Finding Incremental Solutions for Evolving Requirements

Neil A. Ernst
Department of Computer Science
University of Toronto
nernst@cs.toronto.edu

Alexander Borgida
Department of Computer Science
Rutgers University
borgida@cs.rutgers.edu

Ivan Jureta
FNRS & Information Management
University of Namur
ijureta@fundp.ac.be

***Abstract*—This paper investigates aspects of the problem of software evolution resulting from top-level requirements change. In particular, while most research on design for software focuses on finding some correct solution, this ignores that such a solution is often only correct in a particular, and often short-lived, context. Using a logic-based goal-oriented requirements modeling language, the paper poses the problem of finding desirable solutions as the requirements change. Among other possible criteria of desirability, we consider minimizing the effort required to implement the new solution, which involves reusing parts of the old solution. In general, the solution of requirements problems is viewed as an exploration using a "requirements engineering knowledge base" (REKB), whose specification is formalized. The paper reports on experience implementing the REKB on top of a so-called "reason-maintenance system", and provides evidence that incremental solution finding is indeed more efficient.**

***Keywords*-Requirements; evolution; incremental; knowledge-level.**

I. Introduction

Many requirements problems must be solved in the context of existing, so-called "brownfield systems". Systems which were assumed to last for only a few years are now decades old. And it has long been known that the maintenance phase of a software system's lifecyle consumes the lion's share of the resources, while consisting mostly of *adaptive* rather than corrective changes. Apart from strictly practical reasons (e.g., corporate inertia or the challenge of legacy systems updates), there are other good reasons why working with existing requirements is important. For many projects a common way to elicit requirements is by reusing requirements from previous projects, which is associated with lower requirements volatility [1]. Furthermore, a large amount of engineering effort is devoted to planning requirements for new software releases. To support re-use and planning, we must maintain requirements, their existing implementations, and domain assumptions throughout the software's lifecycle. We need a way to solve requirements problems not as one-off challenges, but as evolving and context-dependent problems.

The objective of this paper is to re-consider the problem of finding a solution to the requirements problem in a goal-oriented requirements modeling and analysis framework, when the requirements and domain assumptions are undergoing change. In particular, we start with the key aspects of the Techne approach to requirements [2], where solutions are specified as minimal sets of tasks that achieve goals (of various kinds) based on domain knowledge, including rules for goal decomposition into subgoals and tasks that achieve them.

We distinguish the problem this paper addresses, that of *evolving requirements*, from that of *adaptive requirements*. The main difference is that in adaptive requirements frameworks such as [3] and our own paper [4], one makes the assumption that the overall set of requirements Δ, including goals, decompositions, possible solution tasks and domain knowledge do not change; monitoring is however taking place, and alternate valid solutions are selected, choosing among *existing correct alternatives*, when failures of some requirements are detected. For example, if a washing machine is turned off unexpectedly, we can adapt by restarting the machine, filling it with clothes, etc., so long as these countermeasures exist and are already known to the system.

Evolving requirements, on the other hand, are the result of unanticipated changes. Such situations are of course common in requirements. One major source are the changes introduced by new legislation, such as the U.S. healthcare privacy legislation (HIPAA), or the Sarbanes-Oxley financial reporting act. Other sources of changes are requests for enhancements by current users. These are all changes to the original requirements Δ, and cannot be modeled a priori. One might argue that in this case the default action is to take the requirements model 'offline' and recompute possible new solutions. However, this is almost certainly a duplication of effort. What we may want instead is a suitable *repair/modification* to the existing requirements model that will re-use as much of the old solution as possible, and minimize the number or cost of new implementations. Finding these incremental repairs is the focus of this paper.

In particular, this paper makes the following contributions:

- States explicitly and formally for the first time, to our knowledge, that what appear to be suboptimal solutions to a requirements problem (RP) might in fact be preferred, if the problem was the result of modifying a previous RP, for which a solution has already been adopted and implemented.
- Identifies a variety of criteria for preferring certain

solutions to the modified requirements problem. These criteria can be based on the degree to which they reuse elements of the previous solution.

- Introduces a workbench metaphor for the solution of goal-based RPs, especially after problem evolution. Finding solutions is an exploration carried out by users with the support of a requirements engineering knowledge base (REKB), and specifies its operators at the logical knowledge-level (rather than the implementation level), according to principles advocated by Levesque [5].
- Implements an REKB on top of a Reason Maintenance System[1] as back-end, which, among others, is designed to work in an *incremental* manner.
- Provides experimental evaluation of the proposed implementation, including evidence that it indeed solves problems incrementally in ways that are more efficient than starting from scratch.

The remainder of the paper introduces in Section II the requirements problem, the Techne version of it, and the details of the requirements evolution problem in its context. Section III introduces the REKB, specifying its core operations and illustrating their use. Section IV briefly mentions algorithms for implementing the operations and the complexity of the problems they try to solve, while Section V discusses the manner in which the ATMS reason maintenance system can be used to implement the REKB. Section VI concerns experimental evaluation, while Section VII considers related and future work, followed by our conclusions.

II. Aspects of the requirements problem

Requirements engineering for software elicits the desired characteristics of a machine (specification S) that will bring about some desired properties (the requirements R) in an environment or world W. Finding this specification is the *requirements problem*, formalized by Zave and Jackson [7] as satisfying the condition:

$$W, S \vdash R \qquad (1)$$

where $W \cup S$ is consistent, and $\vdash$ is some form of logical deduction relationship. Note that a final, running system, achieving the requirements R, is obtained when the specifications S are actually implemented.

Subsequent research [8], [9] demonstrated the need to expand on this relationship. First, given W and R, there can be *many* possible specifications S which solve the requirements problem. Therefore, provisions should be made for exploring the space of solutions and comparing them. In our case, this will be a "workbench" metaphor based on a requirements database. Second, requirements are not all alike, and include, among others, ones that are *mandatory* ("must-have") and others which are *optional*, those the stakeholders either consider "nice to have", or prefer to other requirements. Additional requirements emerge during analysis and serve to structure the problem space using refinements.

[1]While commonly known as truth-maintenance systems, this is not 'truly' reflective of what such systems do [6].

A. The variant of the requirements problem in Techne

To address these deficiencies, [10] describes an ontology for requirements engineering called CORE, which is based in part on: *goals* G of various kinds and attitudes to them; *tasks* T referring to behaviours of the system-to-be or in its environment; and *domain assumptions* D which are conditions believed to hold, and which help operationalize goals into tasks achieving them. This was used as a basis for a new requirements modeling language, Techne [2]. In this paper we use a simplified version of Techne, where we ignore some variants of goals (e.g. soft vs hard) as well as "quality constraints" because they do not affect our solutions.

Example II.1. *Our objective is to build a software system for an online music service (such as Pandora.com). Our elicitation with the stakeholders has produced a set of communications sorted into tasks T, goals G, domain assumptions D, including:*

- G_{GenRev}*: The system shall generate revenue. (A mandatory goal.)*
- G_{Ads}*: The system shall display text ads.*
- G_{Sub}*: The system shall charge users a subscription to listen.*
- T_{Target}*: Target text ads based on profiles.*
- T_{Acct}*: Enable account-free login.*
- T_{GoogAd}*: Leverage Google Ads to serve advertising.*
- $T_{Restrict}$*: Restrict the music player to subscribers.*
- T_{Secure}*: Encrypt user payment page with SSL.* ■

A key part of solving requirements problems is finding ways to refine requirements using other requirements, to further refine certain requirements into tasks, and to record conflicts between requirements, giving rise to refinement and conflict relations. The existence of individual relations is considered to be part of D in Techne.

Example II.2. *The previous example must be augmented with refinements and conflicts connecting goals, tasks, etc.:*

- i_1*: Satisfying* G_{Sub} *satisfies* G_{GenRev}.
- i_2*: Satisfying* G_{Ads} *satisfies* G_{GenRev}.
- c_1*:* T_{Secure} *and* T_{Accnt} *are in conflict (we cannot simultaneously allow open access and also secure access to subscribers).*
- i_3*: Doing* $T_{Restrict}$ *and* T_{Secure} *satisfies* G_{Sub}.
- i_4*: Doing* T_{Accnt} *and* T_{Target} *and* T_{GoogAd} *satisfies* G_{Ads}.

Fig. 1 shows a sample REKB *represented graphically, using the stakeholder communications from above.* ■

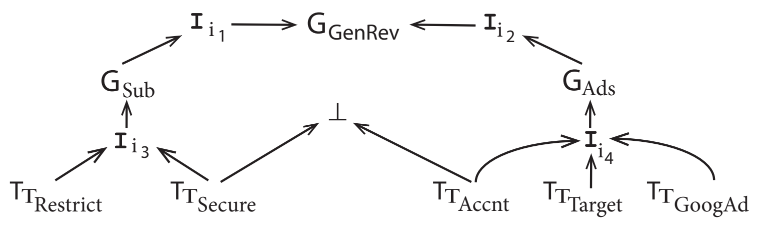

Figure 1. A simple REKB showing goals, tasks, refinements and conflicts.

Example II.3. *What sets of tasks, if implemented, will satisfy the mandatory goal G_{GenRev}? There are two alternatives for accomplishing G_{GenRev}, and so the acceptable solutions are precisely those tasks which satisfy these alternatives. Therefore S_1 : $\{T_{Restrict}, T_{Secure}\}$, and S_2 : $\{T_{Accnt}, T_{Target}, T_{GoogAd}\}$.* ■

In Techne, the requirements problem is re-stated (approximately here) as the search for a subset S of the known tasks T, and refinements in D, which together explain how one can achieve a subset $\acute{G}$ of goals G, such that all mandatory requirements are present in $S \cup \acute{G} \cup D$, and ideally also as many attractive requirements as possible. This implies that we require

$$D, S \vdash \acute{G}$$

and that $D \cup S$ be consistent. Since it does not make sense to have extraneous steps/tasks in such sets S, we expect them all to be minimal, although there may be many solutions S.

Techne balances a reasonably simple representation of requirements with support for automated reasoning. The closer one is to the start of the RE process, the more important it is to keep representations of requirements simple, since requirements are often vague and ambiguous at this point. This avoids wasted effort in subsequent revisions and changes. Representation of requirements in a propositional formalism such as Techne does not harm this aim; at the same time, it allows automated reasoning to be used to answer questions of interest in the very early structuring and exploration of the requirements problem and solution space. In a propositional formalism, natural language statements of requirements remain as they are, and thereby readable to any stakeholder: what is formalized are some specific relations between these statements, and it is by looking at these relations that automated reasoning is performed (rather than by looking "into the content" of natural language statements). The formalization thereby introduces no burden on the modeler, other than that which the modeler has whenever she uses a visual language, in which modeling involves adding nodes (i.e., requirements) and edges (i.e., relations between requirements) to a model of requirements.

B. The Requirements Evolution Problem

Suppose now that we have acquired and solved a requirements problem RP1. "Solved" here means we selected a particular collection of tasks S_0 among the ones specified above, and have a running solution, consisting of some implementation for the tasks in S_0. As we argued earlier, we are likely to encounter unanticipated changes to the problem, which include changes to all aspects of RP1, including the goals, tasks, and various forms of knowledge about refinement. Suppose this results in a new requirements problem RP2. Formally, the obvious path to follow in this case would be to view RP2 as a completely new problem, and simply search for a new solution. However, this view is unrealistic in the real wold: for example, one does not throw away an entire software system and start design from scratch, when a new feature is desired. Instead, the solution is likely to be *incremental*: start from the current solution and try to move towards one that meets the problem captured in RP2.

A key goal of this paper is to explore this evolutionary aspect of the requirements problem. It can be restated semi-formally as:

> **Problem statement:** Given (i) goals G, domain knowledge D, and (ii) some chosen *existing* solution S_0 of tasks (i.e., one that satisfies $D, S_0 \vdash G$), as well as (iii) modified requirements (δ(G),δ(D),δ(T)) that include modified goals, domain knowledge and possible tasks, produce a subset of possible specifications $\hat{S}$ to the changed requirements problem (i.e., $\delta(D), \hat{S} \vdash \delta(G)$) which satisfy some desired property Π, relating $\hat{S}$ to S_0 and possibly other aspects of the changes.

Note that $\hat{S}$ is no longer required to be minimal.

We present below some plausible alternative properties Π for choosing solutions, together with illustrative examples based on a case where: $S_0 = \{a, b, c, d, e\}$ was the initial solution (the set of tasks that were implemented); and $S_1 = \{f, g, h\}, S_2 = \{a, c, d, f\}$ and $S_3 = \{a, b, c, d, f\}$ are minimal sets of tasks identified as solutions to the new requirements:

1) *The standard solutions*: this option ignores the fact that the new problem was obtained by evolution, and looks for solutions in the standard way. In the example, one might return all the possible new solutions $\{S_1, S_2, S_3\}$, or just the minimum size one, S_1.
2) *Minimal change effort solutions*: These approaches

look for solutions $\hat{S}$ that minimize the extra effort $\hat{S} - S_0$ required to implement the new "machine" (specification). In our view of solutions as sets of tasks, $\hat{S} - S_0$ may be taken as "set subtraction", in which case one might look for (i) the smallest difference cardinality $\mid \hat{S} - S_0 \mid$ (S_2 or S_3 each require only one new task to be added/implemented on top of what is in S_0); or (ii) smallest difference cardinality *and* least size $\mid \hat{S} \mid$ (S_2 in this case).

3) *Maximal familiarity solutions*: These approaches look for solutions $\hat{S}$ that maximize the set of tasks used in the current solution, $\hat{S} \cap S_0$, while implementing the current machine. One might prefer such an approach because it preserves most of the structure of the current solution, and hence maximizes familiarity to users and maintainers alike. In the above example, S_3 would be the choice here.
4) *Solution reuse over history of changes*: Since the software has probably undergone a series of changes, each resulting in newly implemented task sets $S_0^1, S_0^2, ..., S_0^n$, one can try to maximize reuse of these (and thereby even further minimize current extra effort) by using $\bigcup_j S_0^j$ instead of S_0 in the earlier proposals.

The above list makes it clear that there is unlikely to be a single optimal answer, and that once again the best we can hope for is to support the analyst in exploring alternatives.

III. The Functional REKB approach

Returning to the original requirements problem, the example in Section II-A omitted one communication from the stakeholder: p_1*: It is preferable to display text ads (G_{Ads}) rather than charging subscriptions (G_{Subs}).* Our current methodology [2] suggests that preferences should be dealt with by humans in choosing between acceptable solutions T_1 and T_2 to the original requirements problem, and hence preferences will henceforth not be explicitly discussed as part of the requirements. However, they further motivate the need to *enumerate* multiple solutions.

This leads us to a view where we support a *requirements engineering knowledge base* (REKB), which is used by the problem solver as a tool for: (i) storing the information acquired during requirements acquisition and domain modeling, as well as justifying problem decomposition; and (ii) asking a variety of questions that can help them compute and compare alternative solutions. We emphasize that what the problem solver does with the basic kinds of answers received, i.e., which solutions it chooses, remain separate. Solving the requirements problem is not the same as operations on the REKB, and answers from the REKB may, or may not, be relevant to solutions.

To fill out this schema, we must provide details of the capabilities of the REKB. We follow Levesque's seminal account of knowledge bases (KBs) [11], which starts with an abstract data type view of KBs that hides implementation details. We must therefore provide **a)** a list of *operations* with their syntactic signatures, and **b)** an implementation-independent *specification* of the effect of each operation. Of course, this must be completed by an implementation.

As stated in [11], "the capabilities of a KB are strictly a function of the range of questions it can answer and assertions it can accept." Therefore we must specify query answering based on what has been told. The "knowledge level" approach to specification, advocated by Levesque, is to use logical formulas as key languages to interact with REKB, and model theory to specify question answering. The advantage of this approach is that to solve the requirements evolution problem, we can choose repair engines according to various independent criteria (e.g., speed, cost, completeness), as long as they achieve the specification.

A. Operators for REKB

More precisely, when working with requirements, two abstract families of operations are necessary: ones for adding/modifying information of any form listed in the requirements problem definition above; and extracting information in the form of solutions to our problem. This requires (several) TELL operators/languages and ASK operators and associated languages:

$$\textbf{TELL} : \text{REKB} \times \mathcal{L}_{tell} \rightarrow \text{REKB}$$
$$\textbf{ASK} : \text{REKB} \times \mathcal{L}_{ask} \rightarrow \mathcal{L}_{answer}$$

The basic logical language we use to express requirements ($\mathcal{L}_{tell}$) is quite simple: propositional Horn logic built up from atoms. Its syntax is captured by

$$\begin{aligned} formula \ ::= \ & atom \mid \\ & (\textstyle\bigwedge_{i=1}^{n} atom_i) \rightarrow atom \mid \\ & (\textstyle\bigwedge_{i=1}^{n} atom_i) \rightarrow \bot \end{aligned}$$

Horn clauses are sufficient for our purposes, as they can represent Techne concepts of AND/OR decomposition and refinement, as well as the notions of conflict. We distinguish 3 kinds of formulas, using values from the domain SORT = $\{goal, task, domain\ assumption\}$. (We will use GOAL, TASK and DA for the set of formulas of the respective sort.) Formulas become well-formed (wff) by first assigning each atom a unique value from SORT, and then, if our REKB should respect a given methodological view, specifying how sorted implications (which are in DA) can be assembled (e.g., no goals may imply tasks).

Finally, in order to make it easier to refer to wffs, and represent requirements as graphs, we will attach *labels* λ in front of formulas ψ, to obtain *labelled wffs* $\lambda : \psi$.

B. TELL operations

As usual in logic, one needs to introduce the atoms allowable in the language. For this, we distinguish a symbol table REKB.ST in the REKB from the collection of formulas

REKB.TH in it, and provide an operation for declaring new atoms:

Operation 1 — DECLARE_ATOMIC
Domain: REKB $\times$ ATOM $\times$ SORT $\times$ LABEL
Co-Domain: REKB
Effect: Add the atom to the symbol table REKB.ST, with the appropriate sort and label.
Throws: Raise exceptions if the atom is already declared or the label used.[2]

Note that a declaration just introduces a symbol – it is not the same as asserting it to be true. The later is achieved for any formula by:

Operation 2 — ASSERT_FORMULA
Domain: REKB $\times$ WFF $\times$ SORT $\times$ LABEL
Co-Domain: REKB
Effect: Add the labelled formula to the theory REKB.TH
Throws: Gives a warning if REKB.TH becomes an inconsistent theory.[3]

Finally, we need to be able to distinguish formulas (especially, but not exclusively goals) that are mandatory, etc., using

Operation 3 — ASSERT_ATTITUDE
Domain: REKB $\times$ WFF LABEL $\times$ $\{$*mandatory, optional*$\}$
Co-Domain: REKB
Effect: Add to the (label of the) formula an indication of its optative nature in the symbol table.
Throws: Raise exception if the label already has an opposite attitude asserted.

For example, we could introduce G_{GenRev} in Example II.1 by executing:
DECLARE_ATOMIC(``The system shall generate revenue.", goal, G_{GenRev});
ASSERT_ATTITUDE(G_{GenRev}, mandatory);

C. UNTELL operations

As part of evolution, clearly one needs inverse operators for DECLARE_ATOMIC and ASSERT_ATTITUDE, to be called UNDECLARE and RETRACT_ATTITUDE, in order to revise the symbol table appropriately. In the case of assertions, we only allow retracting formulas that have been previously explicitly asserted (and hence labelled), using operation RETRACT_FORMULA(LABEL).

D. ASK operations

In any practical situation one would want to retrieve information stored about atoms and formulas in the symbol table REKB.ST. These operations are too numerous and simple to warrant description.

[2]Exceptions leave the REKB unchanged, and provide a cleaner way to specify special error cases than **return** values. We omit trivial exceptions henceforth.

[3]This could be an exception if we want a particularly simple and efficient subcase where the entire set of actions is a (non-minimal) solution.

The next two operations are meant to support standard requirements problem solution. First, we may want to know whether a goal can be achieved using some set of tasks based on the domain knowledge/refinements accumulated so far in the REKB. We use $\wp$(V) to represent the set of subsets of V.

Operation 4 — ARE_GOALS_ACHIEVED_FROM
Domain: REKB $\times$ assumeT $:\wp($GOAL $\cup$ TASK$)$ $\times$
concludeG $:\wp($GOAL$)$
Co-Domain: Boolean
Effect: returns true iff REKB.TH $\cup$ $assumeT \models \bigwedge concludeG$
Throws: throws exceptions if assumeT or concludeG are inconsistent in the sense that REKB.TH $\cup$ assumeT or REKB.TH $\cup$ concludeG entail $\perp$.

This operation supports a fine-grained exploration of what can be achieved using specific actions or from certain subgoals. By itself, it does not solve the requirements problem, but on the other hand it has a much lower computational complexity for our DA language.

Example III.1. *Using example II.2 from before, a sample call for this operator might be ARE_GOALS_ACHIEVED_FROM($\{T_{Restrict}, T_{Secure}\}, G_{GenRev}$), to which the answer is TRUE.* ■

Operation 5 — MINIMAL_GOAL_ACHIEVEMENT
Domain: REKB $\times$ concludeG $:\wp($GOAL$)$
Co-Domain: $\wp(\wp($TASK$))$
Effect: returns a set that contains all sets S of tasks such that REKB.TH $\cup S \models \bigwedge concludeG$, no subset of S has this property, and REKB.TH $\cup S$ is consistent.
Throws: throws exception if concludeG is inconsistent with REKB.TH.

These answers are essentially *abductive explanations* of how the goals can be achieved from the tasks[12]. In the case when concludeG contains all mandatory goals, this provides "candidate solutions" to the requirements problem according to the terminology in [2].

Example III.2. *A call to this operator might be MINIMAL_GOAL_ACHIEVEMENT($\{G_{GenRev}\}$), to which the answer is $\{\{T_{Target}, T_{Acct}, T_{GoogAd}\}, \{T_{Restrict}, T_{Secure}\}\}$. Which set to choose to implement is not determined by the operator. For example, we may choose the set with fewer members, or we may use a cost function to determine the less costly set.* ■

The above operations could have variants such as defaulting to the set of all tasks if assumeT is omitted, or always including mandatory goals in concludeG. We could also define additional operations such as ones to maximally satisfy "attractive" requirements, or optimizing the solution sets using objective functions such as cost or implementation effort. Space does not permit us to treat these in sufficient depth.

E. ASK operations for evolution

The following generalized operator is intended to help find solutions to a subset of problems in incremental evolution. The distance function DIST_FN could be one of the properties Π described in Section II-B.

Operation 6 — GET_MIN_CHANGE_TASK_ENTAILING
Domain: REKB $\times$ goalsG :$\wp$(GOAL) $\times$
originalSoln :$\wp$(TASK) $\times$ DIST_FN
Co-Domain: $\wp(\wp(\text{TASK}))$
Effect: Return the set of task sets S which solve REKB.TH, S $\models$ goalsG, and which minimize DIST_FN($S, originalSoln$).

Example III.3. *Assume our music service implemented $\{T_{Restrict}, T_{Secure}\}$ from Example III.2. Some of the users of the music service have asked for the ability to have third-party applications (such as iTunes) connect to an API (G_{API}). To do so we must continue to exclude non-subscribers (G_{Sub}), limit the rate of all API calls (T_{Rate}) to minimize server load, and add a field to the database to add API access permission (T_{DB}).*

We update our REKB *with this new information: declare atoms $G_{API}, T_{DB}, T_{Rate}$; assert wffs representing the refinement of G_{API} by the two tasks, and G_{Sub} by G_{API}. Figure 2 reflects the new* REKB. *A call for this operator might be GET_MIN_CHANGE_TASK_ENTAILING*($\{\{G_{GenRev}\}\}$, $\{\{T_{Restrict}, T_{Secure}\}\}$, minChange)*, to which the answer is $\{T_{Restrict}, T_{Secure}, T_{DB}, T_{Rate}\}$, since this solution involves implementing just two new tasks, versus three for the solution based on G_{Ads}. Other distance functions might return different results, of course.* ■

All the alternatives discussed in Section II-B can obviously be covered by appropriate set-theoretic distance functions.

IV. Algorithms and Complexity

We mention briefly the most straightforward implementations (as upper bounds) and known complexity results (as lower bounds) for implementing these operations.

The TELL and UNTELL operations are just book-keeping ones, while ASSERT_FORMULA checks whether the addition of its argument results in REKB.TH being inconsistent. Although in general this is an NP-complete problem, it is well-known [13] that for Horn clauses this can be done in time linear in the size of REKB.TH.

The ARE_GOALS_ACHIEVED_FROM operation could again be implemented as propositional theorem proving, but using a proof by refutation, the ability to achieve a single goal can be found by testing the consistency of a Horn theory, which once again can be performed in linear time.

The MINIMAL_GOAL_ACHIEVEMENT operator is no longer a propositional entailment question, since we are looking for *minimal sets of tasks* that entail a goal. This is the *abduction* problem. It is is much more difficult because there may be exponentially many solutions, and even if there is only one solution, it is known to be NP-complete to find it even for Horn clauses (see [14] for a good summary of propositional abduction complexity). While our experiments (below, in §VI) show that this worst-case limit is rare, it is possible to construct pathological examples which render the reasoning intractable. More empirical experience will provide further insight into the viability of the approach.

The GET_MIN_CHANGE_TASK_ENTAILING operation can be implemented by finding all ordinary solutions (using MINIMAL_GOAL_ACHIEVEMENT), and then filtering them through the condition DIST_FN(_,X_0). Its cost is therefore the same order of magnitude as MINIMAL_GOAL_ACHIEVEMENT since the tests are set-theoretic operations which take polynomial time.

V. Implementation

We are interested in obtaining **incremental** implementations of the core problematic function, MINIMAL_GOAL_ACHIEVEMENT(concludeG). We want an incremental implementation so that the complexity of the calculations is not repeated more than is necessary. In the evolving requirements problem, we anticipate opportunities for re-use of previous calculations, particularly if the change was limited.

Our representation of REKB as sets of nodes with Horn clauses linking them to form a graph is exactly the structure underlying a *truth maintenance system (TMS)* [15], [16]. In particular, we chose the A(ssumption-based)TMS of DeKleer [16]. If an explanation is a minimal set of tasks that refine a given goal, the advantage of ATMS is that it a) provides (all) explanations for why a particular goal is believed; b) ATMS inherently works with minimal explanations for a given goal; c) ATMS automatically eliminate explanations containing inconsistencies, and d) ATMS can incrementally compute new explanations for newly added atoms or wffs. By way of contrast, the CAKE reasoner that formed part of the Requirements Apprentice [17] was based on the TMS of [15]; while supporting incremental reasoning, CAKE does not handle minimal explanations.

In an ATMS each node has associated a set of possible *explanations*, in which that node is `:IN` (interpreted as derivable). Explanations are the sets of assumptions which ultimately justify that node (i.e., from which that node can be derived from assumptions via definite Horn-rules called justifications.) The *label* for a given node N will take one of three values: if there is no justification for N, the label is said to be empty; if the node is always `:IN`, i.e., it is an assumption (in our case, a task), then the label has an empty explanation; finally, in all other cases, the node is labelled with explanations: all sets of assumptions for which it can be derived `:IN`. Most importantly, these sets are minimal – no nodes can be removed from such an explanation without

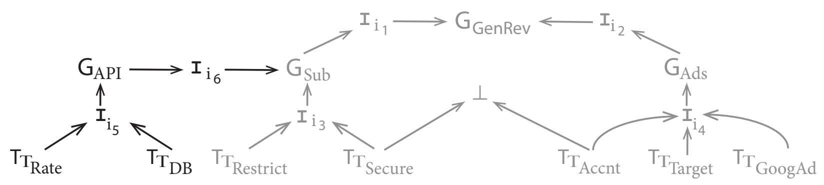

Figure 2. The revised REKB from Figure 1 showing added elements.

losing the full justifications, and the sets are consistent in the sense that no contradictions ($\perp$) can be derived from them. We encode atoms in the REKB as ATMS nodes; atoms with sort TASK become assumptions, and refinements or contradictions become justifications and contradictions, respectively, with a special CONTRADICTION node added as necessary.

The ATMS provides low-level operations, for which we leveraged existing code from [18]. However, alone it cannot implement all operations we described in §III. We extended the ATMS module with a module which implements the operations we desire which are not supported in the ATMS itself, such as consistency checks, retraction and distance metrics between sets.

Given a call MINIMAL_GOAL_ACHIEVEMENT(concludeG), if we created a super-node representing the conjunction of the goals in concludeG, an ATMS would allow us to simply read off the answers required. Short of that, we have to perform operations outside the ATMS which combine the explanations of the desired goals, and test them for consistency.

One problem with ATMS is that they do not normally support retraction. Therefore, when a leaf node (task) t is retracted, the program using the ATMS must filter out explanations that contain t. We simulate this by storing a Techne rule such as $\lambda : a \wedge b \rightarrow c$ as $\lambda \wedge a \wedge b \rightarrow c$, i.e., by including its label as a conjunct. Then, once again, when answering questions based on the solution provided by the ATMS, we must filter the explanations being returned to remove all those that contain the label of retracted rules (since they can no longer be derived). This could cause performance problems since the ATMS calculations seems to grow exponentially with the number of assumptions, and such labels act as assumptions. However, they are not assumptions in the full sense: there is no need to minimize their occurrence or eliminate duplicate labels in explanations; also, such labels do not participate in inconsistencies. If we therefore modify the ATMS implementation to split explanations into two sets: one with tasks and one with rule labels, the operations on the second sets (actually bags) are much cheaper.

Finally, we implemented the distance measures described in §II-B in order to support GET_MIN_CHANGE_TASK_ENTAILING.

A. Alternative implementations to consider

An ATMS is not the only implementation we could use for the REKB.

Two additional approaches suggest themselves. One is to implement GET_MIN_CHANGE_TASK_ENTAILING using pseudo-Boolean minimization [19], which allows weights to be assigned to boolean formulas, and then looks for satisfying assignments that are maximal/minimal. This requires translating the problem into full propositional logic (capturing that the *only* way a goal can be true is if one of the rules makes it true), and then allows us to request minimal-cost assignments, corresponding to abduced sets, by weighing task atoms.

Unfortunately, the complexity of this problem may be even higher, and practical success depends on the precise form of the formulas encountered.

A different, approximate approach to solutions would to leverage the recent progress on SATisfiability testing (e.g., [20]), and try to find minimum-cost SAT assignments by repeated calls to SAT with random starts. Some SATisfiability solvers provide mechanisms for incremental reasoning, but stumble badly in the face of inconsistency, nor do they give explanations for a particular set of goals.

Finally, one could forego guarantees of global optimums in favour of speed by using search heuristics, as we reported in [21]. In that paper we used a Tabu search approach to identify locally optimal solutions, but without reference to incremental changes between solutions.

VI. Evaluation

We evaluated the ATMS implementation on large, randomly generated requirements models. We examine two empirical questions. The first concerns the size of model this approach can tackle. The second concerns the benefits to using an incremental algorithm for updates, to support requirements model revision.

We first define what we consider to be reasonable performance. Achieving consensus on this question is a perennial problem in requirements research. Models mean different things to different people, and there is a tension between scaling to models with thousands of elements that are used in some industrial settings, such as automotive product lines, and models that are used in academic settings.

Our position is that a formal reasoning tool must handle *early* requirements models with an upper size ranging in the hundreds of nodes. One reason for this limit is that comprehension of such large models is very difficult. For example, in the industrial case study described in [22], the models ranged in size from 100-200 nodes. Reports from the study indicated that even at these sizes, models were very difficult to work with. Industrial experience with the KAOS methodology reports model sizes that were, on average, 540 goals and requirements [23]. Early requirements models are complex, and more decision- and analysis-oriented than specification models. Working on models beyond this range requires modularization and separation of concerns, as argued in [24]. It would be useful to define some sample models and model metrics for comparison purposes. Models should be characterized in terms of overall size, branching complexity (e.g. out-degree), number of alternatives, number of inputs, and so on.

The source code for our experiments is available at github.com/neilernst/Techne-TMS. The experiments were run using (primarily) Clozure Version 1.5-r13651 on a Macbook Pro 2.4 Ghz, with 4 GiB of RAM. There are several opportunities for optimizing the code which we have not yet undertaken. We created our models using a growing network attachment model [25], where each new node is added to an existing node with a certain probability based on the number of existing attachments to that node. This produced a digraph where each node had out-degree of one. Since this seems unrealistic in a requirements model (where lower-level requirements might refine multiple higher-level requirements), we randomly added new edges between the nodes. Finally, we classified some of the edges in the tree as either alternatives or contradictions.

Nodes	Tasks	Contrad.	Connectivity	Load Time
50	30	10	5.00%	0.05s
100	66	20	2.70%	0.07s
150	100	30	1.70%	0.19s
200	131	40	1.20%	0.46s
250	162	10	0.95%	0.32s
250	166	50	0.95%	0.33s
300	200	12	0.79%	4.02s
300	195	60	0.79%	2.59s
400	265	80	0.59%	8.23s
500	335	20	0.46%	63.4s
600	398	12	0.33%	110.0s

Table I
LABEL CALCULATION TIMES FOR REQUIREMENTS MODELS USING ATMS. CONNECTIVITY MEASURES THE MEAN PERCENTAGE OF THE MODEL TO WHICH A NODE IS IMMEDIATELY CONNECTED.

Table I shows our experimental results on random model permutations. There are three constraints on reasoning time. The first is obviously the overall size of the model, in terms of the number of nodes. The second is the number of assumptions, or tasks, which are used. The final constraint is the number of connections between nodes. Our results reflect this, with excellent performance for models with a few hundred nodes, and declining performance as more nodes are added. There does not seem to be any meaningful relationship between number of contradictions and evaluation time. Instead, the constraint is the number of nodes and their connectivity.

We expect our reasoner to take a lengthy period of time to initialize the model. This is a deliberate trade-off with the benefit of being able to quickly calculate incremental changes to the model. Consider the scenario presented in §II-B, regarding updates to an existing requirements problem (RP1). We show that a) adding new information to RP1 is handled quickly; b) generating new solutions for RP2 and comparing them to the old solutions from RP1 is quick. Our tool therefore supports some degree of model exploration and scenario analysis.

We evaluated our tool on three scenarios. We start with a 400-node model, and then apply the changes described in the scenarios. Our scenarios are:

- **high-level**: new mandatory goals and refinements are added. This example has 4 new mandatory goals with 8 refinements;
- **new-task**: new tasks are available. Consists of 10 new operationalizations;
- **conflict**: stakeholders identify more contradictions. This example contains 15 new contradictions.

We then evaluated the performance of two operations with these scenarios. The first operation measures how long it takes to load the new model, either starting again or incrementally; the second concerns how long it takes to answer MINIMAL_GOAL_ACHIEVEMENT for some set of mandatory goals. Here, stakeholders might be asking whether the new updates can still produce a viable solution.

Table II shows our results. Our numbers suggest that the incremental algorithm constitutes a clear improvement on starting from scratch. For example, looking for alternative solutions can be done nearly instantly, allowing stakeholders to use our tool as a workbench for solution identification. While the naive algorithm (adding new changes to the REKB and re-calculating the labels) is not terribly slow, there is a large relative difference we expect to see in larger models as well.

The timing results for finding minimal new changes are also nearly-instant, allowing the REKB to support interactive decision-making.

VII. RELATED WORK

Evolving requirements models in order to handle unanticipated changes can be considered requirements management. The typical approach to manage evolution is to treat it as the addition of new requirements, and re-calculate any evaluation algorithms from scratch, rather than incrementally, as we do. A common approach is impact analysis, which can be used to workbench different scenarios: this is used in the AGORA tool [26].

Scenario	**Naive add (s)**	**Incremental add (s)**	**Min_Goal_Achieve (s)**
high-level	1.89	0.070	0.029
new-task	2.49	0.620	0.130
conflict	1.91	0.048	0.023

Table II
INCREMENTAL OPERATIONS ON A LARGE REQUIREMENTS MODEL (N=400). **NAIVE ADD** IS THE TIME TAKEN TO EVALUATE THE SCENARIO PLUS THE ORIGINAL MODEL; **INCREMENTAL ADD** USES INCREMENTAL SUPPORT IN THE ATMS; **MIN_GOAL_ACHIEVE** IS THE TIME IT TAKES TO IDENTIFY THE MINIMAL TASKS TO SATISFY MANDATORY GOALS.

Traceability refers to the ability to describe and follow the life of a requirement [27]. The approach to the evolution of requirements described in this paper distinguishes itself from the research on requirements traceability by its focus on the evolution of the requirements problem and its solutions. Applications of operators that change the REKB can be recorded, and these recordings act as traces of changes to the requirements problem and its solutions.

Working with existing requirements has been studied under the domain of software product lines and feature modeling. Evolving a single set of requirements over time is somewhat analogous to maintaining several sets of requirements for different products. There has been a number of papers looking at the problem of automated reasoning with feature models [28]. Most of this work is consistency checking, that is, determining whether a given configuration is satisfiable, and not at enumerating all solutions, like we do here. Tun et al. [29] use problem frames to incrementally model sets of features for a product line. Temporal logic is used to minimize feature interaction. The chief difference between product line approaches and our approach is our insistence on minimizing the distance between subsequent solutions, permitting incremental reasoning over existing configurations.

Previous work on requirements models and abduction has focused on using abduction for diagnosis (a related problem), in order to derive repairs to inconsistent models (such as [30]). Our approach leverages the ability of an abductive framework to manage incremental updates to the models, which are themselves consistent. Thus, we are checking the implication relation $D, T \vdash G$ and not just the internal consistency of the requirements. Zowghi and Offen [31] deal with evolving requirements using a revision operator to maintain a consistent set of requirements. Revisions are chosen using several postulates, one of which, like us, prefers minimal changes, and another, epistemic entrenchment, which partially orders formulas in the REKB. Our knowledge-level representation captures several aspects of this while remaining agnostic, as they do, about specific mechanisms for partially ordering formulas. Finally, we implement and evaluate our approach using a large-scale example.

VIII. CONCLUSIONS

The paper first recapitulated the specification of the goal-oriented approach to stating requirements, and for judging solutions to the requirements problem, introduced in [2]. The paper then concentrated on solving the *requirements evolution problem*, when one is given an original requirements problem *and* a particular (presumably already implemented) solution, as well as a new, modified version of the requirements problem. This requirements evolution problem focuses on reusing/modifying the previous solution. Thus one novelty was allowing the conceptual possibility that the solution to the new problem was no longer an (optimal) solution in the original sense of [2]. Second, we provided a list of intuitively reasonable potential metrics for preferring changed solutions, showing that in fact the problem is not likely to have a domain-independent solution. We therefore believe that exploring the space of possible solutions to the RE evolution problem is best done by humans with tool support. As a result, we provided the precise formal specification of a rather minimal (but functionally complete) REKB to be used for this purpose. Given this specification, the REKB we presented admits various implementations. We then described how it was implemented on top of a reason maintenance system (ATMS) [16], which supports desirable features such as *incrementality*, *removal of inconsistent sets of actions* and especially *finding minimal support for implementing goals*. Since ATMS does not handle removal of formulas, we described a technique for simulating this, which need not impose exorbitant cost. Finally, we carried out experiments, which showed that this implementation does indeed find incremental solutions more efficiently.

IX. ACKNOWLEDGEMENTS

We are very grateful to John Mylopoulos for suggesting and clarifying at crucial points the fundamental notion of requirements evolution explored in with this paper.

REFERENCES

[1] S. Ferreira, D. Shunk, J. Collofello, G. Mackulak, and A. Dueck, “Reducing the risk of requirements volatility: findings from an empirical survey,” *Journal of Software Maintenance and Evolution: Research and Practice*, pp. n/a–n/a, Oct. 2010.

[2] I. J. Jureta, A. Borgida, N. A. Ernst, and J. Mylopoulos, "Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling," in *International Conference on Requirements Engineering*, Sydney, Australia, Sep. 2010.

[3] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy goals for requirements-driven adaptation," in *International Conference of Requirements Engineering*, Sep 2010.

[4] I. Jureta, A. Borgida, and N. A. Ernst, "Mixed-variable requirements roadmaps and their role in the requirements engineering of adaptive systems," Tech. Rep. arXiv:1102.4178, Feb 2011. [Online]. Available: http://arxiv.org/abs/1102.4178

[5] H. J. Levesque, "A formal treatment of incomplete knowledge bases," Ph.D., University of Toronto, 1981.

[6] J. Doyle, "The ins and outs of reason maintenance," in *International Joint Conference on Artificial Intelligence*, Karlsruhe, 1983, pp. 349–351.

[7] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 1–30, 1997.

[8] L. Chung, J. Mylopoulos, and B. A. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," *Trans. Soft. Eng.*, vol. 18, pp. 483–497, 1992.

[9] E. Letier and A. van Lamsweerde, "Reasoning about partial goal satisfaction for requirements and design engineering," in *International Conference on Foundations of Software Engineering*. Newport Beach, CA: ACM Press, 2004, pp. 53–62.

[10] I. J. Jureta, J. Mylopoulos, and S. Faulkner, "Revisiting the Core Ontology and Problem in Requirements Engineering," in *International Conference on Requirements Engineering*, Barcelona, Sep. 2008, pp. 71–80.

[11] H. Levesque, "Foundations of a functional approach to knowledge representation," *Artificial Intelligence*, vol. 23, no. 2, pp. 155–212, Jul. 1984.

[12] H. J. Levesque, "A knowledge-level account of abduction," in *International Joint Conference on Artificial Intelligence*, 1989, pp. 1061–1067.

[13] W. Dowling and J. Gallier, "Linear-time algorithms for testing the satisfiability of propositional horn formulae," *Journal of Logic Programming*, vol. 1, no. 3, pp. 267–284, 1984.

[14] T. Eiter and G. Gottlob, "The complexity of logic-based abduction," *Journal of the ACM*, vol. 42, no. 1, pp. 3–42, Jan. 1995.

[15] J. Doyle, "A truth maintenance system," *Artificial Intelligence*, vol. 12, no. 3, pp. 231–272, Nov. 1979.

[16] J. de Kleer, "An assumption-based TMS," *Artificial Intelligence*, vol. 28, no. 2, pp. 127–162, Mar. 1986.

[17] H. Reubenstein and R. Waters, "The requirements apprentice: automated assistance for requirements acquisition," *Trans. Soft. Eng.*, vol. 17, no. 3, pp. 226–240, Mar 1991.

[18] K. D. Forbus and J. de Kleer, *Building problem solvers*. Cambridge, MA: MIT Press, 1993.

[19] E. Boros and P. L. Hammer, "Pseudo-boolean optimization," *Discrete Applied Mathematics*, vol. 123, no. 1-3, pp. 155–225, 2002.

[20] E. Di Rosa, E. Giunchiglia, and M. Maratea, "Solving satisfiability problems with preferences," *Constraints*, vol. 15, no. 4, pp. 485–515, Jul. 2010.

[21] N. A. Ernst, J. Mylopoulos, A. Borgida, and I. J. Jureta, "Reasoning with Optional and Preferred Requirements," in *International Conference on Conceptual Modelling*, Vancouver, Nov. 2010, pp. 118–131.

[22] S. M. Easterbrook, E. S. Yu, J. Aranda, Y. Fan, J. Horkoff, M. Leica, and R. A. Qadir, "Do viewpoints lead to better conceptual models? An exploratory case study," in *International Conference on Requirements Engineering*, 2005, pp. 199–208.

[23] A. van Lamsweerde, "Goal-oriented requirements enginering: a roundtrip from research to practice," in *International Conference on Requirements Engineering*, 2004, pp. 4–7.

[24] A. Fuxman, L. Liu, J. Mylopoulos, M. Roveri, and P. Traverso, "Specifying and analyzing early requirements in Tropos," *Requirements Engineering J.*, vol. 9, pp. 132–150, 2004.

[25] P. Krapivsky and S. Redner, "Organization of growing random networks," *Physical Review E*, vol. 63, no. 6, p. 066123, May 2001.

[26] D. Tanabe, K. Uno, K. Akemine, T. Yoshikawa, H. Kaiya, and M. Saeki, "Supporting requirements change management in goal oriented analysis," in *International Conference on Requirements Engineering*, Barcelona, 2008, pp. 3–12.

[27] O. Gotel and A. Finkelstein, "An analysis of the requirements traceability problem," in *International Conference on Requirements Engineering*, Colorado Springs, 1994, pp. 94–101.

[28] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, Sep. 2010.

[29] T. T. Tun, T. Trew, M. Jackson, R. Laney, and B. Nuseibeh, "Specifying features of an evolving software system," *Software: Practice and Experience*, vol. 39, no. 11, pp. 973–1002, 2009.

[30] T. Menzies, S. M. Easterbrook, B. Nuseibeh, and S. Waugh, "An empirical investigation of multiple viewpoint reasoning in requirements engineering," in *International Conference on Requirements Engineering*, Limerick, Ireland, Jun. 1999, pp. 100–109.

[31] D. Zowghi and R. Offen, "A logical framework for modeling and reasoning about the evolution of requirements," in *International Conference on Requirements Engineering*, 1997, pp. 247–257.