An Ina Jo® Proof Manager for the Formal Development Method

by

Daniel M. Berry[1]

Computer Science Department
University of California
Los Angeles, CA 90024
U. S. A.

9 February 1985

Abstract

This paper describes methods for decomposing large conjectures into smaller ones in order to make their proof easier and for limiting the amount of reproving that occurs when a specification is modified. It proposes a tool, based on these methods, for managing the proofs of conjectures about an evolving specification.

1 Introduction

The Formal Development Method (FDM) is a method of arriving at a formal specification of a system which is verifiably guaranteed to meet requirements that are also stated in the specification. The method is intended to support iterative design of these specifications in which the designer is continually modifying parts of it both to add new features to the specification and/or to correct errors. Errors in the specification are detected often in response to a failed attempt to prove the specification *correct*, i.e., to prove that the part of the specification that states what is happening is consistent with the part that puts constraints and criteria on what is happening. The specifications are written in the Ina Jo specification language [LSSE80] and use a state machine model. The conjectures asserting the desired properties are generated by the Ina Jo processor from the specifications. One attempts to prove these conjectures to be theorems with the aid of the Interactive Theorem Prover (ITP) [SS84].

The two major obstacles in the iterative process of specification and verification are

1. the sheer size of the conjectures to be proved and
2. the large amount of reproving that must be done when a specification is modified.

In a typical real-life application, the number of transforms that make state modifications and the numbers of constraints on the transforms and criteria on the states are large. For example, a toy university database [Bry82] specified by this author has 98 transform and 21 criteria and invariants. The specification of a real university database would have more than double of each. In any case, even in this toy example, each of the conjectures that one transform preserves the criteria and the invariants has an antecedent involving 22 conjuncts and a consequent involving 21 conjuncts. There are 98 of these conjectures to prove — a monumental task! However, if the specification is that big, then there will be many conjectures to prove, and there is nothing that can be done to avoid them. It seems to this author that the real difficulty is not in the numbers of the conjectures to prove, but in their individual complexity as measured by the number of conjuncts making up the statement of what is to be proved. There seems to be an inordinate amount of mental effort to prove a 21 conjunct consequent from a 22 conjunct antecedent, especially since the statement of the conjecture fills more than one screen or sheet of paper. In the jumble of the screen or paper it is difficult to pick out any given consequent conjunct and to locate possibly relevant antecedent conjuncts. This is enough to discourage even the most fastidious "grunt mathematician" [DLP79].

Ina Jo processor should be modified to decompose a specification into *smaller* conjectures, i.e., those with fewer conjuncts in their antecedents and consequents, than it presently generates. This would be done in such a way that proving all these conjectures is sufficient to guarantee that all of the original conjectures are theorems. It is possible that the ideal Ina Jo scheme would be one that allowed the specifier or verifier to designate, for each specification, the particular decomposition of conjectures with the single proviso that the conjunction of all the conjectures generated implies the original set of conjectures. Such an ambitious plan is beyond the scope of this paper, as further research is needed before this becomes feasible. In the meantime, however, an approach consisting of the generation of a standard set of smaller conjectures has been devised and is assumed in this document.

Another significant burden of the FDM process is the tremendous amount of reproving that must be done in order to be certain that no part of the proof is being skipped or improperly done. Each time a change is made to a specification, strictly speaking, the entire proof of correctness must be attempted again. Since there is no way at present to save and reuse proofs or portions thereof that have been done before, each attempt at the proof of correctness means doing the whole job from scratch. This is particularly frustrating when the changes are minor in their logical impact; that is if it is perfectly clear to the human that most of the previously completed proof is still applicable. For example, if only one constraint is changed and no transforms are changed, then it should not be necessary to reprove that all of the transforms imply the unchanged constraints. If only one transform is changed and no criteria or constraints are changed, then it should not be necessary to reprove that all the unchanged transforms imply the constraints and preserve the criteria.

In fact because of this frustration, there is a tendency not to do entire proofs. Instead, in practice, only those parts of the proof that *appear* to the prover to have been affected are reproved. While this saves time, the practice is not without its danger. It is easy enough to have made a mistake in assuming that it was not necessary to reprove a particular theorem. Thus, if it is desired to avoid redoing some parts of the proof, it is necessary to have a way to ascertain that parts of the proof that are being skipped can in fact be skipped.

Attempts to automate the discovery of conjectures that do not have to be repoved have not worked too well because of the space and time required to keep all proofs or at least a trace of them and to compare new conjectures with old theorems to find reuseable theorems. This comparison involves a veritable combinatorial explosion. For a specification with cr criteria and tr transforms, one is talking about $(cr \times tr)^2$ comparisons. ISI, which has had some experience in this area, seems to have abandoned the hope of automating the proof memory and reuse process [PC80]. Evidently human assistance is needed to guide the process in order to limit the number of comparisons that have to be made.

Another scheme to avoid reproof, namely that of keeping editable scripts of proving sessions, in order that these scripts be replayed is not a complete solution. While it saves the human effort, still the considerable computing resources to carry out the proofs are spent again. A true proof management system which keeps track of what is still applicable in order to avoid reproof entirely is necessary.

[1] This work was supported by SDC, A Burroughs Company.

® Ina Jo is a trademark of SDC, A Burroughs Company.

The only other proof management system known to this author is the Designer/Verifier's Assistant of Moriconi [Mor79]. This is an ambitious system in which the entire design, specification, and verification process is done under control of the Assistant. The system permits interrogation of the effect of a change before it is made. Then only if the effects are perceived as reasonable by the human, the Assistant is asked to carry out the change and the required new verifications. This system is felt to be too ambitious for the FDM effort. To implement it in the FDM would require even more substantial changes than are envisioned in this paper. Instead, a collection-of-tools approach is taken in which each tool does a more limited part of the job, and the tools may be arbitrarily composed under human control.

Thus some proof management system is needed for keeping track of changes to specifications and the statuses of proofs of parts of the correctness conjecture. All of the writing and changing of the specifications would take place under control of the manager, which is cognizant of the successive versions of the specification. The manager would be able to detect what parts, if any, of the specification has changed since the previous version. Proofs of parts of the correctness conjecture would be requested and monitored by the manager; for any version of the specification, it would request a proof of a part of the correctness conjecture only if the statement or the previous proof, if any, of the part involves parts of the specification that has changed. Gerhart lists the development of such a system as one of the outstanding technique problems for program verification in the 1980's [Ger78].

The function of such a proof manager strongly resembles that of existing configuration controllers for software development such as UNIX's® **make** [Fel78], Tichy's *RCS* [Tic81], and the proposed Ada® [ADA83] Programming Support Environment (APSE) [DOD81].

Note what the goal of the proof manager is. Its goal is to reduce and possibly even minimize the number of conjectures that must be proved as a specification is changed. The number is minimized if all theorems that are still applicable after a change to the specification are recognized as still applicable without reproof. Ultimately, the only way to determine if a theorem is still applicable is to reprove it. It may be that a particular theorem was proved using declarations of the specification that have been changed even though, in fact, it is still applicable; this is because there may be other proofs of the theorem not making use of any changed declarations. Necessarily, the proposed proof manager will be using only syntax-level means to determine if a theorem is still applicable, for to use proof to determine if a theorem is still applicable is defeating the whole purpose of the proof manager. The syntactic means will be to accept as still applicable all those theorems whose statements and proofs make use only of unchanged portions of the specification. The proof manager will be considered minimally successful if it avoids reproving at least one still applicable theorem without slowing down the proof process at another stage. It will be considered reasonably successful if in most cases, small changes give rise to only a few theorems requiring reproof. It will be considered optimally successful if nearly all theoretically still applicable theorems do not have to be reproved.

2 What is to be decomposed and managed

It is assumed that any time, the Ina Jo specification whose proof of correctness is to be managed consists of

1. a set of front matter declarations, $FM_1, \ldots, FM_{fm}$,
2. a set of initial condition declarations, $IC_1, \ldots, IC_{ic}$,
3. a set of transform declarations, $TR_1, \ldots, TR_{tr}$,
4. a set of criterion declarations, $CR_1, \ldots, CR_{cr}$,
5. a set of invariant declarations, $IN_1, \ldots, IN_{in}$,
6. a set of constraint declarations, $CO_1, \ldots, CO_{co}$, and
7. a set of mapping declarations, $MP_1, \ldots, MP_{mp}$.

The front matter consists of some numbers of each of the following kinds of declarations:

1. type declarations, $TY_1, \ldots, TY_{ty}$,
2. constant declarations, $CN_1, \ldots, CN_{cn}$,
3. axioms declarations, $AX_1, \ldots, AX_{ax}$,
4. variable declarations, $VA_1, \ldots, VA_{va}$, and
5. defined variable declarations, $DV_1, \ldots, DV_{dv}$.

The conjunctions of some of these declarations deserve special names, namely,

1. the conjunction IC_1 & $\cdots$ & IC_{ic} is IC, the *initial condition*,
2. the conjunction CR_1 & $\cdots$ & CR_{cr} is CR, the *criterion*,
3. the conjunction IN_1 & $\cdots$ & IN_{in} is IN, the *invariant*,
4. the conjunction CO_1 & $\cdots$ & CO_{co} is CO, the *constraint*, and
5. the conjunction AX_1 & $\cdots$ & AX_{ax} is AX, the *axiom*.

From such a specification, the Ina Jo processor generates a number of conjectures to be proved as theorems. These include:

1. initial condition criteria satisfaction (that the initial condition implies the criteria),
2. transform criteria satisfaction (that each transform preserves holding of criteria),
3. constraint satisfaction (that each transform implies the constraints), and
4. criteria mapping (that the range space transforms preserve the criteria under the map)

Other conjectures may be generated in the future. In order to *verify the correctness of a specification* one must prove all of these generated conjectures to be theorems. Verification of the correctness of a specification will be called simply *verification of the specification*.

The history of the production of a verified specification is as follows: A first version of the specification is written and when all of its syntax errors are eliminated, it is submitted to the Ina Jo processor for conjecture generation. These conjectures are subjected to a proof attempt by a human prover using the ITP. If and when this proof fails, the specification and proof are examined carefully to determine what changes must be made in order to render the specifications verifiable. The changes are made and the process is repeated with the new version of the specification. The process ends when a version is constructed such that all of the conjectures can be proved. Thus any proof management system must deal with changing specifications and determining which proofs from the previous version remain applicable to the current version.

®UNIX is a trademark of AT&T Bell Laboratories

®Ada is a trademark of the U. S. Department of Defense (AJPO).

3 Intuition

In describing the intuition, the discussion focuses on the problem of decomposing and of managing one particular class of conjectures among all of those mentioned above, namely the transform criteria preservation conjecture. Experience has shown this particular conjecture to be well-suited for motivating the techniques. The reader should not assume that the methods are applicable only to these conjecture and should not conclude that the other conjectures are not to be covered. After explaining the intuition behind the techniques, the details covering all classes of conjectures are given.

3.1 Conjecture Decomposition

Since the purpose of generating smaller conjectures is to make human job of directing the interactive proof easier, human methods of dealing with such conjectures should be examined for clues on how to decompose the statements of the conjectures. Consider the transform criteria preservation conjecture of formula (1).

(1) For each i, with $1 \le i \le tr$,
CR_1 & $\cdots$ & CR_{cr} & TR_i -> **N"**CR_1 & $\cdots$ & **N"**CR_{cr}.

Recall that CR_1 & $\cdots$ & CR_{cr} is called CR, and therefore **N"**CR_1 & $\cdots$ & **N"**CR_{cr} is called **N"**CR. The Ina Jo processor generates the transform criteria preservation conjecture as tr separate conjectures, one for each transform. The conjecture for transform i asserts the the second line of formula (1) above. In doing the proof of this conjecture, each conjunct of **N"**CR must be demonstrated, using whatever is known in the front matter, CR and TR_i. The numbers of front matter declarations, criteria, and transforms are very big in any appreciable, realistic specification. Thus, the intellectual effort to guide the proof of the transform criteria preservation conjecture is considerable.

When a human being is faced with having to produce such a proof, what does he or she do? He or she breaks it down into manageable pieces thusly:

(2) For each i and j, with $1 \le i \le tr$ and $1 \le j \le cr$,
CR & TR_i -> **N"**CR_j

It is necessary that the antecedent above include all of CR, because any of its conjuncts may be required in order to prove the desired consequent, **N"**CR_j. However, experience has shown that the number of conjuncts of CR that is actually needed to carry out the proof is usually quite small, that at least CR_j is usually needed, and that in many cases, only CR_j is needed. Thus, the human's strategy is: Attempt to prove that

(3) For each i and j with $1 \le i \le tr$ and $1 \le j \le cr$,
CR_j & TR_i -> **N"**CR_j

and be prepared to assume whatever of the conjuncts of CR that are needed to complete the proof.

Note that in decomposing the conjectures this way, many conjectures may trivialize, e.g., if transform T_i does not modify any variable appearing in C_j, either directly or indirectly via defined variables, then trivially,

(4) CR & TR_i -> **N"**CR_j .

It is proposed to have the Ina Jo processor output a set of conjectures to prove, i.e., namely for each i and j with $1 \le i \le tr$ and $1 \le j \le cr$,

(5) CR_j & TR_i -> **N"**CR_j ,

together with a list of assumable criteria, $CR_{1,\ldots,}$ CR_{j-1}, CR_{j+1}, ... CR_{cr}. The Ina Jo processor will have simplified these conjectures before outputting them in an effort to find the trivial ones. The proof manager would then send off to the ITP each nontrivial element of the set in turn and the human prover could direct the ITP to assume any of CR_1, . . . , CR_{cr} he or she thinks is appropriate to complete the proof.

There are other kinds of theorems to prove about a specification. For each of these kinds of conjectures, an appropriate basis must be found of dividing the required proofs into manageable chunks. (With this understanding that the decomposition idea can be adapted to all the kinds of proofs that must be done, the remainder of this proposal is couched in terms of the requirements of the transform criteria preservation conjectures.)

Basically the decomposition question is one of how to break up the formula

(6) P -> Q,

where P and Q each are conjunctions of other formulae. Breaking Q into Q_1 & $\cdots$ & Q_q allows development of q smaller conjectures each with a different Q_i as a consequent. Breaking P into P_1 & $\cdots$ & P_p allows identification of some of the P_ms as antecedents to the theorem and some of the P_ms as assumables. This decomposition involves a number of engineering trade-offs.

1. The more that is included in the statement of a conjecture, the more likely a conjecture is to be affected by a change in a declaration.
2. The less that is included in the statement of a conjecture, the more conjectures there are to prove.
3. The less that is included in the antecedent of a conjecture the less likely the prover will have enough to prove the conjecture.

On the other hand, in some cases, it may be better to leave some conjuncts of the criteria together because they simplify together well or they are acted upon together in many transforms. On the other, other hand, in these latter cases, perhaps these criteria should have been anded together in the specification to form one criterion. What is critical is that the specifier and/or prover have a means to control the decomposition. One possibility is to have a description, independent of the specification, that states how conjectures are to be decomposed, a "*thermcap*" if you will. Another is to find a single decomposition scheme for each class of conjectures such that the specifier and/or prover has complete control over the decomposition through how declarations are merged in the specification. There are still other possibilities.

Thus, there may be any number of ways to decompose a conjecture into smaller ones, and the right one must be found, possibly by experimentation. A particular decomposition can effect the ability of a simplifier to simplify, how proofs are carried out, and how easily reuseable proofs can be found. Good decompositions may turn out to be a function of the style of specification and/or the style of verification. Clearly, more work needs to be done.

The effect of this proposal is to get the Ina Jo processor, the proof manager, and the ITP collectively to approach the proof job much the same way a human being would.

For substantiation that the above described approach is in fact what a human prover is doing, when he or she is faced with doing a proof essentially by hand, please see the work leading to the Ph. D. thesis of R. A. Kemmerer [Kem79] in which a specification similar to Ina Jo specification is proved (in Ina Jo terminology) to preserve its security criteria. Among the working papers of this work, one finds exactly

the set[2] described above, with trivial theorems marked as such. Also in "Validation of the PDP-11/45 Security Kernel: Upper Level Formal Specification" by J. K. Millen of Mitre Corporation [Mil7?], one finds finds the same set[2] organization and the step to locate trivial theorems.

The reason that the conjectures, as broken down above, are more manageable is that in this broken down form, they involve only the individual criteria of the specification and not a conjunction of them. During the writing of the specification itself, the large criterion, CR, was broken down into the $CR_1, \ldots, CR_z$ in the first place by the specifier just so that he or she could manage them.

3.2 Avoiding Reproving Already Proved Theorems

As noted above, one of the most frustrating aspects of the job of verification of a specification is the fact that as modified specifications are submitted for proof, the same proofs have to be done in their entirety. It would be nice if portions of a proof that are still applicable in the modified specification can be remembered and re-used.

Again, the question must be asked: How would the human remember and reuse proofs in subsequent versions of a specification? The human would not reprove any conjectures whose statement or proof involves any piece of the specification which had not changed. This is exactly the behavior that the proposed automated systems were trying to simulate. However, the human has one advantage. Because each piece of a specification has an abstracted statement in the human's natural language, the human can identify unchanged pieces much more readily than any automated system, which must use brute force comparison to do the same.

What is needed is a system that can keep track of the pieces contributing to the statement and the proof of a theorem much the same way a human can. It turns out that the conjecture set organization described in Section 3.1 provides a good basis for such a system.

Specifically, the proof manager would keep track with each conjecture in the set,

1. a list of all declarations used in the conjecture's statement and simplification,
2. whether or not it had been proved by the ITP, and
3. if proved, a list of all declarations actually used in the proof — this will contain front matter declarations and a sublist of the list of assumables provided with the conjecture.

If and when a specification is modified, the proof manager would compare the old and new specifications and the old and new conjecture sets. It would determine which declarations of the specification had been modified or removed. It would then mark as proved any new conjecture which is identical to an old conjecture whose used declaration list does not contain any modified or removed declaration. These already proved theorems would inherit the information from the old set, and only the unproved new conjectures would be sent to the ITP to be proved.

It is hoped that as the number and impact of modifications to a specification get smaller and smaller, the number of still applicable theorems gets larger and larger and the number of conjectures to prove gets smaller and smaller.

[2] Kemmerer's and Millen's sets are actually expressed as matrices whose elements are indexed the relevant transform and the relevant criterion.

4 Details of Proof Manager System

The proposed proof manager is one part of the complete system illustrated below in Figure 1. Its legend is provided as Figure 2.

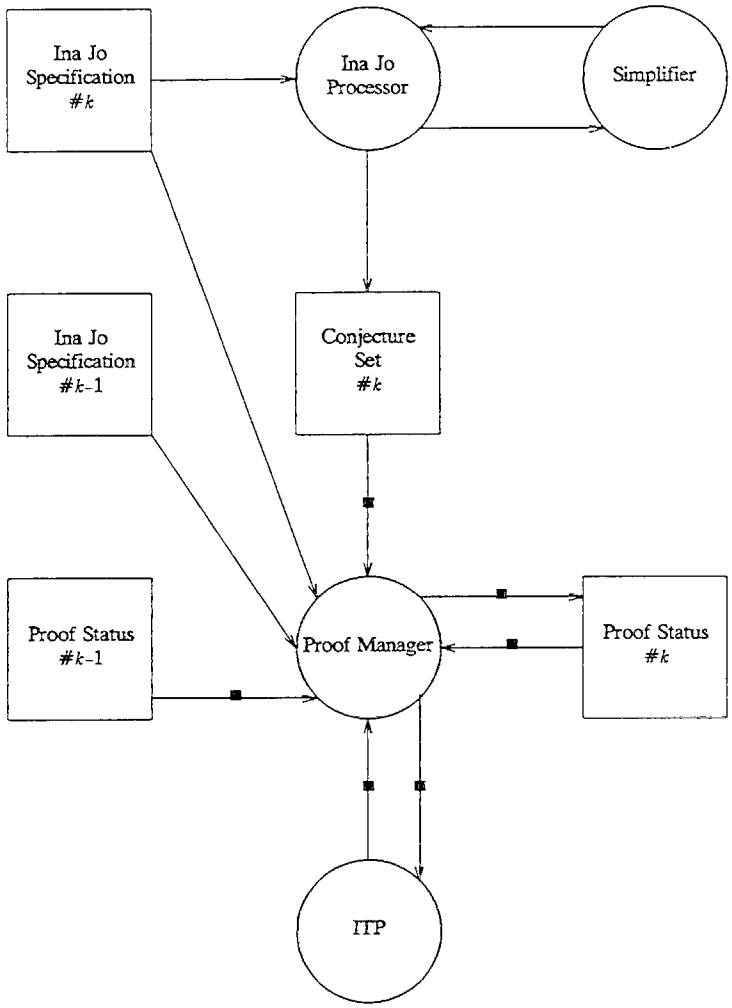

Figure 1: Proof Manager

The running example focuses on the transform criteria preservation conjecture. The detailed description of the proof management system offered in this section is general.

Legend for Figure 1:

Each position of Conjecture Set Contains:
1. Original conjecture to be proved
2. Simplified conjecture
3. Assumable declarations
4. Declarations used in statement and simplification of conjecture

Each position of a Proof Status Contains:
1. Original conjecture to be proved
2. Simplified conjecture
3. Assumable declarations
4. Declarations used in statement and simplification of conjecture
5. Proved?
6. If proved, declarations used in proof of conjecture

The boxes on some of the arrows are positions of the Conjecture Set or of a Proof Status being transported for analysis.

Figure 2

The system consist of a number of parts, some of which already exist, some of which must be created by modification of currently existing existing software, and some of which must be created anew. Its parts are:

1. the Ina Jo language, which already exists,
2. the Ina Jo processor, which has to be modified,
3. the Simplifier, which must be developed anew,
4. the Proof Manager, which must be developed anew, and
5. the ITP, which has to be modified.

Each element of a conjecture set contains the following information about one of the conjectures generated by the Ina Jo processor.

1. the conjecture to be proved in its original unsimplified form
2. the simplified form of the conjecture to be proved (which may already have been simplified to `true`),
3. the list of assumable declarations,
4. proved/not proved (if the conjecture is simplified to `true` then it is proved), and
5. the used declaration list, i.e., the list of all declarations used in the statement and the simplification of the conjecture.

In the case of the transform criteria preservation conjecture, an element might contain the following:

1. CR_j & TR_i -> **N"** CR_j ,
2. `true`
3. $CR_1, \ldots, CR_{j-1}, CR_{j+1}, \ldots, CR_{cr}$.
4. proved
5. $CR_j, TR_i, D_1, \ldots, D_d$

where it is assumed that $D_1, \ldots, D_d$ are the declarations used to simplify the conjecture to `true`. Alternatively, the element might contain the following:

1. CR_j & TR_i -> **N"** CR_j ,
2. CR_j & TR_i -> **N"** CR_j ,
3. $CR_1, \ldots, CR_{j-1}, CR_{j+1}, \ldots, CR_{cr}$.
4. not proved
5. CR_j, TR_i

assuming that simplification failed to to reduce the statement of the conjecture at all.

Each element of a proof status contains the following information about the ITP proof of one of the conjectures generated by the Ina Jo processor.

1. the conjecture to be proved in its original unsimplified form
2. the simplified form of the conjecture to be proved (may even be `true`),
3. the list of assumable declarations,
4. proved/not proved (by the ITP), and
5. the used declaration list, i.e., the list of all declarations used in the statement, the simplification, *and the proof* of the conjecture.

The organizations of the conjecture sets and the proof statuses are such that each element can be uniquely selected by giving the declarations that go into the original, unsimplified statement of the element's conjecture. In the two example conjecture set elements mentioned above, CR_j and TR_i together select the example elements. In addition, their organizations are such that is is possible to locate each element that has a particular declaration, D_k, in its used declaration list. As changes in a declaration will be the key to changes in the proof status of all conjectures whose proof used it, it will be important to be able to readily identify all those conjectures.

The flow of control through the system is as follows: It is assumed that the specifier has written the k^{th} version, S_k, of the specification, where $1 \leq k$. In most cases, i.e., when $k>1$, there is a previous version, S_{k-1}. The description below is written assuming that there is a previous version, and when the steps differ for the first version, what is done differently is indicated inside parentheses.

1. S_k is sent through the Ina Jo processor. The Ina Jo processor with the help of the simplifier produces the conjecture set CS_k.
2. The proof manager takes S_k and S_{k-1} in order to be able to determine which declarations of S_{k-1} have changed[3] to produced S_k. (If $k=1$, then all declarations are assumed to have been changed.)

[3] In order to make this comparison computationally feasible, it may be necessary to require that all declarations have a name. Only like-named declarations would be compared for being unchanged. Any name appearing only in the second specification would be presumed to name an added declaration, and any name appearing only in the first specification would be presumed to name a removed declaration. In the Ina Jo language, as it is now, declarations of types, constants, variables, defined variables, and transforms have names, namely the identifier of the declaration. The declarations of axioms, initial conditions, criteria, constraints, and invariants do not have any identifier that can serve as the name. Thus it would be necessary to add to these latter kinds of declarations, naming identifiers, e.g.,

3. The proof manager builds PS_k to have one element for each element that is in CS_k. For each element E in PS_k, suppose that its original, unsimplified conjecture is C, indexed by declarations $D_1, \ldots, D_d$. Then the contents of E is determined as follows:

   a. The element C of CS_k indexed by $D_1, \ldots, D_d$ is copied into E. Any such element of CS_k which has been simplified to `true` is marked as *proved* in PS_k. All others are marked as *not proved*.

   b. If the element C indexed by $D_1, \ldots, D_d$ exists in PS_{k-1}, it is marked *proved* in PS_{k-1}, all declarations in its use list are unchanged in S_k, *and* if the element indexed by $D_1, \ldots, D_d$ is *not* simplified to `true` in CS_k, then that element of PS_{k-1} is copied into E. Such a copied element remains marked as *proved*. Thus, if in the latest version a conjecture is proved by simplification, the simplification is accepted as the proof rather than any previous ITP proof, even if the ITP proof is still applicable[4]. (If $k=1$, then E is unchanged.)

4. The proof manager begins to request ITP proof of all elements of PS_k which are not marked as proved.

5. If and when all elements of PS_k are marked proved, the specification S_k is considered verified. If for some reason, some element cannot be proved, then it is necessary for the human specifier to examine S_k with an eye toward modifying it to yield a provable S_{k+1}.

In order that the above described proof management system work as specified, the existing software must be modified as described below:

1. The Ina Jo processor must be modified so that it outputs the conjecture set as described above. This will require that it generate a different collection of conjectures than it is doing now. In addition, it will need to keep track of what declarations are used to simplify a conjecture and to report for each conjecture these declarations.[6]

2. A simplifier must be developed so that besides simplifying, it reports to its invoker which declarations it used to carry out its simplifications.

3. The ITP must be modified so that

   a. one can present to it an element of the proof status for proof, that is, so that one can give it only a conjecture and a list of assumables (at present, one must give it whole specifications or only conjectures without a list of assumables), and

   b. it keeps track of and reports to its invoker all declarations actually used in the proof.

5 Testing the System

The development of the proof management system entails some risk. Most pieces of the software are relatively easy to build from scratch or as a modification of existing software. However, other pieces involve major modification of existing software and doing these modifications cannot be undertaken lightly. The risk stems from the the possibility that the system will not yield the desired benefits. Possible negative results of using the proposed system include

1. the initial proof burden is increased as a result of the different decomposition,
2. many fewer theorems than anticipated are found to be reusable, and
3. fewer trivial theorems than anticipated are eliminated by the simplifier.

A prototype proof manager will be developed. While the prototype proof manager is being written, another version of the Ina Jo processor generating the new decomposed conjecture set will be written as a modification of the production version. The proof manager would then proceed from this point to analyze the conjecture sets to determine which conjectures need to be proved.

The effectiveness of the tool will be tested by applying the prototyped proof manager to a modified specification and measuring the proof manager's capability of identifying theorems that have remained applicable, and thus may be retained without reproof. A further assessment will be made to determine if the number of theorems identified for retention grows as the magnitude of the changes to the specification decreases. If a real specification effort in progress is unavailable, a series of representative artificial changes to a specification will be used.

6 Discussion

If one thinks about how specifications are modified during the debugging process, one realizes that in a well-structured, well-modularized specification, changes tend to be very local, restricted to a few pieces of the specification. In such a case, the proof manager will find that much of the database is unchanged, and much of the reproving effort will be saved.

In the relatively rare case that a specification undergoes a complete revision, of course, none of the old proofs will be reuseable, the proof manager will find nothing reuseable, and it will appear that nothing had been gained from having the proposed proof manager. However, in this case, what would need to be done under the proof manager is exactly what would have to be done under the current Ina Jo processor and ITP. Thus the proposed proof manager would seem to be effective at reusing proof in the evolutionary case, in which much of the proof would be reuseable, and would be no worse than what is available now for the revolutionary case, in which almost none of the proof would be reuseable anyway.

Note that the structure of the proof manager is such that to use it effectively, one quickly learns to modularize one's specifications so as to minimize the number of changed database elements. As observed by Glenford Myers and David Parnas [Mye75, Par72], such modularized specifications weather maintenance better and are cheaper overall to implement.

Finally, the system itself is structured in such a way that its parts can be developed as independent projects without disturbing the current work with or on the Ina Jo processor and the ITP. The proof manager and the database are completely separate from the existing software. The required changes to the Ina Jo processor are in isolated separable parts, i.e., the grammar, the theorems generator. Certainly, the changes required of the ITP are not minor. However, it will not be modified until it is ascertained that the system will have the desired benefits.

---

criterion x_always_positive
x>0.

From the human specifier's point of view, such a name, if well-chosen, is better to use as an index than an ordinal number, because a well-chosen name is not likely to change as the specification changes while an ordinal number will surely change as pieces are inserted and removed.

[4] Thus, the still applicable proof is deleted from the record. Experimentation may show that not saving the old ITP proof causes unnecessary reproof later. In this case, the condition above "*and* if the element indexed by $D_1, \ldots, D_d$ is *not* simplified to `true` in CS_k" will be removed.

[6] In addition, if it is discovered that it is necessary to have named declarations, the Ina Jo processor must be modified so that it requires named axioms, initial conditions, criteria, constraints, and invariants.

Acknowledgements

The author thanks Ann Barton, Debbie Cooper, John Scheid, and Gerrie Smith for their comments on an earlier draft. Even though they were originally thought to be a pain to take into account, taking them into account immensely improved this paper. Also the author thanks Mike Melliar-Smith for showing SRI's system and for useful discussions.

Bibliography

[DOD81] "Requirements for the Ada Programming Support Environment: STONEMAN," Technical Report, U.S. Department of Defense (1981).

[ADA83] "Ada Language Reference Manual," MIL-STD-1815A, U.S. Department of Defense (1983).

[Bry82] Berry, D.M., "The Application of the Formal Development Methodology to Data Base Design and Integrity Verification," *Proceedings of Fourth Israel Conference on Software Quality Assurance* (1982).

[Fel78] Feldman, S.I., "Make — A Program for Maintaining Computer Programs," Technical Report, Bell Laboratories, Murray Hill, NJ (1978).

[Ger78] Gerhart, S.L., "Program Verification in the 1980s: Problems, Perspectives, and Opportunities," ISI/RR-78-71, USC Information Sciences Institute, Marina Del Rey, CA (August, 1978).

[Kem79] Kemmerer, R.A., "Formal Verification of the UCLA Security Kernel: Abstract Model, Mapping Functions, Theorem Generation, and Proofs," Ph.D. Dissertation, Computer Science Department, UCLA (1979).

[LSSE80] Locasso, R., Scheid, J., Schorre, D.V., and Eggert, P.R., "The Ina Jo Reference Manual," TM-(L)-6021/001/000, System Development Corporation, Santa Monica, CA (June 27, 1980).

[Mil7?] Millen, J.K., "Validation of the PDP-11/45 Security Kernel: Upper Level Specification," Technical Report, Mitre Corp., New Bedford, MA (197?).

[DLP79] Millo, R.A. De, Lipton, R.J., and Perlis, A., "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM* **22**(5), pp.271-280 (1979).

[Mor79] Moriconi, M.S., "A Designer/Verifier's Assistant," *IEEE Transactions on Software Engineering* **SE-5**(4), pp.387-401 (July, 1979).

[Mye75] Myers, G.J., *Reliable Software through Composite Design*, Petrocelli/Charter, New York, NY (1975).

[Par72] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* **15**(2), pp.1053-1058 (December, 1972).

[SS84] Schorre, D.V. and Stein, J., "The Interactive Theorem Prover (ITP) User Manual," Tech. Report TM-6889/006/01, System Development Corporation, Santa Monica, CA (1984).

[PC80] Smallberg, D.A. and London, R., "Private Communication," USC ISI (1980).

[Tic81] Tichy, W., *Revision Control System*, Purdue University, Lafayette, IN (1981).