Retrenchment and the B-Toolkit

Richard Banach and Simon Fraser

Department of Computer Science, University of Manchester,
Manchester M13 9PL, UK,
{banach,sfraser}@cs.man.ac.uk

Abstract. An experiment to incorporate retrenchment into the B-Toolkit is described. The syntax of a retrenchment construct is given, as is the proof obligation which gives retrenchment its semantics. The practical aspects of incorporating these into the existing B-Toolkit are then investigated. It transpires that the B-Toolkit's internal architecture is heavily committed to monolithic refinement, because of B-Method philosophy, and this restricts what can be done without a complete rebuild of the toolkit. Experience with case studies is outlined.

1 Introduction

The B-Method [2, 14, 19, 18, 17] has enjoyed what can only be called spectacular success in terms of vindicating the view that model based refinement, despite its theoretical depth, can, via the enabling effects of appropriate mechanisation, lead to highly significant benefits for the practical engineering of systems of the highest criticality. By now, B-engineered systems are widespread on the railways in France, and in other countries, where the French success has convinced the appropriate authorities [10, 9, 11].

It is well appreciated by practitioners of refinement, that for all its desirable properties, the technique displays a certain brittleness. The abstract and concrete models have to be in just the right relationship before the refinement proof obligations (POs) can be discharged. Unfortunately this state of affairs takes no account of the human-centred needs/requirements engineering that must contribute to system design, and depending on circumstances, can be a greater or lesser impediment to a transparent system construction process. In order to improve matters in this regard, retrenchment was introduced so that almost-but-not-quite-refinements could be described within a formal framework similar to that used for refinement [5, 6, 7, 4, 15, 16]. The ability to describe not-quite-refinements leads to the capacity to describe and analyse much more general system evolution scenarios [8, 3]. Needless to say this flexibility comes at a price; the guarantees offered by refinement are forfeit.

If refinement greatly benefits from mechanisation then so does retrenchment. The aim of this paper is to describe an experiment to incorporate retrenchment into the B-Toolkit [1], one of the two commercially available implementations of the B-Method, the other being Atelier-B. In fact retrenchment was first conceived

in the context of the B-Method [5], precisely so that the impact of the issues surrounding mechanisation could be taken on board right at the outset.[1]

The rest of this paper is as follows. Section 2 covers the theoretical aspects of the integration, namely: the syntax and the PO it describes, a small example, and what it means for a syntactically correct retrenchment construct to be type correct. Section 3 covers the B-Toolkit's architecture, and how it interacts with the theoretical aspects of incorporating retrenchment. Section 4 covers evaluation, and Section 5 concludes.

Acknowledgements and Note. The authors are indebted to BCore (UK) Ltd. for access to the source of the B-Toolkit. According to the terms under which the access was granted, the IPR residing in the experimental tool described in this paper remains the property of BCore (UK) Ltd.

2 Extending The B-Method for Retrenchment

The B-Method [2, 17] is a methodology in which abstract models can be described and then refined all the way down to code; all this in a manner that lends itself to extensive and integrated machine checkability at all stages. As noted above, retrenchment was introduced to enable the benefits of formal decription and machine checkability to migrate beyond the confines mapped out by strict refinement. The original retrenchment proposal [5] employed a syntax that combined the syntax of abstract machines with the flavour of refinement machines. And while it is adequate for most theoretical investigations into the system engineering aspects of retrenchment, it proves less convenient for implementation within an existing toolkit, since it necessitates extensive modification to the code for processing abstract machines. So for the present experiment, a different syntactic strategy was employed.

2.1 Syntax, the POs, and an Example

In the B-Method, refinement characterises the target as an extension of the source abstraction [2]. Retrenchment however, is a relationship between two abstract machines, and so it was appropriate to introduce a new RETRENCHMENT construct, which refers to the relevant machines, but which (conveniently enough) does not impact their syntax and processing. Table 1 describes the syntax. The RETRENCHMENT keyword introduces the construct, and the FROM and TO keywords indicate the source and target abstract machines respectively of the retrenchment. The RETRIEVES predicate gives the desired retrieve relation between the machines, and the OPERATIONS clause lists the ramifications of the operations common to source and target abstract machines.

Since [5], a number of different flavours of retrenchment have been investigated, including the original or 'primitive' form, the 'sharp' form [6], and the

[1] The choice of the B-Toolkit was dictated principally by familiarity from its use in teaching the B-Method at Manchester.

Table 1. Syntactic Categories for Retrenchment Relationship

Syntactic Category	Definition

Retrenchment Relationship	RETRENCHMENT
	    Identifier
	FROM
	    Identifier
	TO
	    Identifier
	RETRIEVES
	    Predicate
	OPERATIONS
	    Ramifications
	END

Ramifications	*Ramifications* ; *Ramification_Declaration* *Ramification_Declaration*

Ramification_ Declaration	RAMIFICATIONS
	    Identifier
	LVAR
	    Id_List
	WITHIN
	    Predicate
	CONCEDES
	    Predicate
	OUTPUT
	    Predicate
	NEVERTHELESS
	    Predicate
	END

Id_List	*Id_List* , *Identifier* *Identifier*

'output' form [4]. All of these can be viewed as special cases of a common 'sharp output' form, and so the ramifications of the RETRENCHMENT construct were designed to cater for this more general variant. Thus for a given operation, the RAMIFICATIONS clause consists of an LVAR clause (allowing the introduction of 'logical variables' for remembering before-values in the context of the after-state), the WITHIN clause, for constraining the antecedent of the operation PO,

and the CONCEDES, OUTPUT, and NEVERTHELESS clauses for use in the operation PO consequent. The operation PO itself is:

$$IC_f \wedge IC_t \wedge (Q_f \wedge R \wedge W) \Rightarrow Q_t \wedge [S_t]\neg[S_f]\neg(((R \wedge O) \vee D) \wedge E)$$

where IC_f, IC_t are source/target static contexts, Q_f, Q_t are source/target preconditions, S_f, S_t are source/target predicate transformers, and R, W, O, D, E are the retrieve, within, output, concedes, and nevertheless relations respectively.

Of the top level clauses, the RETRIEVES and OPERATIONS clauses are optional, whilst the RETRENCHMENT, FROM and TO clauses are mandatory. If an operation is ramified, then only the RAMIFICATIONS clause itself is mandatory, the remaining clauses are optional (although if an LVAR clause is present, then it is mandatory for a WITHIN clause to also be present that allows for the type checking of the variables declared).

Of course the operation PO is complemented by the intialisation PO:

$$IC_f \wedge IC_t \Rightarrow [I_t]\neg[I_f]\neg(R)$$

We give a small example of retrenchment in this syntax. The example shows how some of the clauses of the retrenchment may be omitted, and indirectly, how the more elaborate structures of the sharp or output forms can increase expressivity: the two conjuncts of the concession could justifiably be separated, putting the $TRUE$ case in an output clause.

MACHINE	abc	MACHINE	def
VARIABLES	aa, bb, cc	SEES	$Bool_TYPE$
INVARIANT	$aa \in \mathbb{N} \wedge$	CONSTANTS	$MaxNum$
	$bb \in \mathbb{N} \wedge$	PROPERTIES	$MaxNum \in \mathbb{N}$
	$cc \in \mathbb{N}$	VARIABLES	dd
INITIALISATION	$aa := 0 \;\|\|$	INVARIANT	$dd \in \mathbb{N}$
	$bb := 1 \;\|\|$	INITIALISATION	$dd := 0$
	$cc := 2$		
OPERATIONS		OPERATIONS	
$my_plus \mathrel{\widehat{=}} aa := bb + cc$		$resp \longleftarrow my_plus(ee, ff) \mathrel{\widehat{=}}$	
		PRE	$ee \in \mathbb{N} \wedge ff \in \mathbb{N} \wedge$
			$ee \leq MaxNum \wedge ff \leq MaxNum$
		THEN IF	$ee + ff \leq MaxNum$
			THEN $dd := ee + ff$
			$\|\| \; resp := TRUE$
			ELSE $dd := 0 \;\|\|\; resp := FALSE$
			END
		END	
END		END	

The abstract machines abc and def.

RETRENCHMENT	abc_to_def
FROM	abc
TO	def
OPERATIONS	
RAMIFICATIONS	my_plus
WITHIN	$bb = ee \wedge cc = ff$
CONCEDES	$(resp = TRUE \wedge dd = aa) \vee$
	$(resp = FALSE \wedge dd = 0)$
END	
END	

The retrenchment construct between the abstract machines abc and def.

2.2 Type Checking

The B-Method requires that, before a predicate involving set-theoretic variables be proved, it must be type-checked. Here we show how the retrenchment construct can be type-checked by extending the 'check' predicate of [2]; we use the same techniques as [2]. We assume that a retrenchment relationship as described above[2] holds between a source machine M_f and a target machine M_t (see Table 2), with operations op_f and op_t (see Table 3).

Table 2. Source and Target Abstract Machines

Source Machine		Target Machine	
MACHINE	$M_f(X_f, x_f)$	MACHINE	$M_t(X_t, x_t)$
CONSTRAINTS	C_f	CONSTRAINTS	C_t
SETS	$S_f; T_f = \{a_f, b_f\}$	SETS	$S_t; T_t = \{a_t, b_t\}$
CONSTANTS	c_f	CONSTANTS	c_t
PROPERTIES	P_f	PROPERTIES	P_t
VARIABLES	v_f	VARIABLES	v_t
INVARIANT	I_f	INVARIANT	I_t
ASSERTIONS	J_f	ASSERTIONS	J_t
INITIALISATION	U_f	INITIALISATION	U_t
OPERATIONS	o_f	OPERATIONS	o_t
END		END	

[2] The fields R, W, D, O and E refer to the RETRIEVES, WITHIN, CONCEDES, OUTPUT and NEVERTHELESS clauses respectively.

Table 3. Source and Target Operations

Source Machine	Target Machine
$u_f \longleftarrow op_f(w_f) \mathrel{\widehat{=}}$ PRE Q_f THEN V_f END	$u_t \longleftarrow op_t(w_t) \mathrel{\widehat{=}}$ PRE Q_t THEN V_t END

Table 4. Type Checking Rules for Retrenchment Constructs

Antecedents	Consequent
M_f,M_t,N,v_f,v_t,$rmDup(c_f, c_t)$, $rmDup(S_f, S_t)$,$rmDup(T_f, T_t)$,$rmDup(a_f, a_t)$, $rmDup(b_f, b_t)$,X_f,X_t,x_f,x_t are all distinct Operation names of o_f are identical to operation names of o, and are all included in the operation names of o_t given(X_f), given(X_t), given(S_f), given(S_t), given(T_f), given(T_t), $a_f \in T_f$, $a_t \in T_t$, $b_f \in T_f$, $b_t \in T_t$ $\vdash$ check($\forall x_f, x_t \bullet (C_f \wedge C_t \Rightarrow$ $\forall c_f, c_t \bullet (P_f \wedge P_t \Rightarrow \forall v_f, v_t \bullet (R \wedge o))))$	check (RETRENCHMENT N FROM M_f TO M_t RETRIEVES R OPERATIONS o END)

Table 4, presents the type checking rule for a retrenchment construct. The validity of three antecedents implies the validity of the 'check' predicate for the whole construct. The first antecedent asserts the distinctness of the various lexical elements listed. Note that *rmDup* removes duplicates *prior to* distinctness checking, therefore *permitting* limited sharing of identifiers (more details are given below). The second antecedent checks the inclusion of source operation names in target operation names. The third antecedent succeeds provided: given the set parameters, and abstract and declared sets of the source and target machines, assuming the numerical parameters and the machine constraints, and assuming the constants and their properties, then the retrieve relation typechecks.

Table 5 presents the type checking rules for the ramifications of operations. The first rule allows lists of ramifications to be checked elementwise. The second

Table 5. Type Checking Rules for Ramifications

Antecedents	Consequent
$ENV \vdash \text{check}(o)$ $ENV \vdash \text{check}(q)$	$ENV \vdash \text{check}(o\ ;\ q)$
$S_f \longleftarrow op(T_f)$ occurs in o_f $S_t \longleftarrow op(T_t)$ occurs in o_t l,u_f,u_t,op,w_f,w_t are all distinct l,u_f,u_t,op,w_f,$w_t \setminus ENV$ ENV, $u_f \in S_f, w_f \in T_f$ $u_t \in S_t, w_t \in T_t$ $\vdash$ $\text{check}(\forall\, l \bullet (W \Rightarrow D \wedge O \wedge E))$	$ENV \vdash$ check (RAMIFICATIONS op LVAR l WITHIN W CONCEDES D OUTPUT O NEVERTHELESS E END)

Table 6. Visibility of Abstract Machine Variables

	R	W	D	O	E
Machine Variables	√	√	√	√	√
Logical Variables		√	√	√	√
Operation Inputs		√	√	√	√
Operation Outputs			√	√	√

rule infers the validity of the 'check' predicate for a ramification on the basis of five antecedents. The first two check the presence of the operation *op* in the (previously typechecked) source and target machines, and extract their I/O types (S_f, S_t, T_f, T_t). The next two check that I/O variables and logical variables are distinct from each other and the environment. The final antecedent succeeds provided: given the environment, and the I/O variables in their types, assuming

the logical constants and the within relation, then the concedes, output, and neverthless relations all typecheck.

Note that the rules outlined above take no account of the SEES, USES or INCLUDES mechanisms. These work in the standard way and are not discussed further here.

2.3 Visibility

The syntactic validation of a retrenchment necessitates the enforcement of a visibility discipline. Table 6 shows which clauses of a retrenchment can access which variables.

3 The B-Toolkit and its Support for Retrenchment

The B-Toolkit is proprietary software of B-Core (UK) Ltd. Its architecture is shown in Fig. 1. The workhorse of the toolkit is the B-Platform (also known as the B-Tool or B-Kernel). This is a theorem proving assistant,[3] whose capabilities include various side effects such as the writing of files. Thus, although it maintains no state of its own, it can affect externally managed B-Toolkit state. To this end it is put to work for all sorts of B-Toolkit tasks such as parsing and typechecking ... which goes some way towards explaining the tool's sometimes eclectic responses to syntactic errors etc.

Maintaining a grip on the state of a B-Toolkit development is the job of the Construct Manager module. And acting as intermediary between the Construct Manager and the B-Platform is the Process Manager. So: users express their demands via the User Interface, these get digested by the Construct Manager, who translates them into a suitable series of commands for the B-Platform, which then get sent to it via the Process Manager. The B-Platform processes them one at a time, making appropriate reference to the B-Toolkit Libraries as necessary.

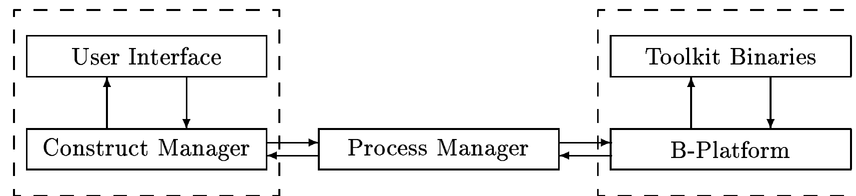

Fig. 1. Architecture of the B-Toolkit

[3] So it can perform inferences, but from user-supplied axioms and theories.

3.1 The Machine Development Structure

A B-Toolkit construct is either an abstract machine (AM), a refinement machine (RM) or an implementation machine (IM). A B-Toolkit machine development is a collection of such constructs that provide different views of a single model. The B-Toolkit considers the development of one (main) abstract machine to proceed linearly from the abstract to the concrete (Fig. 2).

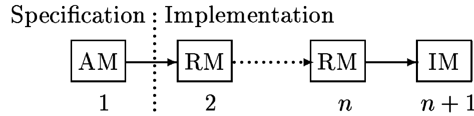

Fig. 2. B-Toolkit Development Structure

Incorporating retrenchment via a separate retrenchment construct (rather than a retrenchment machine), means that the B-Toolkit's mechanisms for refinement and implementation remain unaltered. However the B-Toolkit limits the use of abstract machines, allowing only one abstract machine per development, and restricting it to be only at the start. This raises some problems for the support of the retrenchment construct, as retrenchment fundamentally involves at least two abstract machines. (Typically, there is an 'idealised' abstract model, that undergoes one or more retrenchments until an abstraction refinable to code emerges.) A change to the structure of a machine development was thus required that allowed for at least the structure of machine development as just discussed. See Fig. 3, in which a retrenchment relation (RR) connects successive pairs of abstract machines until a machine refinable to an implementation is reached.

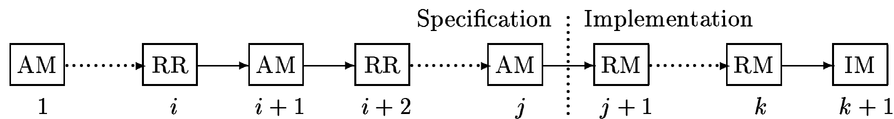

Fig. 3. Proposed Machine Development Structure

The B-Toolkit's limitations on the use of abstract machines turn out to be pervasive. The integration of the refinement relationship with the target machine is not just a syntactic convenience, but is maintained in all representations, internal and external. So the concept of a relationship distinct from (some flavour of) machine did not exist in the B-Toolkit, necessitating extensive redesign.[4]

[4] For this reason the possibility of allowing the retrenchments to form an arbitrary (loop-free) directed graph between abstract machines was not entertained.

3.2 Lifecycle of a Retrenchment Construct

Each construct under configuration control in the B-Toolkit has a state, recorded in the Construct Manager. This can be one of: uncommitted, unanalysed, analysed, unproved or proved. The state changes as the construct is moved through the construct lifecycle, and can be altered by changes occurring elsewhere in the development. This all applies equally to retrenchment constructs.

Introduction. The basic introduction of a retrenchment construct is a straightforward extension of the existing introduction mechanism; especially since the possibility to 'Introduce a retrenchment of Analysed Machines' was not pursued. The latter would have entailed a more extensive reworking of the introduction mechanism.

Committing and Dependency Analysis. The commit process basically has two phases. The first determines and resolves any dependencies on the construct, while the second verifies its syntactic correctness.

In the B-Method, a refinement machine is semantically an extension of the abstraction it refines. In the B-Toolkit therefore, when the abstraction is changed, any refinement of it can no longer be trusted, and it, and any further refinements are set to unanalysed and removed from the (B-Toolkit's internal view of the) state of the machine development.

By contrast, the target of a retrenchment is emphatically *not* an extension of its source machine. Both source and target machines need to be self contained consistent machines. Thus an alteration to the data of any retrenchment construct that connects them need affect neither the source or target machines themselves, nor the B-Toolkit's view of their state.

Fig. 4 illustrates a dependency chain starting with a series of retrenchments, and continuing with a series of refinements beyond machine $i+2$. Fig. 5 shows what happens to this when retrenchment $i+1$ is altered. This relies on the simple development structure implemented during this experiment, which allows only for zero or more retrenchments followed by zero or more refinements.

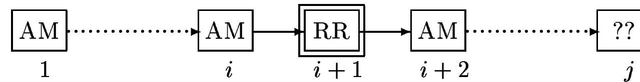

Fig. 4. A Retrenchment Construct in a Machine Development

We turn now to abstract machines. There are three distinct types of relationship that can cause dependency on an abstract machine – refinement, retrenchment and inclusion/importation. The refinement and retrenchment relationships are restricted to a single machine development, and are examined first. Fig. 6 shows the senarios we need to consider.

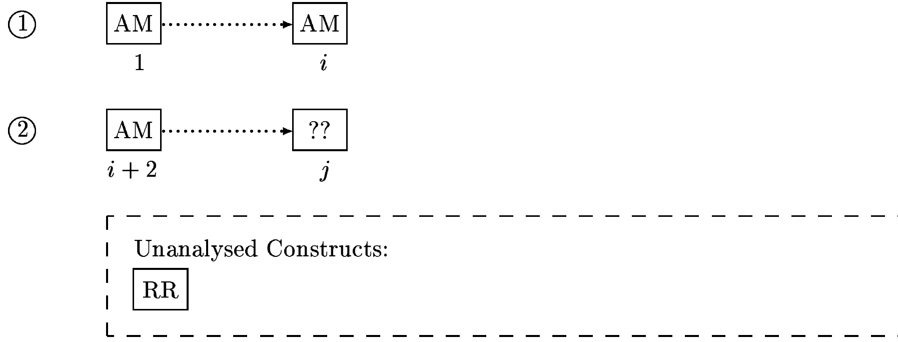

Fig. 5. Resolution of a Retrenchment Construct Commit

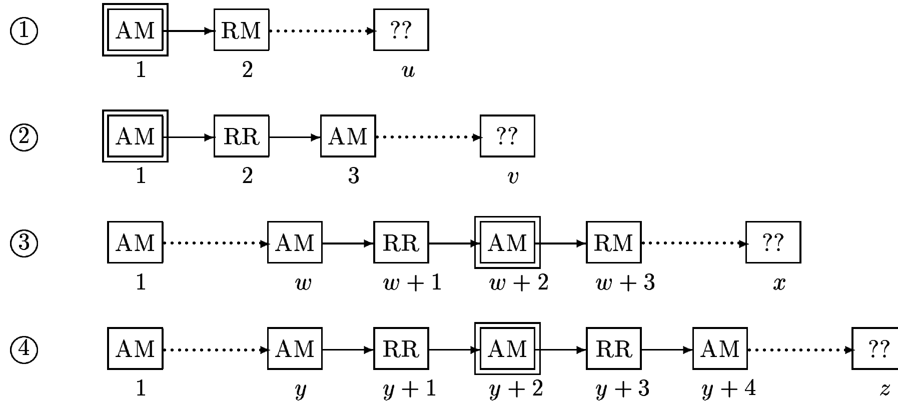

Fig. 6. Abstract Machines in Machine Developments

Development ① in Fig. 6 shows an abstract machine at the head of a (possibly empty) refinement chain. This is a standard B-Toolkit refinement picture, and needed no alteration in dependency analysis.

Development ② in Fig. 6 shows an abstract machine which is retrenched. Here the only construct dependent on the abstract machine is the retrenchment construct. When the machine is altered, it and the retrenchment construct become unanalysed, and the remainder of the chain forms a separate development. See Fig. 7

Development ③ in Fig. 6 shows an abstract machine which is the target of a retrenchment and the source of a refinement. It is clear that when the machine is altered, it, its parent retrenchment construct, and all its refinement descendants, must become unanalysed. Fig. 8 illustrates.

Finally, development ④ in Fig. 6 shows an abstract machine which is both the source and target of retrenchments. In this case, the development splits into

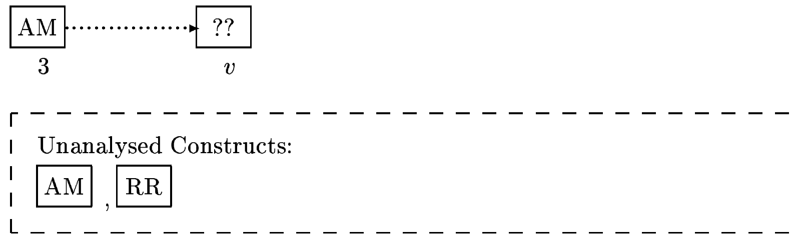

Fig. 7. Resolution of an Abstract Machine Commit ②

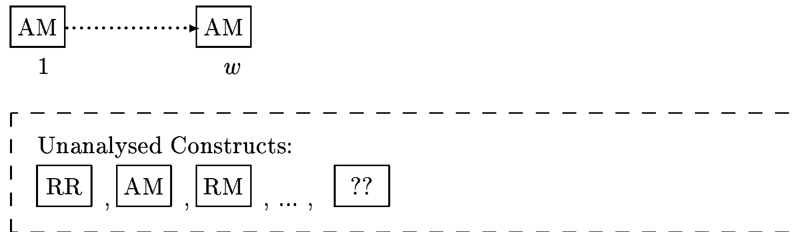

Fig. 8. Resolution of an Abstract Machine Commit ③

two: the initial part up to the most concrete ancestot of the machine in question forms one piece, and the other part consists of the most abstract descendant of the machine in question up to the end. Fig. 9 illustrates.

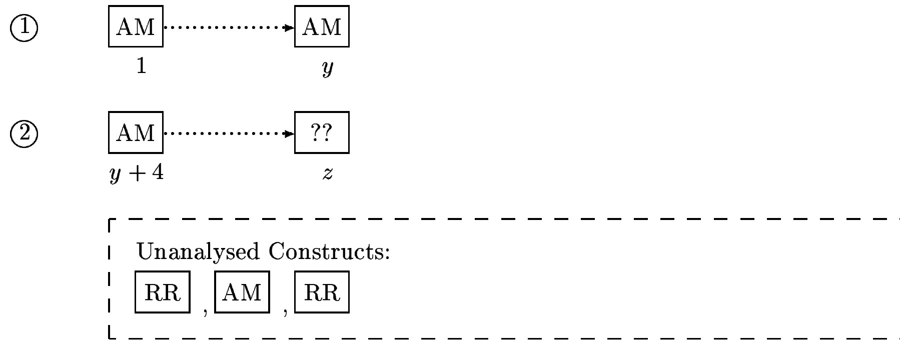

Fig. 9. Resolution of an Abstract Machine Commit ④

An abstract machine can also be included in another (via the AMN INCLUDES clause), and although it can only be included in *one* other machine, that machine

may itself be included in another . . . and so on indefinitely. Fig. 10 shows a typical scenario.

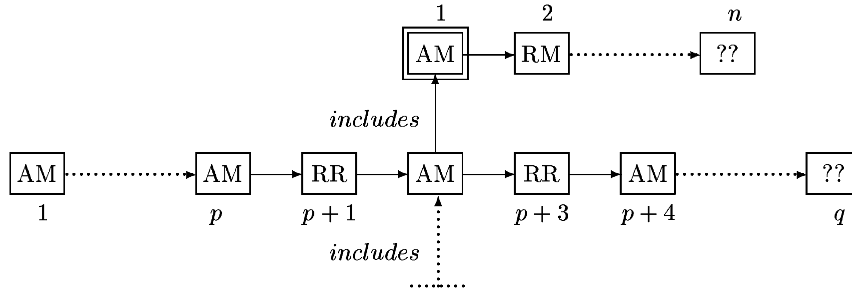

Fig. 10. Cross Machine Development Dependencies

The inclusion dependencies must be resolved before the refinement or retrenchment dependencies. The machine including the one at issue is located; then the one including that one, and so on until the end of the chain. All of these machines must become unanalysed. The last one has its refinement and retrenchment dependencies resolved according the rules above. Then its predecessor, and so on until the original machine is unanalysed. For example, committing a change in the indicated machine in Fig. 10 would result in the state shown in Fig. 11.

Importation dependencies are handled in a way similar to this, the only difference being that an abstract machine may be imported by many different implementation machines.

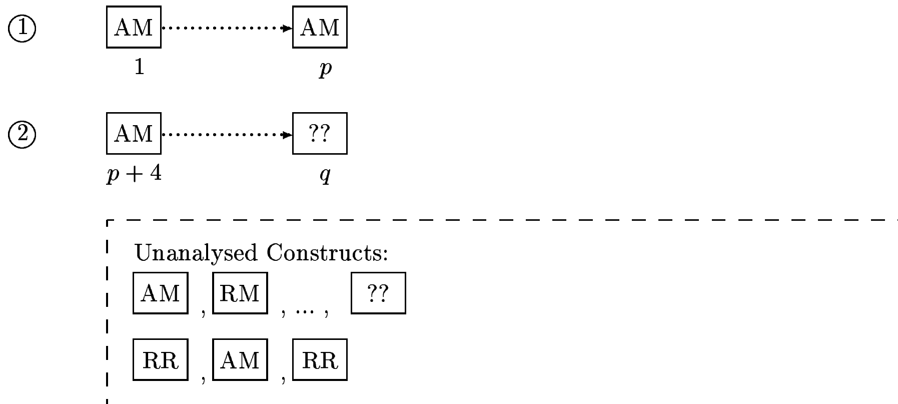

Fig. 11. Resolution of Inclusion Relationship

Once the dependencies of a putative commit of a construct are resolved, the new definition of the construct is parsed. If it fails to parse, the changes remain uncommitted and of course any constructs set to unanalysed during dependency resolution remain unanalysed.

Analysis. The analysis phase consists of three stages: Normalisation, Syntax Checking and Type Checking. The aim is to ensure that the user's definition conforms to the rules of the B-Method, and to produce an internal representation of it.

Normalisation begins with a parse check of the user's construct. It should be noted that this parse check is different to the one in the commit phase, and can uncover different errors. The commit phase parse check simply determines whether the user's definition can be parsed by the B-Platform, and confirms that all keywords and operators have been used correctly. The parse check of the analyse phase determines whether the user's definition satisfies the restrictions encoded in the toolkit binaries.

It was necessary therefore, to create rules for the parsing of a retrenchment construct in the toolkit binaries, and to ensure that retrenchment keywords were not used in other constructs (and vice versa). For example, declaring a FROM clause in an implementation machine would cause an error as would using a VARIABLES clause in a retrenchment. Since the construct manager relies on the file extension of a construct to determine its type, checks were introduced to ensure that a `.rmt` extension corresponded to a retrenchment construct.

It was also necessary to introduce an acyclicity check into the normalisation stage. In principle, two machines can be retrenchments of each other. And while it is theoretically desirable to permit this and other pathologies, the resulting breaking of the linear development structure would have required a drastic re-design of the B-Toolkit due to its extensive internal dependence on linearity, so it was excluded.

After normalisation, a construct progresses to what the B-Toolkit calls syntax checking. In this stage, the B-Toolkit checks that the contents of each clause conform to the expected syntax. For example, each identifier is checked to ensure that its length is between two and sixty characters. Checks are also performed to ensure that the rules governing clause-use have been followed. For example, it is forbidden to have a CONSTANTS clause without a PROPERTIES clause.

Little of the latter is needed for retrenchments. The only clause in which new variables can be declared is the LVAR clause, and the identifiers of these variables must be checked in the same way as any other new identifier. If an LVAR clause is used however, it must have an associated WITHIN clause (so that the variables declared can be given some before-values and types). All the clauses consisting of predicates can be assumed to be well-formed (as they would not otherwise have passed the commit phase parse), but it is still necessary to check these clauses to ensure that typing errors have not occurred.

Once the basic checks described above have been performed, the list of constants, sets and variables of each construct is examined for duplication. When checking a machine, the B-Toolkit will fail with any duplication in any of these

clauses. As a retrenchment construct inherits these lists from its source and target machines however, there is some scope for valid duplication. For example, both source and target abstract machine may see a common library machine which defines a constant used by both machines. Since seen constants are contained within the internal definition of the CONSTANTS clause, the retrenchment relationship will have two instances of this constant in its own list of constants. However, it is clear that this is not an error but a consequence of the difference in modelling machines and relationships. When examining the lists of sets and constants for a retrenchment construct therefore, the B-Toolkit will produce a warning when finding duplicate declarations. Any errors (where a constant has the same identifier, but different properties) will be caught in the type checking stage of the analysis. Duplications in the list of variables, however, will produce errors as it is necessary to be able to distinguish the variables of the source machine from those of the target.

For refinement and implementation machines, the B-Toolkit checks that the set of operation names matches that of their abstraction. For retrenchments, this is relaxed to an inclusion of source operation names in those of the target, requiring a slightly different check.

Having survived thus far, a construct passes to the type checking stage. For a retrenchment construct, the constants, sets, properties and variables clauses are derived from its source and target machines. The combined lists of sets and constants are each type checked against the combined properties clause, and it is at this stage that problems due to the duplication of constant or set identifiers can be uncovered. For example, if abstract machine *aa* defines a constant *const* with the property $const \in \mathbb{N}$, and abstract machine *bb* also defines a constant *const*, but with the property $const \in \mathbb{N}1$, then an attempt to relate the two abstract machines via a retrenchment will cause type checking errors in the retrenchment as the constant cannot have both type $\mathbb{N}$ and $\mathbb{N}1$. Clearly, this type checking cannot guarantee that duplicate constants or sets are valid when their types do not disagree. For whilst two constants may have the same type, it is possible that they can have different values. Within the B-Toolkit's architecture there is no simple way to check that this is not the case. It was decided that the warnings given in the syntactic check and during type checking were sufficient for the purposes of this experiment, and it is left to the user to spot any erroneous duplication when attempting to prove their retrenchment relationship correct. Although this is not ideal, the framework of the B-Toolkit was designed for use with refinement, and a drastic reworking would have been needed to provide the complete checking required in this instance.[5]

The type checking stage results in the production of a file, stored in the `TYP` sub-directory of a development, that stores the type information for the variables and operations of an analysed construct. When type checking a retrenchment, the B-Toolkit uses the type files of source and target machines to ensure that

[5] The arguably preferable option of generating proof obligations to settle such unresolved issues was not pursued.

all the variables used in the RETRIEVES clause have been defined (and typed) in the machines involved.

Likewise, for every operation, the ramifications are checked to ensure that the variables used, conform to the typing of that operation's inputs and outputs in source and target machines (any duplication of inputs and outputs between source and target machine will again cause errors). It is also necessary to check that the type of any logical variables declared in an LVAR clause can be derived from the associated WITHIN clause.

Once type checking is complete, the only task remaining is to add the analysed object to a machine development. For the existing constructs, this happens just as before. For retrenchment constructs, the only thing that needs to happen is to join up the development chains of the source and target machines. For example, Fig. 12 shows the state just before, and Fig. 13 shows the state just after, the moment when source machine AM_x and target machine AM_y get related via a retrenchment construct.

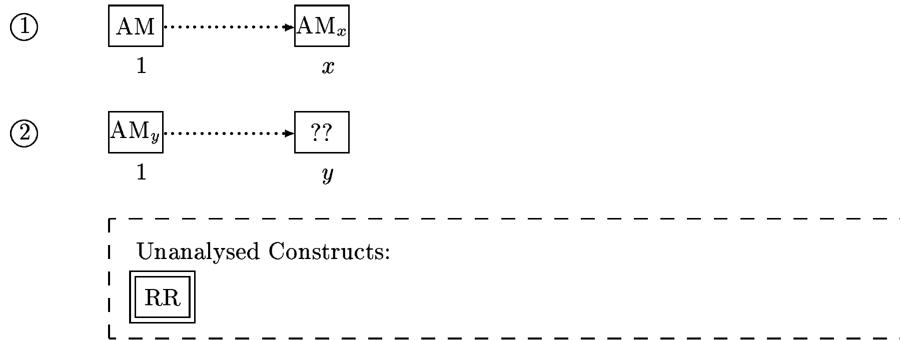

Fig. 12. Machine Developments Before Analysis of a Retrenchment Construct

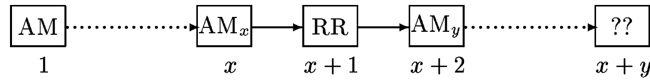

Fig. 13. Resolution of Retrenchment Construct Analysis

Generation of Proof Obligations. Once a construct has been analysed, proof obligations can be generated (since being in the analysed state is the only prerequisite for PO generation for any construct). The discharge of the POs will prove that the construct's definition is consistent. Of course, subsequent change to a construct discards any previously established proofs regarding it.

In generating the POs for a retrenchment from a source to a target, there are three sets of obligations to create. Two sets concern the internal consistency of the source and target machines themselves, and the third concerns the validity of the claimed retrenchment relationship between them. Each of these PO generation activities is tied to the requisite syntactic construct; this must be in the analysed state as noted previously. For the retrenchment relationship, obviously all three participating constructs must be analysed.

The first stage in PO generation for a retrenchment construct is the creation of the initialisation PO. This is a simple task, and the form of the obligations depends only upon the presence of a RETRIEVES clause in the retrenchment construct. The second stage involves the generation of a PO for each ramified operation. The precise form of these proof obligations depends on which of the available clauses (see Table. 1) have been used in the ramifications for the operation.

Once the proof obligations have been generated, the GSL definitions of the initialisation and operations are used to replace the jokers within these obligations. The B-Toolkit then applies its special and implicit tactics to reduce these obligations to a number of predicates. Typically the substitution tactic will be used to reduce the proof obligation to a predicate, and then the deduction and conjunction tactics are used to break the PO into smaller chunks. These putative lemmas are then written to a file which contains all the obligations that must be discharged to validate the associated construct.

4 Evaluation

The development of the extended B-Toolkit described above involved the usual levels of functional and unit testing, which revealed that the basic ingredients were working satisfactorily. More extensive testing came via two case studies, one small the other larger.

The small case study was the little example that we saw in Section 2, involving machines abc and def, and the retrenchment of the addition of unbounded numbers to the addition of finite numbers. The small size of this example meant that the extended toolkit dealt with all aspects of it unproblematically.

The larger case study was based on a case study focused on requirements engineering via retrenchment in the area of telecoms feature interaction [8]. Although, compared to the normal scale of things in real applications this was very much still a toy, as regards exercising the extended B-Toolkit it proved to be very much not a toy.

The case study centred on the operations of an atomic call model, with the inclusion or not of various additional features. Here is the most basic connect operation:

$$
\begin{array}{l}
calls \text{ -}(i, connect_n, o)\text{->}\ calls' \quad \texttt{iff} \\
\qquad free(n)\ \wedge \\
\qquad \texttt{if} \quad free(i)\ \wedge\ (n \neq i) \\
\qquad \texttt{then}\ \ o = OK\ \wedge\ calls' = calls \cup \{n \mapsto i\} \\
\qquad \texttt{else}\ \ o = NO\ \wedge\ calls' = calls
\end{array}
$$

As is clear, this was written in a transition system notation, and its size is hardly enormous by today's standards (enhanced versions of the $connect_n$ operation typically had an additional clause).

A typical concedes relation from one of the retrenchments in [8] is reproduced below:

$$\begin{aligned}
&C_{CF,connect_n}(u, v, o, p; i, j, u, v) = \\
&\qquad (busy(j) \ \wedge \ j \in dom(fortab) \ \wedge \\
&\qquad fortab^{+}(j) = z \ \wedge \ free(z) \ \wedge \ z \neq n \ \wedge \\
&\qquad u' = u \ \wedge \ v' = (calls \cup \{n \mapsto z\}, fortab) \ \wedge \\
&\qquad o = NO \ \wedge \ p = OK)
\end{aligned}$$

(This concession captures the difference in behaviour between the connect operation in a simple system and in an enhance system incorporating call forwarding, with the forwarding data held in the $fortab$ function.) Again the size is hardly excessive, and there are a number of slightly more complicated models and more complicated retrenchments in [8].

For processing by the extended B-Toolkit, the above were hand translated into B syntax. After translation, and using the resources of a typical desktop machine, proving even the simplest of these retrenchments correct, turned out to be all but beyond the capabilities of the system. Upon closer investigation, the reason revealed itself to be that the B-Toolkit's prover took a rather naive approach to proving statements making heavy use of disjunctions (as retrenchment proof obligations invariably do). With a little bespoke optimisation, the toolkit was eventually persuaded to discharge the POs for the simplest of the retrenchments in [8] involving the concession above. When the more complex cases in [8] were attempted, it became clear that available machine resources were decidedly insufficient and the fully mechanised route was not pursued. Visual inspection confirmed however, that although it was unable to prove them, the toolkit had generated appropriate proof obligations, and that these were in fact true statements. A more extensive treatment of this case study can be found in [12] and is supported by [13].

5 Conclusions

In the preceding sections we described the essential tasks addressed in incorporating retrenchment into the B-Toolkit. We gave the syntax of the retrenchment construct and the proof obligation that that syntax represented, and then described how the data was processed within the toolkit's architecture. The latter details revealed that various aspects of the B-Toolkit's internal design were very heavily tied to its original objective of monolithic refinement, this being a result of the underlying B-Method philosophy that a refinement machine is really a kind of extension of its abstraction, rather than an independent entity. The consequences of this were principally that the development structure was restricted to linear as regards retrenchment/refinement dependencies. Moreover the feasibility of proving nontrivial retrenchments correct on today's typical desktop

machines turned out to be heavily compromised by the relatively unsophisticated nature of some aspects of the B-Toolkit's prover.

Thus retrenchment was incorporated in a limited way, and it rapidly became clear that any attempt to extend this limited integration would yield very much diminishing returns. For this reason the implementation here described should be viewed principally as an experiment in the design of mechnical assistance for retrenchment, rather than an ideal solution. The experience gained quickly convinced us that addressing the full array of possibilities opened up by retrenchment would be much better served by a tool built from scratch. Such a tool is the objective of the second author's current doctoral work, for which the present experiment (described at greater length in [12]) provides valuable experience of course.

References

[1] J.R. Abrial. *The B-Tool Reference Manual, Version 1.1.* B-Core (UK) Ltd.

[2] J.R. Abrial. *The B Book.* Cambridge University Press, 1996.

[3] R. Banach and R. Cross. Safety requirements and fault trees using retrenchment. In Heisel, Liggesmeyer, and Wittmann, editors, *Proc. SAFECOMP-04*, volume 3219 of *Lecture Notes In Computer Science*, pages 210–223. Springer, 2004.

[4] R. Banach and C. Jeske. Output retrenchments, defaults, stronger compositions, feature engineering. *Submitted*, 2002.

[5] R. Banach and M. Poppleton. Retrenchment: An engineering variation on refinement. *Lecture Notes In Computer Science*, 1393:129–147, 1998.

[6] R. Banach and M. Poppleton. Sharp retrenchment, modulated refinement and simulation. *Formal Aspects of Computer Science*, 11(5):498–540, 1999.

[7] R. Banach and M. Poppleton. Engineering and theoretical underpinnings of retrenchment. *Submitted*, 2001.

[8] R. Banach and M. Poppleton. Retrenching partial requirements into system definitions: A simple feature interaction case study. *Req. Eng. Journal*, 8:266–288, 2002.

[9] P. Behm, P. Desforges, and J-M. Meynadier. Meteor: An industrial success in formal development. In Bert, editor, *Proc. B-98*, volume 1393 of *Lecture Notes In Computer Science*, page 26. Springer, 1998.

[10] P. Desforges. Using the b-method to design safety-critical software for railway systems. *Recherche et Developpements - Fatis Marquant 97*, 1998.

[11] D. Essame. Handling safety critical requirements in system engineering using the b formal method. In Heisel, Liggesmeyer, and Wittmann, editors, *Proc. SAFECOMP-04*, volume 3219 of *Lecture Notes In Computer Science*, page 115. Springer, 2004.

[12] S. Fraser. *Mechanised Support for Retrenchment in the B-Toolkit*, 2004. Master's thesis, School of Computer Science, University of Manchester.

[13] S. Fraser. *Specifications, Proof Obligations and Proofs Supporting a Case Study of Retrenchment in the B-Toolkit*, 2004. Available online at `http://www.cs.man.ac.uk/~frasers/casestudy`.

[14] Haughton H. Lano, K. *Specification in B: An Introduction Using the B-Toolkit.* Imperial College Press, 1996.

[15] M. Poppleton and R. Banach. Retrenchment: Extending the reach of refinement. In *Proc. ASE-99*, IEEE, pages 158–165, 1999.

[16] M. Poppleton and R. Banach. Controlling control systems: An application of evolving retrenchment. In Bert, Bowen, Henson, and Robinson, editors, *Proc. ZB-02*, volume 2272 of *Lecture Notes In Computer Science*, pages 42–61. Springer, 2002.

[17] S. Schneider. *The B-Method: An Introduction*. Palgrave, 2001.

[18] E Sekerinski and K. Sere, editors. *Program Development by Refinement*. Springer, 1999.

[19] J.B. Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.