TECHNICAL REPORT

from the Department of

Computer Science

at

Keele University

title: **Building Software by Deduction: Why and How**

authors: Paul Singleton PhD MBCS CEng
Pearl Brereton PhD MBCS CEng

ref: TR92-17

ISSN: 1353-7776

date: December 1992

post: Dept. of Computer Science, Keele University,
Keele, Newcastle, Staffs ST5 5BG, U. K.

email: paul@cs.keele.ac.uk
pearl@cs.keele.ac.uk

telephone: +44 (0)782 583477 (Paul) or 583079 (Pearl)

fax: +44 (0)782 713082

Abstract

This paper describe the principles, design and implementation of ***DERIVE***, a deductive database for software configurations, in which primitive items (e.g. source files) are stored as *facts*, configurations are described by *rules*, compilers (and other derivation tools) are regarded as pure *functions* (over byte strings), and system building is realised by *query evaluation* meta-programmed in Prolog.

Configuration items can have unlimited independent variance, which is inherited by derived items, and which interacts combinatorially. New degrees of variance can be added to an existing configuration.

DERIVE makes extensive use of *memoisation* to achieve minimal recompilation; it employs *partial evaluation* to produce partly built product configurations which are described by mechanically generated make files or build scripts; and it employs *abstract interpretation* to answer queries about product feasibility. It demands and exploits referential transparency in all configuration items, infers all derivation dependencies, and perceives and exploits all potential concurrency in the execution of tools in the course of a build. Comparisons are drawn with *make* and its derivatives.

Contents

Abstract ii
Contents iii
1 Introduction 1
1.1 What do we mean by "building"? 1
1.2 Why is building non-trivial? 2
1.3 Building by Deduction? 2
2 Enabling techniques 3
2.1 File bodies as pure values 3
2.2 Derivation tools as functions 4
3 Philosophy and rationale of ***DERIVE*** 5
4 ***DERIVE***'s architecture 6
4.1 ***DERIVE***'s model of configuration items 6
5 Advantages of building by deduction 7
5.1 Proof strategies 7
5.2 Abstract evaluation (AE) 8
5.3 Partial evaluation (PE) 8
5.4 Memoisation 9
6 ***DERIVE*** versus make 9
6.1 Analysis and criticism of 'make' 9
6.2 The make-oids and allied developments 10
6.3 ***DERIVE***'s advantages over make 12
7 Summary of ***DERIVE*** 12
8 Conclusions 15
9 References 15

Building Software by Deduction: Why and How

Paul Singleton (paul@cs.keele.ac.uk)
O. Pearl Brereton (pearl@cs.keele.ac.uk)

1 Introduction

The research summarised here had the objective of exploring the application of deductive meta-programming to the building of software products from their components, performing *version selection* and *minimal recompilation* in a well-founded framework with established theoretical properties, and demonstrating the applicability and usefulness of several logic programming techniques, especially *memoisation*, *abstract interpretation* and *partial evaluation*.

1.1 What do we mean by "building"?

By "building" we mean *system integration*: the construction of a product release (whether for testing or for delivery) from a complex collection of components (e.g. fragments of source

code, executable modules, documentation) using existing derivation tools (e.g. compilers, linkers, text formatters).

Our proposal places no restrictions upon the updating of the database, except that it must be *declarative* (i.e. changes must be made by adding new versions of components rather than by destructively updating existing components). It is intended to be compatible with any change control policy.

We describe a prototype system called ***DERIVE***, a deductive database in which version selection and minimal recompilation are performed in the course of query evaluation.

In addition to concrete builds, ***DERIVE*** supports the following:

- *abstract builds*: e.g. a "dry run" build of a product, without performing compilations, but perhaps performing version selection, and showing the feasibility and general structure of the build.
- *hypothetical builds*: like abstract builds, but if a needed component (or tool) is missing, then it can be hypothesised (in no great detail) to allow the build to proceed. This answers questions such as: What tools and header files would we need to build *GIZMO 3.1.1* for *VMX 2.2a*?

- *generic builds*: using partial evaluation we can partly build a target, deliberately leaving some version selection still to be done, and some tools yet to be invoked. This is the general case for commercial product delivery: final parameterisation and linking of a product is generally done within the customer's system.

1.2 Why is building non-trivial?

A single configuration item may vary in many independent dimensions, e.g.

- different *bugs* (such variants are typically called "revisions");
- different *specifications*;
- different *performance* (e.g. different time/space trade-offs);
- compatibility with different *operating system* environments (e.g. UNIX, MVS);
- compatibility with different *user interface* systems (e.g. X, Windows 3);
- compatibility with different *communications protocols* (e.g. TCP/IP, X.25);
- employing different *natural languages* in their user interfaces;
- being either a fully coded procedure, or just a "stub" for testing.

Variance of components propagates through builds, giving rise to variance of products. Independent variance of components interacts combinatorially. Variance arises not only from versions of source files, but also from flags and options passed to tools, from versions of tools, and from versions of build rules. The independent dimensions in which source files, rules and tools vary cannot generally be anticipated: they will accrue during product development and evolution. Inadequate global control of version selection gives rise to version skew, whereby distinct versions of the same component are incorporated into a product via different dependency paths.

1.3 Building by Deduction?

How can we *deduce* a complex structured entity such as a deliverable software package? With propositional logic, we can only deduce the truth of formulae composed of simple atomic propositions, e.g.

query: `fuse_has_blown` ?

answer: *yes*

With predicate logic, however, we can instantiate variables (bind them to values) in the course of the search of a proof of a query:

query: `square( 3, X) ?` ⟵ *Does there exist some **X** such that the square of 3 is **X** ?*

answer: *yes* ⟵ the logical answer

*where **X** = 9* ⟵ a useful bonus

and this reply can, furthermore, be structured:

```
query:   prime_factors( 12345678, X) ?  <--  Does there exist
                                              some X such that
                                              the prime factors
                                              of 12345678 are X?

answer:  yes                            <--  the logical answer
         where X = [2,3,3,47,14593]     <--  a useful structured bonus
```

If we allow variables to range over large values such as source texts and executable programs, and if we implement compilers (etc.) as predicates, then we can pose queries to build deliverable software products, e.g.

```
query:   build( Gizmo, V6.2.28.1, Sun4, UK-English, X) ?

answer:  yes
         where X = [[..],     <--  complex structured value
                    [..,..],       denoting source and binary
                    [..,..]]       files, libraries, documentation ...
```

We shall show how this can be achieved, and discuss its advantages over established mechanisms.

2 Enabling techniques

There are two major technical prerequisites for building by deduction:

- file bodies, however large, must be treated as atomic *values*;
- derivation tools, however complex, must be treated as *functions*.

We consider these in turn.

2.1 File bodies as pure values

We need to manipulate the bodies of files as pure values, but we don't want to store them within the database (since they can be arbitrarily large), so we implement a strict naming scheme, using compact identifiers within the database and storing the file bodies within the host operating system's filestore.

Many-to-one naming schemes are easily implemented: it is necessary merely to allocate a novel identifier to each newly-created file body. But we need *one-to-one* naming, otherwise we may undermine the deduction, particularly if *negation* is employed. If a file body is created which happens to be identical to one which already exists, then it must be given the same name as the existing one. Thus the equality (or otherwise) of two file bodies is reflected in the equality (or otherwise) of their compact names.

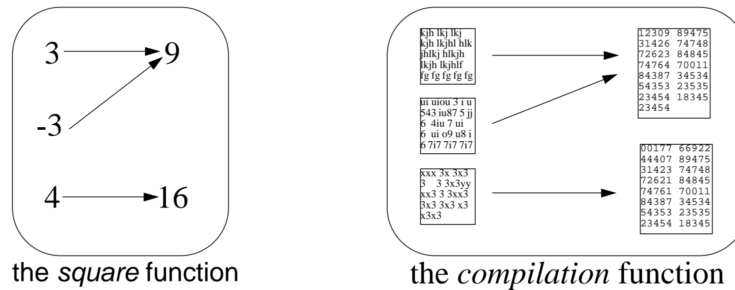
We call this scheme *bigtexts with equality* (hereafter just *bigtexts*), and implementation details are in *[Sing92a]* and *[Sing92b]*.

Bigtexts are <u>never</u> destructively updated: they begin life as uniquely-named empty scratch files, created as necessary to capture the outputs of derivation tools; then they are interned as bigtexts. Ultimately they may be deleted in the course of garbage collection.

The mapping between bigtext names and bigtext values may be seen as a dynamic symbol table, such as is used in many other interpreted language systems (e.g. Prolog and Lisp).

2.2 Derivation tools as functions

By “function” we refer to the mathematical concept of a (many-to-one) mapping from values to values, e.g.



the *square* function the *compilation* function

We regard compilation as a function, not as a process. In practice, a compiler may have several source file inputs and several options and flags, and is thus a function of many variables. (It may have several outputs, in which case it is strictly a *directed relation*, but we use the term “function” hereafter.)

Just as we have made file bodies behave like the most appropriate concept within the deductive programming model (i.e. atomic values), so we make compilers behave like the most appropriate concept: functions (implemented as predicates).

query: `compilation(` [file] `, X) ?` — *"Does there exist some **X** such that the compilation of* [file] *is **X** ?"*

answer: *yes*
*where **X** =* [file]

We require that the behaviour of each derivation tool is as follows:

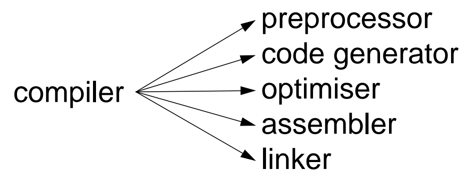
- “sound” - not yielding any results which we don’t need. Example of misbehaviour: a compiler may report the date on which it was built, or produce a running commentary on the progress of its activity. These are inappropriate and distracting, although not necessarily harmful.
- “complete” - yielding all the information which we need. Example of misbehaviour: a compiler may report failure merely by sending a textual message to some file or terminal, rather than by returning a formal error code to whichever process spawned it.
- “benign” - without side-effects. When invoked, a tool must behave as cleanly as an arithmetic operator. It must simply deliver a value (or values), and must not alter the state of its environment (e.g. by writing to public files).
- “deterministic” - without side-infects. The results of each tool invocation must be determined solely and explicitly by its arguments, and not implicitly and insidiously by its environment. Example of misbehaviour: a text processor may observe preferences defined within the user’s home directory.

The terms “sound” and “complete” are borrowed from database theory, where they have similar but more formal meanings.

In general, derivation tools will have been implemented independently, without regard to their possible use in declarative systems such as ***DERIVE***: indeed, they may have been written on the assumption that they will be invoked directly by human users, and they often have features which obstruct their use within a declarative build system.

We therefore attempt to clean up tool behaviour as follows:

- by unbundling tools, where feasible, into simpler components For example, a compiler may comprise a driving program for several underlying stages:



- by repackaging tools or their components within calling envelopes which suppress or correct their undesirable behaviour (examples are given in *[Sing92a]*).

We urge all designers of derivation tools to make provision for this mode of use, by allowing noise to be switched off, defaults to be overridden, failures to be denoted by the exit status, etc.

3 Philosophy and rationale of *DERIVE*

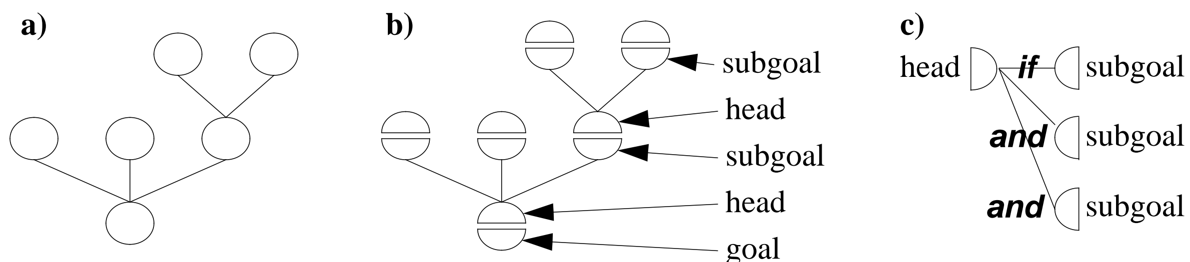
A software developer may find commercial advantage in having full control of variance in builds and releases: when sending software upgrades or bug fixes to customers, it will be advantageous to send only those items which are essential, and to be able to identify this set of items mechanically. When investigating bug reports, it may be necessary to reconstruct exactly the product version in which the bug was encountered. We seek rigorous control and traceability of variance within builds.

We seek the most abstract description of configurations, from which other representations (e.g. bills of parts, diagrams) can be derived mechanically (and hence cheaply and reliably).

A software configuration can be regarded as a program, whose operators are derivation tools, whose primitive values are source files, and whose parameters include the flags passed to compilers and the variables associated with version selection. Building a product can be seen as an evaluation of this program.

Software products under development are typically highly generic: many independent version selection decisions must be made to achieve a single concrete product. Products delivered to customers will, in general, still be generic, pending final specialisation and customisation to the target environment. Construction of a generic product corresponds to partial evaluation of the configuration, and exploring the feasibility or structure of a build corresponds to abstract evaluation.

Configurations can be described by trees or graphs **a)**, but these representations are not primi-



tive: they can be derived **b)** from *rules* **c)** by a proof procedure. These rules may be reused in many configurations.

For maximum abstractness, we thus seek a rule-based model for configurations, and for partial and abstract evaluability we seek a purely declarative model. The deductive programming model satisfies these requirements admirably.

4 *DERIVE*'s architecture

DERIVE is essentially a deductive database, implemented on top of three major components:

- a general-purpose networked operating system and filestore (UNIX + NFS);
- a networked relational database (Oracle + SQL*Net);
- a (relational) host language (Prolog).

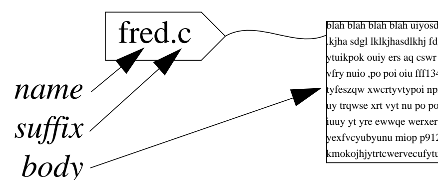
Oracle provides a large, shareable, persistent store for configuration rules and items (other than bigtexts); UNIX provides a large shareable persistent store for bigtexts and an environment in which to execute derivation tools; Prolog supports evaluation of build queries by interpretation of build rules written in ***DECL*** (***DERIVE***'s Evaluable Configuration Language). In addition, we employ 'C' for interfacing these components.

Each ***DERIVE*** user interacts with a private stateless front-end process (implemented in Prolog and C), which in turn accesses the shared network database.

4.1 *DERIVE*'s model of configuration items

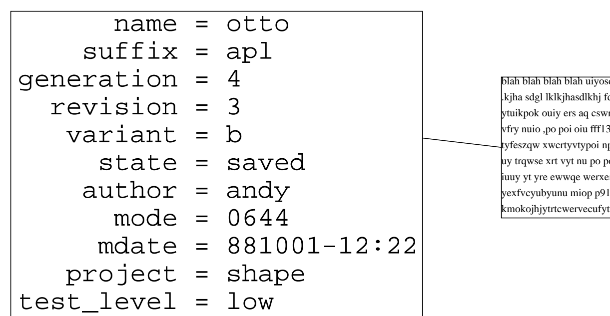
DERIVE's item model is simple and general: it is a finite map from names to values (possibly to bigtext values). We compare it with two other models of configuration items.

A UNIX file (below) has an updateable file body and a fixed set of attributes (name, suffix,



owner, create date, etc.).

An AFS object (below) refines and generalises this to a non-updateable body associated with



an extensible set of user-defined attributes *[Lamp88]*.

A ***DERIVE*** item is an extensible set of user-defined attributes, some of which may have bigtext values. This model is simpler and more expressive that either the UNIX or AFS model. Within

```
  name = fred
suffix = exe
  body = blah blah blah blah uiyosd
         .kjha sdgl lklkjhasdlkhj fdg
         ytuikpok ouiy ers aq cswr
         vfry nuio ,po poi oiu
         tyfeszqw xwcrtyvtypoi npiu
         uy trqwse xrt vyt nu po pop-
         iuuy yt yre ewwqe werxer
         yexfvcyubyunu miop p
         kmokojhjytrtcwervecufytug
 state = saved
author = derek
```

our deductive model, the *most general unifier* of two items is the union of the two maps which represent them. If, for some attribute, the items have incompatible values, then they do not unify.

5 Advantages of building by deduction

We have taken considerable care to integrate the peculiarities of software building (i.e. the awkward size of the data items which it manipulates, and the unclean behaviour of the tools with which it manipulates them) into a purely deductive framework, and this is repaid in our freedom to exploit powerful and general techniques from deductive programming:

- various proof strategies;
- abstract evaluation;
- partial evaluation;
- memoisation.

These techniques are application independent: they do not need to be adapted to our application, because we have already adapted the application to the deductive paradigm. Much of the accumulated expertise and understanding of deductive programming can immediately be reused.

5.1 Proof strategies

First-order deduction has well-defined semantics, and the various proof strategies (no matter how complex or subtle) should all give the same results for the same query, and should differ only in their consumption of computational time and space resources.

We currently employ only two strategies:

- simple depth-first backward chaining (to which we have added cycle detection *[Brou84]* and subgoal memoisation);
- breadth-first forward chaining.

The backward chaining strategy is appropriate for building well-specified targets (e.g. for concrete builds); the forward chaining strategy is more efficient for answering vague queries such as “What can we build”, or for *opportunistic manufacture [Kais87]*.

Proof techniques are the subject of current research, and new results in e.g. *magic templates [Rama92]* should be readily applicable within ***DERIVE***.

5.2 Abstract evaluation (*AE*)

This concept is not peculiar to deductive programming, and we illustrate it by examples from the imperative 'C' programming language.

In general, a 'C' routine cannot be executed unless all input parameters are bound to specific values, i.e. it is only amenable to *concrete* evaluation.

There are, nevertheless, three respects in which 'C' routines are commonly subject to *AE*:

- type checking, as in:

  `WARNING: type conflict of operands`

- reachability checking, as in:

  `WARNING: statement not reached`

- initialisation checking, as in:

  `WARNING: variable used before set`

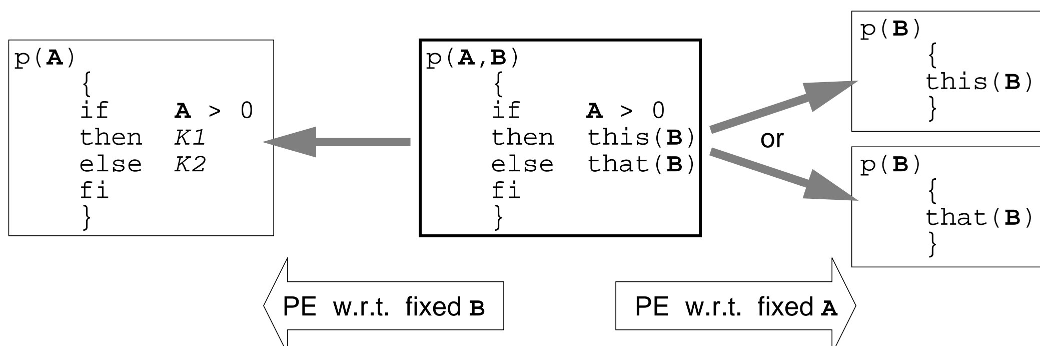
In each case, the 'C' routines are "executed" abstractly, not with specific values for parameters and variables, but with abstract values such as *integer* or *uninitialised*.

Abstract evaluation of ***DECL*** allows the feasibility or structure of a build to be explored, without incurring the cost of a concrete build.

5.3 Partial evaluation (*PE*)

PE, like *AE*, generalises concrete evaluation, and again we illustrate it with reference to an imperative language.

Suppose a routine `p(A,B)` is called with some input parameters bound to specific values, but with other input parameters still unbound. Can we sensibly evaluate it?



Yes: if **B** is known then we can evaluate the routine (above centre) to yield a simpler routine (above left) which is parameterised only by **A**; and if **A** is known then we can evaluate it to one or other of two routines (above right) which are parameterised only in **B**.

Unfortunately for those who use imperative languages, *PE* is obstructed by destructive assignment and by referential opacity. For instance, we can only replace the expression `this(B)`

by a constant *K1* if `this(B)` is referentially transparent and free from side-effects. Declarative languages, however, are quite amenable to *PE*, and ***DERIVE*** exploits it extensively.

Partial evaluation of ***DECL*** permits the construction of generic product releases.

5.4 Memoisation

Wherever referential transparency prevails, a particular function call or expression should always yield the same result, and it is possible to record (*memoise*) and reuse the result of any particular evaluation *[Kell86]*. But if a procedure has side effects, or accesses global variables, then it is infeasible to capture and reuse the impact (destructive or otherwise) of any particular call.

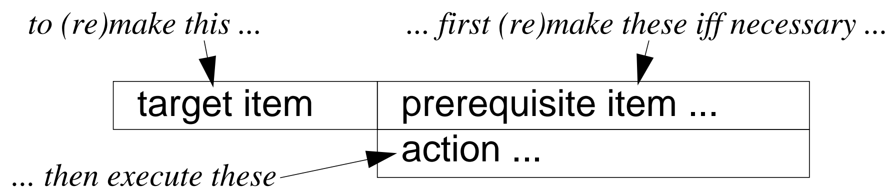
Declarative languages, whether deductive, functional or relational, are amenable to memoisation, and ***DERIVE*** employs it to achieve minimal recompilation. Indeed, ***DERIVE***'s minimal recompilation is more minimal than the timestamp-based scheme of make *[Sing92a]*.

6 *DERIVE* versus *make*

Stuart Feldman's widely-used UNIX utility make *[Feld79]* was the starting point of this research, as it was for many other developments, several of which are described below.

6.1 Analysis and criticism of 'make'

make describes a configuration by a set of rules in a make file, where a rule has this structure:



Essentially, make is a rule-based minimal recompilation utility for configuration items (whether primitive or derived) which are kept as files within a conventional (UNIX, VMS, MS-DOS or similar) filestore. It attempts to rebuild targets with minimal compilation activity, after primitive files have been updated in place.

Remarkably, it maintains no record of the status or derivation history of those items which it purports to manage: it attempts to infer *need-to-recompile* by inspecting the times-of-last-modification (or of creation) of whichever primitive and derivable files it can find.

It is a pragmatic and cleverly-judged engineering compromise which has been widely adopted by those software developers whose basic tools are a text editor and a compiler, thanks to its remarkably low entry threshold (simply creating and using a two-line make file can bring immediate benefit to anyone working in a compile-test-edit cycle).

Its shortcomings have been treated at length *[Mill86]*, although it almost suffices to say that they stem from

- lack of *referential transparency* in the configuration items (the primitive and derived files);

- flaws in its timestamp-based deduction;
- the non-declarativity of the actions which it executes.

In particular, it has these faults:

- it employs *backward chaining*, but implements only two special cases, one of which is severely deficient (it cannot chain its generic, or “suffix”, rules);
- it doesn’t understand the effects of the actions which it executes;
- it doesn’t record the options passed to (or assumed by) the tools which it invokes;
- it is unsafe when multiple make processes run concurrently;
- it spawns tools strictly serially, failing to exploit any potential for concurrency;
- it doesn’t provide formal traceability from products back to their components;
- it doesn’t facilitate, or even accommodate, version selection;
- it doesn’t propagate build parameters through rules;
- it recompiles unnecessarily in certain common circumstances, e.g. whenever an intermediate file is reconstructed but retains its previous value.

In short, it does only what was claimed for it.

6.2 The *make-oids* and allied developments

The shortcomings of make have been addressed (mostly in make’s original spirit of pragmatism) in several ways:

- *make-oids*: many have re-implemented it in the hope of providing a popular successor; Fowler *[Fowl90]* seems to have succeeded, after a false start *[Fowl85]*;
- *bolt-ons*: these typically address one of two main issues:
    - version selection;
    - automatic dependency extraction;
- *generalisations*, especially to support concurrent building (typically, this is advantageous only in local-area networks, or on multiprocessor machines *[Sequ87]*).

The “Rounder Wheel” *make-oid [Hirg83]* adds iterative constructs for applying rules to sets of files, and crudely solves the unsafeness of concurrent make invocations, not by locking the affected files, but by preventing other builds from starting.

build *[Eric84]* uses a modified file naming scheme to overlay filestore subtrees for related configurations and thereby factor out common files.

shape *[Mahl88]*, is built over an attributed file system *[Lamp88]* and attempts to provide uniform treatment of several forms of variance. Its designers pessimistically assume that a compilation which was performed by one machine within a network will not be valid for reuse on another machine, due to differences in the “compile environment”: this suggests inadequate control of “side-infects”.

Further *make-oids* are embedded in various experimental SCM systems, including Adele *[Estu84]* and SAGA *[Camp87]*.

The main *version selection bolt-ons* are SCCS *[Roch75]* and RCS *[Tich85]*, both of which are archive-based version managers, essentially supporting checkin/checkout of items, and both implementing textual compression of revisions (SCCS uses interleaved deltas and RCS uses reverse deltas). Each has also been bolted more intimately onto make, yielding *make-oids* which attempt to extract from the archive any item which they can't find in the filestore *[Tich85]*.

The *dependency extraction bolt-ons* include Walden's scheme *[Wald84]* to track down the insidious dependencies caused by use of the (referentially opaque) `#include` construct in *cpp*, and a scheme to extract module interdependencies from modular programs *[Olss89]*. Both of these schemes employ utility programs to scan primitive files and generate make rules; as the authors observe, you have to remember to run these programs first. It would be better if the dependencies were visible to make.

Most *concurrent make-oids* (i.e sequential *make-oids* which spawn several compilations concurrently) rely on explicit user-defined annotation of the make rules to indicate potential safe concurrency: these include DYNIX make *[Sequ87]*, pmake *[Baal88]* and nmake *[Fowl90]*, and it would be better if:

- compilations were not invoked within a public namespace (they may unwantedly overwrite each other's intermediate and output files);
- each *make-oid* sufficiently understood the actions and their interdependencies so as to be able to schedule them safely and optimally (rather than relying on the makefile author to indicate potential concurrency).

Another pmake *[Doug91]* provides not only build concurrency but also transparent process migration (within Sprite, a prototype UNIX-like network operating system). Despite reported process eviction times of "a few seconds" (on SPARC machines), the dynamic load-balancing of builds around a network of intermittently-used personal workstations apparently yields useful speed-ups, and is reportedly taken for granted by the user community. Given the likely erratic utilisation of personal workstations in a local area network, there may be scope for similar process migration within UNIX-based networks, and this paper should provide food for thought to anyone with strategic responsibility for the development of any of the fifty-seven varieties of UNIX.

The inventor of make has reviewed some of the *make-oids* (sympathetically) in *[Feld88]*, and some ritual abuses of make (e.g. for installing software as well as merely building it) have been catalogued (tediously) in *[Sing86]*.

Kielmann *[Kiel92]* has described a simple Prolog-based *make-oid* which exploits Prolog's pattern-matching facilities to extend the expressiveness of make rules: ***DERIVE*** exploits them rather more to extend it much further.

The most radical *make-oid* is Baalbergen's Amake *[Baal89]* which manipulates attributed objects stored in a specialised fileserver, and uses heuristic "hints" to aid minimal recompilation.

There are undoubted technical challenges in trying to fix make's deficiencies without abandoning its framework and infrastructure of

- timestamped destructive file update in public namespaces;
- referential opacity and hidden dependencies in configuration items;
- imperative actions with obscure side-effects.

For us, however, there was a rather different challenge: to find a computational framework in which these problems simply do not arise. It is disappointing that none of the *make-oid* authors have observed the similarity between make's building algorithm and formal deduction, and ***DERIVE*** rectifies this omission with a vengeance.

6.3 *DERIVE*'s advantages over *make*

We can summarise the advantages of ***DERIVE*** w.r.t. make as follows:

- it handles cyclic dependencies better;
- several builds may proceed concurrently, without any risk of adverse interaction;
- dependencies are inferred automatically;
- generic rules are correctly chained;
- it performs version selection, ensuring global consistency (no version skew);
- it allows variants of rules, and of tools;
- its minimal recompilation is more minimal;
- it handles tool failure better;
- it can detect and exploit all potential tool concurrency;
- it accommodates tools which have several outputs;
- it can treat composite items (e.g. sets of related files) as single entities.

DERIVE has a well-developed body of theory behind it: the foregoing properties are not bolted-on "features", but arise naturally from the adopted deductive programming model.

7 Summary of *DERIVE*

We conclude that a build-oriented programming-in-the-large language should be not merely descriptive but evaluable, and that furthermore it should be amenable to partial evaluation, in order that parameterisation (or variance: they are much the same thing) can be fixed or left unbound as required. This suggests that the formalism should be declarative (since non-declarative constructs obstruct partial evaluation).

We also conclude that the language should be as abstract as possible, so as to minimise the costs of maintaining inherently redundant configuration documents such as "build scripts" and "bills of parts" by allowing them to be mechanically constructed.

We advocate a goal-based model, in which the user never needs to specify actions, but instead specifies goals, and leaves the build system to determine (by searching its rules) what needs to be done (and to do it).

These two properties ("goal-based" and "declarative") suggest deductive computation: the practical need for concurrent usage (by tens or hundreds of users per system?) over a long period (tens of years?) suggests a database, hence deductive database.

The need to interpret a formal language suggests meta-programming, and we can implement meta-interpreters for deductive languages more concisely in Prolog than in any other general-purpose language (we need a general-purpose language for implementing the many functions

besides the central meta-interpretation of the configuration descriptions). Since the most mature commercial implementations of Prolog still only provide a clause base which is neither shareable, persistent nor scalable (i.e. arbitrarily "growable"), we have implemented an external virtual clause base by coupling Prolog to a RDBMS with some specialised interface mechanisms.

In order to integrate derivation tools into deductive software building, it is necessary to regard them as functions from byte strings to byte strings, not as processes which destructively update a filestore. We conclude that many user-friendly features of proprietary compilers are undesirable for formal, mechanical software building: for example, defaults undermine repeatability (of a build, given the same goal) and traceability (of components through to products). We suggest that tools should not attempt to provide piecemeal support for these sorts of SCM functions which can only be successfully implemented by an overall controlling system. At least, each derivation tool should be capable of being run in a quiet, repeatable, well-behaved mode.

Traditionally, Prolog-based deductive systems have manipulated structures of compact symbols: we have shown how arbitrarily large strings of bytes (such as program source text, or executable code) can be represented within a Prolog program by compact surrogates which display the most important property of those byte strings which they represent (i.e. equality). The actual values of particular byte strings are of interest only to derivation tools, to which they are passed by reference (i.e. filename) whenever a tool is executed.

The total cost of the compilations (etc.) involved in building a large software product can be very great, both in machine time (perhaps hours or days) and in delay to product development (e.g. when further development awaits the result of testing some modification to a source item). In a purely deductive system, the general principle of *memoisation* can be specialised and exploited to allow reuse of previous compilations, and we show how this can support minimal recompilation in a natural and sound way (and that this can be more efficient than schemes based on timestamps). The ability to use memoisation is a pay-off of declarativity.

While it has often been recognised that the (vague) notion of *attribute* subsumes many details of configuration items such as their names, types, revision levels etc., we have taken this generalisation further by proposing that a file's body is just another attribute, and we have described and exploited mechanisms for treating them as such. This allows a configuration item to have several bodies, or none, and it allows items to be retrieved associatively (i.e. by the value of their bodies). We claim that the extreme simplicity of this model of *item* (a homogenous map from keys to values) contributes to the simplicity both of the various query interpreters and of the memoisation procedures. We believe that this model is, in its combination of simplicity and expressiveness, superior to anything yet implemented and published.

Just as views in relational databases define virtual tables, so ***DERIVE***'s rules define virtual (or "derivable") items. The fact that considerable computation may be needed to realise these virtual items is hidden from the user. Sometimes, however, a user wants only an abstract summary of these virtual items, e.g. "What varieties of X could be built, on the assumption that all the compilations succeed?". We conclude that any build system will need to perform *abstract interpretation*, and it is fortunate that deductive systems are quite amenable to this.

We note that *module interconnection languages [Tich79] [Somm92]* are more concerned with expressing detailed module compatibilities that they are with defining the derivation dependencies of items (i.e. how and from what an item can be derived). In ***DERIVE***, module compatibilities must be encoded in shared attributes, which can be structured (as arbitrary Prolog terms), and thus there is scope for more sophisticated compatibilities. Alternatively, we can regard this compatibility checking as an abstract form of compilation, which operates not on

concrete source text but on abstract interface specifications (as are supported by several programming-in-the-small languages). Whichever approach to type-safe building is taken, the basic mechanisms of ***DERIVE*** appear to provide an appropriate framework.

We conclude that build systems which destructively update a shared filestore have many problems of their own making, and that a deductive approach avoids these problems. In particular, concurrent deductive builds cannot corrupt each other's workspaces: no locking mechanisms are required to prevent malignant interaction.

As a general philosophy, we claim that avoiding problems by adopting an appropriate, well-behaved computational model is preferable to solving them by adding ingenious fixes to badly-behaved machinery.

Some SCM research has recognised that the logical operations ***and*** and ***or*** are at the heart of version and configuration management *[Tich88]*. ***DERIVE*** takes this to its logical conclusion by employing predicate calculus as its computational framework.

We note that, in general, software products must be built with some parameterisation left unfixed, and that this is an example of partial evaluation. We conclude that Prolog-based deductive building provides exceptionally good support for partial evaluation, and that even concrete builds are best performed in several stages of partial evaluation (in order that potential tool concurrency can be identified and exploited). We hope that this perspective will be regarded, not as a wilfully sophisticated and unnecessary application of an arcane concept, but as an illuminating insight which shows the practical feasibility of (and offers a theoretical basis for) what is otherwise a challenging computational task.

Memoisation, like partial evaluation, is generally feasible only in declarative systems. Since ***DERIVE*** is purely declarative, we can employ memoisation at almost any point in the proof procedures. We have chosen to memoise tool goals (e.g. compilations), since these are both expensive and liable to recur. We also memoise retrievals from the RDBMS, but this would be less important if either the retrieval bandwidth was much better (i.e. closer to the typical rate of retrieval from flat files) or the hybrid Prolog+RDBMS system was replaced by a deductive database which lacked the former's retrieval bottleneck.

We observe that many large software products (e.g. the UNIX, Prolog and RDBMS systems which comprise ***DERIVE***'s infrastructure) are supplied as generic configurations of files whose further concretisation is controlled by make files, and that these make files are clearly maintained by human experts. Where a software product is targeted at many different hardware and operating system environments, there may be many versions of these make files, and their maintenance may be expensive (if done thoroughly) or unreliable (if not). Any build system (such as ***DERIVE***) which can generate these make files mechanically from more abstract (and more cheaply maintainable) configuration descriptions may offer significant cost savings.

Internally, ***DERIVE*** uses a single representation both for primitive configuration items (i.e. facts) and for rules: a fact is a rule whose body is `true`. This simplicity is unwanted from the point of view of the user, who intuitively regards facts and rules as distinct notions, and we therefore present them to her as separate categories. This simplicity is beneficial, however, in that rules can have variants by the same mechanisms that facts can have variants: few alternative build systems offer versioning of configuration descriptions with such fine granularity.

Although complex, the ***DERIVE*** prototype is well modularised: *bigtexts* are a tidy abstract data type, and the memoisation algorithms are implemented quite independently of the meta-interpreters to which they are applied, as is the caching of facts, rules and tool results, and the complex machinery which invokes derivation tools as if they were pure functions. All user-oriented

features such as defaults, visualisations etc. are implemented in a separate outer layer (the ***Deriver's WorkBench***).

8 Conclusions

DERIVE employs deductive meta-programming, which allows a *build* to be realised as pure query evaluation. This approach also supports the powerful but essentially simple techniques of abstract interpretation and partial evaluation, which both find natural applications here. Memoisation provides sound support for minimal recompilation, and when tools are treated as pure functions (and repackaged if necessary to make them behave so) it is possible to schedule the tasks in a build with maximum concurrency. Redundant (i.e. derived) items can be identified and destroyed automatically, easing the trade-off between storage space and rebuild costs. Many builds may proceed simultaneously without harmful interaction, and any build can be explained by reference to its proof, which is not merely a history of the build but a justification of it. Unlimited independent variance can be accommodated, and this general facility can represent a multitude of particular forms of variance. Generic products can be built and delivered (using partial evaluation to specialise the stored configuration), and provision can be made for continuing their building elsewhere.

Our overall conclusion, then, is that deductive meta-programming is an elegant, practicable and illuminating mechanism for software building, with considerable scope for accommodating further programming-in-the-large functionality.

9 References

[Baal88] E. H. Baalbergen, "Design and Implementation of Parallel make", *Journal of Computing Systems*, vol. 1, no. 2, pp. 135-158 (1988).

[Baal89] E. H. Baalbergen, K, Verstoep and A. S. Tanenbaum, "On the design of the Amoeba Configuration Manager", in *[SCM89]*, pp. 15-22 (1989).

[Brou84] D. R. Brough and A. Walker, "Some Practical Properties of Logic Programming Interpreters", *Proceedings of the International Conference on Fifth Generation Computer Systems* (FGCS), pp. 149-156 (1984).

[Camp87] R. H. Campbell and R. B. Terwilliger, "The SAGA Approach to Automated Configuration Management", *Lecture Notes in Computer Science*, vol. 244, pp. 142-155 (1987).

[Doug91] F. Douglis and J. Ousterhout, "Transparent process migration - design alternatives and the Sprite implementation", *Software - Practice & Experience*, vol. 21, no. 8, pp. 757-785 (1991).

[Eric84] V. B. Erickson and J. F. Fellegrin, "Build - A Software Construction Tool", *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 6, pp. 1049-1059 (1984).

[Estu84] J. Estublier, S. Ghoul and S. Krakowiak, "Preliminary Experience with a Configuration Control System for Modular Programs", *ACM SIGPLAN Notices*, vol. 19, no. 5 (May 1984).

[Feld79] S. I. Feldman, "make - a program for maintaining computer programs", *Software - Practice & Experience*, vol. 9, pp. 255-265 (1979).

[Feld88] S. I. Feldman, "Evolution of make", in *[SVCC88]*, pp. 413-416 (1988).

[Fowl85] G. S. Fowler, "The Fourth Generation make", *Proceedings of the 1985 Usenix Conference (Portland, Oregon, USA)* (1985).

[Fowl90] G. Fowler, "A Case for make", *Software - Practice & Experience*, vol. 20, no. S1, pp. S1/35-S1/46 (1990).

[Hirg83] E. Hirgelt, "Enhancing make or Re-inventing a Rounder Wheel", *Proceedings of the Usenix Conference* (Toronto), pp. 45-58 (1983).

[Kais87] G. E. Kaiser and P. H. Feiler, "An Architecture for Intelligent Assistance in Software Development", *Proceedings of the Ninth International Conference on Software Engineering (ICSE9)* (1987).

[Kell86] R. M. Keller and M. R. Sleep, "Applicative Caching", *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 1, pp. 88-108 (Jan 1986).

[Kiel92] T. Kielmann, "Using Prolog for Software System Maintenance", *Proceedings of the 1st. International Conference on the Practical Application of Prolog* (Vol. 1) (1992).

[Lamp88] A. Lampen and A. Mahler, "An Object Base for Attributed Software Objects", *Proceedings of the Fall '88 EUUG Conference* (Cascais), pp. 95-105 (1988).

[Mahl88] A. Mahler and A. Lampen, "An Integrated Toolset for Engineering Software Configurations", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, in SIGSOFT Software Engineering Notes, vol. 13, no. 5, pp, 191-200 (Nov 1988).

[Mill86] W. Miller and E. W. Myers, "Side-effects in Automatic File Updating", *Software - Practice & Experience*, vol. 16, no. 9, pp. 809-820 (1986).

[Olss89] R. A. Olsson and G. R. Whitehead, "A simple technique for automatic recompilation in modular programming languages", *Software - Practice & Experience*, vol. 19, no. 8, pp. 757-773 (1989).

[Rama92] R. Ramakrishnan, "Magic Templates: A Spellbinding Approach to Logic Programs", *Technical Report*, Computer Sciences Dept., University of Wisconsin-Madison (1992).

[Roch75] M. J. Rochkind, "The Source Code Control System", *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 364-370 (1975).

[Sequ87] "DYNIX Programmer's Manual", Sequent Corporation (1987).

[Sing86] P. Singleton, "Makefile Usage and Abusage - a Case Study", *Technical Report srg/ps/220*, Computer Science Dept., Keele University (1986).

[Sing92a] P. Singleton, "Applications of Meta-Programming to the Construction of Software Products from Generic Configurations", *Ph.D. Thesis*, Keele University (1992).

[Sing92b] P. Singleton, "Bigtexts with equality for deductive databases", *Technical Report*, Keele University (1992).

[Somm92] I. Sommerville and R. Thomson, "Configuration Specification using a System Structure Language", *Proceedings of the International Workshop on Configurable Distributed Systems*, Imperial College (London) (1992).

[Tich79] W. F. Tichy, "Software Development Control based on Module Interconnection", *Proceedings of the 4th International Conference on Software Engineering*, pp. 29-41, IEEE Munich (1979).

[Tich85] W. F. Tichy, "RCS - A System for Version Control", *Software - Practice & Experience*, vol. 15, no. 7, pp. 637-654 (1985).

[Tich88] W. F. Tichy, "Tools for Software Configuration Management", in *[Wink88]* (1988).

[Wald84] K. Walden, "Automatic Generation of make Dependencies", *Software - Practice & Experience*, vol. 14, no. 6, pp. 575-585 (1984).