On Posteriori Integration of Software Tools

Dick Schefstrom

TeleLOGIC AB, Aurorum 1, 951 75 Lulea, Sweden.

Abstract.

This paper tries to define and examplify the concept of "integration" in the context of software development support environments. The benefits of tight integration are discussed, together with different approaches on how to achieve this property under the constraint of having to include existing software components and tools. The paper finally reports on the initial experience gained in building an Ada programming support environment according to those principles.

1. The Role of Integration

Much hope is today focussed on the use of tools as a way increasing productivity in software development, shown, for example, by the rapidly growing activity under the label "CASE". While a substantial market of tools, supporting various parts of the software lifecycle, indeed exist, few people would claim that a corresponding major productivity breakthrough has occurred.

In this paper, we will focus on what might be the major obstacle before substantial gains can be shown - the lack of integration. The basic argument is that the semantic loss and overhead imposed by moving between tools that don't know about, and contribute to, each other, is so big that it cannot be bypassed.

Another aspect of the problem is that the tasks performed during software development never are so phased and isolated from each other, as the body of existing tools seem to imply. Infact, software development is never is truly "automatic", and the role of tools is therefore most often in providing the user with the information he needs, and doing so with short response time and without requiring him to engage in errorprone or context breaking dialogues with different other tools. The information needed often spans over phases, obvious examples being the need for accessing requirements documents, and design descriptions, in direct connection to programming and maintenance.

In building support systems, the approach has often been to concentrate on selecting the "right" functions, quite independently of each other. While this decomposition-oriented approach is easier to deal with, both from a technical and project administrative point of view, it usually fails to recognize the large potential of cross-service cooperation, and the obvious fact that the value and usefulness of a service always is relative to its cost, the latter which has three components:

(1) The cost of activating a service in terms of finding it and providing the right parameters.

(2) The cost, in waiting time, for the user while the computer is working.

(3) The cost of finally utilizing and assimilating the result of the service in the context where the user needs it.

We call the sum of those costs the *turnaround time*, and the purpose of our work is to minimize it.

Observe that we discuss not only the turnaround time of currently identified atomic services, but the minimization of the cost for any typical pattern of work. For example, if the looking up of an imported interface is a common task, one could expect its turnaround time to be large in a traditional environment since it probably involves leaving an editor to go out in a file system, explicitly locating a file holding the interface, studying the interface and remembering it, and then returning to the first point of editing without any other result than what could be memorized during the break.

Given the above view of the goal, it is clear that *integration* is a a key issue. The first and third components of the turnaround cost can obviously be very effectively attacked by utilizing context information - making one tool explicitly utilize knowledge about the detailed state and context of the other tool. As will be described later, it also turns out that much less CPU-cycles are consumed in a carefully built integrated system, making it effective in reducing even the second cost component.

1.1. A Definition of Integration

Despite tha fact that integration is often mentioned as a highly desirable property of support environments, it still lacks a more precise commonly agreed on definition, and most publications never even try to be more explicit on the nature and meaning of the concept. However, as a concluding remark on the subject of integration, we propose the following:

In the context of software development environments, we discuss integration in terms of three characterizing properties of tools: *uniformity*, *communicational ability*, and *openness of representation*. Each of those are explained below together with examples.

1.1.1. Uniformity.

Measures the degree to which different tools behave in a similar way in similar situations.

This spans from reusing the same abstract concepts whenever possible, to the unification of syntactical aspects of the user interface. The word *external integration* has frequently been used to denote the latter part of this spectrum.

Examples:

* Use of the same command language and line editing conventions.
* Use of the same conventions for managing windows and menues.
* Use of the same user interaction style, such as pointing-at-object, iconization, parameter defaults, and command result reporting.
* Use of the same editing, viewing, and searching tool for all texts.
* Use of the same conceptual model for representation of data. In a Entity-Relationship-Attribute based environment it would be natural to view most data in terms of typed, attributed, directed graphs. Other environments present other conceptual models, like strings, lists, and (mathematical) relations.
* Use of similar operations on different data represented using the same model.
* Use of the same definitions of concepts; such as version, configuration, dependency, inheritance, relation, access privilege, transaction, icon, data key, type, etc.

1.1.2. Communicational ability.

Measures the degree to which a tool communicates its findings and actions to other tools, and the degree to which it provides means for other tools to initiate communication with it. This includes the *temporal* aspect of immediate notification, the *selective* aspect of communication with a particular tool, and the *granularity* aspect of communicating changed circumstances with high precision.

This may be achieved either by tools explicitly addressing each other, or by means of a "notify" mechanism provided by the object management system. It also includes the ability to (possibly dynamically) compose tools into larger aggregates.

Examples:

* The notification of a tool presenting a view of a database when the underlying data is changed.

* The notification of the editor when the size of the window is changed.

* Passing of context established in a browser, (pointing to a particular "current" object), to the editor or other more or less independent tools.

* Immediate passing of result of an operation to a (set of) tool.

* Notification of the user, (or the tool he is currently using), when the mail program receives a message to that user.

* Activation of, and passing of context to, the debugger when an exceptional condition occurs.

* Communication from the editor to the compiler, or other processing tool, that the user changed a module, or with higher precision, a particular part of it.

* Notification of involved users when the configuration management tool establishes a new version of an object for public use.

1.1.3. Openness of representation.

Measures the degree to which the data generated by one tool, or data otherwise in its domain, is made accessible for use by other tools.

The openness of representation provides for successive growth of the set of integrated tools. As new tools are built, or old tools are modified, the information utilization and availability will increase as a consequence of tool builders taking advantage of data defined by other tools this far.

Examples:

* The representation of data from the Pert chart tool in a form such that it can be used by the project tracking tool.

* The representation of personnel, their abilities and availability, in a form such as it can be used by the Pert chart project planning tool.

* The representation of the language directed editors internal working format in a way such that it can be used by the compiler, cross-referencer, data flow or complexity analyzer, etc.

* The representation of the Ada program library in a form such that it can be used by the configuration control tool, the graphic browser, etc.

* Representation of both program code and documents in a form such that they can be related to each other by means of a single mechanism.

* Representation of any tools data within the agreed upon datamodel, so that all tools and operations assuming that datamodel may be immediatelly applicable.

2. Approaches to Building Development Environments

The total effort that seem to have been invested in attempting to build development environments over the last decade is impressing. Many of those projects have been highly cooperative, involving participants and tools from several countries and companies. This may have been especially true at the European scene, including projects within the ESPRIT programme, such as PCTE/PACT, (PCTE 1986, PACT 1986, PACT 1988), and within the Eureka cooperation, such as ESF, (Fernstrom et al, 1988), together with national programmes in most countries. An example from the US is (Blumberg et al 1988). A complete reference list over projects in this area could in fact span several pages, and several new ones are in an initial phase.

The creation of such an environment is however a major undertaking, comparable to the development of a new operating system, showing the same need for performance and over all quality before the results could really be used. Consequently, there has been a hope that the environment could be built by putting together existing tools, since developing them again would not be feasible. This approach also builds on the strategy to benefit from a large market of existing software tools, and a desire to be equally open to every tool and technology.

On the other hand, the whole idea of building an *integrated* system out of existing tools is an inherent paradox, at least if we are expected to accept those tools as they are. If the benefits of integration are to be exploited, the integration must be quite tight, and utilize every chance of catching information in one tool which might be of use in another place. Integration also seems to stand in conflict with the desire to be equally open to every tool, since very little can be achieved in terms of integration if not quite far reaching assumptions about involved tools are allowed.

To support the opinion one could observe the international scene where few, if any, environment products of the first kind have evolved. Instead, there is an emerging set of what might be called *integration frameworks*, examples of which are PCTE, (Emeraude 1987), and Software Backplane, (Atherton 1987). Those products are usually not immediatelly useful as end user products, but represent semi-fabricates on top of which tool manufacturers are expected to build their products.

When it comes to ambitious environment products providing tightly integrated end user functionality, examples of which are the Interlisp, (Teitelman & Masinter 1984), Smalltalk, (Goldberg 1984), and the Rational Ada Environment, (Archer & Devlin, Rational 1986, Morgan 1988), we see that they are *not* built out of existing tools. Instead, they are carefully crafted together from pieces built with a single purpose: to fit into the environment and contribute to its high level of integration.

To conclude, we observe that it is hard to see any successful attempts on building a competitive environment out of existing tools.

2.1. Another Approach

We have noted two existing approaches:

(1) *Build the environment out of existing tools purchased on the market.*

We consider this approach unsound in that it can be shown to ruin the major beneficial characteristic of the next generation environments: the high level of *integration.*

(2) *Build the environment from scratch to ensure a high level of integration.*

Although this approach *has* been shown feasible in some cases, it has the drawback of soaking up resources into development of basic capabilities and "tool kernels" which have no real relationship to the original reason for starting from scratch: improving the integration.

As an obvious alternative, we propose an approach which takes the best aspects out of the previous proposals:

Take existing tools, over which we have *full technical and commercial control*, and *modify* them into a high level of integration. Both with respect to each other, and with respect to a possible "integration framework" on top of which the environment is built.

The idea sounds simple and obvious: instead of writing all those tools from the beginning, we modify our existing ones whenever necessary. In this way we can achieve the high level of integration and productivity associated with the second approach, for a price that is closer to the first approach. Furthermore, since the software to use for the environment construction is assumed to be known by in house engineers, there can be a fast start and high speed of progress.

However, few organizations have the necessary control over a sufficiently interesting and complete set of software tools. For a successful project, the following criteria must therefore be met by the involved organizations:

* They must have full technical and commercial control over significant software tools.
* They must have the willingness to modify those tools to meet the requirements of the integrated environment.
* They must be ready to commit to certain tools, and therefore to a particular technology and philosophy of integration.

The rest of this paper reports on the applying of this philosophy in one particular context - the Ada product development efforts as carried out within the companies TeleLOGIC and TeleSoft.

The necessary prerequisites seemed to be there in that it included full control over a production quality Ada compilation system and a debugger, (TeleSoft 1988), and a set of programmer support tools as collected within the Arcs system, (Ada editor, browser, recompilation tools, teamwork support, configuration control, program library manipulation, query tools, etc), (Schefstrom 1988). What is discussed here is therefore probably best described as the second, and maybe definitive, step in the development of Arcs, (Schefstrom 1988, Schefstrom 1987, Schefstrom 1986a, Schefstrom 1986b, Schefstrom 1986c, Narfelt & Schefstrom 1985, Narfelt & Schefstrom 1984).

3. An Overview of Arcs

Although integration takes away many of the traditional distinctions between tools, one could divide Arcs into the following components:

* Ada oriented editor.
* Compiler & Binder.
* Library database browser.
* Debugger.
* "Integration framework", which includes program library database, query and command language, common internal datastructures, (such as Graphs (Schefstrom 1988)), and windowing system and associated UI toolkit.
* Teamwork support and configuration management, which includes multiple user coordination, project control, and version management in the program library database.

The primary and most ambitious versions of the software are being built to take advantage of hardware configurations of the category "network of powerful workstations". The purpose is simple: create a significant reduction in programming turnaround time. While not all of the components have reached the stage of a stable released product, prototypes exist and are in use internally.

Some of the components above, together with the philosophy of the integration framework, have been described in earlier publications, (see references at end of chapter 1). However, to make the presentation reasonably complete, we include a short summary of the integration framework.

4. Integration Framework

The Arcs environment is in many respects centered around the Ada program library. In our case this program library is much more general than is strictly required by the Ada LRM, (Narfelt & Schefstrom 1984, Narfelt & Schefstrom 1985). Rather than being tailored to only solving the Ada library problem, as initially perceived, it supports a network, or graph-, oriented datamodel. Arbitrary *nodes* may be introduced, which are connected by arbitrary *relationships*. Both nodes and relationships may have *attributes*, and also have a *type*, or a category. This is used by the compiler to represent Ada compilation units and their interrelationships, as well by other tools to include non-Ada code or other information. Another usage is to impose a meta structure, *subsystems*, on top of the otherwise flat namespace of the Ada library.

Included in this framework is also a query and manipulation language based upon an extended relational algebra. Results of such queries are called *graphs* and *nodelists*, and all tools are made to accept such objects as their input, and produce their output in the same way. This acts as one important integrating factor in Arcs.

The library database is distributed in the sense that a single database is constructed out of arbitrary physically separate parts. This acts both as a means for distribution, sharing, view creation, and version management.

All this is described in more detail in the references above. The similarity in concepts with efforts like PCTE/PACT, (PCTE 1986, PACT 1988), and CAIS, (CAIS 1988), should be noted. Similar ideas, and similar problems, seem to evolve out of the work around Hypertext systems, (CACM 1988).

5. Ada oriented editor

The language oriented editor is probably the most established and well-known component in modern programming environments, (Programming environment workshop 1986, ACM 1984). It is a key component for reasons that will be explained. Nevertheless, different editors that all are called language-oriented differ dramatically in capabilities, and are often talked about interchangeably in a way that is not reasonable. Therefore, let us distinguish between two classes of such editors:

* *Syntax-directed* editors, which know about the syntax of the programming language, but not about the semantics. All of them can check the edited text for syntactic correctness; most of them allow the user to initially produce the text in terms of "syntactic categories"; some of them allow changes to take place in terms of "syntactic categories".

* *Semantics-directed* editors have, usually, all the syntax directed capabilities as a subset. In addition, they check and utilize typing information. This allows them to provide the probably more important extra benefits of type checking, interactive cross-reference, semantic completion, and browsing between modules in a system.

The Arcs editor belongs to the second, semantics-oriented, category.

Since the services we want to provide are based on that syntactic or semantic information is available immediatelly during editing, it usually implies that this information must be maintained incrementally. In the semantics case, in combination with a language like Ada which imposes a strong type checking among separately compiled units, one must also integrate the editor with the program library, since little can be said semantically about a compilation unit in isolation. The intermediate representation used by the editor must then, for things to actually work, be the same as for the batch compiler and code generators.

Although the above reflections are at least partially known from a number of language editor projects, (Reps 84, Kaiser, Kaplan, & Micallef 1987), one could from a general software engineering standpoint observe that not many have a fair chance of actually building a semantic Ada oriented editor, since it requires *very* close coordination with a complete compilation system and its program library database.

When starting the work described here, we had already implemented an editor supporting syntactic support and hierarchical structuring, (Stromfors 1981, Stromfors 1986), as well as certain semantic services like interactive cross referencing of (batch) compiled units. These services utilized the internal form generated by the compiler, and thereby already showed a degree of integration with the compilers internal form and the program library.

This was very useful for static inspection and understanding of a large system, and as a nice interface to the debugger. However, when changing the unit in the editor most of the semantic capabilities had to be turned off and the editor fell back to supporting mainly syntax. This because the semantic aspects of the internal form were not incrementally maintained, and must be updated in batch.

However, closer investigation of the problem revealed that more could be achieved without prohibitive costs, a large part of which can be attributed to a tight integration with the existing compiler.

The compiler used, (TeleSoft 1988), is structured into three phases, a Front-End, a Middle-Pass, and Code Generators. The Front-End includes all the routines for checking the type correctness of a program, and its purpose in the translation process is to produce an internal representation of the compiled unit, which can then be used as input to the Middle-Pass. The Front-End is well separated from other parts, and is run completely before starting later phases of the translation process.

Because of this property, it was possible to import the Front-End software into the editor, and using it for very quick type-checking and internal-form generation. When the user needs to execute his program, the environment checks for which units only the Front-End has been run, and when necessary runs the Middle-Pass and Code Generator. As is discussed in the next section, the importing of all those compiler modules into the editor gives further substantial contributions to decreasing turnaround time.

Further improvements of the editor are underway, including finer incrementality as a way of optimizing the generation of the internal form. Quite naturally, those opportunities are most obvious when a change is limited in scope or is in statements, (as opposed to in declarations).

6. Compiler and Linker

The compiler is the most obvious and basic of tools in the environment, and a rich source of productivity enhancing integration opportunities. To achieve the high goals, the viewpoint must however be much more towards the compiler as one of many cooperating components. The structure of a compiler, to be possible to use in a high-integration environment, must be such that the parts can be used and called quite independently of each other. Different parts must be possible to recombine in new ways, and be possible to call independently and in different orders. Finally, it must be possible to activate routines in many states, such as in a partially completed syntactic or semantic analysis.

The compiler used in this project had those properties to a quite high degree, and made it possible to for a relatively low cost collect what we believe are the major benefits of tight integration. Although incremental update of the internal form, during editing, proved to be economic in many cases, the major benefit of the integration seems to come simply by avoiding to throw away computation results by repeatedly recompute datastructures from more basic forms. Given the current operating system and language technology, this seems to be closely related to whether one can keep datastructures in main memory and let tools coooperate through those, as opposed to repeatedly converting between this form and the (usually different) format that has to be used when communication through files. This seemingly unavoidable loss in both structural and semantic contents has been noted before, and is also developed in an excellent paper by R. Balzer, (Balzer 87).

So, a conclusion is that major gains can be achieved just by allowing tools to communicate in the more precise way that is possible by sharing all datastructures in a single virtual memory space. This opens up new possiblilities, and creates new ways of thinking about tool cooperation than are available when being contstrained to combine tools by pipes or reading each others output files.

It also seems like Ada compilers are unusually rewarding to build this way, due to the dependency on a context of previously compiled units as provided in the program library. Any non-integrated Ada compiler must at every invocation open and read the program library, often doing certain conversions of an externally stored form to a more suitable internal form, followed by a termination where all those internal datastructures are lost. At the next invocation, the same datastructures are created again, just to be thrown away yet another time.

Initial performance results are very encouraging, showing a 75 percent gain in compilation speed at medium sized modules, (400 lines). The results are even better with smaller modules, as well as in cases when errors are detected. The latter is due to that in case of errors, no phase beyond semantics is run, making the (alternative) batch startup overhead relatively higher. Detailed results do of course depend on the characteristics of the actual compiler technology used, but we see no factors that would dramatically change the tendency.

The tight integration also makes it easy and obvious to provide more end user services. Examples of such are fast semantic error feedback to the editor, and quick response and suggested recompilation orderings in case modules have been compiled in wrong order. Another nice example is the direct communication with the browser, which makes it possible to directly update graphical presentations whenever a compilation, (or editing session), causes a structural change.

Finally, any Ada linker must check that the modules linked together have been compiled in the correct order. In the integrated compiler, all the necessary datastructures are immediatelly available making this operation much less costly.

7. Library Browser

In modern programming environments, which are often based on a complex and highly structured "database", the tools for searching and presenting data are of great importance. The most important such tool is usually called the "browser".

In our most basic case, the database contains an Ada program whose development the environment is intended to support. The role of the browser is then to visualize that program to the user, and allow him to move around among the packages, changing view and level of abstraction. The browser also establishes a default context, and provides a means for changing from typing characters into pointing at desired objects. All this immediatelly extends into handling any types of objects, and is therefore the most visible manifestation of the "seamless system".

The browser in Arcs provides a number of different views and modes, and utilizes type information to present library database structures in the most appropriate way. When presenting nodes and relationships that can be identified as Ada modules and their interrelationships, Ada specific knowledge is used to group nodes and select icons as to emphasize that what is viewed is indeed an Ada program. Examples includes hiding bodies and subunits under specification nodes, and grouping together such units that are in the same subsystem.

The browser is tightly integrated with the rest of Arcs, some examples of which are the following:

* When a unit is recompiled, replaced, deleted, or a new unit included into the scope of the current library, this immediatelly affects the view presented by the browser.

* The result of operations using the query and manipulation language are reported in terms of objects, called "graph variables". Those are in fact the variables of the query language, and the browser, as any tool in Arcs, takes such variables as input.

* The unit clicked on using the browser is automatically made the default unit for any other command in Arcs.

* The browser reads its information from exactly the same source, the program library database, as is used by every other tool. This means that graphical pictures presented by the browser are always correct and up to date in the sense that what is seen is always exactly the software that would execute if the linker was called and the resulting program was started.

The browser was however created specifically for Arcs, and did not exist on beforehand.

8. Debugger

Debugging is closely coupled to the inspection and browsing of programs. Therefore, tools like the browser and the editor, with its interactive cross reference ability, fit extremely well in this context. If such tools are immediatelly available in close connection to the debugging session, they not only provide their original service of overview and aiding understanding of the program, but provides the context and presentation mechanisms for the pure debugger activities. In fact, browser and "semantic" editor together makes up a main part of the debuggers user interface. Reusing those components in this new context further strengthens the users feeling of system uniformity.

The tight integration here also shows the usual benefits of sharing internal datastructures. The internal forms used by the compiler in doing typechecking and managing separate compilation, and by the semantic Ada editor to provide interactive cross referencing and semantic completion, is of course also used by the debugger. The immediate availablility of those datastructures, together with the resulting very short debugger initialization time and freedom of context change, further contributes to minimizing turnaround time.

The debugger existed on beforehand as a separate tool with its own user interface and own management of internal datastructures.

9. Teamwork Support and Configuration Control

In practice we seldom work alone, but are members of a team of programmers which concurrently modifies and updates different parts of the same program system. Any serious environment must allow for this dynamic process of continuous change to proceed, without resorting to extensive manual work, recompilations, or purely administrative solutions. This is, however, one of the major challenges in building an Ada development system. Not very surprisingly, the design of the program library database is a very central issue here.

In Arcs, we approach this area in terms of three aspects:

(1) First, different users must be able to have *different views* of the same system under development. While the largest part of the program library is common to the programmers of a project, they must be able to specify arbitrary individual variations. One programmer might be updating one unit, in his view, while another one need to see the older stable version while testing still another unit. This independence must be provided while not forcing duplication of parts that don't need to be so.

(2) Second, partly as a consequence of the first requirement, it must be possible to have *multiple versions* of the same unit in the program library. The first requirement could infact be approached as a question of allowing multiple versions, and allowing different people denote different versions as the default. The related concept of a configuration must then be introduced.

(3) Finally, but very important and sometimes forgotten, there must be good support for the *work process*. The configuration control scheme must be built for an interactive, dynamic, changing, and sometimes inconsistent, environment. We have found this last aspect, of hardening the configuration control system with respect to frequent change, as most fruitful and rewarding to work with.

Arcs, together with the TeleGen2 program library on top of which it is built, meets those requirements to a very high degree.

In our case, an Ada program library is defined by composing a list of sublibraries, each containing an arbitrary set of compilation units. Sublibraries are independent entities, and may be combined completely freely to construct full libraries; it is just that certain combinations of sublibraries will make Arcs generate more complaints about recompilation needs and missing units than others. This property, independence of sublibraries, infact showed to be more important and useful than we fully foresaw when designing it.

The scheme is very general, and can be used in many ways. However, the method discussed here might be called "incremental baseline update", and builds on that a given set of sublibraries are said to represent the most recent stable state of the project, in other words, a "baseline". Work then proceeds by programmers reserving units in a baseline sublibrary, transefering them to a private sublibrary for modification and test. When finished, the unit can be moved back to the origin sublibrary, but maybe more typically to an intended Next Version of the Baseline sublibrary, since other people may not want to see the change immediatelly.

The following shows a typical such sublibrary list:

Typical sublibrary list:

1. Programmers_Private
2. Product_Baseline_0_1
3. Standard_Sublibrary

Showing Next Version sublibrary:

1. Programmers_Private
2. Product_Baseline_0_1

    2.1 Product_Baseline_0_2

3. Standard_Sublibrary

At some suitable point in time the configuration manager then decides to establish a new Baseline, and the point here is that this can be done with a single command, in a structured and highly automatic way given the information in the Baseline and Next Version sublibraries.

The following summarizes the basic cycle:

(1) *Get* a unit from Baseline sublibrary into a private sublibrary.

(2) *Modify* and test the unit in private area without disturbing others.

(3) *Return* the unit to Next Version baseline sublibrary when finished.

(4) *Generate new baseline* from old Baseline and Next Version sublibraries.

Going into the details of this kind of multiple version, multiple people environment, reveals a large number of questions and special cases. However, most of them can be handled using this basic structure. Let us consider some examples:

Assume now that other people have updated and returned units, and you want to make those visible in your view for testing against, and you don't want to wait until a complete new baseline makes it visible. To solve this problem, the RESYNC command can be used, which synchronizes the local view with what has been released as Next Versions. This is done by giving the user local versions of the Next Version units he doesn't already have.

As another example, consider what happens if you you have reserved and changed a central spec, and therefore need to recompile many depending units, many of which currently are in the baseline, (and therefore are shared between many people).

This problem is directly solved by the COMPILE command, which computes the necessary recompilations, and makes sure that this takes place only in the local view.

Finally, consider the case that one want to reserve a unit for update, but there is a newer one than the Baseline version.

The answer is that the GET command will automatically select the newest version, (if it is not reserved, in which case reservation is denied).

Similar schemes for doing configuration control and coordination of project groups have been presented within NSE, (Sun Microsystems 1988) and Eclipse, (Pierce 1987). What differs Arcs from those systems is the high level of integration in Arcs, between the tools and subsystems mentioned above, the program library database, and the characteristics of the Ada language. In that sense it is similar to the Rational environment, (Morgan 1988).

The most important component, the Ada program library, existed on beforehand. However, Arcs teamwork support and configuration management could never have been built, as is true for all of the services presented, without the unrestricted access to internal interfaces.

10. Experience

All parts of what is described above have been developed at least to the stage of prototypes stable enough for internal use. A subset of the functionality is currently offered, and has been delivered, as a product to external customers, and over the next year the externally available functionality will be successively increased. However, this means that the major experience this far comes from the internal use, and the following comments are based on this fact.

10.1. *User Interface*

Arcs was originally built in a bottom-up philosophy, emphasizing basic primitives and combinational ability. The quite large body of functionality was provided by means of basic building blocks, which could then be combined into different more elaborate services.

As an example of this approach, consider the often wanted service of iterating through all places in a system where a particular procedure is called. In a large system, this obviously involves searching through many modules. According to the building block approach, the basic service of stepping through units was made extremely simple: The command

```
Arcs> ixref init <list_of_units>
```

just causes the given <list_of_units> to be searched. However, the problem of creating that list of compilation units, containing all those units where it could be meaningful to search for calls of the given procedure, is not part of the "ixref" command. Instead, the generally available program library database query language should be used for this task, utilizing the fact that the only units that could possibly contain a call to a given procedure are those that 1) directly "withs", or 2) are secondary units, to the unit in which the procedure is declared. The Arcs command language allows such queries to be expressed directly at the same command line.

This kind of generality provided a lot of power, and created numerous unexpected new usages and combination opportunities. However, only for the expert user. Experienced people having the necessary curiosity, and time to invest, find it very powerful, while other people have found the generality overwhelming and even disturbing.

As a result of this experience, we are now working on providing more prepacked services corresponding to often wanted patterns of combinations of basic functions. The trick here is to do this while keeping the combinational abilities and avoiding to create an ever more complex system.

10.2. *The editor*

While the Ada oriented editor provide a substantial body of well appreciated functionality not available otherwise, the mere introduction of another editor is an obstacle in many cases. Despite the fact that most people learn the editor in less than a day, the intrusion into what is in some cases considered to be the programmers innermost privacy - the editor - is creating much discussion. The opinion has been that one should find a way of providing all the services while letting the user still use standard "vi", for example.

However, the previous discussion in this paper, together with several investigations dedicated to the editor problem, seem to show that accepting the limitations of not having "administrative and technical control" over the editor would be too hard. So, instead of trying to utilize an arbitrary editor, we invest more, and extend, the one that is built into Arcs. The development goes along two lines: 1) we create a number of special versions of the built in editor to make it appear like different popular editors. 2) We introduce even more functionality dedicated to software development, to further motivate the change to a new editor.

10.3. *Configuration control*

The configuration control in Arcs, or as it also is called, "Teamwork support", is an example of functionality that successively grew out of the need and experience of the work in the development group. Different specifications were written and evaluated, but it was not until we had used Ada for a large system, with multiple people, for a longer time, that we could develop a functionality that was really felt helpful. The reason it took time before reaching a matured standpoint in this subject is probably just that it is a complex area where few well known principal results are available. It is furthermore an area where the semantics of modern separate compilation languages like Ada interact with the environment in a very elaborate way.

This inherent complexity, together with the fact that many people, just as in the case with the editor, already have some own way of working, makes this part of Arcs into one of the most difficult to explain. However, once the problem is understood, and the Arcs configuration control functionality has been in use for a while, the service turns out to be among the most valuable and appreciated.

11. Acknowledgements

The ideas and results reported here are part of what has been discussed, argued about, and worked on over the years at one of the most creative places I know: The Ada Development Environments corridor at TeleLOGIC/Lulea. Mikael Backman, Nils Livingstone, Hans

Ohman, Ola Stromfors, and Johnny Widen, are all responsible for any good ideas presented in this paper.

12. References

ACM (1984),
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Peter Henderson, ed., Pittsburgh Pennsylvania, April 1984.

Archer & Devlin
Rational's Experience using Ada for Very Large Systems, "White Paper", Rational, 1501 Salado Drive, Mountain View, CA 94043.

Atherton (1987),
The Software Backplane, Product description material, Atherton Technology, 1333 Bordeaux Drive, Sunnyvale, CA.

Balzer (1987),
Living in the Next-Generation Operating System, in (IEEE Software 1987).

Barstow, et al, (1984),
Interactive Programming Environments, Barstow, Shrobe, & Sandewall. McGraw-Hill 1984.

Blumberg et al (1988),
NASA Software Support environment: Configuring an Environment for Ada Design, Blumberg, Cantor, McNickle, and Reedy. in "Ada in Industry", Proceedings of the 1988 Ada-Europe Conference, Munich. Cambridge University Press.

CACM (1988),
Communications of the ACM - Special Issue on Hypertext, July 1988.

CAIS (1988),
Military Standard Common APSE Interface Set (CAIS). Proposed Military Standard, US Dep of Defense.

Carr et al (1987),
Implementation of a Prototype CAIS Environment, in ACM Ada Letters, Volume VII, number 2, March,Äpril 1987.

Emeraude (1987),
Emeraude PCTE General Presentation, Product Description Material, G.I.E. Emeraude, 38, bd H.-Sellier, 92154 Suresnes Cedex, France.

Fersnstrom et al (1988),
Eureka Software Factory Design Guidelines and Architecture, Fernstrom, Narfelt, Reenskaug, Schaefer, and Weber. ESF consortia 1988.

Gallo, Minot, & Thomas (1986),
The Object Management System of PCTE as a Software Engineering Database Management System, in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto, Cal, Dec 1986.

Goldberg (1984),
The Influence of an Object-Oriented Language on the Programming Environment, in

(Barstow 1984).

IEEE Computer (1987),
IEEE Computer, Nov 1987, Special issue on programming environments.

IEEE Software (1987),
IEEE Software, Nov 1987, Special issue on programming environments.

Kaiser, Kaplan, & Micallef,
Multiuser, Distributed Language-Based Environments, IEEE Software, November 1987.

Morgan (1988),
Configuration Management and Version Control in the Rational Programming Environment, in "Ada in Industry", proceedings of the 1988 Ada-Europe Conference, Munich, Cambridge University Press.

Narfelt & Schefstrom (1984),
Towards a KAPSE Database, in Proceedings of the 1984 IEEE Conference on Ada Applications and Environments, StPaul, Minnesota 1984.

Narfelt & Schefstrom (1985),
Extending the Scope of the Program Library, in 1985 International ACM Ada Conference, Paris 1985.

Programming Environment Workshop, (1986),
Proceedings from International Workshop on Advanced Programming Environments, Trondheim, Norway, June 1986.

PCTE (1986),
A Basis for a Portable Common Tool Environment, Functional Specification. European Strategic Programme for Research and Development in Information Technology.

PACT (1986),
PACT General Description, Bull S.A. Louveciennes, France, 1986.

PACT (1988),
PACT Commion Services - C Language, Bull S.A. Louveciennes, France, 1988.

Pierce (1987)
ECLIPSE - An APSE Based on PCTE, in Proceedings of the Ada-Europe International Conference, Stockholm, May 1987.

Reps (1984),
Generating Language-Based environments, MIT Press, Cambridge, Mass.

Software Sciences (1987),
Eclipse Overview, Software Sciences Ltd, UK, October 1987.

Rational (1986),
Application of the rational Environment to Lifecycle Software Development, "White Paper", Rational, 1501 Salado Drive, Mountain View, CA 94043.

Teitelman & Masinter (1984),
The Interlisp Programming Environment, in (Barstow 1984).

TeleSoft (1988),
TeleGen2 - The TeleSoft Second Generation Ada Development System, User's Guide. TeleSoft, 5959 Cornerstone Court West, San Diego, 92024 California

Schefstrom (1986a)
Integrating an Ada- Library System into the Unix Configuration Management Toolset,

2:nd IEEE Conference on Ada Applications and Environments, Miami Beach, April 1986.

Schefstrom (1986b),
Integrating Ada in an Existing Environment - the Arcs Example, 1986 ACM Ada Conference, Edinburgh, May 1986.

Schefstrom (1986c),
Project Support Building on the Ada Program Library, in Proceedings of MILCOMP-86, London, September 1986.

Schefstrom (1987),
The System-Oriented Editor - A Tool for Managing Large Software Systems, 1987 ACM SIGAda Conference, Boston, Dec 1987.

Schefstrom (1988),
Programming-in-the-Large with the System-Oriented Editor, 1988 AdaEurope Conference, Munich, Germany, june 1988.

Stromfors (1981),
The Implementation and Experiences of a Structure-Oriented Text Editor, in Proceedings of ACM SIGPLAN/SIGOA Symposium on Text Manipulation, SIGPLAN Notices, vol 19, no 6, 1981.

Stromfors (1986),
Editing Large Programs Using a Structure-Oriented Text-Editor, Ola Stromfors, Linkoping University. In (Programming Environment Workshop, 1986).

Sun Microsystems (1988),
Introduction to NSE, March 1988.