Synthese (2021) 198 (Suppl 23):S5719–S5744
https://doi.org/10.1007/s11229-019-02305-7

S.I.: RELIABILITY

A theory of change for prioritised resilient and evolvable software systems

Giuseppe Primiero[1] · **Franco Raimondi**[2,3] · **Taolue Chen**[4]

Received: 11 November 2018 / Accepted: 25 June 2019 / Published online: 28 June 2019

Abstract

The process of completing, correcting and prioritising specifications is an essential but very complex task for the maintenance and improvement of software systems. The preservation of functionalities and the ability to accommodate changes are main objectives of the software development cycle to guarantee system reliability. Logical theories able to fully model such processes are still insufficient. In this paper we propose a full formalisation of such operations on software systems inspired by the Alchourrón–Gärdenfors–Makinson (AGM) paradigm for belief revision of human epistemic states. We represent specifications as finite sets of formulas equipped with a priority relation that models functional entrenchment of properties. We propose to handle specification incompleteness through ordered expansion, inconsistency through ordered safe contraction and prioritisation through revision with reordering, and model all three in an algorithmic fashion. We show how the system satisfies basic properties of the AGM paradigm, including Levi's and Harper's identities. We offer a concrete example and complexity results for the inference and model checking problems on revision. We conclude by describing resilience and evolvability of software systems based on such revision operators.

Keywords Software evolution · Software reliability · Software resilience · Software evolvability

✉ Giuseppe Primiero
giuseppe.primiero@unimi.it

Franco Raimondi
f.raimondi@mdx.ac.uk

Taolue Chen
taolue@dcs.bbk.ac.uk

1 Department of Philosophy, University of Milan, Via Festa del Perdono 7, 20122 Milan, Italy

2 Department of Computer Science, Middlesex University London, The Borroughs, London NW4 4BT, UK

3 Dipartimento di Matematica e Fisica, Universitá degli Studi della Campania, Salerno, Italy

4 Department of Computer Science and Information Systems, Birkbeck, University of London, Malet Street, London WC1E 7HX, UK

1 Introduction

The process of designing software starts usually from a list of requirements, intended as properties expressing desires of a stakeholder concerning the software to be developed. Given a certain domain knowledge, the requirements are meant to be implemented and satisfied by corresponding specifications, intended as properties of the system.[1] The notion of reliability for software systems has been mainly formulated in terms of continuity of correct service and is an attribute of dependability along with availability, maintainability, safety and security. Software reliability is heavily complicated by change in the life-cycle of computing systems. The process of modifying or re-defining systems specifications is required by increasing architectural complexity of the actual implementations, or improving software quality. In either case, "software maintenance has been regarded as the most expensive phase of the software cycle".[2] A considerable amount of research has already been dedicated to the understanding, planning and execution of software evolution, in particular for requirements evolution, see e.g. Ernst et al. (2009). Typically, this occurs as part of the *late* life-cycle of the system and it is dictated by

- Architectural degeneration, i.e. the violation or deviation of the architecture, increasing with changes being made to the original, see e.g. Eick et al. (2001) and Lindvall et al. (2002);
- Flexibility requirements, i.e. the system property that defines the extent to which the system allows for unplanned modifications, see e.g. Port and Liguo (2003);
- Requirements prioritisation, i.e. the design choice which defines the relevance of corresponding functionalities, and in turn their resilience in view of future changes, see e.g. Fellows (1998) and Firesmith (2004).

In this context, the persistence of service delivery when facing changes is referred to as resilience and it is combined with correct evolvability, as the ability to successfully accommodate changes, see Laprie (2008). They can be taken as building blocks for defining and proceduralising a notion of reliability for software systems. The laws of software evolution for computational systems linked to a real environment (Lehman 1980; Lehman et al. 1997) express the importance of an appropriate understanding of software change. Change classification schemes, assessing the impact and risk associated with software evolution, present challenges (Mens et al. 2005) which include integration in the conventional software development process model. This, in turn, means that a model of software change at design and implementation stages is essential to assess and anticipate errors and to determine system's reliability in view of threats to functionalities. Late life-cycle *misfunctions*, where the system produces negative side-effects absent in other systems of the same type, require *corrective changes* after testing on the actual code (i.e. excluding model-based testing). *Disfunctions*, where the system is less reliable or effective than one expects in performing its function are more likely to be assessed at *early* stages, where *perfective changes* result from new or changed requirements, see Lientz and Swanson (1980) and Sommerville (2004).[3]

[1] For these standard meanings, see Zave and Jackson (1997).

[2] Williams and Carver (2010, p. 32).

[3] For the definitions of misfunctions and disfunctions in software systems, see Floridi et al. (2014).

A third classification is that of *adaptive changes*, where the system or its environment are evolving.[4] The understanding, modelling and development of a theory for software evolution are thus crucial tasks (Mens et al. 2005).

Let us consider a concrete example, extracted from Zowghi et al. (1996, sec. 2):

> Consider the requirements engineering process involved in developing a word-processor. The initial problem statement only specifies that this wordprocessor is intended to be used by children. Two assumptions may be made immediately that are related to the domain knowledge and usability. Firstly, since all word processors by default have a spell check functionality, we may specify a requirement for existence of a spelling check function. Secondly, since it is intended for use by children, we may add a set of requirements for the ability to change the colour of screen and text etc. These assumptions are added to the initial statement to represent our current state of belief about the software we are to develop and are then presented to the problem owners for validation. They, in turn, confirm that spell check is indeed a requirement but since they will only have monochrome terminals available there is no need for colour change. So we need to revise our set of beliefs to contract those requirements related to colour.

Besides requirements evolution, such a revision may also be induced at a lower level of abstraction. Consider the formulation of a specification satisfying given requirements, and an implementation thereof. The specification can be seen as a model of the physical artefact. When the latter violates some of the properties expressed by the former, a revision of the examined system in one of the above mentioned ways becomes necessary. Consider the following variant of the above case:

> The initial problem statement only specifies that this wordprocessor is intended to be used by children. Two assumptions may be made immediately that are related to the domain knowledge and useability. Firstly, since all word processors by default have a spell check functionality, we may specify a requirement for existence of a spelling check function. Secondly, since it is intended for use by children, we may add a set of requirements for the ability to change the colour of screen and text etc. These assumptions are added to the initial statement to represent our current state of belief about the software we are to develop. The system is then developed accordingly, but the functionality to change the colour of the screen is not implemented. We extract the specification of the current implementation and compare it with the intended one. We notice the two are not logically equivalent, hence we wish to modify the latter to accommodate the required change. We consider this a perfective change. In the new implementation, the developers add the ability to insert graphs and figures. Again, a specification may be extracted and compared to the intended one: as the product is intended for children, the new functionality is considered superfluous and we wish to remove it. We consider this a corrective change. In the next development cycle, the functionality to spellcheck is made dependent on the ability to switch languages: for each language, an appropriate spellcheck is developed. The comparison with the intended model shows the logical difference, but it

[4] Here we explicitly ignore the other classification, namely *preventative changes*.

also indicates the dependency of a required functionality from a non-required one. The model is changed to accommodate the latter, in order not to loose the former. We consider this an adaptive change.

Despite the triviality of the above example, it seems clear that the above operations, possibly automatically performed, would be a significant aid to the process of software evolution. Even more so, if the products under considerations are no longer word-processor, but safety-critical systems, where the removal operation might induce significant effects.[5]

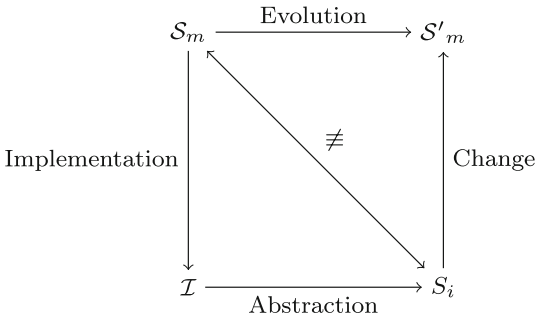
One way to account for corrective, perfective and adaptive changes on an implementation diverging from the specification is to treat them similarly to change operations in scientific theories. In particular, in this paper we define formal operations inspired by the AGM belief change theory, see Alchourrón et al. (1985). This area at the intersection of software engineering and theory change has been only very little explored: the only approach explicitly based on AGM is to be found in Zowghi et al. (1996), offering a framework to reason about requirements evolution in terms of belief change operations. In Dam and Ghose (2014), belief revision is used to deal with change propagation in model evolution. In Mu et al. (2011) and Booth's (2001) negotiation-style for belief revision is used to model change from current to to-be system requirements, aiming at some form of compromise based on prioritisation. AGM belief revision has been investigated for logic programming under answer set semantics in Delgrande et al. (2013a, b). While notoriously a number of methods in software engineering have focused on developing implementation from specifications (Spivey and Abrial 1992; Abrial 2005), our analysis concentrates on the modelling of perfective, corrective and adaptive changes to design *new* specifications from early (incorrect) implementations that are the object of change. Here the passage from model to implementation to new model is crucial.

The case under consideration can be reformulated as follows in the process of software development. We start with $\mathcal{S}_m$, intended as a specification for a software system: translated in an abstract formal model, this is considered as a logical *theory*, i.e. a set of formulas closed under logical consequence, i.e. where all the properties implied by the formulas should be considered valid. An implementation of $\mathcal{S}_m$ will be denoted by $\mathcal{I}$. This actual artefact will have its own formal model, possibly automatically extracted, and it will be denoted by S_i. This new model *cannot* be treated as a logical theory, because it will verify only a finite number of formulas. Therefore, S_i needs to be considered as a *base*. Assuming a discrepancy is found between the intended model $\mathcal{S}_m$ and S_i, some change is performed on the latter to obtain a new specification $\mathcal{S}'_m$.[6] This dynamics, which can be labelled Specification Evolution, is illustrated in Fig. 1. To model this process concretely, we start with considering $\mathcal{S}_m$ as a software theory, i.e., the deductive closure of a (finite) set of formulas, each representing a property of our system. Our aim is to define some operation that allows the construction of

[5] For another example of software uninstall operations where dependencies affect system efficiency and reliability, see Primiero and Boender (2018). A more realistic and complex example of specification revision in view of inconsistent implementation is presented below in Sect. 4.

[6] Notice that a Software Engineer might be interested only in the evolution of $\mathcal{I}$, while we explicitly address the specification evolution in terms of the logical change of the system to be reflected in the corresponding theory.

Fig. 1 Specification evolution



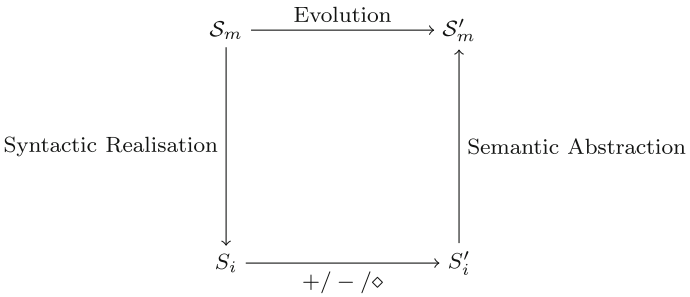


Fig. 2 System change

a new theory from the previous one by performing some perfective, corrective or adaptive change. In order to manipulate such a theory in an algorithmic fashion and define concrete operations, we deal with a *finite base* S_i representing the software system which offers a syntactic representation of the theory $\mathcal{S}_m$ and is not logically closed. In other words, we assume that the specification of any however large software and its manipulation should be accounted for in terms of a finite representation. This concrete formulation of the specification evolution is labelled System Change and it is illustrated in Fig. 2.

We consider the computational aspect of the operators introduced in this paper, i.e. the computational complexity of reasoning with the operators. In literature, there are mainly two questions assuming a finite propositional language as in this paper:

- *Inference* Given a knowledge base S_i, a new formula ϕ_i and a query ψ_j (represented as propositional formulas), decide whether ψ_j is a logical consequence of $S_i * \phi_i$, the revised knowledge base. (Here $*$ is interpreted as a revision operator.) The complexity of this problem was first studied by Eiter and Gottlob (1992).
- *Model checking* Given a knowledge base S_i, any such knowledge base can be equivalently represented by the set of its models, denoted as $\mathcal{M}(S_i)$. A model M is *supported* by the knowledge base S_i iff $M \in \mathcal{M}(S_i)$, i.e., $M \models S_i$. The model checking problem is thus to decide whether a model is supported by the revised base. Formally, given a knowledge base S_i, a new formula ϕ_i, and a model M (represented by a valuation of propositional letters), decide whether $M \in \mathcal{M}(S_i * \phi_i)$.

In AI literature, the complexity of various belief revision and update operators has been extensively studied by Liberatore and Schaerf (2001). We show that for our revision operator with reordering, the inference problem is in co-NP, whereas the model checking problem is in the second level of the polynomial hierarchy, i.e., Σ_2^{P}.

The rest of this paper is structured as follows. In Sect. 2 we introduce our formal machinery. In Sect. 3 we formulate our definitions of expansion, safe contraction and revision for system evolution and offer their algorithmic translations. In Sect. 4 we present a notorious example of a broken algorithm and its redesign through the operators introduced in this paper. In Sect. 5 we present the complexity results. In Sect. 6 we offer resilience and evolvability properties on our theory. We conclude with remarks on future research.

2 Preliminaries

The alphabet of a propositional formula is the set of all propositional atoms occurring in it. A valuation of an alphabet X is a truth assignment to all the propositional letters in X. An interpretation of a formula is the truth assignment when the valuation to the atoms of its alphabet is given. A model M of a formula ϕ_i is an interpretation that satisfies ϕ_i (written $M \models \phi_i$). Interpretations and models of propositional formulas will usually be denoted as sets of atoms (those which are mapped into true). A theory T is a logically closed set of formulas. An interpretation is a model of a theory if it is a model of every formula of the theory. Given a theory T and a formula ϕ_i we say that T entails ϕ_i, written $T \models \phi_i$, if ϕ_i is satisfied by every model of T.

We consider a software theory as the deductive closure of a finite set of formulas $\mathcal{S}_m = Cn(S_m)$ where $S_m = \{\phi_1, \ldots, \phi_n\}$, i.e. $\mathcal{S}_m := \{\phi_i \mid S_m \models \phi_i\}$, where each ϕ_i expresses a specific behaviour that the intended software system $\mathcal{S}_m$ should display. In the following we will use respectively theory and system to refer to these two distinct formal objects. The consequence relation $\vDash$ for $\mathcal{S}_m$ is classical with the following essential properties:[7]

1. $\mathcal{S}_m \vDash \top$.
2. $\mathcal{S}_m \vDash (\phi_i \rightarrow \phi_j)$ and $\mathcal{S}_m \vDash \phi_i$, implies $\mathcal{S}_m \vDash \phi_j$.
3. $\mathcal{S}_m \vDash \phi_i$ implies $\mathcal{S}_m \nvDash \neg\phi_i$.

$\vDash$ intuitively reflects property expressiveness: $\phi_i \vDash \phi_j$ says that a property specification ϕ_i holding for a system $\mathcal{S}_m$ *induces* property specification ϕ_j in the corresponding theory $\mathcal{S}_m$.

We call *functional entrenchment* an ordering $\phi_i \leq \phi_j$ which says that ϕ_i is at least as embedded as ϕ_j in view of the functionalities of the system. This means that the satisfiability of functionality ϕ_j might depend or be less essential than the satisfiability of functionality ϕ_i. Another way to present the intuitive meaning of the entrenchment relation $\phi_i \leq \phi_j$ in the context of software systems is to say that one would prefer to remove first ϕ_j than ϕ_i. Functional entrenchment is defined by two properties:

[7] We do not require the theory of interest to be neither complete (in the sense of satisfying $\forall\phi, \mathcal{S}_m \vDash \phi_i$ or $\mathcal{S}_m \vDash \neg\phi_i$) nor compact. Completeness is too-strong in view of the system being under-defined with respect to the alphabet of all possible properties; compactness is trivial in the finite setting of the system.

1. Transitivity: if $\phi_i \leq \phi_j$ and $\phi_j \leq \phi_k$, then $\phi_i \leq \phi_k$;
2. Dominance: if $\phi_i \vDash \phi_j$, then $\phi_i \leq \phi_j$;

An argument about the utility of such an entrenchment is the following: assume we have a set $\{p, q\}$ where the two formulas are unrelated and not ordered by any preference; assume that for some reason we are required to remove from that set the formula $(p \vee q)$ and its consequences; in this situation we would be forced to obtain the empty set as a result of the contraction operation, as either formula in the current set implies $(p \vee q)$; on the other hand, obviously, it would be enough to remove either one of the two formulas p or q, if there was some priority order defined over them to allow us choosing.

We now refer to the system $S_i = \{\phi_1, \ldots, \phi_n\}$ as a knowledge base, i.e. a set of formulas *not* closed under logical consequence. Recall that this is due to the need of representing a model of a physical implementation $\mathcal{I}$ in some concrete programming language of the corresponding theory $\mathcal{S}_m$. We say that S_i is consistent if there exists a model for S_i. Let now $(S_i, <)$ denote a finite set of formulas with a partial order. When referring to the model abstracted from S_i we shall use $\mathcal{S}_i$ in order to denote $Cn(S_i, <)$. If S_i is not a faithful translation of S_m (and hence of its theory $\mathcal{S}_m$), in the sense of either not satisfying some property included in S_m, or satisfying some contradictory property or reflecting an undesired functional entrenchment, we then wish to perform changes.

For our complexity results we assume familiarity with basic concepts of computational complexity, and we use standard notations of complexity classes. In particular, the class P denotes the set of problems whose solution can be found in polynomial time by a deterministic Turing machine, while NP denotes the class of problems that can be solved in polynomial time by a nondeterministic Turing machine. The class co-NP denotes the set of decision problems whose complement is in NP. We also use higher complexity classes defined using oracles. In particular P^A (NP^A) corresponds to the class of decision problems that are solved in polynomial time by deterministic (nondeterministic) Turing machines using an oracle for A in polynomial time.

3 Change operations

A change operation is triggered by some ϕ_i satisfying one of the following conditions:

- *Incompleteness handling* $S_m \vDash \phi_i$ and $S_i \nvDash \phi_i$, i.e. the implementation does not satisfy one of the intended functionalities;
- *Inconsistency handling* $S_m \nvDash \phi_i$ or $S_m \vDash \neg\phi_i$ and $S_i \vDash \phi_i$, i.e. the implementation satisfies a functionality not intended by the model, or one whose negation was intended by the model;
- *Priority handling* $S_m \vDash \{\phi_i \leq \phi_j\}$ and $S_i \vDash \{\phi_j \leq \phi_i\}$, i.e. the implementation satisfies a different functional entrenchment than the one intended by the model.

Each of these three cases expresses a form of inconsistency between bases. Automatic techniques for inconsistency checking of systems are available both through theorem proving and model checking, also within the AGM paradigm, and widely reported in the literature (Kolyang and Wolff 1996; Buccafurri et al. 1999; Sousa and Wassermann

2007; Zhang and Ding 2008; Guerra and Wassermann 2010). Formal operations can be defined on S_i so that either the current input in the implementation becomes valid for the model; or the specification that makes our current input invalid is removed; or the order of the base is changed, in combination with the other operations.

- In the first case, S_i is changed to include ϕ_i: we indicate the result of this change as *expansion*. This formal operation reflects the implementation of a new functionality and hence qualifies as a *perfective change*.
- In the second case, S_i is changed to remove ϕ_i (under a complete system, which we do *not* assume, this implies inclusion of $\neg\phi_i$ in S_i): we indicate the result of this revision as *contraction*. A contraction operation should aim at removing the least expressive properties to induce a minimal loss of functionalities; at each stage of the implementation consistency is preserved. This formal operation reflects the removal of an undesired functionality (error fixing) and hence qualifies as a *corrective change*. Such change operation in software design and system evolution should be defined in view of resilience, intended as the maximal preservation of functionalities not related to the removed property.
- We call instead *adaptive change* the category of modifications that result from a required novel priority ordering in the system so as to make one property safer from future corrective changes. This can be defined as a combination of corrective and re-ordered perfective changes and it is related to system evolution.

In all cases, a new model $\mathcal{S}'_m$ is obtained, from which a new implementation can be formulated.

We formalize perfective, corrective and adaptive changes respectively in terms of expansion, safe contraction and revision with a reordering operation: these operations are formally defined from the next subsection, and they modify existing well-known operation from the AGM paradigms to be adapted to prioritised bases. Expansion $+$ is rather easy to associate to functionality extension. Safe contraction $-$ has had only little attention in the large literature in epistemic logic using the AGM paradigm, but it appears essential to the issue of property resilience in system evolution and it also satisfies the criteria of a minimal contraction operator, hence ensuring that as little as possible is lost. Finally, we consider an operator $\diamond$ which is defined to satisfy as much as possible the standard AGM revision $*$ postulates and additionally makes the property object of the operation safer from future contraction operations. This recalls operations of preference change, e.g. in Alechina et al. (2015) where preferences are treated as a special kind of theory, and minimal change contraction and revision operations are defined. Safe contraction defined over bases exists already from the literature (Fuhrmann 1991), where it is called minimal contraction and our postulates for ordered safe contraction match those offered there. An efficient form of AGM belief contraction (linear time) satisfying all but one of the AGM postulates is defined in Alechina et al. (2005) for a realistic rule-based agent which can be seen as a reasoner in a very weak logic (but still over deductively closed sets of beliefs). More importantly, we use such operations to define property and system resilience and offer a definition of system evolution.

In view of our application, some of the properties of the AGM paradigm need to be re-designed, in virtue of the fact that revision operations are not defined on theories

but rather on finite bases. This has the advantage of being computationally far more appealing. Moreover, a relevant addition in our model is the use of an entrenchment relation to define a functional priority relation over the properties of the system and to dictate both removal and reordering. Notice that functionality prioritisation and its mapping to sub-characteristics (induced in our model by a relation between the inference relation and the ordering) is proposed and implemented also in structured methods for architectural evolvability in industrial setting, see e.g. Breivold et al. (2008).

3.1 Ordered expansion

The process of designing a piece of software can be seen as moving from an empty set of functionalities (the trivial system specification, i.e. one that implements no operations) to one that includes *some* property specifications. This process is akin to an *expansion* of the software model abstracted from the trivial implementation $S_i = \emptyset$ with respect to a new functionality ϕ_i. This is denoted by $(S_i)^+_{\phi_i}$. In general, the expansion operation is intended in the following as adding a new functionality that has the least entrenchment with respect to the existing ones. This can be informally justified by considering the new property as the weakest one, in view of the fact that its effect on the system is still unknown. This general rule is obliterated only in the case when the expansion formula validates a formula already in the base, in which case the functional entrenchment requires to insert it directly before that in the order. This means we need to compute such set in order to position our expansion formula.

Definition 1 (*Ordered expansion*) We first define

$$\Xi = S_i \cap Cn(\phi_i)$$

where $Cn(\phi_i) = \{\psi \mid \phi_i \models \psi\}$. Our ordered expansion is denoted as

$$\begin{aligned} S'_m = (S_i)^+_{\phi_i} &:= (S_i \cup \{\phi_i\}, <') \text{ where} \\ <' &= < \cup \{(\phi_k, \phi_i), (\phi_i, \phi_j) \mid \phi_j \in \Xi, \phi_k \in S_i \backslash \Xi\} \end{aligned}$$

In Fig. 3 we provide an explicit algorithm (in pseudocode) to compute the result of the safe ordered expansion operator.

Example 1

$$\{p < q\}^+_{(q \vee r)} = (p < q < (q \vee r)) \tag{1}$$

Example 2

$$\{p < (q \vee r)\}^+_q = (p < q < (q \vee r)) \tag{2}$$

Let us explain the Example 2 step by step: given a base with functionalities expressed by formulas p and $q \vee r$, where the latter is less entrenched than the former (e.g. because

```
1  PROCEDURE Expansion(S_i, <, φ_i)
2
3  Ξ := {};
4  FOR each φ_j ∈ S_i
5    IF φ_i ⊨ φ_j
6       THEN Ξ := Ξ ∪ {φ_j};
7    ENDIF
8  ENDFOR
9
10 RETURN(S_i ∪ {φ_i}, < ∪{(φ_k, φ_i), (φ_i, φ_j) | φ_j ∈ Ξ, φ_k ∈ S_i \ Ξ})
11 ENDPROCEDURE
```

Fig. 3 Algorithm for ordered expansion

it depends from it), we wish to add a functionality expressed by formula q: then we simply add it to our specification, but its positioning in the functional entrenchment requires it comes between p and $q \vee r$, because $q \vDash q \vee r$.

Software system creation has then a starting point $S_i \nvDash \phi_i$, for any ϕ_i. Any expansion operation after the first one should preserve consistency in S_i. Otherwise, each expansion by ϕ_i needs to be accompanied by the implicit elimination of the contradictory $\neg\phi_i$ from the list of feasible property descriptions according to S_i. Hence, each non-consistent expansion requires a minimal set of contraction operations.

3.2 Ordered safe contraction

We now consider contracting S_i in view of a system functionality ϕ_i. We denote this by $(S_i)^-_{\phi_i}$. In general, the contraction operation is intended in the following as removing the minimal number of functionalities that have the least entrenchment with respect to the existing functionalities in order to remove the property ϕ_i at hand. This can be informally justified by considering the contraction as removing the least entrenched or essential properties. In order to define a procedure for this, we require to compute both the set of properties that are implied by and that imply the contraction formula. This does not amount to compute the entire consequence set of S_i (i.e. $\mathcal{S}_i$), but rather to perform a membership check for the contraction formula, to identify whether it implies or is implied by one of the formulas in the base. Then we identify among these the minimal ones in the entrenchment. The resulting operation is also expressive about the properties that are safe with respect to the contraction and the output of our procedure is still taken to be a (contracted) base.

Definition 2 (*Ordered contraction*) We first define

$$\Xi = S_i \cap (Cn(\phi_i) \cup \overline{Cn}(\phi_i))$$

where $Cn(\phi_i) = \{\psi \in S_i \mid \phi_i \models \psi\}$ and $\overline{Cn}(\phi_i) = \{\psi \in S_i \mid \psi \models \phi_i\}$. We then have a sequence of subsets of Ξ, i.e., $(\Xi_i)_{i\geq 0}$, inductively defined as follows:

$$\begin{aligned}
&\Theta_0 = \Xi\\
&\Xi_i = \{\psi \mid \psi \text{ is a minimal element of } \Theta_i \text{ wrt. } <\}\\
&\Theta_{i+1} = \Theta_i \backslash \Xi_i
\end{aligned}$$

```
PROCEDURE SafeContraction(S_i, <, φ_i)

Ξ := {};
FOR each φ_j ∈ S_i
  IF φ_j ⊨ φ_i or φ_i ⊨ φ_j
      THEN Ξ := Ξ ∪ {φ_j};
  ENDIF
ENDFOR

i := 0; Θ := Ξ;
DO
  Ξ_i = ∅;
  FOR each ψ_k ∈ Θ
      IF ψ_k is a minimal element of Θ with respect to <
          THEN Ξ_i := Ξ_i ∪ {ψ_k};
  ENDFOR
  i := i + 1;
  Θ := Θ \ Ξ_i
WHILE Θ ≠ ∅;

k := 0; S_i' := S_i;
WHILE (S_i' ⊨ φ_i) DO
  S_i' := S_i' \ Ξ_k;
  k := k + 1;
ENDWHILE

RETURN (S_i', {(φ_k, φ_j) | φ_j < φ_k and φ_j, φ_k ∈ S_i'}).
ENDPROCEDURE
```

Fig. 4 Algorithm for ordered safe contraction

Note that as Ξ is finite, we have that there is some n such that $\Xi = \bigcup_{i=0}^{n} \Xi_i$. We then define

$$k_0 = \min \left\{ k \mid S_i \setminus \bigcup_{i=0}^{k} \Xi_i \not\models \phi_i \right\}$$

and

$$(S_i)^{-}_{\phi_i} := \left(S_i \setminus \bigcup_{i=0}^{k_0} \Xi_i, <' \right) \text{ where } <' = \left(<\restriction_{S_i \setminus \bigcup_{i=0}^{k_0} \Xi_i} \right)$$

where $\restriction$ indicates a projection function.

This definition expresses the content of the new system as obtained by a function from the current S_i to a new base whose models do not imply ϕ_i. In Fig. 4 we provide an explicit algorithm (in pseudocode) to compute the result of the safe contraction operator.

Example 3

$$\{p < q < r\}^{-}_{(q \vee r)} = \{p\} \tag{3}$$

Example 4

$$\{p < (q \vee r) < r\}_r^- = \{p\} \tag{4}$$

Example 5

$$\{p < (q \vee r) < r\}_{r \wedge p}^- = \{p < (q \vee r)\} \tag{5}$$

Let us explain the Example 5 step by step: given a base with functionalities expressed by formulas $p, q \vee r$ and r, in this functional entrenchment order, we wish to remove the combination of functionalities p and r, i.e. their logical conjunction: in this case, we are not required to remove both functionalities, but just one of them (as we do not wish to eliminate both, but their combination); then we induce a contracted base with the minimal removal required, hence preserving as much as possible the functionalities. The result removes r, which is the least entrenched functionality.

Note that by our definition of ordered safe contraction the preference is always to remove formulas that express less entrenched functionalities. This might appear counter-intuitive, for example in the case where the less entrenched functionalities is the result of a combination of several components, and therefore its loss might potentially result more impactful than the removal of a more entrenched, but less complex functionality. Note that if to express the case of composed functionalities of a given degree we use logical conjunction, our ordered safe contraction will only require the removal of a component, rather than of the whole set of functionalities. The following is a variant of Example 5 which illustrates this case:

Example 6

$$\{p < (q \vee r) < (r \wedge s)\}_{r \wedge s \wedge p}^- = \{p < q < s\} \tag{6}$$

Note that the choice of which component to preserve in this case is purely contextual and it might be dictated by applications, as it might affect other functionalities: here the choice of preserving s means we need to modify also $q \vee r$. Different is the case where a complex functionality is expressed by logical disjunction or implication:

Example 7

$$\{p < (q \vee r) < (r \vee s)\}_{(r \vee s) \wedge p}^- = \{p < q\} \tag{7}$$

Example 8

$$\{p < (q \vee r) < (r \rightarrow s)\}_{(r \rightarrow s) \wedge p}^- = \{p < (q \vee r)\} \tag{8}$$

Here the contraction operation is costly because we are forced to loose the whole complex functionality, and in the Example 7 even to weaken the additional functionality expressed by $q \vee r$. It is only with the more complex operation of revision with reordering introduced in the next subsection that this problem can be avoided.

From the definition of safe contraction and properties of the consequence relation over the contraction formula, the following can be proven about $\mathcal{S}_i$ (i.e. referring to the closure of S_i):

Lemma 1 [Alchourrón and Makinson (1985)] $(S_i \cap Cn(\neg\phi_i)) \subseteq (S_i)^-_{\phi_i}$.

Proof Suppose $\phi_j \in S_i$, $\neg\phi_i \vDash \phi_j$ and $\phi_j \notin (S_i)^-_{\phi_i}$. Then $(\neg\phi_i < \phi_j)$ and there is some minimal $S_i \backslash \Xi_i \vDash \phi_i$. Take $S_i \backslash \Xi_j = S_i \backslash \Xi_i \backslash \{\phi_j\}$ then $S_i \backslash \Xi_i = S_i \backslash \Xi_j \cup \{\phi_j\}$ and $S_i \backslash \Xi_j \cup \{\phi_j\} \vDash \phi_i$. Since by assumption $\neg\phi_i \vDash \phi_j$, then $\neg\phi_j \vDash \phi_i$ and so $S_i \backslash \Xi_j \cup \{\neg\phi_j\} \vDash \phi_i$, but this contradicts the minimality of Ξ_i. □

In the context of Software Engineering, a contraction operation should aim at removing the least expressive properties to induce a minimal loss of functionalities. We capture this formally by the functional entrenchment ordering $<$ on properties, similarly to what is done with epistemic entrenchment (Gärdenfors and Makinson 1988). Hence, in a contraction process, one starts removing from the last element in the order to preserve as much as possible the operational properties of the system. Among the different (although in some ways related, see Alchourrón and Makinson 1986) contraction functions, *safe contraction* is a natural candidate for the contraction on a finite set of property specifications under this ordering preserving system functionalities:

Definition 3 (*Safe contraction*) A property ϕ_j is safe with respect to $(S_i)^-_{\phi_i}$ if and only if $\phi_i \nvDash \phi_j$.

AGM revision is usually characterized by Gärdenfors postulates. These are modified as follows: the closure postulate is missing, provided the output of our procedure is again constrained to be a base; and the recovery postulate is missing, as the result of ordered contraction followed by ordered expansion does not necessarily returns the original order of the base. The resulting postulates match those in Fuhrmann (1991):

1. *Inclusion* $(S_i)^-_{\phi_i} \subseteq S_i$
2. *Vacuity* $(\phi_i \notin Cn(S_i)) \rightarrow ((S_i)^-_{\phi_i} = S_i)$
3. *Success* $(\phi_i \notin Cn(\emptyset)) \rightarrow \phi_i \notin Cn((S_i)^-_{\phi_i})$
4. *Extensionality* $(\phi_i \equiv \phi_j) \rightarrow (S_i)^-_{\phi_i} = (S_i)^-_{\phi_j}$

Proposition 1 *Safe contraction satisfies* (1)–(4).

Proof Similar to the one in Alchourrón and Makinson (1985), except for Success:

1. *For Inclusion* immediate from the definition of $(S_i)^-_{\phi_i}$ from S_i and Ξ.
2. *For Vacuity* if $\phi_i \notin Cn(S_i)$, there is no Ξ_i such that $\Xi_i \vDash \phi_i$, hence every $\phi_j \in S_i$ is safe in $(S_i)^-_{\phi_i}$.
3. *For Success* Assume that $(\phi_i \notin Cn(\emptyset))$ and $(S_i)^-_{\phi_i} \vDash \phi_i$; then there is $(S_i \backslash \bigcup_{i=0}^{k_0} \Xi_i, <') \vDash \phi_i$ and because Ξ_i is finite and $<$ is non-circular, there is a minimal element $\Xi_i \vDash \phi_j$ for which one of the following holds:
   - $\phi_j = \phi_i$: then because $(S_i \backslash \bigcup_{i=0}^{k_0} \Xi_i, <') \vDash \phi_j$ it must be safe in $(S)^-_{\phi_i}$; but by construction ϕ_j cannot be safe in $(S_i)^-_{\phi_i}$ because $\Xi_i \vDash \phi_i$ and $\phi_j = \phi_i$;

– $\phi_j < \phi_i$: because $(S_i \setminus \bigcup_{i=0}^{k_0} \Xi_i, <') \vDash \phi_j$ it must be safe in $(S)^-_{\phi_i}$, but by Dominance $\phi_j \vDash \phi_i$ and so ϕ_i should be safe in $(S_i)^-_{\phi_i}$, but cannot be;
– $\phi_i < \phi_j$: because $(S_i \setminus \bigcup_{i=0}^{k_0} \Xi_i, <') \vDash \phi_j$ it must be safe in $(S)^-_{\phi_i}$, but by Dominance $\phi_i \vDash \phi_j$, and because ϕ_i cannot be safe in $(S_i)^-_{\phi_i}$, so is not ϕ_j.

4. *For Extensionality* if $Cn(\phi_i) \equiv Cn(\phi_j)$, then $(S_i \setminus \bigcup_{i=0}^{k_0} \Xi_i, <') \equiv (S_i \setminus \bigcup_{i=0}^{k_0} \Xi_j, <')$, hence $(S_i)^-_{\phi_i} = (S_i)^-_{\phi_j}$ □

Along the lines of the interpretation of $<$ in terms of security and reliability in Alchourrón and Makinson (1985), if the consequence relation $\vDash$ for S_i is intended to describe specification expressiveness, then the more it can be logically inferred from a property, the more expressive that property is. In turn, our safe contraction module $<$ makes more expressive properties safer, removing first those with the least inferential impact. This justifies our Dominance axiom; with Transitivity, the following continuing conditions hold (Alchourrón and Makinson 1985):

Proposition 2 (Continuing down) *If* $\phi_i < \phi_j$, *and* $\phi_k \vDash \phi_j$, *then* $\phi_i \leq \phi_k$, *for all* $\phi_{i,j,k} \in S_i$.

This is shown easily by Dominance and Transitivity. It means that if ϕ_i is more functionally entrenched than ϕ_j and ϕ_k induces ϕ_j, then ϕ_i is also at least as functionally entrenched as ϕ_k.

Proposition 3 (Continuing up) *If* $\phi_i \vDash \phi_j$, *and* $\phi_i < \phi_k$, *then* $\phi_j \leq \phi_k$, *for all* $\phi_{i,j,k} \in S_i$.

Again shown by Dominance and Transitivity. This says that if a property ϕ_i induces ϕ_j and is also as functionally entrenched as ϕ_k, then ϕ_j is also as functionally entrenched as ϕ_k.

3.3 Revision with reordering

In the standard literature on belief revision, revision is understood as the operation of adding new information to a knowledge base because of new information received about the world. In the following we analyse a revision induced by a new formula that requires to be prioritised over the existing ones. Intuitively, this is the case of a new property that we want to maximally protect from any later contraction. We obtain this by an operator that satisfies all properties of the AGM revision, and additionally re-defines the partial order in the base.

Definition 4 (*Revision with reordering*)

$$(S_i)^{\diamond}_{\phi_i} = (((S_i)^-_{\neg\phi_i})^+_{\phi_i}, <') \text{ where } <' = < \cup \{(\phi_i, \phi_j) \mid \phi_j \in ((S_i)^-_{\neg\phi_i})\}.$$

Below we provide an explicit algorithm (in pseudocode) to compute the result of the revision operator. As expected, we shall call the expansion and safe contraction procedures defined in the previous algorithms.

```
PROCEDURE Revision(S_i, <, φ_i)

S_i' := (Safe Contraction(S_i, <, ¬φ_i))
    DO Expansion(S_i', <, φ_i))

RETURN S'_{i+1} = {(φ_i, φ_j) | φ_i < φ_j for each φ_j ∈ S'_i}).

ENDPROCEDURE
```

This procedure defines an AGM revision operator $*$, in that it satisfies the Levi's identity, with the additional property that the revised base has acquired a new priority relation $<'$. Note that while expansion only performs a reordering if the added functionality implies some functionality present in the older base, by placing the former before the latter, revision with reordering ensures this happens (because it is defined in terms of expansion) but additionally prioritise the new functionality over any other unrelated one. To see this, consider the following three examples, one for each possible position of the contracted formula relatively to $<$:

Example 9

$$\{p < q\}^{\diamond}_{\neg p} = ((\{p < q\}^{-}_{p})^{+}_{(\neg p)})^{<'} = \{\neg p < q\} \tag{9}$$

Example 10

$$\{p < q\}^{\diamond}_{\neg q} = ((\{p < q\}^{-}_{q})^{+}_{(\neg q)})^{<'} = \{\neg q < p\} \tag{10}$$

Example 11

$$\{p < q < r\}^{\diamond}_{\neg q} = ((\{p < q < r\}^{-}_{q})^{+}_{(\neg q)})^{<'} = \{\neg q < p < r\} \tag{11}$$

Let us explain the Example 11 step by step: given a base with functionalities expressed by formulas p, qr and r, in this functional entrenchment order, we wish to perform ordered revision by $\neg q$. The following is the ordered series of changes:

$$\begin{aligned}
&\{p < q < r\}^{\diamond}_{\neg q} = \\
&(\{p < q < r\}^{-}_{q} = \{p < r\} \\
&(\{p < r\})^{+}_{(\neg q)} = \{p < r < \neg q\} \\
&(\{p < r < \neg q\})^{<'} = \{\neg q < p < r\}
\end{aligned}$$

In the first step, we remove q from the base, which is conflicting with the desired new functionality; in the second step, we add the new desired functionality $\neg q$; in the third step we reorder the base, bringing up the new functionality in the order over any other functionality p, r which has survived the initial removal operation (and preserving the order among them).

For the case illustrated above in Example 8 of a complex functionality expressed by logical implication, the operation of revision with reordering gives us the possibility

to remove a higher ordered functionality and to preserve a lower ordered but more complex one:

Example 12

$$\{p < (q \vee r) < (r \rightarrow s)\}^{\diamond}_{\neg((r\rightarrow s)\wedge p)} = \{(r \rightarrow s) < (q \vee r)\} \tag{12}$$

Ordered revision also satisfies the following:

Proposition 4 (Harper's identity)

$$(S_i)^{-}_{\phi_i} = S_i \cap (S_i)^{\diamond}_{\neg\phi_i}$$

This equation identifies the contraction operation with the intersection of the original base with the revised one. To show this, we refer to the result of the identities above using their example number:

Example 13

$$(6) \cap \{p < q\} = \{q\} = \{p < q\}^{-}_{p} \tag{13}$$

Example 14

$$(7) \cap \{p < q\} = \{p\} = \{p < q\}^{-}_{q} \tag{14}$$

Example 15

$$(8) \cap \{p < q < r\} = \{p < r\} = \{p < q < r\}^{-}_{q} \tag{15}$$

An informal way to justify this is as follows: a minimal revision $(S_i, <)^{*}_{\neg\phi_i}$ should keep the difference between the revised base and the original base minimal, i.e. keep as much as possible in common; hence, the contextual overlap among the two will be as large as it can be while conforming with $\neg\phi$; this makes the intersection $(S_i, <) \cap (S_i, <)^{*}_{\neg\phi_i}$ a plausible candidate for a minimal contraction of S_i on ϕ_i. In turn, this confirms that our contraction is indeed a minimal change operator. As $\diamond$ can be defined in terms of $-$ and $+$, it preserves the Gärdenfords postulates.

4 Example

It has been recently shown that some implementations of the Mergesort algorithm are broken, including the Timsort hybrid algorithm (de Gouw et al. 2015). We present here briefly the specification evolution from the broken implementation to the fixed specification, with remarks adapted to our analysis. The main loop of Timsort is presented in Fig. 5.

Consider this system specification as our S_m, whose theory is $\mathcal{S}_m$. This loop will satisfy an instance with `stackSize`$=4$, which we refer to as our formula ϕ_i. We

```
do {
    int runLen = countRunAndMakeAscending(a, lo, hi, c);
    if (runLen < minRun)
        {
        int force = nRemaining <= minRun ? nRemaining : minRun;
        binarySort(a, lo, lo + force, lo + runLen, c);
        runLen = force;
        }
    ts.pushRun(lo, runLen);
    ts.mergeCollapse();
    lo += runLen;
    nRemaining ?= runLen;
    }
while (nRemaining != 0);

assert lo == hi;
ts.mergeForceCollapse();
assert ts.stackSize == 1;
```

Fig. 5 Main loop of Timsort

```
private void mergeCollapse() {
        while (stackSize > 1) {
            int n = stackSize - 2;
            if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
                if (runLen[n - 1] < runLen[n + 1])
                    n--;
                mergeAt(n);
            } else if (runLen[n] <= runLen[n + 1]) {
                mergeAt(n);
            } else {
                break;
            }
        }
    }
```

Fig. 6 Java implementation of Timsort

now consider a Java implementation $\mathcal{I}$ of the above theory and the resulting system S_i, which will have a corresponding formulation of ϕ_i above, namely of the loop with `stackSize`$= 4$: S_i will hence include a disjunctive formula (to mimic the while loop) where each element is an implication with the antecedent assigning a value size to the stack and to the length of the run, and the consequent executing the merge:

$$\begin{aligned}\phi_i = \{&(\texttt{stackSize}, \texttt{runLen} == 4 \rightarrow \texttt{mergeCollapse}())\vee\\ &(\texttt{stackSize}, \texttt{runLen} == 3 \rightarrow \texttt{mergeCollapse}())\vee\\ &(\texttt{stackSize}, \texttt{runLen} == 2 \rightarrow \texttt{mergeCollapse}())\vee\\ &(\texttt{stackSize}, \texttt{runLen} = 1 \rightarrow \texttt{assert ts.stackSize} == 1)\}\end{aligned}$$

The implementation is presented in Fig. 6. In this implementation, it is the case that $S_i \nvDash \phi_i$ after violation of the invariant `ArrayIndexOutOfBoundsException` in `pushRun`, see de Gouw et al. (2015).

In order not to loose generality, one does not want to just remove ϕ_i. Hence one must individuate some $\phi_j < \phi_i$ and proceed to revise the model $S_i \mapsto S_i'$. We first proceed

```
1  private void newMergeCollapse() {
2    while (stackSize > 1) {
3      int n = stackSize - 2;
4      if (n > 0   && runLen[n-1] <= runLen[n] + runLen[n+1] ||
5          n-1 > 0 && runLen[n-2] <= runLen[n] + runLen[n-1]) {
6        if (runLen[n - 1] < runLen[n + 1])
7          n--;
8      } else if (n<0 || runLen[n] > runLen[n + 1]) {
9        break;
10     }
11     mergeAt(n);
12   }
13 }
```

Fig. 7 New Java implementation

by contraction $S_i' = (S_i)^-_{\phi_j}$ and then formulate some ϕ_k and proceed by expansion $S_i'' = (S_i')^+_{\phi_k}$. Our ϕ_j is now identified as the `mergeAT(n)` commands obtained by the satisfied `if` clauses. This will match the `mergeCollapse()` commands on lines 10 and 17 of the Timsort loop.

$$\begin{gathered}\phi_j = \{((\texttt{stackSize} > 0 \wedge \texttt{stackSize[n]} - -2) \rightarrow \\ (((\texttt{runLen[n} - 1\texttt{]} <= \texttt{runLen[n]} + \texttt{runLen[n} + 1\texttt{]}) \wedge \\ (\texttt{runLen[n} - 1\texttt{]} < \texttt{runLen[n} + 1\texttt{]})) \rightarrow \\ \texttt{[n} - 2\texttt{]} \wedge \texttt{mergeAT(n)} \vee \\ (\texttt{runLen[n]} <= \texttt{runLen[n} + 1\texttt{]}) \rightarrow \\ \texttt{mergeAT(n)}))\}\end{gathered}$$

We then proceed with formulating ϕ_k as `OR` clauses in the `if else` loops;

$$\begin{gathered}\phi_k = \{((\neg(\texttt{stackSize, runLen} > 1) \vee (\texttt{runLen[n]} <= \texttt{runLen[n} + 1\texttt{]})) \vee \\ ((\texttt{runLen[n} - 1\texttt{]} <= \texttt{runLen[n]} + \texttt{runLen[n} + 1\texttt{]}) \wedge \texttt{mergeCollapse()}) \vee \\ \texttt{assert ts.stackSize} == 1))\}\end{gathered}$$

finally, ϕ_j is again added $S_i''' = (S_i')^+_{\phi_j}$. The resulting new implementation is shown in Fig. 7. Notice that with respect to ϕ_j, a reordering in the functional entrenchment is also taking place.

5 Model checking

In this section, we consider the complexity of the model checking and inference problems for the change operations introduced in Sect. 3, along the line of Liberatore and Schaerf (2001) and Eiter and Gottlob (1992). We will mainly focus on the revision with reordering introduced in Sect. 3.3, as this is the most interesting one. Its definition subsumes the safe contraction, while the expansion is trivial.

We first examine the model checking problem. Recall that we are given a knowledge base $(S_i, <)$ where S_i is given as a set of propositional formulas and $<$ is the associated

order over S_i, a model M which is given as a valuation, a formula ϕ_i which is to be updated with respect to S_i, the model checking problem determines whether $M \models (S_i)^{\Diamond}_{\phi_i}$.

Proposition 5 *The model checking problem is in co-NP.*

Proof We first check whether $M \models \phi_i$, which can be in done in polynomial time. (As a matter of fact, it is in ALOGTIME, which is a uniform version of NC^1, so probably much lower than P.) If the answer is negative, we conclude that the model is not supported by the revision. Otherwise we proceed to check whether $M \models$ Safe Contraction$(S_i, \neg\phi_i)$. To this aim, we first compute $\Theta = \{\psi_i \in S_i \mid M \models \psi_i\}$.

Recall that for the safe contraction, we order the formulas in S_i in the way such that

- $S_i = \biguplus_{k=1}^{i} S_k$, i.e., a disjoint union of S_k's for $k = 1, \ldots, i$, and
- for all $1 \leq j \leq k$, S_j is the set of minimal elements of $S_i \setminus \bigcup_{1 \leq k \leq j-1} S_k$.

As a more succinct notation, we usually write $S_1 < S_2 < \cdots < S_m$. The partition can be obtained by standard topological sorting in polynomial time. Let

$$K = \min \left\{ \ell \mid \bigcup_{\ell \leq k \leq m} S_k \subseteq \Theta \right\}.$$

If $K = 1$, then clearly $M \models$ Safe Contraction$(S_i, \neg\phi_i)$ and we are done. We then assume that $K > 1$. In this case we claim that

$$M \models \text{Safe Contraction}(S_i, \neg\phi_i) \text{ iff } \bigcup_{K-1 \leq k \leq m} S_k \models \neg\phi_i.$$

To see this, suppose $M \models$ Safe Contraction$(S_i, \neg\phi_i)$, i.e. Safe Contraction$(S_i, \neg\phi_i) \subseteq \Theta$. Note that, according to the definition of K, $\bigcup_{K-1 \leq k \leq m} S_k \nsubseteq \Theta$. It follows that

$$\text{Safe Contraction}(S_i, \neg\phi_i) \subsetneq \bigcup_{K-1 \leq k \leq m} S_k,$$

and thus

$$\text{Safe Contraction}(S_i, \neg\phi_i) = \bigcup_{j \leq k \leq m} S_k \text{ for some } j \geq k.$$

According to the definition of safe contraction, we have that

$$\bigcup_{K-1 \leq k \leq m} S_k \models \neg\phi_i.$$

For the other direction, suppose that $\bigcup_{K-1 \leq k \leq m} S_k \models \neg\phi_i$. According to the definition of K, it must be the case that

$$\text{Safe Contraction}(S_i, \neg\phi_i) \subseteq \bigcup_{K \leq k \leq m} S_k \subseteq \Theta$$

Fig. 8 Model checking algorithm

```
1   PROCEDURE ModelChecking(M, (S_i, <), φ_i)
2
3   IF M ⊨ φ_i RETURN FALSE
4   ELSE
5     Θ := {ψ_i ∈ S_i | M ⊨ ψ_i}
6     FOR d FROM 1 TO m
7       IF ⋃_{d≤k≤m} S_k ⊆ Θ
8       THEN BREAK ENDIF
9     ENDFOR
10    IF d == 1
11       THEN RETURN TRUE
12       ELSE RETURN ⋃_{d≤k≤m} S_k ⊨ ¬φ_i
13    ENDIF
14  ENDIF
15  ENDPROCEDURE
```

which implies that $M \models$ Safe Contraction$(S_i, \neg\phi_i)$. We observe that checking $\bigcup_{K-1\leq k\leq m} S_k \models \neg\phi_i$ amounts to checking the validity of the formula

$$\bigwedge_{\xi\in\bigcup_{K-1\leq k\leq m} S_k} \xi \implies \neg\phi_i,$$

which can be done in co-NP. This gives a co-NP algorithm for checking $M \models$ Safe Contraction$(S_i, \neg\phi_i)$. Overall, we have a co-NP algorithm for checking $M \models (S)^{\diamond}_{\phi_i}$. This completes the proof. □

Below we give an example to show that the order associated with the knowledge base has to be considered during model checking, because with different orders the results vary.

Example 1 (**Order matters**) Given $S_i = \{p, q\}$, $\phi_i = \neg(p \wedge q)$ and $M(p) = 1, M(q) = 0$. Obviously $M \models \phi_i$. If we have $p < q$, then the revision would be $\{q, \phi_i\}$. If p and q are unordered, then the revision would be $\{\phi_i\}$. In the former case, $M \not\models \{q, \phi_i\}$ while in the latter case, $M \models \{\phi_i\}$. □

A decision algorithm for the model checking problem is presented in Fig. 8.

We now turn to the inference problem. Again we focus on the revision with reordering. Formally, we are given a knowledge base $(S_i, <)$, two formulas ϕ_i and ϕ_j where ϕ_i is to be updated wrt S_i, the inference problem determines whether $(S_i)^{\diamond}_{\phi} \models \phi_j$.

Proposition 6 *The complexity of satisfiability checking is in* Σ_2^{p}*, i.e.,* $\mathrm{NP}^{\mathrm{NP}}$.

Proof As in the proof of Proposition 5, we first compute $\Theta = \{\psi_i \in S_i \mid M \models \psi_i\}$. Recall that for the safe contraction, we order the formulas in S_i in the way such that

- $S_i = \biguplus_{k=1}^{i} S_k$, i.e., a disjoint union of S_k's for $k = 1, \ldots, i$, and
- for all $1 \leq j \leq k$, S_j is the set of minimal elements of $S_i \setminus \bigcup_{1\leq k\leq j-1} S_k$.

We observe that $(S_i)^{\diamond}_{\phi_i} \models \phi j$ iff there exists some d such that

(i) $\bigcup_{d\leq k\leq m} S_k \cup \{\phi_i\} \models \phi_j$, and
(ii) $\bigcup_{d\leq k\leq m} S_k \not\models \neg\phi_i$.

```
1   PROCEDURE InferenceChecking((S_i, <), φ_i, φ_j)
2
3   Θ := {ψ_i ∈ S_i | M ⊨ ψ_i}
4   FOR d FROM 1 TO m
5     IF ⋃_{d≤k≤m} S_k ∪ {φ_i} ⊨ φ_j ∧ ⋃_{d≤k≤m} S_k ⊭ ¬φ_i
6       THEN RETURN TRUE
7     ENDIF
8   ENDFOR
9   RETURN FALSE
10  ENDPROCEDURE
```

Fig. 9 Inference algorithm

The "only if" direction is trivial, as according to the definition of safe contraction,

$$\text{Safe Contraction}(S_i, \neg\phi_i) = \bigcup_{K-1\leq k\leq m} S_k$$

such that $\text{Safe Contraction}(S_i, \neg\phi_i) \not\models \neg\phi_i$ and $\text{Safe Contraction}(S_i, \neg\phi_i) \cup \{\phi_i\} \models \phi_j$.

For the "if" direction, assume such d exists, then it must be the case that

$$\bigcup_{d\leq k\leq m} S_k \cup \{\phi_i\} \subseteq \text{Safe Contraction}(S_i, \neg\phi_i)$$

because of (ii), and we must have that

$$\text{Safe Contraction}(S_i, \neg\phi_i) \cup \{\phi_i\} \models \phi_j$$

because of (i). Now note that both (i) and (ii) can be checked in co-NP and NP respectively, we can easily obtain an NP^{NP} algorithm, which completes the proof. □

A decision algorithm for the inference checking problem is presented in Fig. 9.

6 Remarks on resilience and evolvability

In the introduction we have suggested that the concept of reliability for software systems can be defined in terms of notions of resilience and evolvability.

Resilience for a computational system reflects its (graded) ability to preserve a working implementation under changed specifications. The above analysis of software theory change allows us to provide a precise definition of resilience in the presence of removal or failure of certain components. In the literature on software change, this process corresponds to preservation of *behavioural safety* by specification approximation, see e.g. the taxonomy offered in Buckley et al. (2005). Various attempts have been made to formalise perseverance of validity under change. The most common one encountered in this research area is that of *system robustness* (Bloem et al. 2010a, b). One (older) interpretation is given in terms of the inability of the system to distinguish

between behaviours that are essentially the same, see Peled (1997). More recently, the term resilience has been used to refer to the ability of a system to retain functional and non-functional identity with the ability to perceive environmental changes; to understand their implications and to plan and enact adjustments intended to improve the system-environment fit (De Florio 2013).

Evolvability is the ability to successfully accommodate changes, the capacity to generate adaptive variability in tandem with continued persistence (Cook et al. 2005) and more generally the system's ability to survive changes in its environment, requirements and implementation technologies (Ciraci and van den Broek 2006). The crucial need to accommodate changes in requirements and corresponding intended functionalities with the least possible cost while maintaining architectural integrity has been stressed since Rowe et al. (1998). Our analysis of resilience and evolvability has focused on functional entrenchment and the change in view of prioritised functionalities.

In view of the operation of ordered contraction, resilience of functionalities in a software system can be defined by functional entrenchment via logical consequence:

Definition 5 (*Property resilience*) Consider property specifications $\phi_i, \phi_j \in \mathcal{S}_m$ and a relevant implementation S_i. Then ϕ_i is resilient in $(S_i)^-_{\phi_j}$ iff $\phi_j \nvDash \phi_i$.

This holds immediately by Definition 3. Generalising, one can say that a software system specification S_i is resilient with respect to a property specification ϕ_i if the latter is safe in any contracted subsystem that preserves minimal functionalities of S_i. System resilience as the resistance to change of property specifications (as in Definition 5) can be essential to determine system antifragility (De Florio 2014). Software antifragility has been characterized as self-healing (automatic run-time bug fixing) and adaptive fault-tolerance (tested e.g. by fault-injection in production) (Monperrus 2017). An inferential notion of resilience helps characterizing a certain degree of fault-tolerance; the latter is considered strictly intertwined with self-healing properties: while not all fault-tolerant systems are self-healing, one can argue that self-healing techniques are ultimately dependable computing techniques (Koopman 2003). Our resilient core, intended as the persistence of service delivery (Laprie 2008) in view of functionalities contraction, allows to determine the adaptation required by changes in terms of valid and invalid properties of its contractions and can anticipate results of its expansions. In particular, given the non-resilient part of the system, it is possible to establish which properties will still be instantiated in any subsystem. In this sense, resilience is a function of dependability between functionalities, expressed as an inferential relation.

Proposition 7 (Accountability) *For any $\phi_j \in \Xi_i$ as per Definition 2 such that ϕ_j is minimal w.r.t $\phi_i < \phi_j$, then $(S_i)^-_{\phi_i} \nvDash \phi_j$. For any $\phi_j \in \Xi_i$ as per Definition 2 such that ϕ_j is not minimal w.r.t $\phi_j < \phi_i$, then $(S_i)^-_{\phi_i} \vDash \phi_j$.*

This holds immediately by Definition 2, Dominance and Definition 3. Furthermore, we have qualified our ordered revision operator as a function of evolvable software systems. In the context of prioritised functionalities, we have more precisely formulated evolvability as the property of a software to be updated to fulfill a *newly prioritised* set of functionalities. This allows to deal explicitly with one of the main problem of architecture design (Breivold and Crnkovic 2010). The advantage of our analysis is

again based on the relation between the priority order and the inferential relation, as expressed by the Dominance property. This can be useful to guarantee a prevision property on formulas affected negatively by a revision operation:

Proposition 8 (Prevision) *For any properties* $\phi_i, \phi_j \in S_i$ *such* $\phi_i < \phi_j$*, there is always a* $S_i' = (S_i)^{\diamond}_{\phi_k}$ *such that* ϕ_i *becomes an element w.r.t.* $<$ *exposed to a further contraction.*

This holds by Dominance and Definition 4. In other words, revision with reordering (i.e. a novel prioritisation of evolvable sub-characteristics for a system) always implies the possibility of building a system with a module including a previously safe property which is no longer safe to future contractions.

In view of such prevision property, one can re-factor the impact of evolvability on sub-characteristics, in order to know how much a novel prioritisation will make the system exposed. To do so, it is sufficient to consider:

1. The cardinality of the consequence set of each property $\mid Cn(\phi_i) \mid$;
2. An order on their sizes, denoted by $\prec$;
3. And a preference weighting on selected revisions, denoted by $\preccurlyeq$.

Definition 6 (*Preference on revision*) Consider a system $S_i := \{\phi_i < \phi_j < \phi_k < \cdots < \phi_n\}$, such that $\phi_i \nvdash \phi_j$ and $\phi_j \nvdash \phi_k$. Then $(S_i)^{\diamond}_{\neg\phi_j} \preccurlyeq (S_i)^{\diamond}_{\neg\phi_k}$ iff $\mid Cn(\phi_j) \mid \prec \mid Cn(\phi_k) \mid$.

In other words, if a reorder revision should be selected between two options which do not have a logical relation, the impact of the removed properties in the system should be taken into account in terms of the respective consequences on the models. Note that preference on revision requires to move our analysis from the specification base S_i to the model $\mathcal{S}_m$.

Software tools based on either theorem proving or model checking techniques can implement this theory to several aims. In view of Property Resilience from Definition 5 and Proposition 7, it is possible to express direct and indirect dependency relations between software packages in the presence of uninstall operations. A similar task is performed proof-theoretically by a natural deduction calculus with an explicit notion of trust in Primiero and Boender (2018). More importantly, it will be possible to provide information as to the resulting state of the system after uninstall operation have been performed, anticipating possible resulting unstable conditions and the eventual impossibility of the system to perform critical operations. The utility of Proposition 8 in combination with Definition 6 is again in view of assessing the state of the system in the presence of uninstall operations, by anticipating which functionalities may result threatened and establishing removal options of minimal impact on the system among possible ones. Although the present work does not aim at the development of any such tool, we believe that a solid theoretical basis to this aim is a valuable contribution.

Different authors have quantified software maintenance between 40% and $>$ 90% of build costs, depending on the software project considered, project development methodology used and best practices applied. A checklist used to explore the realism and accuracy of maintenance requirements (see for example Hunt et al. 2008) should include, among others, the following questions:

1. Which parts of the system will be preserved and allow incremental maintenance?
2. Are healthy chunks of the original code being rewritten or changed?
3. Which pieces of software will need (non-incremental) maintenance?
4. Can you assess which (non-incremental) maintenance operations are the least invasive on the current system?

We believe our formal model can be used to provide answers to such questions, and thus be of further help to the constraints of software maintenance costs.

7 Conclusion

We have presented a theory of software change inspired by techniques used in belief revision theory. We have highlighted how operators on contraction, expansion and revision with reordering over a base with functional entrenchment allow to identify resilience and evolvability properties for software systems. We have moreover identified the complexity of such operations and gave a real case scenario on a recent example of a broken algorithm. Future research will focus on enhancements of this model that can be of further interest to the software engineering community, e.g. by operations of multiple contraction and selective revision. From the conceptual point of view, the interest is in the modelling of anti-fragility properties for software systems in the light of revision and update operations.

Acknowledgements Giuseppe Primiero is partially supported by the Project PROGRAMme ANR-17-CE38-0003-01. He moreover wishes to thank Hykel Hosni for useful discussions held during the early stages of this research. Franco Raimondi was supported by a Visiting Professorship grant from Dipartimento di Matematica e Fisica, Università degli studi della Campania "Luigi Vanvitelli", Caserta, Italy. Taolue Chen is partially supported by UK EPSRC Grant (EP/P00430X/1), Birkbeck BEI School Project (ARTEFACT), NSFC Grant (No. 61662035), and Guangdong Science and Technology Department Grant (No. 2018B010107004).

References

Abrial, J.-R. (2005). *The B-book: Assigning programs to meanings*. Cambridge: Cambridge University Press.

Alchourrón, C. E., Gärdenfors, P., & Makinson, D. (1985). On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, *50*, 510–530.

Alchourrón, C. E., & Makinson, D. (1985). On the logic of theory change: Safe contraction. *Studia Logica*, *44*(4), 405–422.

Alchourrón, C. E., & Makinson, D. (1986). Maps between some different kinds of contraction function: The finite case. *Studia Logica*, *45*(2), 187–198.

Alechina, N., Jago, M., & Logan, B. (2005). Resource-bounded belief revision and contraction. In *Declarative Agent Languages and Technologies III, Third International Workshop* (pp. 141–154), DALT 2005, Utrecht, The Netherlands, July 25, 2005, Selected and Revised Papers.

Alechina, N., Liu, F., & Logan, B. (2015). Efficient minimal preference change. *Journal of Logic and Computation*, *28*, 1715–1733. https://doi.org/10.1093/logcom/exv027.

Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T. A., & Jobstmann, B. (2010a). Robustness in the presence of liveness. In T. Touili, B. Cook, & P. Jackson (Eds.), *Computer aided verification* (Vol. 6174, pp. 410–424), Lecture notes in computer science. Berlin: Springer.

Bloem, R., Greimel, K., Henzinger, T. A., & Jobstmann, B. (2010b). Synthesizing robust systems. *Acta Informatica*, *51*(3–4), 193–220.

Booth, R. (2001). A negotiation-style framework for non-prioritised revision. In *Proceedings of the 8th Conference on Theoretical Aspects of Rationality and Knowledge* (TARK2001) (pp. 137–150), Siena, Italy, July 8–10, 2001.

Breivold, H. P., & Crnkovic, I. (2010). An extended quantitative analysis approach for architecting evolvable software systems. In *Computing Professionals Conference Workshop on Industrial Software Evolution and Maintenance Processes* (WISEMP-10), IEEE.

Breivold, H. P, Crnkovic, I., Land, R., & Larsson, M. (2008). Analyzing software evolvability of an industrial automation control system: A case study. In *ICSEA 2008* (pp. 205–213).

Buccafurri, F., Eiter, T., Gottlob, G., & Leone, N. (1999). Enhancing model checking in verification by AI techniques. *Artificial Intelligence*, *112*(1–2), 57–104.

Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel, G. (2005). Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, *17*(5), 309–332.

Ciraci, S., & van den Broek, P. (2006). Evolvability as a quality attribute of software architectures. In *International ERCIM Workshop on Software Evolution 2006* (pp. 29–31), 6–7 April 2006, Universite des Sciences et Technologies de Lille, France.

Cook, S., Harrison, R., & Wernick, P. (2005). A simulation model of self-organising evolvability in software systems. In *Proceedings of the 2005 IEEE International Workshop on Software Evolvability (Software-Evolvability'05)*, IEEE Computer Society.

Dam, H. K., & Ghose, A. (2014). Towards rational and minimal change propagation in model evolution. CoRR. arXiv:1402.6046).

De Florio, V. (2013). On the constituent attributes of software and organisational resilience. *Interdisciplinary Science Reviews*, *38*(2), 122–148. Maney Publishing.

De Florio, V. (2014). Antifragility = elasticity + resilience + machine learning models and algorithms for open system fidelity. *Procedia Computer Science*, *32*, 834–841. Elsevier.

de Gouw, S., Rot, J., de Boer, F. S., Bubel, R., & Hähnle, R. (2015). OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case? In D. Kroening & C. Pasareanu (Eds.), *Computer aided verification* (Vol. 9206, pp. 273–289), CAV 2015. Lecture notes in computer science. Cham: Springer.

Delgrande, J., Peppas, P., & Woltran, S. (2013a). AGM-style belief revision of logic programs under answer set semantics. *Logic Programming and Nonmonotonic Reasoning*, *8148*(2013), 264–276. LNCS.

Delgrande, J., Schaub, T., Tompits, H., & Woltran, S. (2013b). A model-theoretic approach to belief change in answer set programming. *ACM Transactions on Computational Logic*, *14*(2), 264–276.

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, *27*(1), 1–12.

Eiter, T., & Gottlob, G. (1992). On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Artificial Intelligence*, *57*(2–3), 227–270.

Ernst, N. A., Mylopoulos, J., & Wang, Y. (2009). Requirements evolution and what (research) to do about it. In K. Lyytinen et al. (Eds.), *Design requirements workshop* (Vol. 14, pp. 186–214), LNBIP.

Fellows, L. (1998). A case for priority classifying requirements. In *Eighth Annual International Symposium on Systems Engineering*.

Firesmith, D. (2004). Prioritizing requirements. *Journal of Object Technology*, *3*(8), 35–47.

Floridi, L., Fresco, N., & Primiero, G. (2014). On malfunctioning software. *Synthese*, *192*(4), 1199–1220. https://doi.org/10.1007/s11229-014-0610-3.

Fuhrmann, A. (1991). Theory contraction through base contraction. *Journal of Philosophical Logic*, *20*(2), 175–203.

Gärdenfors, P., & Makinson, D. (1988). Revisions of knowledge systems using epistemic entrenchment. In *Second Conference on Theoretical Aspects of Reasoning about Knowledge* (pp. 83–95).

Guerra, P. T., & Wassermann, R. (2010). Revision of CTL models. In A. Kuri-Morales & G. R. Simari (Eds.), *Advances in artificial intelligence?* (Vol. 6433), IBERAMIA 2010. Lecture notes in computer science. Berlin: Springer.

Hunt, B., Turner, B., & McRitchie, K. (2008). Software maintenance implications on cost and schedule. In *2008 IEEE Aerospace Conference* (pp. 1–6), Big Sky, MT. https://doi.org/10.1109/AERO.2008.4526688.

Kolyang, S. T., & Wolff, B. (1996). A structure preserving encoding of Z in Isabelle/HOL. In G. Goos, J. Hartmanis, J. van Leeuwen, J. von Wright, J. Grundy, & J. Harrison (Eds.), *Theorem proving in higher order logics* (Vol. 1125), TPHOLs 1996. Lecture notes in computer science. Berlin: Springer.

Koopman, P. (2003). Elements of the self-healing system problem space. In *Workshop on Architecting Dependable Systems/WADS03*, May 2003. http://users.ece.cmu.edu/~koopman/roses/wads03/wads03.pdf. Accessed May 2019.

Laprie, J.-C. (2008). From dependability to resilience. In *Proceedings of IEEE International Conference on Dependable Systems and Networks* (Vol. Supplemental, pp. G8–G9).

Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, *68*(9), 101–103.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). Metrics and laws of software evolution—The nineties view. In *Proceedings of 4th International Software Metrics Symposium (METRICS '97)* (pp. 20–32). https://doi.org/10.1109/METRIC.1997.63715.

Liberatore, P., & Schaerf, M. (2001). Belief revision and update: Complexity of model checking. *Journal of Computer and System Sciences*, *62*(1), 43–72.

Lientz, B., & Swanson, B. (1980). *Software maintenance management*. Boston: Addison-Wesley.

Lindvall, M., Tesoriero, R., & Costa, P. (2002). Avoiding architectural degeneration: An evaluation process for software architecture. In *Proceedings of the 8th IEEE Symposium on Software Metrics* (pp. 77–86).

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., & Jazayeri, M. (2005). Challenges in software evolution. In *Proceedings of the 2005 Eighth International Workshop on Principles of Software Evolution*.

Monperrus, M. (2017). Principles of antifragile software. In J. B. Sartor, T. D'Hondt & W. De Meuter (Eds.), *Companion to the First International Conference on the Art, Science and Engineering of Programming* (Programming '17). ACM, New York, NY, USA, Article 32, 4 pages. https://doi.org/10.1145/3079368.3079412.

Mu, K.-D., Liu, W., Jin, Z., Hong, J., & Bell, D. (2011). Managing software requirements changes based on negotiation-style revision. *Journal of Computer Science and Technology*, *26*(5), 890–907.

Peled, D. (1997). Verification for robust specification. In E. Gunter & A. Felty (Eds.), *Theorem proving in higher order logics* (Vol. 1275, pp. 231–241), Lecture notes in computer science. Berlin: Springer.

Port, D., & Liguo, H. (2003). Strategic architectural flexibility. In *Proceedings of the International Conference on Software Maintenance* (pp. 389–396).

Primiero, G., & Boender, J. (2018). Negative trust for conflict resolution in software management. *Web Intelligence*, *16*(4), 251–271.

Rowe, D., Leaney, J., & Lowe, D. (1998). Defining systems evolvability—A taxonomy of change. In *International Conference and Workshop: Engineering of Computer-Based Systems* (page 45+). Maale Hachamisha, Israel, IEEE Computer Society.

Sommerville, I. (2004). *Software engineering* (7th ed.). Boston: Addison-Wesley.

Sousa, T. C., & Wassermann, R. (2007). Handling inconsistencies in CTL model-checking using belief revision. In *Proceedings of the Brazilian Symposium on Formal Methods*.

Spivey, J. M., & Abrial, J. R. (1992). *The Z notation*. Hemel Hempstead: Prentice Hall.

Williams, B. J., & Carver, J. C. (2010). Characterizing software architecture changes: A systematic review. *Information and Software Technology*, *52*, 31–51.

Zave, P., & Jackson, M. (1997). Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *6*(1), 1–30.

Zhang, Y., & Ding, Y. (2008). CTL model update for system modifications. *Journal of Artificial Intelligence Research*, *31*(1), 113–155.

Zowghi, D., Ghose, A., & Peppas, P. (1996). A framework for reasoning about requirements evolution. *PRICAI'96: Topics in artificial intelligence* (Vol. 1114, pp. 157–168), Lecture notes in computer science. Berlin: Springer.