the functional set of operations and tools manipulating the data base structure and its objects. This set of functions is discussed in more detail in the following sections.

(3) the data base manager

Depending on the user's access rights and authorization the data base manager establishes a user's personal domain realized by libraries as introduced in the next chapter. It consists of all private and shared objects, of which the user is owner. In addition references to shared objects, which belong to other users, are elements of the user's personal domain.

The data base manager provides mechanisms for controlling and synchronizing access to shared objects.

(4) the data base

All information and components created during the development and life time of a program family are stored in a common data base. Storage and retrieval of elements is performed by the data base manager. The structure of the data base and its elements is introduced in more detail in the next chapter.

Considering the functional view one must keep the underlying process model in mind: As pointed out above the development, production, and maintenance process is divided into different stages and activities. CHEKS supports the following activities:

(1) programming in the large

When designing an APS characterized by the properties shown above partitioning and refinement are essential. Specifying the skeleton of the APS by different layers and a hierarchy of library (sub-) units' interfaces is well supported by CHILL. Identifying different revisions and variants, which is essential to any kind of system's evolution, is realized by means of the program family concept.

(2) checking external references

The validation completeness and correctness of external references is performed by the context manager. It is most important for guaranteeing the system's consistency property. It is performed by checking whether granted and seized entities match or not. In case of program updates, i.e. revisions, the context manager delivers the program units which depend on this update. In addition it can decide whether these program units have to be recompiled in an appropriate order, updated, or changed, too.

(3) programming in the small

Obviously CHEKS contains components both for designing the internal structure of local data and functions of the program units, and its implementation and coding. Creation and update is performed by a common text editor. Syntax driven incremental program creation is not yet planned. After checking the syntactic and semantic correctness, which is part of the compilation process, the program unit is translated into the intermediate representation based on the abstract syntax tree. This intermediate representation is the common interface to all tools and functions available within CHEKS. The first step toward a complete unit's test bench is the existing symbolic debugger.

(4) program synthesis

The production of an APS can conceptionally be performed in two different ways: Following the more conventional way one has to generate target code for each program unit, to select the relevant components, and to link them together. The alternative approach - which is fundamental to optimization and testing beyond unit boundaries - is characterized by selecting appropriate components represented by their intermediate representation in accordance with the configuration description, linking them together by performing tree operations, and generating highly optimized target code for the whole program.

(5) validation / documentation

There are no facilities yet available within CHEKS.

4 LIBRARY AND LIBRARY ELEMENTS

All program components belonging to one program family are contained in the data base. This data base may be partitioned physically into one or several libraries. The libraries can be organized according to the people working on this program family.
The database of program components is furthermore structured in a logical way according to the structure of the PFRV (ref.

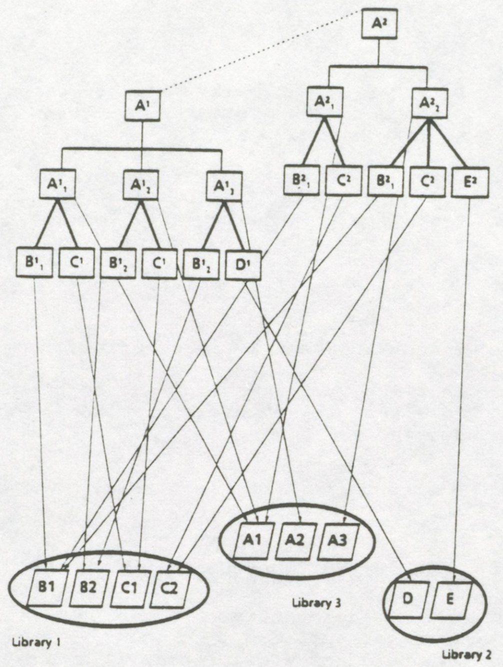

Fig 4. Logical description and physical database

Fig.1 and Fig.2). The structure depicted in these figures can be conceived as a pure configuration description [DK 76, LS 83] which is superimposed over the pool of physical program components. This is depicted in Fig.4. In this figure several typical situations are demonstrated :

- corresponding components in different revisions may be identical, ie a revision may only revise some of the components of the whole program. In Fig.4 the PUs B^1_1 and B^2_1 are implemented by the same program component B1.
- different variants of a PU may be implemented by the same program component. This is possible if the subcomponents only give rise to the variants. In Fig.4 the PUs A^1_1, A^1_2, and A^2_1 are all implemented by the same program component A1.

Each library has one owner, who determines the access rights other users have with respect to this library.

4.1 LIBRARY ELEMENTS

In the following list we give typical kinds of library elements.

- program components in source form
- program components in intermediate form
- program components in object form
- symbol table information possibly structured according to the name spaces
- configuration descriptions (description of revisions and variants)
- results of analyses of program components
- test data and test results
- libraries
- references to elements of other libraries
- documentation
- data of development and production
- etc.

The library does not only contain program components but also a lot of related information. Most of this information can be related directly to the program components represented in the lower part of Fig. 4. This leads to a structure where the boxes representing program components are themselves structured into several components as eg source program, object program etc.

Note that libraries can also be elments of libraries, ie we may have an arbitrary hierarchy of libraries. This makes the introduction of different kinds of libraries unnecessary as eg main library, sub library etc. The libraries of such a hierarchy must not all reside in one computer. They can be arbitarily distributed. This means that CHEKS can also support multi-site development.

4.2 RELATIONS BETWEEN LIBRARY ELEMENTS

A number of different relations exists between the library elements mentioned in the preceding section. The most important of these relations are given in the following list.

- program component <-> symbol tables
- program component <-> contained program component
- source form <-> intermediate form
- intermediate form <-> object form
- program revision <-> program variant
- PU variant <-> PU variant
- library <-> elements
- module <-> spec module
- module <-> context specification
- (program comp.,test data) <-> test result
- program component <-> documentation
- etc.

With respect to the completeness of these relations a library can be in different states. We call such states consistency states of the library. Examples for such consistency states are:

- source completeness : no unsatisfied references on source level;

- object completeness : to each program component in source form exists one component in object form;
- object consistency : object completeness and all object components are valid.

The libraries are managed by a library manager. This manager realizes the access to and the relations between the library elements. The elements themselves may be stored in a database or in files of a conventional file system. In the latter case access to such file should only possible via the library manager.

5 OPERATIONS ON LIBRARIES AND ELEMENTS

According to the structure of libraries there are two kinds of operations : (a) operations on libraries, and (b) operations on library elements that are not themselves libraries.

The operations on libraries are mostly of an organizational nature whereas the operations on library elements are the typical tools of programming environments.

Examples of operations on whole libraries are:

- Create a new library

  This operation creates a new library and defines the owner and the name of the library. A newly created library is initially empty.

- Delete a library

  This operation can be applied only by the owner of the library to be deleted.

- Mark a library as the current library

  Further operations use the current library if no library is explicitely given in their invocation.

- Open a library

  There are different open modes : exclusively for modifying the contents, and nonexclusively, if the user only wants to read the contents.

- Close a library
- List the table of contents of a library
- Join two or more libraries

  The program parts developed by several programmers can thus be put together into one library containing afterwards the whole program.

- Split a library in two or more libraries
- Check the consistency state of a library

Operations on elements are the typical manipulations of programs and program components in a programming support environment.
The main operations on elements are :

- Create element
- Delete element
- Edit element

  If the element is a source program component the editor will also have a syntax oriented mode.

- Translate : source -> intermediate form
- Translate : intermediate form -> object form
- Translate : intermediate form -> source form
- Execute element in intermediate form

  This will be performed by an interpreter for the intermediate language of the system.

- Pretty print a source element

6 IMPLEMENTATION ASPECTS

CHEKS generally follows the host-target-approach, where currently available Siemens mainframes will act as host system, controlled by the time sharing operating systems BS2000 and BS3000, resp. Target systems will be Siemens switching processors used for EWSD [Sie 81].

Host and target systems are not yet interconnected physically. With respect to diagnosis, test, and maintenance an interconnection - e.g. at data link level - would be essential. An advanced architecture (see below) would offer such facilities by selecting a component dedicated for diagnosis and maintenance.

CHEKS itself will be implemented using CHILL and the currently available tool set [Fei 83; Rei 83]. Although its current run time environment is determined, the implementation of CHEKS will be conceptually based on a virtual support environment to provide portability and evolution with respect to advanced architectures. The virtual support environment consists of

(1) a virtual terminal to establish the user interface according to the periperal equipment (high resolution <u>vs</u> character oriented screen, etc.)

(2) a set of services which allow an implementation independent of specific operating system functions, especially using the new CHILL-IO as defined in [MW 84].

The specificationen of implementation dependent interfaces will be influenced by new architectures proposed for development environments. This advanced architectures are characterized by a set of personal computers connected either by a mainframe, which offers central services, or by local area networks, where data are distributed multi site and services may be decentralized.

7 REFERENCES

Boe 76 Boehm, B.
Software Engineering
IEEE Trans. on Computers 25,12(1976)1226..1240.

DK 76 DeRemer, F.L.; Kron, H.H.
Programming-in-the-large versus Programming-in-the-small
= [NS 76: 80..89].

DoD 80 United States Department of Defense
Requirements for Ada Programming Support Environments "Stoneman"
Washington D.C., February 1980.

Fei 83 Feicht, E.J.
CHILL-factory: production and maintenance of a large CHILL software system
Proc. 5th Intern. Conf. on Software Engineering for Telecommunication Switching Systems, Lund 1983, p.98..103.

LS 83 Lampson, Butler W.; Schmidt, Eric. E.
Practical Use of a Polymorphic Applicative Language
= [POP 83: 237..255].

MW 84 Mehner, T.; Winkler, J.F.H.
An Implementation of the New CHILL I/O
This volume.

NS 76 Schneider, H.-J.; Nagl,M. (eds.)
Programmiersprachen - 4. Fachtagung der GI
Springer, Berlin etc. 1976.

POP 83 POPL 83 - Tenth Annual ACM Symposium on Principles of Programming Languages
ACM, New York 1983.

Rei 83 Reithmaier, E.
Compilation Control in a large CHILL Application
Second CHILL Conference, Lisle Illinois, March 1983, p.111..120.

Sie 81 Siemens AG
EWSD Digital Switching System
telcom report, Vol.4 (1981) Special Issue.

Tic 82 Tichy, Walter F.
Adabase: A Data Base for Ada Programs
AdaTEC Conference 1982 p.57..65; ACM, New York 1982.

Win 77 Winkler, J.F.H.
Beschreibung und Realisierung von Programmfamilien
Jahresbericht der Fakultät für Informatik an der Universität Karlsruhe 1977, p.188..197.

Win 84 Winkler, J.F.H.
The Realization of Data Abstractions in CHILL
This volume.

CCITT CHILL

THIRD CHILL CONFERENCE

CAMBRIDGE UNIVERSITY

ROBINSON COLLEGE & UNIVERSITY CENTRE

September 23-28 1984

CONFERENCE PROCEEDINGS

Sponsored by:

ITT Europe Engineering Support Centre

ITT