Creation of a Family of Compilers and Runtime Environments by Combining Reusable Components

Christian Weber

Siemens Nixdorf, Otto-Hahn-Ring. 6, 8000 Munich, Germany

Abstract: When confronted with the requirement to supply language processors for a wide range of languages, hardware architectures, and operating systems, the conventional approach to software reuse - decoupling language specific front ends from hardware specific code generators by some common intermediate representation - proves to be insufficient. A larger set of decoupling interfaces can be defined such that all language processor products, compilers as well as runtime environments, can be created by a simple linking process out of components which have at most one dependency: either on the source language, or on the hardware, or on the operating system. It will be shown that the development costs for a whole family of language processor products can thus be reduced considerably.

1. Introduction

1.1 Description of the Problem

As a major vendor of computer systems, Siemens Nixdorf has to supply language processors, i. e. compilers together with their corresponding runtime environments, for a large range of languages, for various hardware architectures, and for two unrelated operating systems: SINIX®[1] and the mainframe system BS2000®[2]. In order to cover this range of products with limited resources, Siemens Nixdorf established about 5 years ago a common intermediate representation called ULS (Universal Language System) which allowed to compose all compilers out of three independent parts: a front end to map the source language to ULS, a language and hardware independent optimizer to optimize the ULS representation of the source program, and a back end to map the ULS program into machine code.

It had to be observed, however, that several problems still remained:

- Each change within the operating system environment (such as the introduction of the new object format ELF in UNIX®[3] or within the tool environment (such as the introduction of a new global cross-referencing tool) caused extensive adaptation activities in many of the products.
- The ULS approach only supported the rationalization in the development of the compilers. The runtime environments still had to be adapted for each product with each major change of the hardware architecture or the system.

1.) SINIX® is the UNIX®[3] of Siemens Nixdorf Information Systems.
2.) BS2000® is a registered trademark of Siemens Nixdorf Information Systems.
3.) UNIX® is a registered trademark of UNIX System Laboratories.

- The ULS did not even fully specify the decoupling of the language and hardware specific parts within the compilers. Additional interfaces for options transfer, message output, debugging information, etc. were needed especially if the different parts of the compiler were developed at different sites or if the back end was bought directly from the hardware supplier.

It was therefore decided that a more complete set of decoupling interfaces was necessary.

1.2 Solution Approach: Definition of a Component Architecture with Decoupling of the Language, Hardware, and Operating System Specific Parts

About two years ago we started the design of a component architecture for our language processor products which we called SNAP (*S*iemens *N*ixdorf *a*rchitecture for language *p*rocessors). SNAP will, as we believe, provide a high degree of code reuse, and will allow the distribution of development tasks to sites specializing in some particular aspect of language processing. Our approach (figure 1) was as follows:

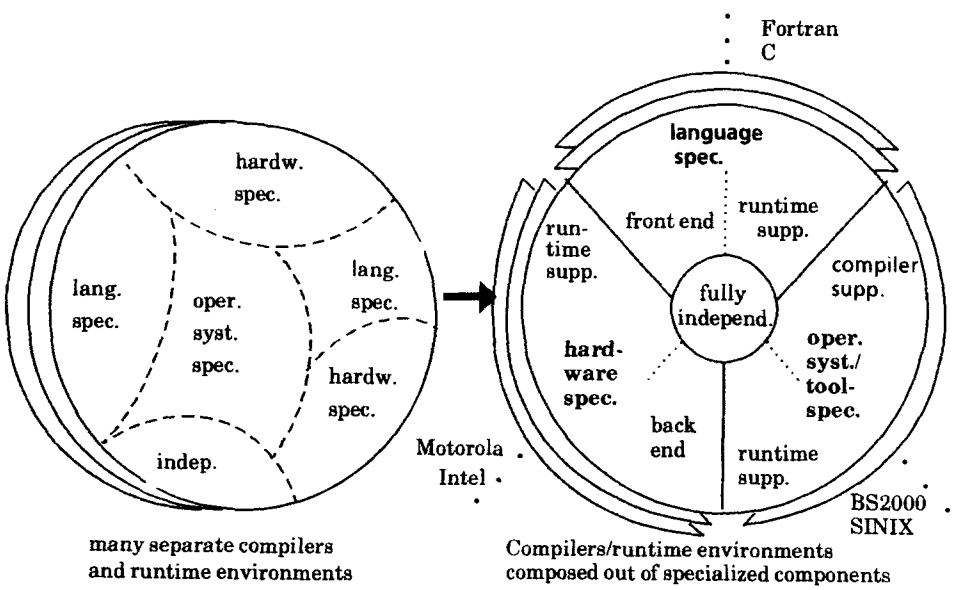

Figure 1: Product Structure vs. Component Structure

- We identified all dependencies on source language, hardware, operating system, and external tools within our most important products: the language processors for C, C++, COBOL, Fortran90, Extended Pascal, Ada, BASIC, and SPL (a PL1 dialect used internally). We believe, though, that our approach could easily be extended to all other procedural languages.

- We defined a set of components, sufficient to cover the functions of all the language processors mentioned above, such that each component had at most one of the possible dependencies: either on the source language, or on the hardware, or on the operating system, or on the tools.
- We further tried to define as many independent components as possible (i.e. components without any of the possible dependencies).

In fact, the interface of the hardware-dependent components, which includes the intermediate representation ULS as the main interface to the back ends, can be considered as a virtual hardware, and the interface of the operating system specific components can be considered as a virtual operating system which we called PROSOS (=*pro*gramming language *s*upport *o*perating *s*ystem).

In the following, the components sharing the same dependency will be referred to as *component groups,* interfaces within one component group will be referred to as *private interfaces*, and interfaces between different component groups as *SNAP interfaces.*

Note that the interface between the objects generated by the compilers and the runtime system are *not* SNAP interfaces: the language specific front end may always generate (within ULS) a call to its language specific runtime system (by a private interface) instead of expanding the corresponding function by ULS terms, and the code generator may always generate a call to its (private) hardware specific runtime system instead of expanding the corresponding hardware instructions.

1.3 Related Work

The decoupling of front ends, optimization, and back ends by some common intermediate representation is a well-known method (cf. e.g. [1]) to achieve code reuse for compiler components. The SNAP approach extends this method

- by identifying and decoupling not only the hardware and language specific parts of the compilers, but also the operating system and tool specific parts, and
- by including the runtime environments into this reuse architecture.

Some individual aspects of this decoupling task have been addressed previously: [2] discusses the separation of the debug information generation from the main compilation process, and [3] describes the decoupling of certain special features from various language specific runtime environments. There apparently does not exist, however, any interface architecture yet which tries to solve the full decoupling problem as described above.

Further rationalization may be gained within each SNAP component group: e.g. the language specific front ends may be partially generated from a formal language description ([7]), or the back ends may be generated from hardware

description tables ([1], [7]). The discussion of such techniques, however, is beyond the scope of this paper.

In section 2 the guidelines which governed the interface design are described, and section 3 gives an estimate for the cost and benefits of SNAP. An overview of the SNAP components is given in section 4, a more detailed description is contained in section 5. In section 6 a report is given about the current status and first experiences.

2. Guidelines for the Design of SNAP Interfaces

The design of the SNAP interfaces was guided by the following principles:

- The interfaces must not have any dependency on language, hardware, operating system, or external tools.
- Each interface must be implementable by a component which has at most one of the dependencies mentioned above (but which may call any other architecture interface to perform its service).
- The performance degradation (e.g. by additional calling layers) must not exceed the following limits:
  * the compilation performance must not be degraded by more than 70%,
  * the execution performance of runtime system parts accessing services of the operating system must not be degraded by more than 10%,
  * the execution performance of the generated object code must not be degraded at all,

  in comparison to a product specially tailored to every combination of language, hardware, and operating system.
- The interfaces should be as narrow as possible (i.e. a new interface should not be offered if the corresponding function can be gained by a combination of services already available).
- If redundant interfaces are introduced because of conflicting requirements, they should be described in a separate document to keep the main architecture description as simple as possible.

To give a non-trivial example for such a conflict of requirements: the basic read/write operations of PROSOS could be just byte-oriented since any record-oriented I/O (demanded e.g. by Fortran, COBOL, ...) could always be mapped to byte-oriented I/O. In mainframe operating systems such as BS2000, however, the native I/O of the operating system is record-oriented, forcing PROSOS to map any byte-oriented I/O to record-I/O again. To avoid a loss of performance we therefore decided to offer both record- and byte-I/O within PROSOS.

3. Estimates of the Costs and Benefits of SNAP

The cost of SNAP is basically some loss of performance which we limited strictly (cf. 2) to stay competitive. There may also be some additional storage

requirements, but we did not particularly limit these since we do not offer products for very small systems.

The main benefit of SNAP will be a reduction of the total code volume of all our language processor products, and hence, as we expect, of development and maintenance costs. We estimated for a realistic set of products (48 language processor products for 10 languages, incompletely distributed across 6 hardware platforms and 2 operating systems) the possible gain caused by SNAP as follows:

	without any reuse	with reuse of only front end, optimization, and back end	with SNAP
total volume in millions of lines of code	9.7	3.4	1.6

A different calculation reviewed roughly the effort (measured in man years) spent for a particular product (FORTRAN77 in BS2000), in the last years to find out how much effort would have been saved if SNAP had been invented earlier:

	without any reuse (=actually spent)	with reuse of only front end, optimization, and backend	with SNAP
product specific costs	39	24	5
share of costs for common components	-	5	12
Total effort	39	29	17

In this calculation we assigned between 25% (for general adaptation activities to new features of the operating system) and 50% (for performance and precision improvements of the mathematical functions) of the costs for common components to the FORTRAN product.

This cost relation may not be representative for our whole product range, but it may be typical for a product in the maintenance phase when there is not much evolution in the language any more, but rather in the tools, operating system, or hardware architecture.

Other benefits which we expect from SNAP are:

- We can concentrate related development topics (e.g. the support of a new hardware by all products) within one organisational unit by assigning the responsibilites in correspondence to the component grouping. Thus, the distribution of development to different sites, and the acquisition of

expertise by the development staff (since concentrated to a certain area) will be facilitated.

- User programs in a mixed-language runtime environment will have a better interoperability due to the common use of SNAP components: the common storage management allows more economic storage administration, the common file handler creates the same physical file formats for each language, etc.
- The user interface will be more uniform between our whole range of products (e.g. one particular file or stream is always accessable by the same identifier).

4. Overview of SNAP

In this section an overview showing the most important control and/or data flow relationships is given for the major components which make up a typical SNAP language processor product. The components are marked according to their component group: *L* = language specific component, *HW* = hardware specific component, *OS* = operating system or tool specific component, *I* = independent component, *P* = product specific component where the particular dependencies cannot be isolated.
It should be noted that the dependency grouping of the components not necessarily matches their calling or data flow relationships.

4.1 Embedding of a Compiler into the Programming Environment

The *compiler* is invoked by the programming environment to compile one source which includes the output of various informations for the debugger or the environment. The transfer of options, e.g. the source file name, or defaults from the environment to the compiler is done via a *name manager* storing named strings to be retrieved individually and decentrally by the different compiler components. The internal name of each such option has to obey certain naming conventions: each component gets reserved some individual name space for its options.

In a stand-alone compiler, the programming environment is replaced by a few small components as indicated in figure 2: the *programming environment controller* may issue start and/or summary messages, calls the options processor, and calls the compiler. The *options processor* reads and/or analyzes the options given by a start command or batch file, or it handles a prompting dialog with the user. This component is specific to the operating system since the options syntax and the expected dialog support are entirely different in each operating system.

The options prompting and/or analysis is parameterized by *options tables* which describe the external syntax and/or dialog for the options and their mapping to the internal names handled by the name manager. The options tables are still specific to each compiler product since they may describe options or contain defaults for all components of the product.

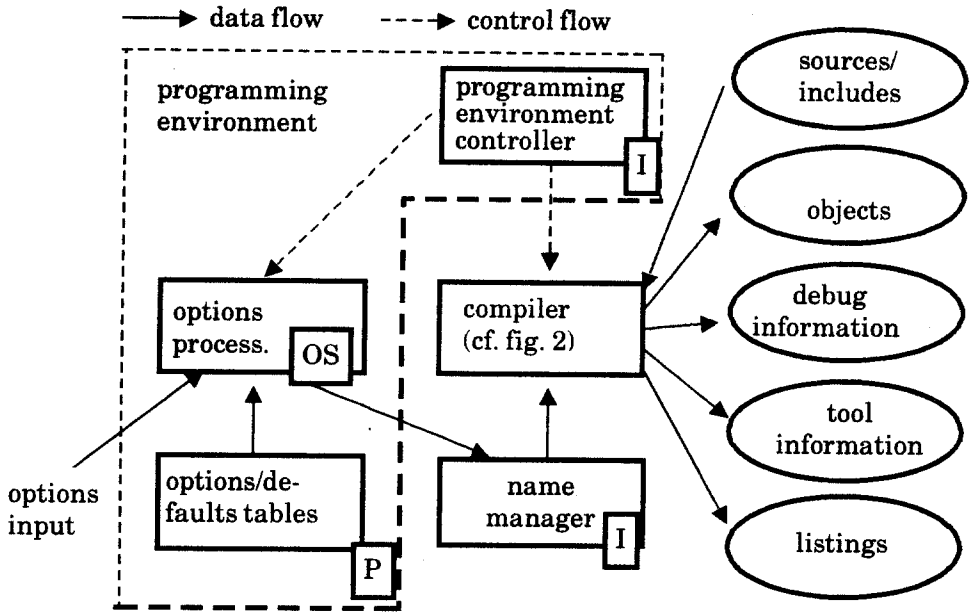

Figure 2: Embedding of a SNAP Compiler into the Programming Environment

4.2 Structure of a Compiler

The *compiler controller* calls - depending on the current options or error status - the different passes of the compiler.

The *front end* reads sources, includes, and possibly interface descriptions by use of PROSOS services and produces several outputs:

- It creates the ULS representation equivalent to the source program which may contain calls to the language specific part of the runtime system using interfaces private to the language specific component group. This mapping is parameterized by certain hardware parameters delivered by the back end which enable the front end to perform (structure-) relative address allocation and all constant expression evaluations or checks.
- It may write interface descriptions of the compiled modules which are deposited in a language specific format using PROSOS services. This data flow is not shown in figure 3 since the interface information is accessed only by the compiler itself and is therefore no interface to the environment.
- It produces information for the programming environment such as error messages, for listings and tools such as cross-referencing data, and for the debugger such as the mappings from source lines to statements, which is output using tool or debug information editing services.

The *middle pass* performs various restructuring tasks on the ULS code (e.g. hardware-independent optimizations).

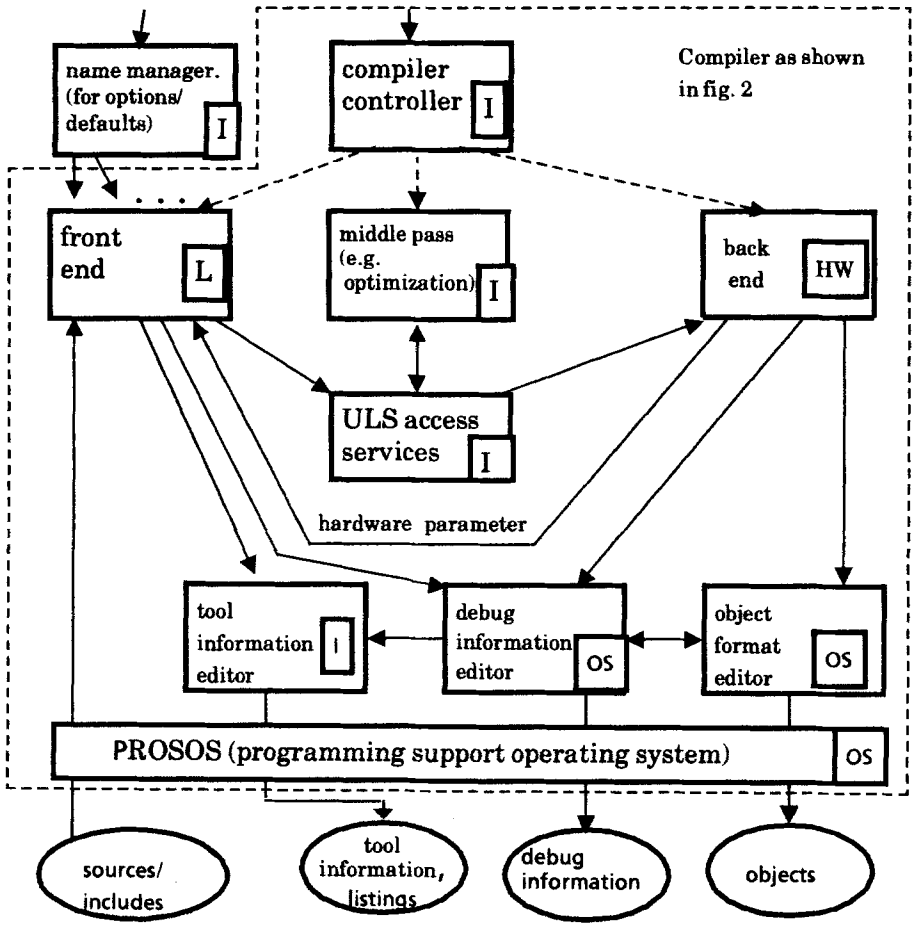

Figure 3: Structure of a Compiler

The *back end*

- maps the ULS program to the code of the target hardware, possibly with calls to the hard-ware-specific runtime system (instead of full code expansions), and outputs the code by interfaces of the object format editor;
- may write an object listing which is output by PROSOS services,
- adds information for the debugger (e.g. statement addresses) by interfaces of the debug information editor, and
- delivers certain parameters of its hardware to the front end.

The *tool information editor* collects information from the front end and produces files and/or data structures in a specialized format for the various

tools (data dictionary, configuration management, error correction dialog, ...) and/or generates various listings.

The *debug information editor* collects information separately from the front end (e.g. data types, ranges of definitions of symbols, names of symbolic constants, mapping from source lines to statements, statement types,...) and back end (addresses of data items / statements, possibly register tables for the debugging of optimized code) and combines these to tables for the debugging tools.
In principle, the debug information editor could be an independent component; in reality, however, the debug information formats are defined by the operating system and are strongly linked to the object formats (which are specific to the operating systems).

The *object format editor* maps some abstract object format interface to the object format demanded by the target operating system and writes the object code into a library by PROSOS services.

The *programming support operating system (PROSOS)* provides basic operating system services such as I/O, storage management, etc.

Each of the components stated above may retrieve options by a call to the name manager.

4.3 Structure of a Runtime Environment

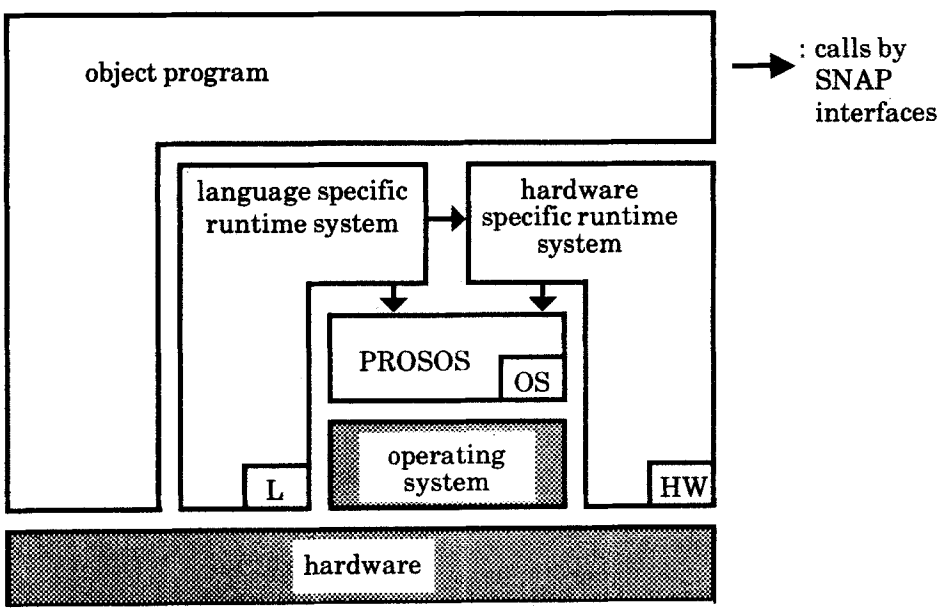

Figure 4: Structure of a Runtime Environment

The runtime system functions called by the generated *object program* are divided into a *language specific runtime system* (LRTS) and a *hardware specific runtime system* (HRTS). The calls to LRTS are generated by the front end within ULS using interfaces private to the language specific component group, and the calls to HRTS are generated by the back end using interfaces private to the hardware specific component group. The LRTS routines may also call HRTS services by SNAP interfaces.

Whenever services of the operating system are needed (e.g. some I/O service needed by LRTS or a new storage extent needed by the stack management within HRTS), PROSOS routines can be called.

5. Description of the Architecture Components

In this section, the main architecture components are described (with their main functions and/or interfaces) in an order corresponding to their dependency grouping and, hence, in an order suitable for the assignment to organisational units.

5.1 Language Specific Components

The language specific component group consists of the front end which maps the source program to ULS (cf. 4.2), and the language specific runtime system LRTS (cf. 4.3) which e.g. performs I/O-formatting, language specific exception handling etc.

Although these components are usually the biggest ones within a language processor, they offer very few SNAP interfaces:

- the front end has one interface for its invocation,
- the LRTS has one interface for its initialization (to be called by PROSOS if a multi-language environment is first initialized by a different language).

The main data flow is not handled by these interfaces, but by calls to the name manager for the retrieval of options or product parameters, by calls to PROSOS for source/include/interface input/output, by calls to the ULS services for ULS output, by calls to the tool information editor for output of various analysis results, and by calls to the debug information editor for debug information output.

5.2 Hardware Specific Components

The hardware specific component group consists of the back end which maps the ULS program to the target code (cf. 4.2), and the hardware specific runtime system HRTS (cf. 4.3).

The back end offers the following interfaces:

- one interface for its invocation;
- interfaces to deliver certain parameters of the target hardware which
  * define the mapping of the basic data types of all languages to the basic hardware types (signed/unsigned integer, float, pointer, ...),

* define the length, precision, and alignment attributes of the basic hardware data types, and
* describe the character code of the target system.

The main data flow is not handled by these interfaces, but by calls to the name manager for the retrieval of options or product parameters, by calls to the ULS services for the retrieval of ULS and the data/control flow information gained by the middle pass (and needed by the back end for optimizations), by calls to the PROSOS services for the output of the object listing, by calls to the debug information editor for the output of debug information, and by calls to the object format editor for the object output.

The HRTS offers (apart from private interfaces used by the corresponding back end for its code generation) the following services:

- conversion routines between hardware types and character strings,
- arithmetic functions (SIN, COS, ...),
- string manipulation/search routines which are hardware specific for reasons of performance,
- stack storage allocation functions,
- routines for the interpretation of a hardware context produced by an interrupt,
- routines for a *longjump* (c. [5]) backwards within the calling hierarchy.

5.3 Operating System/Tool Specific Components

The operating system specific component group consists of a general system embedding layer (PROSOS), the object format editor (cf. 4.2), and the options processing and prompting component (cf. 4.1) which can also be used for the analysis or prompting of runtime options.

The (only) tool/specific component is the debug information editor (cf. 4.2) which (for reasons given in 4.2) is at the same time specific to the operating system. For all other tools (so far) we have defined a system independent file format to store all information gained by the compilation process and needed by the tools: cf. 5.4.2.

The object format editor, debug information editor, and options analyzer have been described in section 4; we therefore describe only PROSOS in more detail here. The PROSOS services consist of

- Input/output services:
  Files are identified by a name (with system dependent syntax) or by a *link name* (with system-independent syntax) to which the user may assign (e.g. by SHELL/environment variables) a real file at runtime. PROSOS also supports the access to library members (necessary for compiler I/O) where library and member can be identified separately (by file name or link name). PROSOS supports byte- as well as record-oriented reading / writing, and supports sequential access (forward/backward), random

positioning to some file position which had been "current" at some time (similar to *fsetpos/fgetpos* of the C library, cf. [5]), random access by record or byte number, and index-sequential access.
The control character support offered for text I/O roughly corresponds to the functions of the C library (c. [5]).

- storage management:
  Functions are offered to request large chunks of storage, storage areas of arbitrary size (similar to malloc/free of the C library), and storage within heap areas which may be freed as a whole.
- environment information retrieval:
  Services are offered to retrieve environment information such as date, time, etc.
- exception handling:
  Interfaces similar to signal/raise of the C library are offered; the real hardware context is replaced, however, by a ULS oriented interrupt context.
- miscellaneous:
  Services are provided for initialisation and termination of a (multilingual) environment, for retrieval of message texts, and for (Ada-) multi-threading.

5.4 Components Without Dependency on Language, Hardware, or Operating System

We identified several independent components which are described in this section. They have nothing in common: therefore they might well be assigned to different organisational units.

5.4.1 ULS Access Services

The ULS access services implement the virtual hardware ULS as an abstract data type offering functions for reading and writing ULS tokens or for positioning within a ULS program. They further allow the storage and the retrieval of the ULS representation of programs into / from external libraries when needed for the inline expansion of external calls (performed by the middle pass).

The ULS is a low-level intermediate language at the level of an assembly language. Numeric constants are stored (by character literals) in a format which is independent of the target hardware. The (structure-relative) data allocation and the expansion of all access paths is laid down explicity in ULS and has therefore to be done by the front end. The ULS does not contain any information necessary only for debugging tools (i.e. names of data items/constants, user types, source line numbers,...: such information is handled by the debug information editor (cf. 4.2)); it only contains a flagging of

those data items or code positions whose addresses have to be reported by the back end to the debug information editor.

The ULS further contains various information for code optimization such as the boundaries of code blocks controlled by exception handling (where therefore code motion and many other optimizations are not allowed), or the results of the data / control flow analysis performed by the middle pass (to be used by the back end for target specific optimizations).
We decided to choose a (slightly) target dependent intermediate representation (as opposed to a fully independent one as OSF's ANDF: cf. [4]) for the following reasons:

- the intermediate language can be considerably smaller and simpler,
- the implementation of the front ends is easier if all constant expressions and checks (also those involving size calculations of data items) can be evalued immediately and need not be postponed,
- the back ends are not burdened by language specific tasks normally done by the front ends (e.g. checks for the correctness of some constant expression).

Therefore, for our requirement (rationalization of the compiler development) the ULS approach is more suitable, even though it does not cover the main requirement of ANDF: the establishment of a software distribution format which is independent of source language as well as target hardware.

5.4.2 Tool Information Editor

The tool information editor collects information from the front end by interfaces suitable for the structuring of a front end into different passes: a preprocessor may e.g. provide the names of the files actually read and the mapping of the input to the output lines, the lexical analysis may provide the mapping from lines and columns to language statements, the syntax analysis may provide the type of a statement or some cross-reference information, the semantic analysis may provide structure layouts or data attributes.
The information is then reordered and restructured to produce various specialized outputs:

- information for the error correction dialog (error messages together with the original erroneous source line and column, cross reference information),
- information for the configuration management,
- information for the data dictionary,
- information for various other tools (provided by some standard data base for compilation results),
- possibly compiler listings (source, cross reference, ...).

Information which is needed both by the tool information editor and the debug information editor has to be provided only once by the front end: the debug information editor will pass the relevant information to the tool information editor.

5.4.3 Name Manager

The name manager offers access (storage / retrieval) to named values; it is used mainly for the transfer of options, defaults, product parameters, or general compilation results.

5.4.4 Middle Pass

The middle pass modifies a program at the ULS level; possible activities are:

- inline expansion of procedure calls (if the ULS representation of the called routine is available),
- data and control flow analysis and target-independent optimization,
- vectorization and / or parallelization,
- generation of code for the collection of various run time data (test coverage, true/false-ratios or average number of loop cycles for a better optimization,..).

5.4.5 Controllers

The controllers for the programming environment of a stand-alone compiler or the compiler itself are described in 4.1/4.2.

5.5 Product-Specific Components

We still have some components which have to be adapted to each product (i.e. which are specific to each combination of language, hardware, and operating system):

- the options tables (c. 4.1), and
- procedures for the compilation and linking of the individual components to generate a full product.

6. Status and First Experiences

The overall design of the architecture is finished. The front ends for FORTRAN77, Pascal XT (some extended Pascal with Ada-like exception handling, cf.[6]), C, C++, and BASIC, and the back ends for /370 mainframe, National Semiconductor, Intel, Motorola, and Mips RISC processors have been adapted to SNAP. The operating system embedding (PROSOS, object format editing, debug information editing) for BS2000 and SINIX, and the middle pass (optimization) are finished in a first version.

Other SNAP components are in planning or development: front ends for COBOL, Fortran90, Ada; decoupled language specific runtime environments for COBOL, Fortran90, Pascal XT, Ada; a back end for vector processor support, decoupled hardware specific runtime systems especially for the mainframe architecture, an enhanced middle pass, and the tool information editing.

Our first experience indicates:

- The process of changing the structure of a whole product suite towards a component architecture is difficult, takes longer than expected and demands frequent adaptation of interface details. We have not encountered so far, however, any difficulties indicating that SNAP fails to work in any particular cases.
- The performance limits (cf. 2.) which have been set can be met, but occasionally require the addition of new, redundant interfaces to the architecture.
- The gain of productivity is hard to measure, but appears to be roughly within the expected range.

Furthermore, we found that SNAP enables us to react much quicker to market demands than before (since development topics for many products may now be concentrated within a small team), which gives rise to a better competitiveness.

Acknowledgements: I would like to thank my colleagues Wolfgang Frielinghaus, Wolfgang Hoyer, and Franz Karl Ress for their major contributions to the global architecture design, and to Franz Karl Ress, Manfred Stadel and George Baumann for their helpful comments on earlier versions of this paper.

References

[1] Tanenbaum, v. Staveren, Keizer, Stevenson: A Practical Tool Kit for making Portable Compilers,
Comm. of the ACM, Vol. 26, Nr. 9, p. 654-660 (1983)

[2] Atkinson, Demers, Hauser, Jacobi, Kessler, Weiser: Experiences Creating a Portable Cedar, Proceedings of the 1989 ACM Sigplan Conference, June 1989, p. 322-328

[3] Weiser, Demers, Hauser: The Portable Common Runtime Approach to Interoperability, Operating Systems Review 1989, Vol. 23, Nr.5, p. 114-122

[4] TDF Specification: Part I (OSF's ANDF-subset) and Part II (features beyond ANDF), Nov. 1991, available at: DRA Electronics Division, St.Andrews Rd., Malvern, Worcs WR14 3PS, UK

[5] American National Standard for Information Systems-Programming Language C, Doc. No. X3J11/90-013, ANSI X3 Secretariat, 1990.

[6] M. Stadel: "Compiler-Familie für Pascal" in H. Schwärtzel (Hrsg.): "Informatik in der Praxis",Springer, Berlin 1986, pp209-219.

[7] Leverett, Cartell, Hobbs, Newcomer, Reiner, Schatz, Wulf: An Overview of the Product-Quality Compiler-Compiler Project, IEEE Computer 13: 8(1980), 34-39.

Key Words and Phrases: compiler family, runtime environment, reusable software, compiler design, runtime system design