Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 52 (2015) 988 – 995

Procedia Computer Science

The 2nd International Workshop on Computational Antifragility and Antifragile Engineering (ANTIFRAGILE 2015)

Software Theory Change for resilient near-complete specifications

Giuseppe Primiero[a,*], Franco Raimondi[a]

[a]Department of Computer Science, Middlesex University, London, UK

Abstract

Software evolution and its laws are essential for antifragile system design and development. In this paper we model early-stage *perfective* and *corrective* changes to software system architecture in terms of logical operations of expansion and safe contraction on a theory. As a result, we formulate an inference-based notion of property specification resilience for computational systems, intended as resistance to change. The individuated resilient core of a software system is used to characterize adaptability properties.

Peer-review under responsibility of the Conference Program Chairs

Keywords: Software Evolution; Theory Change; Property and System Resilience; System Adaptability.

1. Introduction

Software change is a critical step in the life-cycle of computational systems. Modifying or re-defining system specification properties is required by increasing architectural complexity or to improve software quality. In either case, "software maintenance has been regarded as the most expensive phase of the software cycle"[35]p.32. A relevant amount of research has already been dedicated to the understanding, planning and execution of software evolution, in particular for requirements evolution, see e.g.[16]. Typically, this occurs as part of the *late* life-cycle of the system, dictated by architectural degeneration (violation or deviation of the architecture, increasing with changes being made to the original, see e.g.[15,27]) and flexibility requirements (the system property that defines the extent to which the system allows for unplanned modifications[32]). In this context, resilience as functionality preservation under changes is of paramount importance. The laws of software evolution for computational systems linked to the real environment[24,25] express the importance of an appropriate understanding of software change. Change classification schemes, assessing the impact and risk associated with software evolution, present challenges[28] 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*, which include testing, with the exception of model-based testing. *Early* life-cycle

* Correponding author. Tel.: +44-020-841-15146.
E-mail address: G.Primiero@mdx.ac.uk

Peer-review under responsibility of the Conference Program Chairs
doi:10.1016/j.procs.2015.05.091

change can be classified by *disfunctions*, where the system is less reliable or effective than one expects in performing its function, see[17]. At early design and implementation stages, *perfective changes* result from new or changed requirements, see[26,33]. We explicitly ignore here the other two classifications, namely *adaptive changes* and *preventative changes*. A violation of the model specification in the implementation leads to a revision of the examined system. Such process of software change can be modelled similarly to scientific theory change, to account for both perfective and corrective changes in the implementation, when the latter is diverging from the model specification. We model software change as operations on a model of a scientific theory, defined according to the formal operations of AGM belief change, see[2]. We constrain the revision operation to finite bases, as we assume that requirements specification of any however large software should be accounted for in terms of a finite representation. 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[36], offering a framework to reason about requirements evolution in terms of belief change operations. In[10], belief revision used to deal with change propagation in model evolution. In[30], Booth's[8] 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 prioritization. AGM belief revision has been investigated for logic programming under answer set semantics in[13,14]. While notoriously a number of methods in software engineering have focused on developing implementation from specifications[34,1], our analysis concentrates on the modelling of both perfective and corrective changes to design new specifications from early (incorrect) implementations. We formalize such changes in terms of expansion and safe contraction operations. Most importantly, we use such operations to define property resilience and generalise it to a definition of system adaptability. The latter connection, in particular, appears to be entirely missing in the literature. The rest of this paper is structured as follows. In section 2 we offer an informal analysis of software as a theory, using the distinction between model and implementation; in section 3 we offer the formal analysis of requirements evolution as revision on such a theory; in section 5 we link this analysis to property resilience for software systems.

2. Software Theory

A *model* of a computational system $\mathcal{S}_m$ is a representation of the system's intended behaviour: as such, it naturally offers a representation of its specification in terms of law-like expressions formulating requirements of valid input states, and the result of (program) actions in output states. Identified preconditions guarantee *safety* of program execution, i.e. they include sufficient states for some program of type p to obtain a state that satisfies the postconditions. Identifying preconditions that (may) lead to execution failure of an instance of program of type p allows to define *completeness*. A complete description of $\mathcal{S}_m$ is akin to what in software testing research is known as an *oracle*, a "procedure that determines what the correct behaviour of a system should be for all input stimuli with which we wish to subject the system under test"[20]. A complete model $\mathcal{S}_m$ of a correctly functioning computational system can be taken to mimic the notion of *right theory* for a software system. In this sense, $\mathcal{S}_m$ is understood as the perfect description of design-for-test principles, which can be accompanied by a (possibly complete) formal specification of the intended behaviour. *Near-complete* specifications refer to approximations to complete specifications through the use of possible errors classification in weakened postconditions and an algebra on the possible program states, see[23]. A similar sense of near-completeness is offered by approaches to requirements engineering where the relevant specification description can be marked up with refinements given by possible, admissible or acceptable malfunctioning. Where completeness is not achieved, a partial or near-complete oracle is a description that offers postconditions for some given inputs, and could offer alternative means for other inputs (metamorphic testing and derived information from example executions).

An *implementation* of $\mathcal{S}_m$ is an actual realization of the model in some programming language. We assume that such realization is for all purposes and intentions faithful to the spirit of $\mathcal{S}_m$. We assume moreover the possibility of abstracting (possibly automatically) from said implementation a set of law-like expressions of the same syntax as those forming $\mathcal{S}_m$. We currently abstract from the concrete syntax of such expressions. Let us call this new set of specifications $\mathcal{S}_i$. We assume, finally, that $\mathcal{S}_i$ is in fact comparable to $\mathcal{S}_m$, in the sense of containing at least some of the specification properties expressed by the latter. The comparison of requirements implemented in $\mathcal{S}_i$ with $\mathcal{S}_m$ can be performed in terms of log instances expressing observed truths of domain literals or task occurrences, where successful execution of the latter ones in appropriate order leads to goal satisfaction. A working implementation, i.e. a piece of code which *correctly* translates the specification presented in $\mathcal{S}_m$, can then be regarded as an artefact whose every execution realises output states that do not diverge from those expressed by the corresponding law-like idealizations in the model. For a malfunctioning implementation we require a faulty program execution:

Definition 2.1 (Faulty program execution). *An execution of p_i in an instance of $\mathcal{S}_i$ is faulty when the actual post-conditions obtained diverge from those expected according to $\mathcal{S}_m$.*

Definition 2.2 (Malfunctioning implementation). *An implementation of $\mathcal{S}_i$ is malfunctioning with respect to the intended model $\mathcal{S}_m$ iff the former accommodates possibly faulty execution of a program p_i.*

Faulty execution as expressed by divergence from expected or intended postconditions can be seen as a figure of drifting from fidelity, see [12]. Reasons for labelling an instance of a program as malfunctioning with respect to its model can be reduced to either the requirement of stronger preconditions for running p_i as intended by $\mathcal{S}_m$, or the obtaining of different postconditions. Software systems' resistance to change in the environment as graceful degradation is a research topic of its own [6,7], including reference to hardware and material execution conditions, and we abstract from it in the present contribution to model specifically the relation of change in specifications as induced from unexpected postconditions. In the case of diverging descriptions of $\mathcal{S}_i$ from $\mathcal{S}_m$, perfective or corrective changes might be applied to produce a new model $\mathcal{S}_m^{'}$ that either accommodates the divergences, or eliminates some of the current (undesired) properties of the model $\mathcal{S}_m$. This process, known as *requirements evolution*, formally corresponds to a mapping $\mathcal{S}_i \mapsto \mathcal{S}_m^{'}$. Informally, it can be associated with the restructuring of required properties in a model adapted to the current implementation conditions, a typically antifragile process. Formally, we require a non-monotonic logic which, in the case of perfective changes allows to add desired property-specifications and in the case of corrective changes allows to remove undesired ones. Comparing $\mathcal{S}_i$ to $\mathcal{S}_m$, it is possible to identify which elements of the former should be changed to have a better approximation to the latter, i.e. to reduce drifting and restore properties of the originally intended model in the implementation (possibly with a higher level of resistance). The formal operations described in the next section allow to establish which elements from $\mathcal{S}_m$ are safe, when contraction of $\mathcal{S}_i$ is performed.

3. Requirements Evolution by Theory Change

Consider a finite set of formulae $\mathcal{S}_m = \{\phi_1, \dots, \phi_n\}$, where each ϕ_i expresses a specific behaviour that the intended software system $\mathcal{S}_m$ should display. The intended meaning of a formula ϕ_i is an instance of the relation over the state space represented by program actions defining $\mathcal{S}_m$. We refer to S_m as a theory of the model, i.e. its closure under logical implication $S_m = Cn(\mathcal{S}_m) := \{\phi_i \mid \mathcal{S}_m \vDash \phi_i\}$. We say that S_m is consistent if $S_m \vDash \neg(\phi_i \wedge \neg\phi_i)$. We currently abstract away from the definition of the consequence relation $\vDash$ for S_m, which can be thought of as any classical consequence relation without completeness (which appears too strong in the context of real software specifications) and compactness (which is trivial in the present finite setting):

1. $S_m \vDash \top$
2. $S_m \vDash (\phi_i \rightarrow \phi_j)$ and $S_m \vDash \phi_i$, implies $S_m \vDash \phi_j$
3. $S_m \vDash \phi_i$ implies $S_m \nvDash \neg\phi_i$

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

Consider now a new language $\mathcal{S}_i$ for the model abstracted from an implementation of $\mathcal{S}_m$ and so that for some ϕ_i, either a theory $S_i = Cn(\mathcal{S}_i)$ is such that $S_i \nvDash \phi_i$ and $S_m \vDash \phi_i$; or $S_i \vDash \phi_i$ and $S_m \nvDash \phi_i$ or $S_m \vDash \neg\phi_i$. Formal operations can be defined on S_i so that either the current input in the *implementation* becomes valid, or the specification that makes our experimental result wrong is removed. In the former case, S_i is expanded as to include ϕ_i, hence one handles a form of requirement incompleteness: we indicate the result of this change as *expansion*. This formal operation reflects the implementation of a new requirement and hence qualifies as a *perfective change*. In the second case, S_i is contracted as to remove ϕ_i (under a complete theory, which we do not assume, this induces satisfiability of $\neg\phi_i$): the divergence between the implementation and the oracle reflects here a form of inconsistency, while at each stage of the implementation consistency is preserved; we indicate the result of this revision as *contraction*. This formal operation reflects the removal of an undesired requirement (error fixing) and hence qualifies as a *corrective change*. In both cases, a new model S'_m is obtained, from which a new implementation can be formulated.

3.1. Expansion

The process of designing a piece of software can be seen as moving from an empty set of requirements (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 specification requirement ϕ_i, denoted as $S'_m = (S_i)^+_{\phi_i}$ and defined by the logical closure of $\mathcal{S}_i$ and ϕ_i, $(S_i)^+_{\phi} = Cn(\mathcal{S}_i \cup \{\phi\})$. Theory 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 $\mathcal{S}_i$. Hence, each expansion requires a minimal set of contraction operations.

3.2. Contraction

We now consider contracting $\mathcal{S}_i$ in view of a specification requirement ϕ_i, inducing the removal of the minimal set of specifications implying ϕ_i. We denote this by $(S_i)^-_{\phi_i} = Cn(\mathcal{S}_i/\phi_i)$, where the latter indicates the result of S_i once ϕ_i has been removed. The contraction operator is then a mapping function:

$$S_i \mapsto S'_m := \{(S_i, \phi_i)\} \mapsto (S_i)^-_{\phi_i}$$

from the set of theories of the current $\mathcal{S}_i$ to a new model whose implementations have languages that do not imply ϕ_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 reflect this formally by an ordering $\leq$ on properties, similarly to what is done with epistemic entrenchment[18]: $\phi_i \leq \phi_j$ in our context says that ϕ_j is at least as embedded as ϕ_i in the system in view of its functionalities. Hence, in a contraction process, one would remove first the latter in order to preserve as much as possible the operational properties of the system. $\leq$ satisfies the following postulates:

1. Transitivity: if $\phi_i \leq \phi_j$ and $\phi_j \leq \phi_k$, then $\phi_i \leq \phi_k$;
2. (Anti-)Dominance: if $\phi_i \vDash \phi_j$, then $\phi_j \leq \phi_i$;
3. Conjunctiveness: either $\phi_i \leq \phi_i \wedge \phi_j$ or $\phi_j \leq \phi_i \wedge \phi_j$;
4. Minimality: if S_i is consistent, then $\phi_i \notin S_i$ iff $\phi_i \leq \phi_j$ for all ϕ_j;
5. Maximality: if $\phi_j \leq \phi_i$ for all ϕ_j, then $\phi_i \in Cn(\emptyset)$.

Standard Gärdenfors postulates are modified by (Anti-)Dominance, inverting the usual relation with $\vDash$. Among the different (although in some ways related, see[4]) 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 3.1 (Safe Contraction,[3]). *Given a theory S_i of an implementation of S_m, ordered by a transitive non-circular relation $<$ (or hierarchy); let $\phi_i \in S_i$ express a property we wish to eliminate from the corresponding model $\mathcal{S}_i$; then we say that a property $\phi_j \in S_m$ is safe with respect to $(S_i)^-_{\phi_i}$ modulo $<$, if and only if ϕ_j is not a minimal element under $<$ of any minimal S'_m that verifies ϕ_i.*

Under such definition, ϕ_j is never the first property inducing ϕ_i in any sub-model S'_m of S_m. Safe contraction satisfies the following rationality postulates:

1. Closure: $(S_i)^-_{\phi_i} = S_m \cap Cn(\mathcal{S}_i/\phi_i)$
2. Inclusion: $(S_i)^-_{\phi_i} \subseteq S_i$
3. Vacuity: $(\phi_i \notin Cn(\mathcal{S}_i)) \rightarrow ((S_i)^-_{\phi_i} = S_i)$
4. Success: $(\phi_i \notin Cn(\emptyset)) \rightarrow \phi_i \notin Cn(\mathcal{S}_i/\phi_i)$
5. Recovery: $(\phi_i \in Cn(\mathcal{S}_i) \rightarrow S_i \subseteq (S^-_{\phi_i})^+_{\phi_i}$

6. Extensionality: $(\phi_i \equiv \phi_j) \rightarrow (S_i)^-_{\phi_i} = (S_i)^-_{\phi_j}$

Along the lines of the interpretation of $\leq$ in terms of security and reliability in[3], 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 $\leq$ makes more expressive properties safer, removing first those with the least inferential impact. This justifies our (Anti-)Dominance axiom; with Transitivity, the following *counter-continuing* conditions hold[3]:

Proposition 3.1 (Counter-Continuing Down). *If $\phi_i < \phi_j$, and $\phi_k \vDash \phi_j$, then $\phi_i < \phi_k$, for all $\phi_{i,j,k} \in S_i$. In other words, if ϕ_i is less safe for contraction than ϕ_j (hence the former should be easier to remove than the latter) and ϕ_k is more inferentially powerful than ϕ_j (hence the former should be harder to remove than the latter), then ϕ_i is also less safe for contraction than ϕ_k.*

Proposition 3.2 (Counter-Continuing Up). *If $\phi_i \vDash \phi_j$, and $\phi_i < \phi_k$, then $\phi_j < \phi_k$, for all $\phi_{i,j,k} \in S_i$. This says that if ϕ_i is inferentially more powerful than ϕ_j (hence the former should be harder to remove than the latter), and ϕ_i is also strictly less safe from contraction than ϕ_k, then ϕ_j is strictly more safe from contraction than ϕ_k.*

3.3. Revision

Safe contraction leads to the understanding of revision of a software model S_i with respect to a new (possibly inconsistent) specification requirement ϕ_i denoted as $S^*_{\phi_i}$ as the accommodation of ϕ_i involving as little change as possible. The revision operator is then a mapping function

$$S_i \mapsto S'_m := \{(S_i, \phi_i)\} \mapsto (S_i)^*_{\phi_i}$$

from the set of theories of S_i to a new model S'_m whose implementations have languages that imply ϕ_i. Counterparts to the above rationality postulates hold for revision. According to Levi identity, revision is equivalent to contraction followed by expansion: $S^*_{\phi_i} = (S^-_{\neg\phi_i})^+_{\phi_i}$.

4. Example

It has been recently shown that most Mergesort algorithms are broken, including the Timsort hybrid algorithm[19]. 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:

```
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;
```

can be represented as a theory S_m, which among others satisfies a formula ϕ_i, which is an instance of the main loop with `stackSize`$=$ 4. A Java implementation S_i shows that the algorithm is broken with respect to such ϕ_i after violation of the invariant, `ArrayIndexOutOfBoundsException` in `pushRun`, see[19]:

```
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; // Invariant is established
            }
        }
    }
```

A new corrected model S'_m is obtained by contraction of the merging step at a too low entry in the stack and by expansion to specify the right entry in the remaining stack where the merge happens. The new implementation S'_i removes `mergeAT(n)` commands and <= predicates and adds `OR` clauses and `mergeAT(n)` commands in the appropriate loops:

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

Notice that merging of the last 3 elements of `runLen` is preserved, while the corresponding merging is not.

5. Inferential-based Resilience for System Adaptability

Resilience for a computational system reflects its (graded) ability to preserve a working implementation under varied specifications. The above analysis of software theory change allows us to provide a precise definition of resilience in the presence of failing 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 [9]. Various attempts have been made to formalise perseverance of validity to change. The most common one encountered in this research area is that of *system robustness*. One (older) interpretation is given in terms of the inability of the system to distinguish between behaviours that are essentially the same, see [31]. 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 [11]. In [5], a value-based notion of resilience is presented, according to which a prompt in a design process is resilient if it is a feature resisting all value-based relations of currently involved stakeholders and an Information System is called strongly resilient if, in view of possibly conflicting value-based future configurations, the system admits of new relations accommodating them. Resilience of software system specifications evaluated with respect to resistance to contraction operations in the relevant model induces a relation between inferential power of specifications and stronger system behaviour. Accordingly, their removal is more dangerous for system's functionality. Then the following inferential-based definition of resilience can be formulated:

Definition 5.1 (Property Resilience). *Consider property specifications $\phi_{i,j,k} \in S_m$ and a relevant implementation S_i. Then ϕ_k is more resilient than ϕ_j with respect to $S'_m = (S_i)^-_{\phi_i}$ if it is not a maximally vulnerable element of some minimal $S'_m \subseteq S_m$ implying ϕ_i.*

Accordingly, one can generalize to system resilience:

Definition 5.2 (System Resilience). *A software system specification S_m is said resilient with respect to a property specification ϕ_i if the latter is not a maximally vulnerable element in any S'_m preserving minimal functionalities of S_m.*

System resilience as the resistance to change of property specifications (as in Definition 5.1) can be essential to determine system antifragility. 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)[29]. 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[21]. Our resilient core, intended as the persistence of service delivery[22], 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.

The first property establishes that, given the non-resilient part of the system, it is possible to establish which properties will not be instantiated in any subsystem.

Proposition 5.1 (Accountability). *For any maximally vulnerable ϕ_i of some $S'_m \subset S_m$, if $\phi_i \vDash \phi_j$, then $S'_m \nvDash \phi_j$.*

The second property establishes that, given the resilient core of the system, it is possible to establish which properties will be instantiated by any subsystem.

Proposition 5.2 (Evolvability). *For any non-maximally vulnerable ϕ_i of some $S'_m \subset S_m$, if $\phi_i \vDash \phi_j$, then $S'_m \vDash \phi_j$.*

The third property establishes that, given the resilient core of the system, it is possible to know which properties will be instantiated by any of its extensions.

Proposition 5.3 (Prevision). *For any non-maximally vulnerable element ϕ_i of some $S'_m \subset S_m$, if $\phi_i \vDash \phi_j$ then there is $S''_m \subseteq S_m$ such that $S''_m \vDash \phi_j$.*

6. Conclusions

In this paper we have considered software systems as theories, whose implementations show possibly diverging postconditions from the intended specification, and modelled changes following the AGM paradigm. A definition of dynamic resilience for such systems results from safe contraction operations that focus on preserving properties of the originally intended model, while accommodating changes and make it possible to anticipate required adaptation. In future research, we will consider the update operation from the AGM model to formulate *adaptive changes* as responses to variations in the environment.

Acknowledgements

We wish to thank Nikos Gorogiannis, Balbir Barn and Kelly Androutsopoulos for their input.

References

1. Abrial, Jean-Raymond (2005). *The B-Book: Assigning programs to meanings*, Cambridge University Press.
2. Alchourrn, Carlos E., Gärdenfors, P. Makinson, David (1985). On the logic of theory change: Partial Meet Contraction and Revision Functions, *Journal of Symbolic Logic*, vol. 50, pp. 510-530.
3. Alchourrn, Carlos E., Makinson, David (1985). On the logic of theory change: Safe contraction, *Studia Logica*, vol. 44, n.4, pp. 405-422.
4. Alchourrn, Carlos E., Makinson, David (1986). Maps between some different kinds of contraction function: The finite case, *Studia Logica*, vol. 45, n.2, pp. 187-198.
5. Barn, B., Barn, R., Primiero, G. (forthcoming). Value-sensitive Co-Design for resilient Information Systems. Proceedings of IACAP2014, Conference of the International Association of Computing and Philosophy, Synthese Library Series, Springer.
6. Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T.A., Jobstmann, B. (2010). Robustness in the presence of Liveness, in Touili, Cook, Jackson (eds.), *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 6174, pp.410-424.

7. Bloem, R., Greimel, K., Henzinger, T.A., Jobstmann, B. (2010). Synthesizing Robust Systems, *Acta Informatica* vol. 51, issue 3-4, pp. 193-220.
8. Booth, R. (2001). A negotiation-style framework for non-prioritised revision. In *Proc. the 8th Conference on Theoretical Aspects of Rationality and Knowledge* (TARK2001), Siena, Italy, Jul. 810, 2001, pp.137-150.
9. 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*, Volume 17, Issue 5, pages 309–332.
10. Dam H.K, and Ghose, A. (2014). Towards rational and minimal change propagation in model evolution, CoRR, abs/1402.6046, 2014.
11. De Florio, V. (2013). On the Constituent Attributes of Software and Organisational Resilience. *Interdisciplinary Science Reviews*, vol. 38, no. 2, Maney Publishing.
12. De Florio, V. (2014). Antifragility = Elasticity + Resilience + Machine Learning Models and Algorithms for Open System Fidelity. *Procedia Computer Science*, vol. 32, pp.834 - 841, Elsevier.
13. Delgrande, J., Schaub, T., Tompits, H., Woltran, S. (2013). A model-theoretic approach to belief change in answer set programming. *ACM Transactions on Computational Logic* 14(2) (2013) Volume 8148, 2013, pp 264-276.
14. Delgrande, J., Peppas, P., Woltran, S. (2013b) AGM-Style Belief Revision of Logic Programs under Answer Set Semantics, *Logic Programming and Nonmonotonic Reasoning*, LNCS Volume 8148, 2013, pp 264-276.
15. 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), pp. 1-12, 2001.
16. 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*, LNBIP 14, pp. 186-214, 2009.
17. Floridi, L., Fresco, N., Primiero, G. (2014). On Malfunctioning Software, *Synthese*, doi: 10.1007/s11229-014-0610-3.
18. Gärdenfors, P., Makinson, D. (1988). Revisions of Knowledge Systems Using Epistemic Entrenchment, *Second Conference on Theoretical Aspects of Reasoning about Knowledge*, pp. 8395.
19. de Gouw, Stijn, Rot, Jurriaan, de Boer, Frank S., Bubel, Richard and Hähnle, Reiner (2015). OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case?. Submitted to CAV15, available at `http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf`.
20. Harman, M. and McMinn, P. and Shahbaz, M. and Yoo, S. (2013). A Comprehensive Survey of Trends in Oracles for Software Testing, University of Sheffield, Department of Computer Science, CS-13-01.
21. Koopman, P. (2003). Elements of the self-healing system problem space, *Workshop on Architecting Dependable Systems/WADS03*, May 2003. `http://users.ece.cmu.edu/~koopman/roses/wads03/wads03.pdf`.
22. Laprie, J.-C. (2008). From dependability to resilience, in Proc. I*EEE Int. Conf. on Dependable Systems and Networks*, vol. Supplemental, 2008, pp. G8G9.
23. Le, QuangLoc and Sharma, Asankhaya and Craciun, Florin and Chin, Wei-Ngan (2013). Towards Complete Specifications with an Error Calculus, in Brat, Guillaume and Rungta, Neha and Venet, Arnaud (eds.), *NASA Formal Methods*, Lecture Notes in Computer Science, vol. 7871, pp.291-306.
24. Lehman, M. (1980). Programs, life cycles, and laws of software evolution, *Proceedings of the IEEE*, 68(9), pp.101-103.
25. Lehman, M. M.; J. F. Ramil; P. D. Wernick; D. E. Perry; W. M. Turski (1997). Metrics and Laws of Software Evolution - The Nineties View, *Proc. 4th International Software Metrics Symposium (METRICS '97)*. pp. 20?32. doi:10.1109/METRIC.1997.63715
26. Lientz, B., Swanson B., *Software maintenance management*, Addison-Wesley, 1980.
27. Lindvall, M., Tesoriero, R., Costa, P. (2002). Avoiding architectural degeneration: an evaluation process for software architecture, *Proceedings of the 8th IEEE Symposium on Software Metrics*, pp. 77-86, 2002.
28. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M., (2005). Challenges in Software Evolution, *Proceedings of the 2005 Eigth International Workshop on Principles of Software Evolution*, 2005.
29. Monperrus, M. (2014). Principles of Antifragile Software, arXiv:1404.3056v2 [cs.SE].
30. 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*, vol. 26, n.5, pp.890-907, 2011.
31. Peled, D. (1997). Verification for robust specification, in Gunter, E. and Felty, A. (eds.), *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, vol. 1275, pp. 231-241.
32. Port, D., Liguo, H. (2003). Strategic architectural flexibility, *Proceedings of the International Conference on SOftware Maintenance*, pp. 389-396, 2003.
33. Sommerville, I. (2004). *Software Engineering*, 7th ed., Addison-Wesley, 2004.
34. Spivey, J Michael and Abrial, JR (1992). *The Z notation*, Prentice Hall Hemel Hempstead.
35. Williams, B.J., Carver, J.C. (2010). Characterizing software architecture changes: A systematic review, *Information and Software Technology*, vol. 52, pp.31-51.
36. Zowghi, D., Ghose, A., Peppas, P. (1996). A framework for reasoning about requirements evolution, in *PRICAI'96: Topics in Artificial Intelligence*, Lecture Notes in Computer Science, vol. 1114, pp.157-168.