SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 27(4), 447–480 (APRIL 1997)

Exploiting Persistence in Build Management

DAG I. K. SJØBERG
Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway
(email: Dag.Sjoberg@ifi.uio.no)

AND

RAY WELLAND, MALCOLM P. ATKINSON, PAUL PHILBROW, CATHY WAITE
Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow G12 8QQ, Scotland (email: {ray, mpa, pp, cathy}@dcs.glasgow.ac.uk)

SUMMARY

A challenging issue in the construction and maintenance of large application systems is how to determine which components need to be rebuilt after change, when and in which order. Rebuilding is typically recompilation and linking, but may also include update of derivable components such as cross-reference databases and re-creation of library indexes. Type definitions or schema, and data values in a file store, database or persistent store may also need to be rebuilt. The main purpose of this paper is to describe how persistent language technology can be exploited to enhance build management. In particular, the paper describes a method for transactional, incremental linking and the implementation of its support. To help implement this method, and to make it safer and more efficient to carry out rebuild activities in general, we have defined a set of automatically checkable constraints on the software. The build management tool we have implemented, the Builder, derives rebuild dependencies *automatically* and offers *partitioning* of dependency graphs—a means to defer or avoid unnecessary rebuilding. The Builder is implemented in a persistent programming language and provides build management for applications written in the language. It exploits features such as strong typing, runtime linguistic reflection, and referential integrity provided by the language processing technology. The Builder operates over both programs *and* (complex) data, which is in contrast to conventional language-centred tools.

KEY WORDS: build management; recompilation; incremental linking; method support; programming environments; persistent programming

INTRODUCTION

This paper is concerned with the construction and maintenance of large, complex and long-lived software systems. Such systems are constructed from many components that have to be combined in some systematic way to form the executable system. Because such systems evolve over time, it is inevitable that changes will need to be made to the software and other system components. A typical large system may take several hours, or even days, to rebuild to a new operationally consistent state by naive methods after changes have been made to its source components. *Build*

CCC 0038–0644/97/040447–34 $17.50

Received 3 November 1995
Revised 4 June 1996 and 15 August 1996

management techniques and tools attempt to reduce the cost of evolution by finding efficient mechanisms to rebuild a system.

The problem of keeping track of which versions of a component can be integrated with which versions of other components in order to constitute a valid application is referred to as *configuration management*.[1] An important aspect of configuration management is *build management*, which controls the building and rebuilding of a system. Derived components that are used in the executable system are produced from source components and other derived components. Rebuilding is typically recompilation and linking of application components, but in more sophisticated programming environments rebuilding may also include updating cross-reference databases and other automatically derived documentation, re-creating indexes referring to the components of a library, and executing programs to build long-lived data, such as translation tables, on which the new operational programs depend.

This paper focuses on build management and its aims are to:

(a) describe a build management tool that provides:
  (i) automatic construction of *system models*, i.e. identification of the components of an application and the construction of the dependency graph among them;
  (ii) the programmer with a number of rebuild options, each of which operates on partitions of the overall dependency graph;
(b) describe a method and tool support for transactional, incremental linking of application components;
(c) show that a set of automatically checkable software constraints can be defined to support our method for application construction and build management;
(d) illustrate how the power of a persistent programming language that provides strong typing, referential integrity and reflection can facilitate the implementation of build management tools.

In spite of significant growth in the performance of certain components (processors, stores, networks, etc.) supporting computation, recompilations (as well as other rebuilding actions) still represent a significant part of the maintenance costs of a large software system and may thus inhibit system evolution. It has been reported that in a large Ada application more than half of the compilations were redundant.[2] Avoiding unnecessary recompilations is therefore an important issue.

The use of *incremental linking* also reduces the cost of rebuilding the system because in conventional systems every object module must be relinked after a change to any of the modules. Quong[3] proposes an incremental linking technique that can be used in a conventional system. Our work exploits the power of a persistent programming language to implement a type safe, incremental linking method, which in turn supports an incremental development strategy. The motivation for avoiding the ‘big-bang’ approach is both to save processing time and to simplify the management of changes in large application systems. We believe it is easier to control the changes if they are carried out in small increments.

Build management addresses some of the technical problems of changing software but does not handle the problem of the (human) management of the change process. While we recognise its importance, we are *not* concerned with *change control*,[4] i.e. deciding which changes are to be made, evaluating cost/benefit of change, adminis-

tering the change process, proposing guidelines for logging and access rights when changes are to be propagated, etc. Versioning is also not an issue of this paper.

Context

The work described in this paper was carried out in the context of the strongly typed, orthogonally persistent[5] programming language Napier88.[6,7] The concept of persistence tackles the mismatch between database systems and programming languages;[8,9] a uniform model for representations and operations on persistent and transient data is provided. The rights to persistence are orthogonal to the type of the data; all types of data may outlive individual program executions. Application development tools, programs and the data over which they operate may reside in the same store. Many of the benefits of persistent language technology have been described in the literature.[5,10,11] We exploit other properties of Napier88, particularly mutable and scannable sets of bindings. These are described later in the paper.

Experimental approach, assumptions and limitations

We report an experiment with one approach to aspects of build management in this persistent context. Five assumptions are made:

(i) that integration of build management tools with the language or languages is acceptable;
(ii) that it is possible to obtain detailed program analysis information from the compiler or economical to reconstruct it by additional partial analysis of programs;
(iii) that programmers can be persuaded to adhere to a programming methodology;
(iv) that programmers cannot be trusted to supply accurate dependency information on which build-avoidance depends (even if they were reliable, we do not believe it would be a good use of their time);
(v) there is a store available to hold all the data involved, e.g. source fragments, program executable fragments, cross-reference databases, system models, etc.

Our methodology is particular to systems utilising persistence, but we suspect adaptations are easily discovered that allow the approach to be used in other contexts. Assumptions (iv) and (v) above are supportable because we have an orthogonally persistent store available to us. More work would be required if this were not the case.[12] Some aspects of build management have not been investigated:

(a) we have worked with only 100,000 lines of code and therefore have no evidence that industrial scale applications could be supported, we believe this is mainly an issue of object-store technology, which is the work of others;[13–17]
(b) we have not investigated the issue of several programmers working simultaneously on the application;

These issues have also been investigated contemporaneously in Forest[12] and similar techniques could, in principle, be applied in our experimental context.

Persistent programming languages and their environments are intended to support the development of large, complex and long-lived application systems. Such systems are difficult to manage intellectually; for example, it is hard for software maintainers,

without assistance, to keep control over the build dependencies and ensure consistency between the source programs and executables. Our supporting tool for build management, the *Builder*, derives the dependencies automatically by analysing both the source code and data in the persistent store.

The Builder and the components over which it operates all reside in the same persistent store. In combination with referential integrity,[11] this helps guarantee the existence and access of the components the Builder requires during analysis and rebuilding. That is, direct references may be established among components, e.g. between source programs and executables, as opposed to relationships based on naming conventions as in conventional file systems like Unix™. In our context, components include source code, executable code, sets of bindings, type representations and 'constant' data.

The assistance from the persistent context is of two forms:

(a) the identification of referents (components being referenced) is by a supported mechanism, and not by one which we have to implement and validate ourselves;
(b) the referents are guaranteed to be of the correct type, so that our processing does not have to cope with type errors.

However, the latter form is not an absolute guarantee that all required data exists. For example, temporary values of the right type are used as place holders for useful values constructed later. If the construction process is not properly organised, a temporary value will be encountered when a useful one is required. Our tools help with the organisation of this process.

The remainder of this paper is structured as follows. The next section describes a build management model that provides operations to rebuild various partitions of the overall dependency graph. Then follows a section on related work. This section is followed by a description of the major features of Napier88 that are relevant to our work. We then discuss a method for building application systems in a persistent programming language, exploiting separate compilation and referential integrity, and defining an incremental linking method. A set of software constraints that facilitate build management are also described. The subsequent section describes the Builder, which is the supporting tool we have built in the Napier88 environment. We conclude the paper with some observations on the wider applicability of what we have learned and suggestions for future work.

BUILD MANAGEMENT

A typical application is composed of a large number of inter-dependent components, typically collections of code that are edited, rebuilt and maintained as single units. The *build management* of such components is described by Leblang[18] as follows: 'Build management controls the building and rebuilding of software to produce *derived objects* which are the final software products. Build management includes *minimal rebuilding*—reusing as many derived objects as possible and only rebuilding those objects that had a dependency change.' Complementary to minimal rebuilding are techniques for improving efficiency such as parallel distributed rebuilding where multiple rebuild tasks are submitted at once to various machines in a network.[19]

A comment on terminology is now appropriate. For convenience we write that a

component is *built* or *rebuilt*. Being precise, however, what is meant is that an event on a component may trigger a (re)build action. An edit may trigger a recompilation, a changed (i.e. new) component may trigger a linking operation, etc.

We use the following definition of *dependency* in this paper. If there are two components *a* and *b*, then *a* depends on *b* means that *a* needs to be rebuilt if there is a significant change to *b*. The meaning of *significant* in this definition depends on the kind and context of the (re)build operation. An example of an insignificant change could be a change to a comment in a source code component if the operation is compilation—the derived executable does not need to be rebuilt. However, such a change is significant if the operation is automatic extraction of comments by a documentation generator. Of course, a build management system must be able to detect the kind of change in order to judge whether it is significant or not. A discussion of how to discriminate between significant and insignificant changes is given by Borison.[20]

In the discussion in this paper we assume that the dependency graph and the *derivation graph* coincide. In conventional language environments, the derivation graph may be a subset of the dependency graph since components in the dependency graph may have no corresponding derived component. For example, a header file with definitions may depend on another header file, but nothing is rebuilt after a change to the latter file, only a 'touch' of the former. However, our own and many other modern compilation systems separately compile all kinds of components and thus always aim to produce a derived component.

Two common kinds of dependency are *procedural dependency* and *type dependency*. A procedural dependency exists if one component contains a call to a procedure defined in another component. A type dependency exists if one component references a type defined in another component. Type definitions are often contained in separate components exclusively consisting of declarations (cf. the convention of collecting declarations in header files).

Deciding whether a change is significant or not depends upon the sophistication of the build management system. Suppose that a component *c* contains a call to a procedure *x* contained within a component *d*, then there is a procedural dependency between *c* and *d*. A change to the signature (interface) of *x* is significant as component *c* will have to be edited and recompiled.* If the implementation of procedure *x* is changed, but its signature is unchanged, then it is not necessary to recompile *c*, although relinking will normally be necessary. An unsophisticated build management system will simply record that *c* depends on *d* and will always completely rebuild *c* if any change in *d* occurs.

If the software components are structured in such a way that the signatures and implementation of procedures are contained in separate components, then even a simplistic build management system can avoid some unnecessary rebuilding. Extending the above example, if the signature of *x* is contained in a component *e* and the implementation of *x* is contained in the component *d*, then even a straightforward build management system will be able to avoid rebuilding *c* if only the implementation of *x* is changed. Work on eliminating unnecessary recompilations caused by procedural dependencies is reported by Burke and Torczon.[23]

* The editing is not necessary in the special cases of signature changes that can be overcome by *coercion* in conventional languages, for example, an integer value can be used where a real is expected, and *vice versa*.[21,22]

For type dependencies, a simplistic build management system will assume that any change in the type definition component is significant. Suppose that there is a component *f* that uses some of the types defined in a component *t*. The simplistic build management system will record the type dependency between *f* and *t*, and always rebuild *f* if *t* is changed. A more sophisticated model will detect whether any of the types actually used by *f* have been changed and will only rebuild *f* in this case, cf. the problem of the *big inhale* described in the next section.

We now describe a model for rebuilding applications after change that offers programmers the choice of selecting well-defined partitions of the application identifying the components that are the focus of the rebuilding. This model is general in that it is independent of the way dependencies are formed and is thus independent of the programming language and environment used. The five basic operations of the model are as follows:

(i) *Total build, i.e. build all components, regardless of changes*. This option accomplishes a total rebuild of all the components of the whole application, whether or not they have been changed. For example, this may be required when a new release of the compiler is delivered as the dependency of components on the compiler is not represented explicitly.

(ii) *Build target*. This option focuses on one particular component, allowing it to be compiled and tested, for example, independently of the rest of the application, without the overhead of rebuilding the whole application. This removes 'noise' caused by rebuilding other components and saves significant processing time.

(iii) *Build target and transitively propagate the build to dependent components*. This option also restricts the rebuilding to only that part of the application that is currently of interest, but in addition it propagates the build operations necessary to ensure consistency between the target and the components that use it.

(iv) *Build all components that have been changed since the last build*. This option rebuilds all changed components whatever position they occupy in the dependency graph.

(v) *Build all components that have been changed since the last build and transitively propagate the build to dependent components*. This option is a combination of (iii) and (iv).

The basis for build management is a dependency graph where rebuilding is defined according to certain criteria. The nodes of the graph represent components of the application, and the arcs show the dependencies among the components (see Figure 1). A general requirement of the graph is that it should be a DAG, that is, directed and acyclic.* For example, there should be no cycles among a set of type definitions that constitute a dependency graph. In a DAG there exists a partial order among the components constituting the graph. Topological sorting can then embed a partial order in a linear order,[26] which determines an order (not necessarily unique) in which the components can be processed for rebuilding. Determining the partial order may be a non-trivial task; tool support is typically required.

* There are compilation systems that in certain cases support circular dependencies among components (modules), e.g. the CHIPSY[24] programming system for the language CHILL.[25]

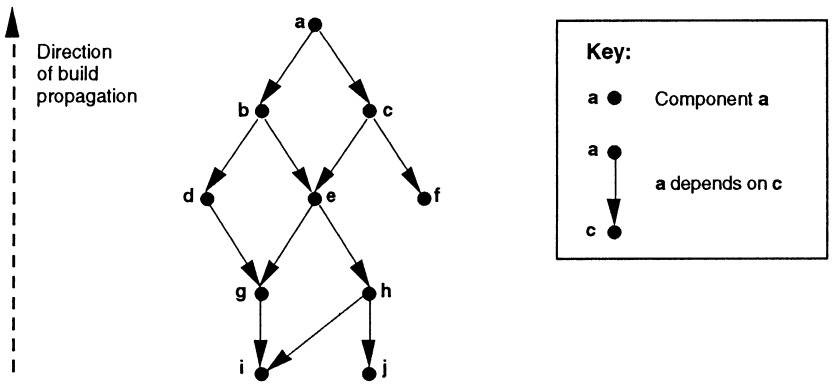

Figure 1. Dependency graph

We partition the dependency graph into affected and unaffected areas. Each of the five basic operations enumerated above is carried out in two parts. The first identifies the affected area; the second invokes the corresponding rebuild function. Figure 1 shows the dependencies among components *a* to *j*. The second of the five operations means rebuilding all the components in the graph preceding a specified target so that the target itself can then be rebuilt. For example, if *e* is the selected target, then *g*, *h*, *i* and *j* must be rebuilt, if they have been changed, before *e* can be rebuilt. The dependency graph is partitioned to exclude all components that are not *e* and its successors. For example, even if *f* had been changed, it would not be rebuilt as it is not part of the affected area of the dependency graph.

The third operation (iii) means that in addition to rebuilding the target, rebuilding is also propagated transitively to all the successors of the target. For example, if the graph represents compilation dependencies and *e* is the target, then *b*, *c* and *a* are also recompiled, in addition to the changed components preceding *e* as described for the second operation. Such propagation is also termed 'cascading' and 'trickle-down'* recompilation in the literature.[27–29]

The issue of access rights is beyond the scope of this paper, but in practice build management systems must consider the problem occurring when rebuild propagation is prevented due to access restrictions.

RELATED WORK

A plethora of experimental prototypes and commercially available products have been developed to support build management.[1,30,31] This section describes some of them according to:

(a) component granularity;
(b) language dependency; and
(c) automation.

We distinguish between coarse and fine granularity. The components dealt with at the coarse granularity are files or some notion of virtual files. They may be of

* The 'trickle-down' metaphor assumes the root is at the bottom.

different kinds depending on their contents such as source text and various kinds of binaries. Apart from this classification, no semantics are associated with the file contents. This is in contrast to fine granularity components such as type definitions, procedures, constants, variables, etc., which can be expressed in a programming language. Hence, fine granularity systems are language dependent and coarse grained systems are typically language independent. Furthermore, a language dependent system will be able to automate more tasks than language independent systems, e.g. derivation of dependency graphs.

Coarse granularity methods and tools

The classical tool to help rebuild applications after change is *Make*.[32] Make considers only file level build dependencies. It is language independent and can be used for a wide range of rebuild operations. However, the programmers must derive the build dependencies and describe them in *Makefiles* manually. Ensuring that the referenced files actually exist is also up to the programmers. It was early experienced that creating and maintaining (possibly hundreds of) Makefiles may be a cumbersome task.

Inter-source file dependencies are typically expressed in terms of *include* statements in languages such as C, Pascal and Fortran. Based on pattern matching, simple methods have been developed for generating Make descriptions of such dependencies.[33] A class of tools have been developed that exploit such methods and feature automatic generation of Makefiles or provide information necessary to maintain them. Most of these tools support Makefile descriptions for C and are based on the C preprocessor macro language. For example, Imake[34] derives inter-file dependencies, but also supports portability in that it separately keeps architecture-dependent information about compiler options, alternate command names, etc., which also affect the specification of Makefiles. GNU Make[35] provides similar Make enhancements. Tools that generate Makefiles also exist for languages other than C.[36]

Many coarse grained, methodology and language independent systems give enhanced support in build management. Examples are Odin,[37,38] DSEE,[39] ClearCase™[18,40] and Vesta.[41] (Vesta also gives some support at the language component level in the form of *bridges*, denoting groups of related tools that support a particular programming language.[42]) The build dependencies among the source components, the derived components and the tools that produce the derived components are described by the user in the system model. Sophisticated features of these systems are management of derived components and automatic generation of 'bill-of-material' documentation of the build processes and their results.

Using Make and classical source code control systems such as SCCS[43] and RCS[44] the users must themselves name and keep track of the derived files. Systems like Odin, DSEE, ClearCase and Vesta provide an internal repository, typically implemented on top of a conventional file system or database management system, in which they can store binary files such as executable code, bitmap graphics, etc. Since such systems understand how to automatically generate the derived files, they can assume responsibility for all the derived file management. The users are only concerned with creating and modifying source files.

Moreover, these systems record information about the current environment at the time of building such as the time, user, system model or build script, kind and

version of the applied tool and operating system, various parameters, etc. Ideally, the system should thus be able to rebuild a component in an environment equal to the one in which the component was previously built. The combination of having intermediate or final derived components cached and a description of how they were built, gives flexibility. When a derived component is requested, the system will incrementally invoke the build tool as appropriate if the component's source has changed, or it will save processing time by reusing existing components if rebuilding is unnecessary.

Fine granularity methods and tools

The systems described in the previous section are of great practical value in that they support a wide range of tools and environments. The methods and tools described in this section are semantically integrated with a given programming language. They are thus less flexible, but offer more automation and support in reducing costs. There are basically two approaches to reducing build and rebuild costs: build avoidance and reduction of the work needed to carry out a single build. As mentioned, tools that generate Makefiles typically infer dependencies on the basis of *include* statements. Any change to any declaration in the included file initiates recompilations of all the program components that include the file—whether or not they use the changed declaration. Tichy[45] has proposed a 'smart recompilation' method for reducing the number of recompilations after a change to such declarations by extending the compiler's symbol table to keep track of finer granularity dependencies between type definitions, constants, variables, macros, etc. Various implementations of this method have been reported in the literature.[24,46]

Another approach to reducing rebuild costs is *selective environment processing*, described by Adams *et al*.[27] as follows:

> Selective environment processing attempts to reduce the amount of environment data that must be processed in each compiler run. The environment of a given unit simply consists of those units on which the given unit depends. Reading the environment has been nicknamed *big inhale* for the usually large amount of information that must be scanned before actual compilation commences.

More than a decade ago Conradi and Wanvik[28] reported that the big inhale may account for 30–50% of compilation time. With increased use of libraries and with programs often being transitively dependent on them, a representative figure for current systems is probably larger than 75%.[47]

The Forest approach[12] to reducing the cost of rebuilding is based on the lazy evaluation strategies developed in functional programming.[48] Each step in the build process is a function application, e.g. compilation, on a combination of build-control parameters, source components and the results of other function applications. The final function application delivers the built system. The results of function application (from earlier builds and other parts of this build) are reused if none of the relevant inputs have changed. Edits to source components generate new instances, so that identity tests can be used to determine whether a function's inputs have been changed. Vesta, preceding Forest, uses a similar approach.[49] The granularity of the source elements has a significant effect on the extent to which savings are achieved.

Vesta is essentially file-based, while Forest is object-based, i.e. it operates at a finer granularity using an OODB as the object repository.

Several commercially available products provide extensive build management support tailored to conventional languages such as COBOL, Ada, Pascal, C and C++. Enabled by an integrated fine granularity environment, dependencies are inferred, changes tracked and necessary recompilation and linking initiated automatically. Examples are application build systems for PCs or Macintoshes from companies such as Symantec,[50,51] Borland[52] and Metrowerks.[53]

Rational's Apex Ada[54] is a sophisticated development environment for Unix used in large-scale production systems. (Apex C/C++[55] is similar product tailored to C and C++.) Apex Ada is based on a persistent intermediate representation of programs, now an industry standard, called DIANA (Descriptive Intermediate Attributed Notation for Ada).[56] DIANA structures combine source code and executable code of Ada objects into one unit and record additional semantic information. The programmer can choose the extent of cascading recompilation and smart recompilation by selecting the granularity of change, which can be compilation units, declarations or statements. The system also avoids unnecessary recompilation by testing the significance of a source change. For example, modification of comments and addition of (not yet used) declarations do not initiate recompilation. The reader can find more detailed information in an evaluation report on the Rational Environment technology, now used in the Apex products, amongst others.[57]

Constraints and types

The issue of constraint checking, in particular verifying that interfaces match, is closely related to work on type systems. Cardelli, in Quest[58] proposed tuples of types and values as interfaces, which can in principle be verified as equivalent by structural comparisons when components have been developed independently. This problem is challenging with modern type systems. The work of Lampson and Burstall[59] explores this issue and the most widely used system that implements such checking is Standard ML.[60] These are precursors of Quest.

The present state of play in module management with sophisticated types, such as those encountered in Napier88, is reported in Connor *et al.*,[61] and Jones has identified a formal model for checking such systems that appears to advance the state-of-the-art.[62]

NAPIER88

Our work has been carried out in the context of the persistent programming language Napier88.[6,7] The features of Napier88 that proved particularly helpful were:

(i) orthogonal persistence;
(ii) strong typing;
(iii) environments as manipulable, scannable and mutable sets of bindings;
(iv) store semantics;
(v) procedures as first-class values;
(vi) a transactional model;
(vii) reflection.

A brief description of each of these is given to help readers unfamiliar with Napier88 understand the sequel.

Orthogonal persistence

A system is said to be orthogonally persistent if it provides equal rights to longevity or transience for all the data it manipulates.[5,63] In particular all values, whatever their type and size, have the right to endure longer than the individual program executions.

Napier88 provides a particularly convenient form of this property, as it complies with the principle of persistence independence,[5] whereby code is the same, i.e. has the same syntactic form and semantic interpretation, when it applies to long-lived and short-lived data. We make significant use of these properties. Orthogonal persistence means that we design our data structures to suit our algorithms and need to take no special action to map them to long-term storage. Persistence independence means that we write our code as if it were operating exclusively on transient, RAM resident, data structures and Napier88 arranges that it works correctly for long-lived data structures.

In Napier88, persistence is defined by *reachability*,[5] which means that data is preserved for as long as it is reachable from the distinguished persistent root. That is, for as long as there is some path of references from the persistent root that would enable a Napier88 program to reach a data item, then Napier88 ensures that that item is preserved. We depend on this property for the integrity of our data structures. Indeed we are able to assume, without recourse to any special operations, that if we traverse a data structure we will not encounter some point where we cannot continue because no arrangements have been made for saving the rest of the structure. Thus as we represent relationships by components of data structures referring to other data structures, Napier88 provides us with a form of referential integrity.

Strong typing

Napier88 ensures that every operation has consistent types before the operation is applied. This is achieved mainly by static analysis during compilation, but flexibility is achieved by a judicious leavening of dynamic checks.[64] The principle of persistence independence ensures that this type safety extends to all long-term data. There are no loopholes in this safety regime.[65]

This type safety protects our data structures from our own errors and from errors in code we are manipulating or examining. Therefore, it greatly accelerated our implementation work. Type safety also establishes and maintains interface compatibility constraints that would otherwise have to be maintained by our Builder and verified by our constraint checker.

Bindings and environments

Bindings are central to our techniques as we examine them to determine dependencies and manipulate them during builds. A *binding* is an association between a name (value identifier) and a value.[21,66] Aho[67] states: ‘When an environment associates

storage location s with a name x, we say that x is *bound* to s; the association itself is referred to as a *binding* of x.' Atkinson and Morrison[68,69] extend the binding concept in Napier88 to comprise an association between a name, a value, a type and constancy (an indication as to whether the value is mutable or not). The same definition is used by Dearle.[70] Although Napier88 has additional means of identifying values, our Builder only needs to consider these named bindings.

As stated in Aho's quotation, *environments* are traditionally considered as the context in which to interpret bindings. Napier88 extended the environment construct by introducing sets of bindings as first-class mutable values.[71] These were implemented by Dearle[70,72] as sets of L-value bindings, i.e. bindings to locations. Code can dynamically bind to locations via these environments and facilities exist for scanning the bindings in an environment, inserting bindings and removing them. It is common programming practice in Napier88 to place environments in the persistent store and to use them to identify and organise the components of an application under construction.[72] The Thesaurus tool scans all the environments it can find in the store using these facilities to capture and report on the properties of bindings in these environments. The Builder will remove a binding and insert one with the same name to implement a type change.

Store semantics

Mutable values in Napier88 can be changed by assignment and other update operations, and therefore behave as typed stores. If these values are themselves persistent, then the new value assigned to the store will persist. This is used by the Builder tool to store program parts, such as procedures. An environment behaves as a store. A newly compiled procedure or a newly constructed value is placed into this store as a binding the first time it is made and subsequently the new value can be assigned to the location provided the type has not changed.

Procedures as first-class values

In Napier88 procedures are first-class values, that is, they have a set of standard operations available that are applicable to all other values.[73] This enables the Builder tool to manipulate the values produced by the compiler, just like any other value. In particular, it means that they can be assigned to a persistent environment, as described above, and then recovered from the persistent store whenever they are needed for some later build.

Transactional model

The traditional definition of transactional is compliant with ACID properties: Atomic, Consistent, Isolated and Durable. We make no use of concurrency and therefore issues of isolation do not arise. The type system, the constraints and the referential integrity combine to provide significant guarantees of consistency. We depend on Napier88's atomic properties.[13] That is, the tools directly update the existing data structures relying on the atomicity provided by Napier88, which restores the persistent store to the last completed stabilisation operation if there is a software failure. This restoration gives some durability although many programmers reinforce this by storing a compressed copy of an earlier state of their store.

Reflection

The Napier88 compiler is a procedure in the persistent store that can be called dynamically and facilitates a particular form of *reflection*. Programs that operate on the persistent store can be generated, compiled and executed at run-time, which means that programs can change their own computational context. This form of reflection in Napier88 is referred to as run-time linguistic reflection.[74,75]

During the execution of a program new source code can be generated, typically on the basis of information becoming dynamically available (information currently in the persistent store, user input, etc.). The newly generated source code can then be analysed by the compiler, which is a procedure taking a string or another source code representation as input. If the compilation is successful, the result can be executed, everything still taking place within the execution of the 'parent' program.

BUILD MANAGEMENT IN A PERSISTENT PROGRAMMING ENVIRONMENT

This section describes a method for constructing applications in the context of a persistent programming language from various kinds of components (executable code or data). Such applications are termed *persistent applications*. If we focus on executable components only, this method corresponds to what is traditionally called linking but provides greater flexibility. However, we treat both data and code consistently, partly because this is the natural approach in systems where procedures and data have the same rights to persistence[73] and partly because we encounter many applications where it is useful to construct and ship systems with collections of data used by the code. The latter case occurs more commonly when persistence is being used. For example, in a health care system, versions of the code were shipped with versions of the database containing icons to be used for displaying things in wards and versions of files containing the templates of forms. These had to match the code.

We now describe the construction method, which is called the *location binding method*, and then how it is implemented in Napier88. The section concludes with a discussion on software constraints that are formulated to support build management.

Location binding method

The location binding method is used by programmers to build persistent applications in order to reduce the costs of rebuilds after a component has been changed. It was invented by Dearle[70,72] and has been described further in several papers.[76–78] Two terms used to describe our use of the location binding method are now introduced:

(a) **code**—any fragment of program, for example a procedure.
(b) **component**—a coherent unit of data or any typed collection of code that is edited, rebuilt (recompiled, linked, etc.) and maintained as a single unit. Components would typically be modelled as *modules* in Pascal, Modula-2[79] and Standard ML,[80,81] *packages* in Ada[82] and Java,[83] etc.

A persistent application is made from a number of components. Each component is separately maintained, and is typically represented in two forms: a source form

suitable for human use and an internal representation suitable for computer use. Some transformation operation is used to create the internal representation from the source. For example, the source might be text representing a procedure and the internal form might be the executable code and the data structures it accesses. In this case, the transformation is achieved by supplying the source to a compiler. As another example, the source might be a map in some standard Geographic Information System format, the internal representation a set of data structures optimised for the application representing a relevant selection of data from the standard format, and the transformation the process that recreates these data structures after the map has been edited. In the remainder of this section, we use *value* to denote the internal representation of a component.

Components in a persistent application need to use other components via bindings. A *persistent binding* is an association between a name and a value or between a name and location that is kept in the persistent store as long as it is needed. In the location binding method, all persistent bindings are between names and locations. The synthesis of an application from its components is achieved by constructing persistent bindings in persistent environments. Each binding associates a name with a persistent location. Components using another component find its location from a persistent environment and statically bind to that location. It is a requirement of the build process that a useful and correct value be placed in a location before any other component attempts to use the value from that location. The type system ensures that there are no type mismatches during this process.

To arrange this, programmers first set up the persistent bindings, by creating persistent locations and giving them names. They may choose to initialise a location with a dummy value or with a first attempt at the final value.

When a component's source is revised (assuming no type change), it is only necessary to re-apply its transformation and assign its result to the correct persistent location identified via the persistent binding. From then on, the components that use this component will use the new value.

It is now apparent why a dummy is assigned in the first instance. There are two reasons:

(a) Initially, many of the required values cannot be filled in because they need to refer to bindings to locations holding other components. Using the dummy value enables the programmer to avoid the effort of topologically sorting the dependencies to write new values in the right order.
(b) When the code to insert the first useful value is written, it is not encumbered with code to set up the persistent location and persistent binding, and to specify their type.

This method avoids the need to discover all the components that use a particular component and rebuild them after that component has been rebuilt. It exploits persistent technology that accommodates persistent locations, persistent bindings and the types of the components used to build the application.

It is preferable if the whole of the persistent store and all the operations on it are strongly typed. This gives the equivalent safety of checking that interfaces match when modules are combined in languages such as Ada (packages).[73] In such languages the interfaces are checked using various kinds of import/export lists, although more

sophisticated interface control mechanisms may be provided by tailored tools, e.g. the AdaPIC tool set[84] in the context of Ada.

Persistence and strong typing, however, imply that if it is necessary to revise the type of some component, more work is necessary. When a component's type is changed, the old persistent binding must be removed and replaced by a new persistent binding with the same name bound to a new location with the new type. All components that are to use the component with its new type then need to be altered and re-transformed.

To illustrate the use of the location binding method, we will work through a simple example outlining how a small application is constructed and how changes can be made to the application. The dependency graph for a very simple application is shown in Figure 2. We use the same notation as was introduced in Figure 1; thus component *a* is dependent on *b* and *c*; *b* is dependent on *d* and *e*; and *c* is dependent on *e*. For each interface of a component there is a corresponding value produced by a transformation operation (e.g. compilation).

To start building the application we create persistent bindings for each of the components and assign a dummy value to each of them, as shown in Figure 3(a). Each of these dummy values will be replaced by a useful value for the associated component, but the order in which the dummy values are replaced is irrelevant. (However, a testing strategy may indicate a particular order of construction.) In Figure 3(b) the insertion of a value for component *a* is shown; the value a_{val} is created and assigned to the persistent location associated with *a*; a_{val} contains references to the persistent locations associated with *b* and *c*. We can now create a value for *b*, for example, and add it to the application without modifying a_{val}.

We can continue adding the values for components until we have completed the application, as shown in Figure 4. Each of the dependencies shown in Figure 2 is represented by a reference to a persistent location.

Suppose that we want to make a change to the component *c*. A new value c'_{val} is created, typically by modifying the source of *c* and transforming it to create the new value. If the type of c'_{val} is unchanged, this new value is assigned to the persistent location associated with *c*, replacing the existing value c_{val}. This is shown in Figure 5. Note that no change is required to a_{val} which references *c* via its persistent location. Notice also that c_{val} can no longer be referenced from this application; it will be garbage collected unless some other object outside the application has a reference to it.

If the type of the new value is changed then the situation is a little more complicated. Again, suppose that we want to make a change to *c* but in this case the new value c'_{val} has a different type from c_{val}. We now have to create a new persistent binding for *c* so that the associated persistent location is of the correct

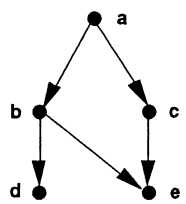

Figure 2. A simple dependency graph

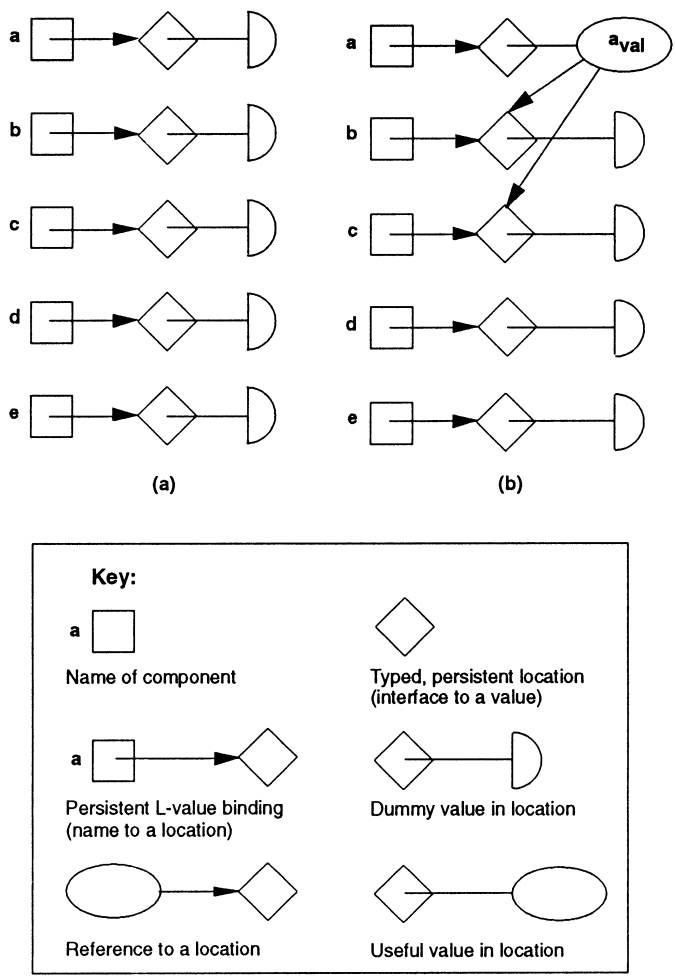

Figure 3. Starting to build a persistent application

type. To use this new value for *c* in the application, it will be necessary to modify *a* and produce a new value a'_{val} that is type compatible with c'_{val}; this new value for *a* will then need to refer to the new persistent binding for *c*. The modified application structure is shown in Figure 6. As the new values of *a* and *c* need to have consistent types, we should commit the changed components, a'_{val} and c'_{val}, in a single transaction. Again, a_{val}, c_{val} and the old persistent location for *c* will be garbage collected if they are unreachable.

Transactional incremental linking

A tool called the Builder, described in a subsequent section, implements a method for transactional incremental linking. The need for such support was perceived by studying the current practice of Napier88 programmers.[85] This section discusses the basic concepts of the linking method. The discussion is initially focused on building

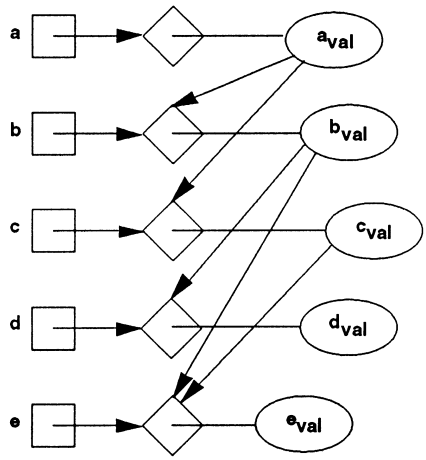

Figure 4. The complete application

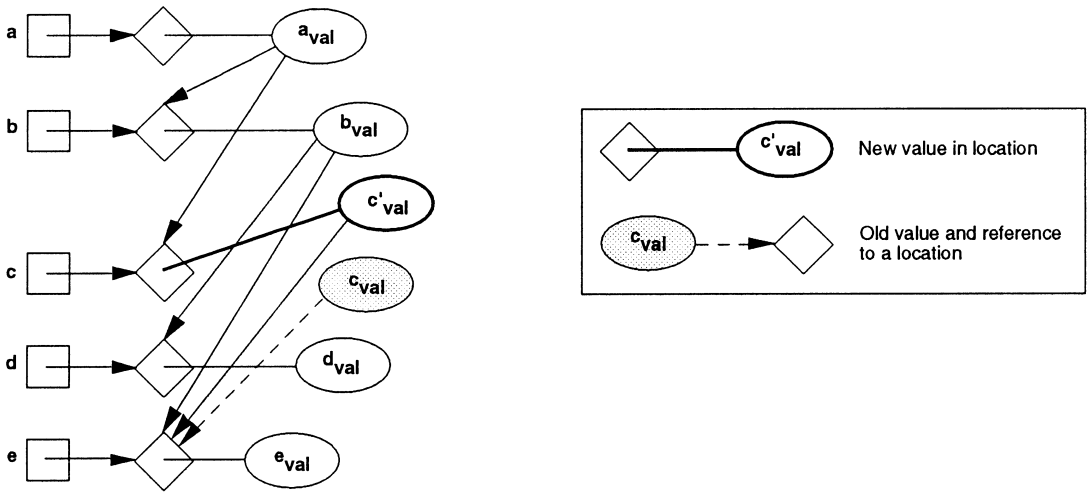

Figure 5. Changing the value of **c**

applications from procedures using source editing, compilation and updating persistent locations. When a persistent location is updated with a new executable version of the procedure, this is the equivalent of linking in a conventional system.

The Builder operations are applied directly to the existing application during a re-build. We rely on the atomicity provided by Napier88 here. If atomicity were not guaranteed then a failure part way through a re-build would leave the application in an inconsistent state from which it might not be easy to recover. However, Napier88 will restart from the last stabilise if a failure occurs. The Builder performs stabilises when the application is in a consistent state, e.g. when a re-build is complete, and thus can directly manipulate the application. We refer to this as transactional linking.

The basic steps of the location binding method are now described in further detail:

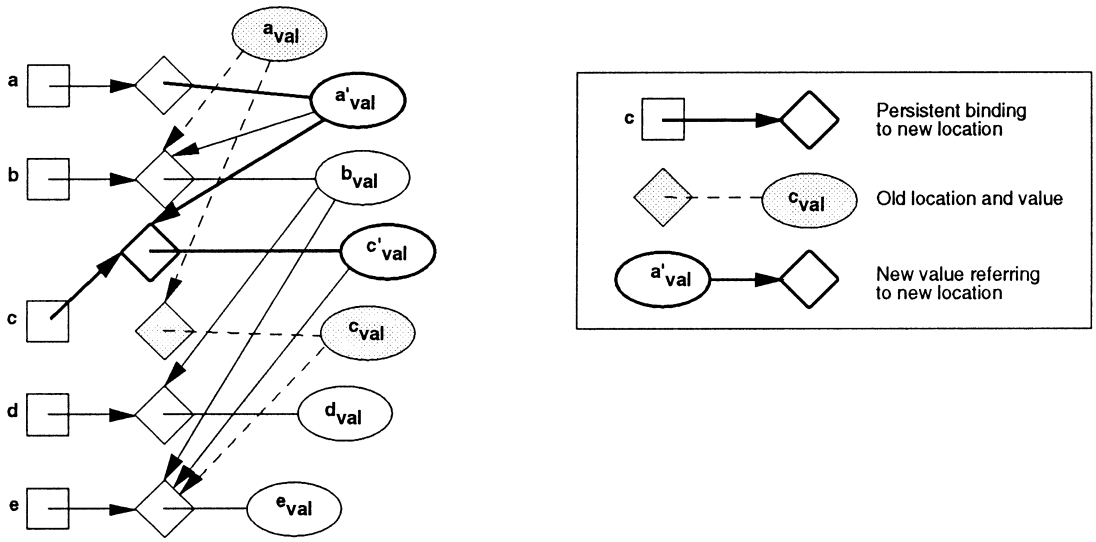

Figure 6. Changing the type of **c**

(i) For each new component, create a persistent location and insert therein a dummy value.
(ii) Create or modify the source corresponding to a useful value of the component, compile the source and update the component's location with the new executable value; this new value will contain required references to the locations of other components.
(iii) If the component's type is changed: delete the existing location, create a new location with the new type and insert a dummy value, as under (i), then repeat step (ii).

To simplify the implementation of these steps, we have defined three categories of transaction (Table I) that include different kinds of operation affecting the persistent store. Insert-transactions execute code that creates locations with dummy values in the persistent store (cf. step (i)).

Implementing step (ii), an update-transaction executes code that finds and establishes references to locations of components required by the component under

Table I. Categories of transaction

Category	Explanation
Insert-transaction	inserts at least one binding into the persistent store, but neither updates a persistent location nor deletes any binding
Update-transaction	updates at least one persistent location and establishes references to other persistent locations, but neither inserts nor deletes any binding
Delete-transaction	deletes at least one binding, but neither updates a persistent location nor inserts any binding

construction, and updates the component's location with the newly constructed version. Incremental development is supported in that the source of a component can be edited and compiled, and a new value, including references, inserted into the location by executing the corresponding update-transaction. There is no need for editing, recompilation or updating of the other components using the new value. Hence, this approach significantly reduces the need for cascades of rebuild operations.

Implementing step (iii) involves all three categories of transaction. A delete-transaction executes code that deletes the location of the component whose type is to be changed. Then insert- and update-transactions are invoked to carry out the work as described in steps (i) and (ii). In addition, all the components that use the changed component must themselves be the subject of the tasks of step (ii), so that the correct references to the new value of the changed component are established.

At present most of the components in Napier88 applications are procedures,[85] representing executable code in the persistent store. As the usage of this method increases, however, one might expect more widespread use of other kinds too. For example, complex data structures such as symbol tables, tables with geographical information, lists of images, etc. can also be initially created as locations with dummy values and be the subject of the method.

The method has been followed by experienced Napier88 programmers and has been taught to students. Hitherto, however, they have had to write and invoke all the transactions and operations themselves. Specific transactional properties are provided by the underlying store technology of Napier88. Following the method, Napier88 provides a general mechanism for transactional incremental linking under programmer control—programmers can replace a set of references in one transaction.

The method has been found to enhance maintainability of applications, but is tedious to apply manually. This provides a justification for implementing a support tool that automates as much as possible of the process (see description later in the paper).

Software constraints supporting build management

As a means to achieve reliable and efficient build management in our environment using the location binding method, we have defined a set of application independent, automatically checkable constraints over the software of an application. We start by describing some general constraints defined in the context of Napier88 and finish by describing constraints that support the incremental linking method in particular. A range of other software constraints, also intended to support application consistency and maintenance but not build management in particular, are described elsewhere.[86] It should be emphasised that a constraint violation is not necessarily an error but may be an anomaly indicating a situation that is liable to errors or inefficient build management.

Constraint (a) of Table II aims to prevent unused declarations, which are confusing, contribute to unnecessarily verbose code and are a potential problem concerning change; a study of FORTRAN programs found a correlation between the proportion of unused variables and fault rate.[87] Unused declarations are a particular problem in build management for two reasons. First, they may initiate unnecessary rebuild operations since dependency graphs are produced on the assumption that an identifier is used within a component if it is declared as used within that component. Second,

Table II. General build management constraints

(a)	All imported names should be used (type definitions and bindings declared in a use-clauses).
(b)	There should be a partial order among components that define type definitions and those that use the types.
(c)	There should be a partial order among components that create bindings in the persistent store and those that use the bindings.

the rebuild itself is usually unnecessarily expensive since the processing environment is unnecessarily large—the big inhale problem.

Empirical data shows that unused declarations are a real problem. In a study we conducted of Napier88 we found that 24% of all type definitions and 10% of the identifiers declared in use-clauses were unused.[85] Other studies of imported names that are unused report similar figures (from 7% to 20%).[28,88] Measurements of violation of the other constraints can be found elsewhere.[85,89]

As mentioned before, dependency graphs defined according to certain criteria are the basis for build management, and a requirement is that they are directed acyclic graphs. We have encountered systems shipped to us that could not be built because they contained cycles. Presumably, they got into this state because of a series of incremental changes that could each individually be rebuilt in the context of the partially constructed system at the original site. Constraint (b) of Table II helps ensure that there are no cycles among a set of type definitions; constraint (c) helps ensure that a binding is inserted into the persistent store before it is used. For example, a data structure must be created before it can be populated, a location must be created before it can be assigned a value, etc.

Table III shows a set of constraints that are tailored to support the incremental linking method and produce atomic increments to minimise the component grain size. Adherence to these constraints indicates that this method is followed. The code available in each component for analysis by the constraint checker comprises two parts. It is predominantly the application code that is being processed and assembled into the application system. The other part is a small amount of code that directly specifies the assembly process. This code is also written in Napier88, which is remarkable in being able to describe binding processes.[70] In principle, this part could be automatically generated by the Builder. The constraint checker is also able to

Table III. Constraints supporting the incremental linking method

(a)	A program component should contain code for at most one of: inserting a binding, deleting a binding or modifying the value of a persistent location, i.e. a program component should correspond to at most one of the three categories of transaction defined in Table I
(b)	For each insert-declaration of a location with a dummy value there should be exactly one corresponding program component capable of performing an insert-transaction, one program component capable of performing an update-transaction and at most one program component capable of performing a delete-transaction
(c)	The transactions under (b) should operate on only one persistent location
(d)	A binding required by a component should be present in the persistent store

scan persistent environments to verify that bindings exist and to inspect their properties. The constraint checking establishes consistencies among these three, potentially independently manipulated, parts: application code in components, assembly specification in components and persistent bindings in the store where the application is being built.

Besides directly supporting and simplifying the implementation of the linking method, constraint III (a) attempts to improve the way applications are organised around the persistent store in that it restricts each component to perform only one kind of operation on the store and thus encourages the creation of small and well-defined compilation units.

Constraint III (b) is defined to help ensure that all needed transactions exist for any binding initially created as a location with a dummy value. For example, if no update-transaction is present, the value will remain dummy except in the case of constant data. There should be exactly one transaction of each kind; several transactions of the same kind complicate the application structure unnecessarily, may cause confusion and will increase the risk of linking errors. Executing more than one insert-transaction may cause an attempt to re-declare a binding; executing several delete-transactions may cause an attempt to delete a binding not present in the store. These faults could be detected and prevented by dynamic checks, but the Builder analysis can normally detect them statically and give more timely and informative warnings.

Defining the transactions at a fine granularity, in compliance with constraint III (c), may lead to improved build performance and programming precision. For example, if a procedure in a persistent location is to be updated, the transaction to be executed will only update that location with a new value. If the transaction had operated on several locations, it would unnecessarily have updated other locations and initiated rebinding to them as well. Even though this would not have lead to inconsistency, since the new values in those locations would have been identical to the existing values, unnecessary work would have been carried out and unnecessary space allocated.

Constraint III (d), which states that a binding specified to be used by a component should be present in the persistent store, aims to increase the likelihood of successful linking. We can guarantee the existence of a location and value of the right type, but we cannot guarantee that the value will be useful. In a conventional programming environment a corresponding constraint would be that a file to be used by a program should exist to avoid potential run-time errors. For example, if such a file is unintentionally deleted or renamed, or if this is done intentionally but the programmer forgets to change the program accordingly, then the inconsistency may not be detected before a program attempts to access the file at run-time. Again, the values in the file may not be useful.

It is generally difficult (in some cases hardly possible) and invariably time consuming to check software constraints manually. Hence, their success depends heavily on a supporting environment that automatically checks constraint adherence and provides relevant information in the case of violation.

The constraints of Tables II and III are part of a larger collection of constraints that can be checked by a tool called SPASMCheck.[86] For each violation of a constraint, SPASMCheck gives a warning and indicates the source of the violation. (It is then the responsibility of the programmer to rectify the inconsistent state.)

Only warnings are given since the constraints are not mandatory. For example, during initial application construction, violation may be the general case. There may also be other cases where the constraints could be relaxed. For example, regarding constraint III (c), there are pragmatic reasons for allowing a 'coherent group of persistent locations', instead of only one location, to become the subject of one transaction.

Constraints III (b) and (c) are strictly enforced internally in the Builder implementation and are a means to ensure that the incremental linking is supported correctly. The parts of these constraints that concern insert- and delete-transactions are not irksome for programmers using the Builder since those transactions are created automatically.

An update-transaction requires the programmers to provide a component with source code corresponding to the value to be put into the persistent location. The Builder cannot prevent a programmer from having several versions of such a component, but it can give support for only one value at a time. The Builder applies to a given version of an application; if a programmer wishes to have several alternative update-transactions, then it is necessary that the version to be used by the Builder be explicitly distinguished. The implementation of the Builder is discussed further in the next section.

THE BUILDER

The Builder tool implements support for the model of selecting dependency subgraphs to which rebuild should apply and the method for transactional incremental linking described earlier in the paper. This section describes the Builder's working context, the input information it requires, its basic features and its user interface, which is implemented using the WIN persistent window management system.[90] Finally, some particular implementation issues are discussed.

Builder context

The Builder is a tool in the *Glasgow Workshop*,[91] which is the basis of an experiment to demonstrate how a persistent working environment can be built using the existing persistent technology. The Workshop consists of a number of *Workbenches*, each of which in turn provides some rudimentary tools for persistent software engineering and also provides us with experience of building a substantial persistent application system.

On a Workbench are a number of *Workitems*, which can be almost anything, such as a collection of data, a string, a program, a suite of programs, a tool or a Workbench. There are tools that create, copy and remove Workitems. All Workitems have certain minimal properties, such as: a name, a history, creation date, modification date, and creator.

Certain Workitems have additional properties. For example, *source* and *executable* are two attributes specific to a *Program Workitem*. The association between the source and the executable is automatically maintained in a name-independent way, which is in contrast to the name-based association in traditional file systems. The Workbench also keeps timestamps recording when a Program Workitem was last edited, compiled, linked, etc. Our design decision to include this as part of a

Workbench's information about Program Workitems, and not let the Builder maintain it separately, has the advantage that timestamps may also be used and updated by other tools. For example, a user can invoke the compiler independently of the Builder.

Workitems that combine to form an application are collected together in a Workbench as a *Persistent Application System* (PAS). A PAS maintains direct references to all its components. Programmers introduce a new component to a PAS by using the appropriate Workbench tool.

The Workshop logs information about any tool operation such as: user, time, tool and operation, name of the components being the subject of the operation. It also keeps a history of the results, e.g. compilation and execution errors.

Creating the dependency graph

To create the needed dependency graphs and to check the constraints of Table II and III, the Builder exploits the information stored in the *thesaurus*, which is a fine-grained, cross-reference database containing timestamped information about all user-introduced identifiers occurring in the source programs of a PAS and the names of the bindings to code and other data in the associated persistent stores.[92] We have provided a user interface to allow selective viewing of all the thesaurus data. However, for consistency checking purposes we deliberately include *all* identifiers, because inconsistency may occur at any level of granularity.

A thesaurus entry holds information such as: the *name*, *type*, *constancy* of an identifier and the *usage* and *context* of identifier occurrences. The property *usage* indicates how the identifier is being used, e.g. declaration or use of a type identifier, or declaration, left context or right context of a value identifier. Context indicates whether the identifier occurs in an environment operation or as a declaration of a type parameter, procedure parameter, structure field, variant tag, etc. or as a dereferenced structure field, projected variant, etc. Various kinds of dependency among components (type dependency, procedural dependency, etc.) are easily inferred from the thesaurus.

All the contents of the thesaurus are automatically maintained. The whole PAS is analysed, and the thesaurus updated, regularly at times specified by the user. A full analysis and update can also be initiated at any time. The part of the thesaurus analyser that processes Napier88 source programs is based on the Napier88-in-Napier88 compiler.[77] The lexical and syntax analysers have been adjusted to conform to the special information needs of the thesaurus. Instead of generating executable code, the thesaurus analyser extracts the needed information and inserts it into the thesaurus. The part of the thesaurus analyser that extracts information from the persistent store re-uses low-level procedures used in the implementation of a Napier88 browser.[93] The principles of the thesaurus have been generalised in an industrial (C, C++, X Window System and relational database) environment.[94]

The major motivation for automatic production of dependency information is *reliability* and *efficiency*. The dependency information inferred from the thesaurus is accurate, unless the thesaurus content is out of date, i.e. the application has changed since the last thesaurus update. The timing of thesaurus update and the Builder's extraction of dependency information is therefore a crucial issue.

One approach is to perform a total PAS analysis and production of dependency information before each build. Applying such a very expensive approach one may

question how much of the gain of deferring or avoiding unnecessary rebuild is lost. A smarter approach is to incrementally update the dependency information before each build, although this approach also incurs an additional cost. We have defined the algorithm for incrementally updating the dependency graph and it is currently being implemented so that it can be included in the next release of the Builder, but at present the dependency information is updated in quiescent periods.

Incremental update of a dependency graph

This section describes the incremental update algorithm (Figure 7). The assumptions are as follows:

(i) An application consists of a set of components **N**.
(ii) There is a set of changed components **C**, where **C** $\subseteq$ **N**.
(iii) An *element* is a binding or type declaration that is defined by (name, component). We assume that **name (e)** returns the name of the element **e**. If a component uses an element provided by another component, then the former depends on the latter. The name of the source component providing the element **e** is returned by **source (e)**.
(iv) Each component **n** in **N** is characterised by the following:
D_n the set of elements (name, source component) upon which **n** depends.
P_n the set of elements (name, using component) provided by **n** that are used by other components.
A_n the set of names of elements of **n** that are available to be used, where $\forall$**p** $\in$ **P_n**: name(p) $\in$ **A_n**.
U_n the set of components that use elements of **n**, which can either be stored explicitly or generated from **P_n**.

The algorithm requires that we identify **C**, the set of changed components, either by incrementally recording changes or by scanning **N. U** is the set of all components that currently use elements in **C**, which is going to be built up during step 1 of the algorithm.

The first step is to analyse each changed component and update the set of elements upon which it depends (**D_c'**); at the same time we can update the set of elements it makes available (**A_c'**). We can then identify the new elements that are required by

0. Discover C
   $U = \varnothing$

1. for each c in C
   {analyse the source of c to construct D_c' and A_c'
   $\Delta D_c = D_c' - D_c$
   $U = U \cup U_c$
   $P_c' = P_c \downarrow A_c'$ *restrict* P_c *to those names occurring in* A_c'}

2. for each u in U
   {for each d in D_u {if name(d) $\notin A_{source(d)}$ then issue error}
   if u $\in$ C then for each d in ΔD_u add (name(d), u) to $P_{source(d)}$ and add u to $U_{source(d)}$}

Figure 7. Update algorithm

this component ($\Delta\mathbf{D}_c$); add all components that currently use the component to $\mathbf{U}$ and ensure that the set of elements that are identified as provided ($\mathbf{P}'_c$) is restricted to those named as available in $\mathbf{A}'_c$.

At the end of this pass through $\mathbf{C}$ we know that the set of dependent elements ($\mathbf{D}_c$) for each changed component is correct and that the set of elements it makes available ($\mathbf{A}_c$) is also correct. However, we cannot guarantee that the newly required dependent elements ($\Delta\mathbf{D}_c$) are actually available in the source components; nor are we sure that the provided set ($\mathbf{P}_c$) and usage set ($\mathbf{U}_c$) of the changed component are accurate.

The second step is to scan $\mathbf{U}$ and check the consistency of each component in this set. For each component in $\mathbf{U}$ we check whether the elements required by this component are still available in the source components, if not an error is issued as a build will fail. If the component in $\mathbf{U}$ is also in the changed set $\mathbf{C}$ then we need to update the set of provided elements in each source component that provides a new required element and the set of used components in each set that uses such components.

Selecting subgraphs and performing builds

The Builder implements the general model for performing build management on sub-graphs as described in the second section of the paper. The programmer can constrain rebuild to a specific part even though other parts may also have been changed. The performance gain of this approach may be significant. Another advantage is that it eliminates the need for programmers to apply operations such as the manual ‘touch’ of derived files in Unix. That is, to avoid rebuilding programs that have been the subject of only cosmetic changes (in the programmers’ opinion), programmers often mark derived files as being up-to-date by ‘touching’ them, but this technique is very *ad hoc* and unsafe since it relies on the programmers to accurately determine the effect of the changes.

The user interface is shown in Figure 8. In that example both compilation and linking are requested. A specific target is selected (indicated by the Workitem name appearing on the right hand side of the ‘Build Target’ button), and the rebuild actions should be propagated (the ‘Propagate Changes’ check box is set). This corresponds to the second operation of the model described earlier.

During the building process the Builder reports the name of the program component being compiled or linked and any error messages. The Builder can also be invoked in the ‘No Execution’ mode in which the rebuild is not actually carried out, but the dependencies are calculated and a trace is displayed as if the rebuild were really carried out, cf. the *-n* parameter of Make.[32]

If a given target has been requested to be built, the Builder reports the name of the components that must be built before the target itself can be built. If the changes should propagate to dependent components, their names are also reported before they are rebuilt.

To process the components in a legal linear order, the Builder performs a topological sort on the dependency graph and carries out the rebuild actions according to a generated legal ordering if such an ordering exists. If a cycle is detected, the Builder reports the names of the components constituting the cycle.

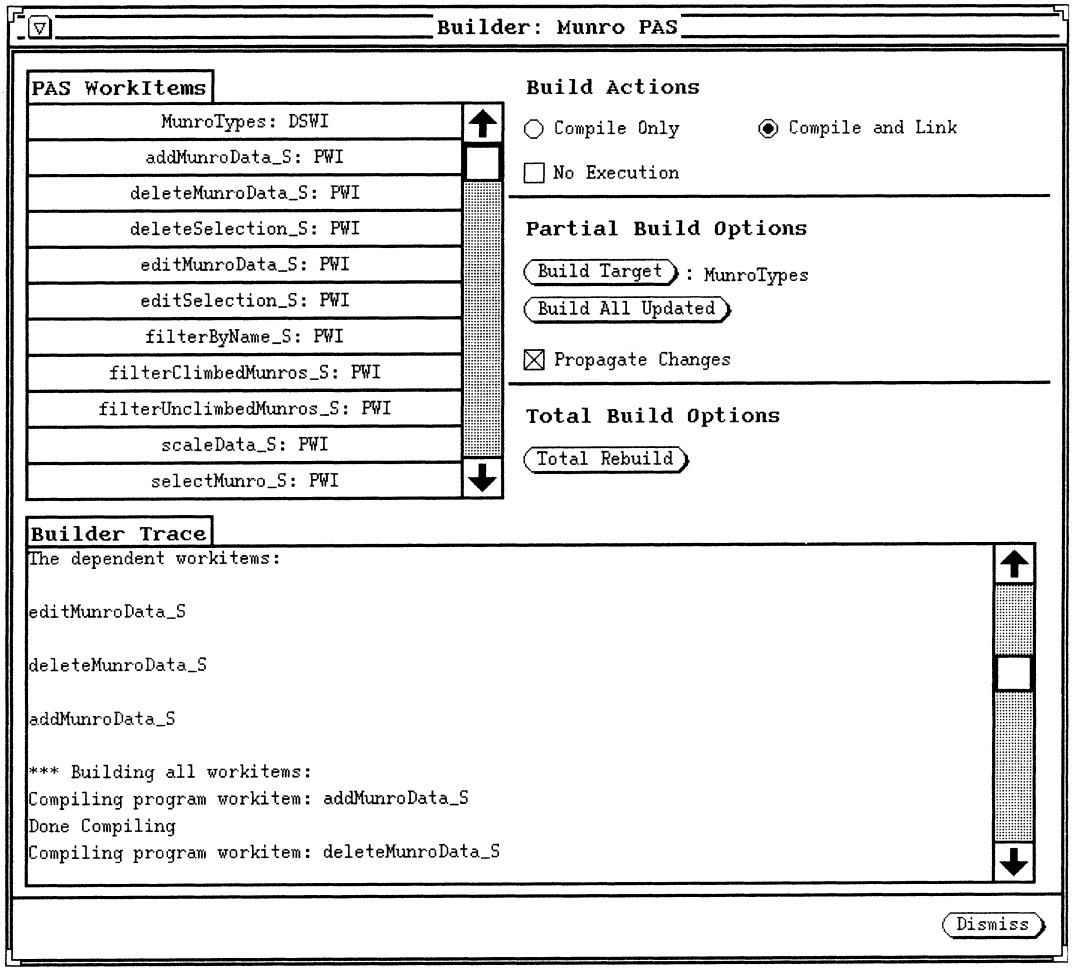

Figure 8. The Builder's user interface

Builder implementation

Managing recompilation includes first recompiling, in a correct linear order, all changed components, within the selected subgraph, that contain commonly used declarations, and then, in any order, all other components needing recompilation. The Builder ensures that the programs are compiled against the needed set of (precompiled) declarations.

Implementing the method described in the 'Transactional incremental linking' section is more complicated. The Builder carries out the four basic tasks as follows:

(a) Detect the source components that have changed since last rebuild.
(b) For each such component, identify the steps of the incremental linking method that must be carried out.
(c) Create or locate the necessary transactions.
(d) Execute these transactions in a correct order.

The timestamps recorded as a property of a Program Workitem, representing components in this context, make the first task trivial.

A value is put into a location (i.e. the location updated) in the persistent store by means of a Program Workitem that contains the name and path of the location and an expression describing the type and value. Such Program Workitems are identified by the Builder on the basis of the source code information in the thesaurus and are the only ones involved in the linking. If the location already exists and the type is unchanged, only an update-transaction needs to be executed.

For each Program Workitem that is supposed to insert a value into a persistent location, the Builder checks whether there exists a location with the appropriate name, path and type. Since we are in a strongly typed environment, the location contains a description of its type. This information is accessible from the Builder. The callable Napier88 compiler is called from within the Builder and it returns the type of the value that should be inserted.

If there is no location that matches the name and path, the Builder generates a corresponding insert-transaction with the appropriate name, path and type. If the name and path match but not the type, then the Builder generates a delete-transaction and then the insert-transaction. The location must be deleted and re-created to change its type so that existing software still refers to the old location, and therefore remains type correct, whereas new transformations of source code will only be able to refer to the new location with its new type. An earlier description of the way in which locations may be re-created manually is given elsewhere.[78]

The Builder's approach to execute the transactions in a correct order is generally applicable. (There might be other legal approaches in certain cases.) The Builder executes all delete-transactions before all insert-transactions, which in turn are executed before all update-transactions. In addition, if an insert-transaction is executed, the corresponding update-transaction and all update-transactions dependent on that insert-transaction, are subsequently also executed.

CONCLUSIONS

Build management is a challenging issue when creating and maintaining long-lived, large application systems. Longevity means that programmers making changes are unlikely to be familiar with much of the system that they change. Large means that most builds reconstruct relatively small parts of the system but in a complex context.

Tool support is crucial to keep track consistently of which components must be rebuilt before a target can be rebuilt and determine how changes should be propagated in a dependency graph. Avoiding unnecessary rebuilding or deferring rebuilding is particularly important in large application systems. A model presented in this paper offers flexibility in that the programmer may confine the rebuilding operations to be applied to sub-graphs of the overall dependency graph. The model is implemented by a tool called the Builder.

More specifically, to identify build management requirements in our programming context, which is centred around the persistent programming language Napier88, we collected empirical data from 20 applications consisting of more than 108,000 lines of source code. The study* provided information about how programmers develop their applications and initiated work on improving the process.

* The full study is reported elsewhere.[85]

In our environment, most of the classical build management has previously been performed either in an *ad hoc* way or by use of Make. Programmers have had to infer dependencies and create and maintain Makefiles manually. The Napier88-specific Builder tool provides derived component management, cascading and smart rebuilding and semi-automatic support for reducing the big inhale.

A persistent programming language unifies database programming and conventional application programming. Code and (complex) data can be stored in persistent locations. The Builder actively supports a programming method that had become popular and that exploits this extended, unified processing environment. The method includes a form of incremental linking.

To further support build management, we defined a set of automatically checkable constraints, some of which help reduce the amount of code that needs to be rebuilt. We believe that the more programmers adhere to such a method and its associated constraints, the more efficient and effective the build management tools can be.

By using an orthogonally persistent system we have gained the benefit of convenient management of the long-term data structures used in the re-building process. As Napier88 bases data lifetimes on whether there is a path that can be followed by a Napier88 program to that data, we have been able to work with the simple assumption that data will still exist if we have a way to access it. This significantly simplified coding and made it unnecessary to verify the existence for applications that were under construction. One of the roles of the Builder is to ensure that data is useful before it is used.

The atomicity of Napier88's persistent store updates meant that we could safely, directly update the application during a re-build. The absolute imposition of type checking provided by Napier88 meant that it was not necessary to carry out equivalent checks in the Builder and constraint checker. We believe that the strong type checking also accelerated our implementation. The provision by Napier88 of accessible and manipulable persistent sets of bindings to locations permits the scanning necessary for constraint checking and the updates necessary for incremental binding. Other data structures, e.g. tables or O_2's named persistent objects,[95] could have performed part of this role.

The Napier88 programming environment provides a callable compiler. Being able to call the compiler from within the language and having control over the typed information returned (enabled by run-time linguistic reflection) make it convenient to implement a tool that provides extensive information to the programmers during the building process.

FUTURE WORK

The current release of the Builder has revealed several possibilities for future work. A long-term goal is to make the build management tool reported in this paper interoperable with general configuration management and version control tools. Although some ideas have been proposed in the literature,[11] it is still an open question whether a persistent programming environment will enable more sophisticated and productive configuration management and version control tools to be built than those found in other language environments. A model for configuration management and version control based on immutable components supported in our persistent program-

ming environment should be compared with the model of Vesta[41] and Forest,[12] which are also based on immutable components.

Current developments are user requested work on the visualisation of dependency graphs and on integration with other tools that automatically derive information from source modules, design data, etc. For example, the documentation tools and dependency graph mechanisms both need to perform scans and present dependencies. It may be possible to save resources and improve uniformity by combining them. Rebuild operations are simulated by the existing Builder in its non-execution mode, but more information could be given to the user if the dependency graphs were displayed graphically. For a realistic application the dependency graph will be large and have many connections; there may be hundreds or even thousands of components in the graph. There are some interesting problems in presenting such large graphs including: how to scale or partition the graph; providing different views of the graph (for example, using 'fish-eye' views); using colour to highlight certain features of the graph or to show the impact of changes. A typical application will use many standard library components, and the visualisation of an application will be simplified if we can 'prune' the graph to remove components that are regarded as immutable, e.g. because they are provided and not modifiable by the programmer.

The current Builder exploits the features of a persistent programming language. To exploit the existence of other rebuild related tools in our environment, work is being carried out on enhancing the Builder to control rebuild activities other than recompilation and linking. The design of the interface of the next Builder release also includes the option of specifying *incremental* or *total* update of the information generated by a fine-grained cross-reference tool,[92] another tool that extracts and displays documentation of program components[96] and an information retrieval tool for exploring library components.[96]

In the context of these tools, incremental update, like separate compilation and incremental linking, is provided for performance reasons and will, if specified, be applied to all newly linked components. Total update is typically useful if the application has been installed in a new environment or if one believes, for some reason, that not all the incremental updates have been successfully carried out.

In this paper we have described a build management system implemented in a persistent programming environment. However, we believe that much of our work is of relevance in a much wider context. We have defined and implemented a build management model that chooses to use a fine granularity in two respects:

(a) the units of dependence are named bindings (named types, named values and named locations) rather than larger units such as programs, modules or files;
(b) the steps in a software rebuild are subdivided into type changes, insert-transactions, update-transactions and delete-transactions, which are each treated differently by the build manager.

This finer granularity increases the size and complexity of the dependency graph, making it infeasible to prepare or maintain it manually, but it allows increased avoidance of redundant rebuilds and reduces the source data examined during rebuilding.

The introduction of convenient ways of specifying different subsets of the graph that are to be rebuilt also provides economies. By utilising these two effects and by

reducing dependencies using the location binding mechanism it has proved possible to arrange acceptably fast rebuild times.

The build method reported in this paper could also be implemented using an OODBMS and some programming language. However, for most programming languages in industrial use, the following additional work would fall on the implementation of the Builder:

(a) a transactional, orthogonally persistent store;
(b) updateable and nameable persistent locations that can store the components being the subject of build management and enable direct references between those components;
(c) a source code analyser that automatically extracts the information required to infer build dependencies and provide consistency checking; and
(d) a callable compiler to simplify the implementation of the build management tools.

Work is under way to develop a system to provide orthogonally persistence for Java.[97] One of the planned uses of this system is support for the development of existing build management tools.[12] This build manager will benefit from similar persistence support to that we have obtained from Napier88, but in the context of a commercially supported language. It will then be interesting to investigate whether the constructs described in this paper will be beneficial in that context.

ACKNOWLEDGEMENTS

The St Andrews persistent programming team provided the underlying language technology for the work described in the paper. We would like to thank Reidar Conradi, Quintin Cutts, Stuart Macneill and the anonymous referees for their detailed, valuable comments. Dag Sjøberg was supported by a post-doctoral fellowship from the Research Council of Norway. All the authors benefited from a collaboration project between Norway and UK funded by the Research Council of Norway and the British Council.

REFERENCES

1. W. F. Tichy, *Configuration Management*, John Wiley & Sons Ltd, 1994.
2. R. Adams, A. Weinert and W. Tichy, 'Software change dynamics or half of all Ada compilations are redundant', *Proc. European Software Engineering Conf.*, Lecture Notes in Computer Science 387, Springer-Verlag, pp. 203–221.
3. R. W. Quong and M. A. Linton, 'Linking programs incrementally', *ACM Trans. on Programming Languages and Systems*, **13**(1), 1–20 (1991).
4. L. Ferraby, *Change Control During Computer Systems Development*, Prentice-Hall, Hemel Hempstead, 1991.
5. M. Atkinson and R. Morrison, 'Orthogonally persistent object systems', *VLDB Journal*, **4**(3), 319–401 (1995).
6. R. Morrison, F. Brown, R. Connor and A. Dearle, 'The Napier88 Reference Manual', *Technical Report PPRR-77–89*, Universities of Glasgow and St Andrews, 1989.
7. R. Morrison, A. L. Brown, R. C. H. Connor, A. Dearle, G. N. C. Kirby and Q. I. Cutts, 'The Napier88 Reference Manual (release 2.0)', *Technical Report CS/93/15*, University of St Andrews, 1993.
8. M. P. Atkinson, 'Programming languages and databases', *Proc. Fourth Int. Conf. on Very Large Data Bases*, Berlin, West Germany, 13–15 September 1978, pp. 408–419.
9. G. Copeland and D. Maier, 'Making Smalltalk a database system', *Proc. ACM SIGMOD 1984 Conf. on the Management of Data; ACM SIGMOD Record*, **14**(2), 316–325 (1984).

10. M. P. Atkinson and O. P. Buneman, 'Types and persistence in database programming languages', *ACM Computing Surveys*, **19**(2), 105–190 (1987).
11. R. Morrison, R. C. H. Connor, Q. I. Cutts, V. S. Dunstan and G. N. C. Kirby, 'Exploiting persistent linkage in software engineering environments', *Computer Journal*, **38**(1), 1–16 (1995).
12. M. Jordan and M. L. van der Vanter, 'Software configuration management in an object oriented database', *Proc. USENIX Conf. on Object-Oriented Technologies*, Montery, Canada, 26–29 June, 1995.
13. A. L. Brown, 'Persistent object stores', *PhD Thesis*, University of St Andrews, 1989.
14. A. L. Brown and R. Morrison, 'A generic persistent object store', *Software Engineering Journal*, **7**(2), 161–168 (1992).
15. E. Moss and A. Sinofsky, 'Managing persistent data with Mneme: designing a reliable, shared object interface', *Advances in Object-Oriented Database Systems*, Springer-Verlag, Berlin, 1988, pp. 298–316.
16. M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White and M. Zwilling, 'Shoring up persistent applications', *Proc. ACM/SIGMOD*, 1994, pp. 383–394.
17. D. S. Munro, R. C. H. Connor, R. Morrison, S. Scheuerl and D. W. Stemple, 'Concurrent shadow paging in the Flask architecture', *Proc. Sixth Int. Workshop on Persistent Object Systems*, Tarascon, Provence, France, 5–9 September 1994, Springer-Verlag and British Computer Society, pp. 16–42.
18. D. B. Leblang, 'The CM challenge: configuration management that works', in W. F. Tichy (ed), *Configuration Management*, John Wiley & Sons Ltd, Chichester, 1994, pp. 1–37.
19. D. B. Leblang and R. P. Chase Jr., 'Parallel software configuration management in a network environment', *IEEE Software*, **4**(6), 28–35 (1987).
20. E. Borison, 'A model of software manufacture', *Proc. Int. Workshop in Advanced Programming Environments*, Lecture Notes in Computer Science 244, Springer-Verlag, pp. 197–220.
21. C. Strachey, *Fundamental Concepts in Programming Languages*, Oxford University Press, 1967.
22. L. Cardelli and P. Wegner, 'On understanding types, data abstraction, and polymorphism', *ACM Computing Surveys*, **17**(4), 471–522 (1985).
23. M. Burke and L. Torczon, 'Interprocedural optimization: eliminating unnecessary recompilation', *ACM Trans. on Programming Languages and Systems*, **15**(3), 367–399 (1993).
24. H. Eidnes, S. O. Hallsteinsen and D. H. Wanvik, 'Separate compilation in CHIPSY', *ACM SIGSOFT, Software Engineering Notes*, **16**(7), 42–45 (1989).
25. 'CCITT High Level Language (CHILL)', Recommendation Z.200, CCITT, ITU, Geneva, 1980.
26. D. E. Knuth, *Fundamental Algorithms, The Art of Computer Programming*, Addison-Wesley, Vol. 1, 1973.
27. R. Adams, W. Tichy and A. Weinert, 'The cost of selective recompilation and environment processing', *ACM Trans. on Software Engineering and Methodology*, **3**(1), 3–28 (1994).
28. R. Conradi and D. H. Wanvik, 'Mechanisms and tools for separate compilation', *Technical Report 25/85*, The Norwegian Institute of Technology, University of Trondheim, Norway, 1985.
29. M. Rain, 'Avoiding trickle-down recompilation in the MARY2 implementation', *Software—Practice and Experience*, **14**(2), 1149–1157 (1984).
30. S. Dart, 'Concepts in configuration management systems', *Proc. Third Int. Workshop on Software Configuration Management*, Trondheim, Norway, 12–14 June 1991, pp. 1–18.
31. J. Estublier, *Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops*, Lecture Notes in Computer Science 1005, Springer-Verlag, 1995.
32. S. I. Feldman, 'Make—a program for maintaining computer programs', *Software—Practice and Experience*, **9**(4), 255–265 (1979).
33. K. Walden, 'Automatic generation of make dependencies', *Software—Practice and Experience*, **14**(6), 575–585 (1984).
34. T. Brunhoff and J. Fulton, 'Imake—C preprocessor interface to the make utility', *Technical Report, Unix Programmer's Manual, X Window System Version 11, Release 5*, 1994.
35. R. M. Stallman and R. McGrath, 'GNU Make, a program for directing recompilation', *Edition 0.48, Version 3.73 Beta*, Free Software Foundation, Inc. Cambridge, MA, April 1995.
36. R. A. Olsson and G. R. Whitehead, 'A simple technique for automatic recompilation in modular programming languages', *Software—Practice and Experience*, **19**(8), 757–773 (1989).
37. G. M. Clemm, 'ODIN—an extensible software environment', *Technical Report CU-CS-262–84*, Department of Compouter Science, University of Colorado, Boulder, USA, 1984.
38. G. M. Clemm and L. Osterweil, 'A mechanism for environment integration', *ACM Trans. on Programming Languages and Systems*, **12**(1), 1–25 (1990).

39. D. B. Leblang and R. P. Chase Jr., 'Computer-aided software engineering in a distributed workstation environment', *ACM SIGPLAN Notices*, 104–112 (May 1984).
40. 'ClearCase Concepts Manual', Atria Software Inc., 1992.
41. R. Levin and P. R. McJones, 'The Vesta approach to precise configuration of large software systems', *Technical Report 105*, DEC Systems Research Center, Palo Alto, CA, USA, June 1993.
42. M. R. Brown and J. R. Ellis, 'Bridges: tools to extend the Vesta configuration management system', *Technical Report 108*, DEC Systems Research Center, Palo Alto, CA, USA, June 1993.
43. M. J. Rochkind, 'The source code control system', *IEEE Trans. on Software Engineering*, **SE-1**(4), 364–370 (1975).
44. W. F. Tichy, 'RCS—a system for version control', *Software—Practice and Experience*, **15**(7), 637–654 (1985).
45. W. Tichy, 'Smart recompilation', *ACM Trans. on Programming Languages and Systems*, **8**(3), 273–291 (1986).
46. R. W. Schwanke and G. E. Kaiser, 'Smarter recompilation', *ACM Trans. on Programming Languages and Systems*, **10**(4), 627–632 (1988).
47. R. Conradi, Personal communication, May 1996.
48. S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, Hemel Hemstead, UK, 1987.
49. C. B. Hanna and R. Levin, 'The Vesta language for configuration management', *Technical Report 107*, DEC Systems Research Center, Palo Alto, CA, USA, June 1993.
50. 'THINK Pascal™ User Manual, Version 4.0.1', Symantec Corporation, 1990.
51. 'Symantec Café for Windows 95/NT, 1.0', Symantec Corporation, 1996.
52. 'Borland C++, User's Guide, Version 4.5', Borland International Inc., 1994.
53. 'CodeWarrior® User's Guide', Metrowerks, Inc., Austin, TX, USA, Dec. 1995.
54. 'Rational Apex Ada', User's guide, Release 2.0, Rational Software Corporation, 1995.
55. 'Rational Apex C/C++', User's guide, Release 2.1, Rational Software Corporation, 1995.
56. G. Goos, W. A. Wulf, A. J. Evans and K. J. Butler, *DIANA—An Intermediate Language for Ada*, Lecture Notes in Computer Science 161, Springer-Verlag, Berlin, 1983.
57. P. H. Feiler, S. A. Dart and G. Downey, 'Evaluation of the rational environment', *Technical Report CMU/SEI-88-TR-15*, SEI, Carnegie Mellon University, Pittsburgh, PA, USA, 1988.
58. L. Cardelli, 'Typeful Programming', *Digital Systems Research Center Report 45*, Digital Eqipment Corp., Palo Alto, CA, USA, May 1989.
59. R. Burstall and B. Lampson, 'A kernel language for abstract data types and modules', *Proc. Int. Symposium on Semantic of Data Types*, Lecture Notes in Computer Science 173, Springer-Verlag, 1984, pp. 1–50.
60. R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1989.
61. R. Connor, G. Ghelli and P. Manghi, 'Modules and type abstraction in persistent systems', *Proc. Seventh Int. Workshop on Persistent Object Systems*, Cape May, NJ, 29–31 May 1996.
62. M. P. Jones, 'Using parameterized signatures to express modular structure', *Proc. 23rd ACM SIGACT–SIGPLAN Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, 21–24 January 1996.
63. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott and R. Morrison, 'An approach to persistent programming', *Comp. J.*, **26**(4), 360–365 (1983).
64. R. Morrison, A. L. Brown, R. Carrick, R. C. H. Connor, A. Dearle and M. P. Atkinson, 'The Napier type system', *Proc. Third Int. Workshop on Persistent Object Stores*, Newcastle, New South Wales, Australia, Springer-Verlag and British Computer Society, 10–13 January 1989, pp. 3–18.
65. R. Morrison, A. L. Brown, R. C. H. Connor, Q. I. Cutts, G. N. C. Kirby, A. Dearle, J. Rosenberg and D. Stemple, 'Protection in persistent object systems', in J Rosenberg and J. L. Keedy (eds), *Security and Persistence*, Springer-Verlag, 1990, pp. 46–66.
66. R. D. Tennent, *Principles of Programming Languages*, Prentice-Hall, 1981.
67. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.
68. M. P. Atkinson and R. Morrison, 'Types, bindings and parameters in a persistent environment', in M. P. Atkinson *et al*. (eds), *Data Types and Persistence, Proc. First Workshop on Persistent Object Systems*, Appin, Scotland, August 1985, Springer-Verlag, 1988, pp. 3–20.
69. R. Morrison, M. P. Atkinson, A. L. Brown and A. Dearle, 'On the classification of binding mechanisms', *Information Processing Letters*, **34**(1), 51–55 (1990).

70. A. Dearle, 'Environments: a flexible binding mechanism to support system evolution', *Proc. 22nd Int. Conf. on Systems Sciences*, Hawaii, USA, 1989, pp. 46–55.
71. M. P. Atkinson and R. Morrison, 'Polymorphic names and iterations', *Proc. Advances in Database Programming Languages, First Workshop on Database Programming Languages*, Roscoff, France, September 1987, Addison-Wesley and ACM Press, 1990, pp. 241–256.
72. A. Dearle, 'On the construction of persistent programming environments', *PhD Thesis*, University of St Andrews, 1988.
73. M. P. Atkinson and R. Morrison, 'Procedures as persistent data objects', *ACM Trans. on Programming Languages and Systems*, **7**(4), 539–559 (1985).
74. D. Stemple, R. B. Stanton, T. Sheard, P. C. Philbrow, R. Morrison, G. N. C. Kirby, L. Fegaras, R. L. Cooper, R. C. H. Connor, M. P. Atkinson and S. Alagic, 'Type-safe linguistic reflection: a generator technology', *Technical Report FIDE/92/49, ESPRIT Basic Research Action, Project Number 3070—FIDE*$_1$, 1992.
75. D. Stemple, R. Morrison, G. N. C. Kirby and R. C. H. Connor, 'Integrating reflection, strong typing and static checking', *Proc. 16th Australian Computer Science Conf.*, Brisbane, Australia, 1993, pp. 83–92.
76. R. C. H. Connor, 'Types and polymorphism in persistent programming systems', *PhD Thesis*, University of St Andrews, 1991.
77. Q. I. Cutts, 'Delivering the benefits of persistence to system construction and execution', *PhD Thesis*, University of St Andrews, 1993.
78. A. Dearle, Q. Cutts and R. Connor, 'Using persistence to support incremental system construction', *Microprocessors and Microsystems*, **17**(3), 161–171 (1993).
79. N. Wirth, *Programming in Modula-2*, Springer-Verlag, New York, 1983.
80. D. B. MacQueen, 'Modules for Standard ML', *Polymorphism*, **2**(2), (1985).
81. A. W. Appel and D. B. MacQueen, 'Separate compilation for Standard ML', *Proc. Int. Conf. on Programming Language Design and Implementation*, Orlando, FL, USA, 1994, pp. 13–23.
82. J. G. P. Barnes, *Programming in Ada plus Language Reference Manual*, Addison-Wesley, New York, 1991.
83. J. Gosling and H. McGilton, 'The Java language environment: a white paper', Sun Microsystems, Inc., CA, 1995.
84. A. L. Wolf, L. A. Clarke and J. C. Wileden, 'The AdaPIC tool set: supporting interface control and analysis throughout the software development process', *IEEE Trans. on Software Engineering*, **15**(3), 250–263 (1989).
85. D. I. K. Sjøberg, Q. Cutts, R. Welland and M. P. Atkinson, 'Analysing persistent language applications', *Proc. Sixth Int. Workshop on Persistent Object Systems*, Tarascon, France, 5–9 September 1994, Springer-Verlag and British Computer Society, pp. 235–255.
86. D. I. K. Sjøberg, M. P. Atkinson and R. Welland, 'Maintaining consistency using software constraints', *Preprint Report 1996 No. 1*, Department of Informatics, University of Oslo, February 1996.
87. D. N. Card, V. E. Church and W. W. Agresti, 'An empirical study of software design practices', *IEEE Trans. on Software Engineering*, **SE-12**(2), 264–270 (1986).
88. R. F. Kamel, 'Further experience with separate compilation at BNR', *Proc. IFIP WG2.4/ IFORS System Implementation and Languages: Experience and Assessment*, Canterbury, UK, September 1984.
89. D. I. K. Sjøberg, 'Thesaurus-based methodologies and tools for maintaining persistent application systems', *PhD Thesis*, University of Glasgow, 1993.
90. Q. I. Cutts, A. Dearle and G. N. C. Kirby, 'WIN programmers' Manual', *Research Report CS/90/17*, University of St Andrews, 1990.
91. D. I. K. Sjøberg, R. Welland, M. P. Atkinson, P. Philbrow, C. Waite and S. MacNeill, 'The persistent workshop—a programming environment for Napier88', *Nordic Journal of Computing*, **4** (in press) (1997).
92. D. I. K. Sjøberg, M. P. Atkinson and R. Welland, 'Thesaurus-Based Software Environments', *Proc Workshop on the Intersection Between Databases and Software Engineering*, Sorrento, Italy, 16–17 May 1994, IEEE Press, 1995, pp. 41–46.
93. G. N. C. Kirby and A. Dearle, 'An adaptive graphical browser for Napier88', University of St Andrews, 1990.
94. D. I. K. Sjøberg, 'Quantifying schema evolution', *Info. and Softw. Technology*, **35**(1), 35–44 (1993).
95. F. Bancilhon, C. Delobel and P. Kanellakis, *Building an Object-Oriented Database System: The Story of O*$_2$, Morgan-Kaufmann, San Mateo, CA, 1992.
96. C. A. Waite, R. C. Welland, T. Printezis, A. Pirmohamed, P. C. Philbrow, G. Montgomery, M. Mira da Silva, S. D. Macneill, D. O. Lavery, C. Herzig, A. Froggatt, R. L. Cooper and M. P. Atkinson,

'Glasgow Libraries for orthogonally persistent systems—principles, organisation and contents', *Technical report FIDE/95/132, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$,* University of Glasgow, 1995.

97. M. P. Atkinson, M. Jordan, L. Daynès and S. Spence, 'Design issues for persistent Java: a type-safe, object-oriented, orthogonally persistent system', *Proc. Seventh Int. Workshop on Persistent Object Systems,* Cape May, NJ, 29–31 May, 1996.