Controlling Software Architecture Erosion: A Survey

Lakshitha de Silva*, Dharini Balasubramaniam

School of Computer Science, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SX, UK

Abstract

Software architectures capture the most significant properties and design constraints of software systems. Thus, modifications to a system that violate its architectural principles can degrade system performance and shorten its useful lifetime. As the potential frequency and scale of software adaptations increase to meet rapidly changing requirements and business conditions, controlling such architectural erosion becomes an important concern for software architects and developers. This paper presents a survey of techniques and technologies that have been proposed over the years either to prevent architecture erosion or to detect and restore architectures that have been eroded. These approaches, which include tools, techniques and processes, are primarily classified into three generic categories that attempt to *minimise*, *prevent* and *repair* architecture erosion. Within these broad categories, each approach is further categorised reflecting the high level strategy adopted to tackle erosion such as: process-oriented architecture conformance, architecture evolution management, architecture design enforcement, architecture to implementation linkage, self-adaptation and architecture restoration. Some of these strategies contain sub-categories under which survey results are presented.

We discuss the merits and weaknesses of each strategy and argue that no single strategy can address the problem of erosion. Further, we explore the possibility of combining strategies and present a case for further work in developing a holistic framework for controlling architecture erosion.

Keywords: software architecture, software architecture erosion, design erosion, controlling architecture erosion, survey

1. Introduction

Software systems are under constant pressure to adapt to changing requirements, technologies and social landscapes. At the same time these systems must continue to deliver acceptable levels of performance to users. Often, modifications made to a software system over a period of time damage its structural integrity and violate its design principles. As a result the system may exhibit a tendency towards diminishing returns as further enhancements are made. Such software is no longer useful for its intended purpose nor is it economically viable to maintain. Eroded software often goes through a process of re-engineering, though this may not always yield the expected benefits. The alternative is to build a replacement system from scratch, which clearly would require a sizeable investment. Moreover, software has become a key asset in organisations that sell software as well as those who use them. Ensuring these systems continue to perform adequately over long periods of time is vital for the sustainability of these organisations.

Software erosion is not a new concept. Parnas [1] argues that software aging is inevitable but nevertheless can be controlled or even reversed. He highlights the causes of software aging as obsolescence, incompetent maintenance engineering work and effects of residual bugs in long running systems. However, later works in this area such as those carried out by Huang et al. [2] and Grottke et al. [3] define software aging as the gradual degradation of performance in executing software processes due to changes in the runtime state (e.g. memory leaks). In this paper we regard erosion as the overall deterioration of the *engineering quality* of a software system (see section 2.2) during its evolution. Erosion can be thought of as a contributory factor to the kind of software aging studied by Huang et al. and Grottke et al.

*Corresponding author. Tel: +44 1334 463253
Email addresses: `lakshitha.desilva@acm.org` (Lakshitha de Silva), `dharini@cs.st-andrews.ac.uk` (Dharini Balasubramaniam)

The impact of erosion is profound when the damage affects the architecture of a software system. Software architecture [4, 5] establishes a crucial foundation for the systematic development and evolution of software and forms a cycle of influence with the organisation to which the system belongs [6]. It provides a high level model of the structure and behaviour of a system in terms of its constituent elements and their interactions with one another as well as with their operating environment. Architecture also encompasses rationale, which forms the basis for the reasoning and intent of its designers [4].

In this work we focus on *architecture erosion* which possibly plays the biggest role in accelerating software erosion. Architecture erosion is usually the result of modifications to a system that disregard its fundamental architectural rules. Although a single violation is unlikely to produce an adverse effect on the system, the accumulation of such changes over time can eventually make the software incompatible with the architecture. The effects of architecture erosion tend to be systemwide and, therefore, harder to rectify. Other forms of software degeneration such as inferior quality code are typically easier to repair.

Architecture erosion and its effects are widely discussed in literature. Perry and Wolf [4] differentiate *architecture erosion* from *architecture drift* as follows: erosion results from violating architectural principles while drift is caused by insensitivity to the architecture. As the underlying causes for both are the same, we will not consider this difference for the purpose of our survey. Additionally, the notion of software architecture erosion is discussed using a number of different terms such as architectural degeneration [7], software erosion [8], design erosion [9], architectural decay [10], design decay [11], code decay [12, 13] and software entropy [14]. Although some of these terms imply that erosion occurs at different levels of abstraction (for instance code decay may potentially be considered insignificant at the architectural level), the underlying view in each discussion is that software degenerates because of changes that violate its design principles.

Mechanisms for controlling architecture erosion have been traditionally centred on architecture repair as evident from the large body of published work (e.g. [15, 16, 17]). Architecture repair typically involves using reverse engineering techniques to extract the implemented architecture from source artefacts (recovery), hypothesising its intended architecture (discovery) and applying fixes to the eroded parts of the implementation (reconciliation). Subsequent research in this area focuses more on erosion prevention schemes (e.g. Mae [18] and ArchJava [19]) and explores concepts from other areas of computer science such as artificial intelligence to increase the accuracy of architecture discovery and recovery techniques (e.g. Bayesian learning-based recovery [20]).

This paper presents a classification of strategies for controlling architecture erosion and a survey of currently available approaches under each category in the classification. Section 2 of the paper describes architecture erosion with the aid of a few industrial examples and briefly discusses related work in the form of other surveys with a similar aim. Section 3 introduces the classification scheme while sections 4 to 9 present the survey results under this classification. For each approach, we provide evidence of adoption where available, and a discussion of efficacy and cost-benefit analysis based on our own experience and material from literature. In section 10 we discuss some of the factors that may influence the selection of an erosion control strategy along with possibilities for using strategies in combination with one another. In conclusion, section 11 outlines the state-of-the-art with respect to current strategies and presents some thoughts on future work.

2. Background

In this section we provide the context and motivation for the rest of the paper. Based on industrial case studies, we recognise that architecture erosion is often an inevitable outcome of the complexity of modern software systems and current software engineering practices, requiring approaches for controlling erosion at different stages of the software life cycle. We also introduce the terminology used in the remainder of the article and build a case for this survey by highlighting the strengths and drawbacks of previous survey attempts.

2.1. Context

The deterioration of software systems over time has been widely discussed since the late 60s debate on the "software crisis" [21, 22]. Evolving software systems gradually become more complex and harder to maintain unless deliberate attempts are made to reduce this complexity [23]. At the same time, these software systems have to be continually upgraded to adapt to changing domain models, accommodate new user requirements and maintain acceptable levels of performance [23]. Therefore, complexity becomes a necessary evil to prevent software from becoming obsolete too soon.

Complexity, however, makes it harder to understand and change a design, leading to programmers making engineering decisions that damage the architectural integrity of the system. Eventually, the accumulation of architectural violations can make the software completely untenable. Lack of rigorous design documentation and poor understanding of design fundamentals make a complex system even harder to maintain [1]. Furthermore, software architectures that have not been designed to accommodate change tend to erode sooner [1].

Architecture erosion can also result from modern software engineering practices. Architectural mismatches can arise in component-based software engineering (CBSE) due to assumptions that reusable components make about their hosting system [24]. New challenges in CBSE like trust, re-configurability and dependability create enormous demands on the architectures of evolving software systems [25]. In addition, modern iterative software development processes (such as agile programming methods) may cause the occurrence of architecture erosion sooner rather than later because they place less emphasis on upfront architectural design [9].

A number of case studies indicate that architecture erosion is widespread in the industry. Eick et al. [12] present a study of a large, 15-year old telecommunication software system developed in C/C++. They derive a set of indices from change request data to measure the extent of erosion and its impact on the system. Although termed *code decay*, the module level changes have architectural level impact. The study shows a clear relationship between erosion and increased effort to implement changes, increased number of induced defects during changes, increased coupling and reduced modularity. In another commonly cited example of architecture erosion, Godfrey and Lee [26] describe their analysis of the extracted architectures of the Mozilla web browser (which subsequently evolved into Firefox) and the VIM text editor. Both these software products showed a large number of undesirable interdependencies among their core subsystems. In fact, the badly eroded architecture of Mozilla caused significant delays in the release of the product and forced developers to rewrite some of its core modules from scratch [9, 27]. Other similar findings have been reported on popular open source projects such as FindBugs [28], Ant [8] and version 2.4 of the Linux kernel [9].

The impact of architecture erosion is far reaching and has an associated cost. In the worst case an eroded software system that is not salvageable requires complete re-development. Even if a system does not become unusable, erosion makes software more susceptible to defects, incurs high maintenance costs, degrades performance and, of course, leads to more erosion. Consequently, the system may lose its value, usefulness, technical dominance and market share, as experienced by the Mozilla web browser. Mozilla lost its leading position and a large portion of the browser market to Microsoft's Internet Explorer mostly due to its inability to introduce new features on time [27].

2.2. Terminology

In this paper we define *architecture erosion* as the phenomenon that occurs when the *implemented architecture* of a software system diverges from its *intended architecture*. The implemented architecture is the model that has been realised or built-in within low-level design constructs and the source code. The term concrete architecture also refers to the implemented architecture [29]. The intended architecture is the outcome of the architecture design process, also known as the conceptual architecture [29] or the planned architecture [7]. The divergence itself is not caused by malicious human actions (though it could be intentional), but rather by routine maintenance and enhancement work typical for an evolving software system.

Architecture erosion leads to the gradual deterioration of the engineering quality of software systems. For the purpose of this survey we define engineering quality as a subsumption of architectural integrity (i.e. completeness, correctness and consistency), conformance to *quality attribute* requirements [6], adoption of sound software engineering principles (e.g. modularity, low coupling, etc.), and adherence to architecture rationale and corresponding design decisions. The loss of engineering quality makes software hard to adapt [4] and, therefore, become less useful over time.

Engineering quality of a system may not always equate to the quality of system performance. A well-performing system may have a badly eroded architecture. However, such a system is extremely fragile and has a high risk of breaking down whenever modifications are made. Similarly, a well-engineered system may not perform as expected because its architecture has not been designed to cater to user requirements. We do not consider an architecture incompatible with user requirements an eroded architecture. However, such an architecture erodes faster if developers resort to ad-hoc repairs to match requirements instead of addressing fundamental design issues. On the other hand, it is possible to progressively improve a flawed architecture with carefully crafted upgrades to the system. The topic of correcting original design flaws in software architectures during or after implementation is not covered in this work.

2.3. *Other surveys*

A number of previously published articles have discussed the topic of architecture erosion and various methods available for dealing with the issue. In this section, we consider existing surveys of techniques for controlling architecture erosion and their advantages and limitations.

In their survey, Hochstein and Lindvall [7] focus comprehensively on tools available at the time for detecting architectural erosion at source code level and for applying refactoring transformations to repair erosion. In addition, they discuss possible approaches for preventing architecture erosion though the paper does not provide a classification of these approaches.

Pollet et al. [30, 31] present a taxonomy that includes goals, processes, inputs, techniques and outputs for reconstructing eroded architectures. The emphasis of this taxonomy is on restoration and therefore, the authors do not cover techniques for minimising and preventing erosion.

Stringfellow et al. [13] evaluate an architecture reverse engineering method that leverages data from version control systems and change requests against other reverse engineering techniques. This work focuses on detecting and subsequently repairing architecture erosion through architecture recovery approaches.

Knodel and Popescu [32] present a comparison of architecture compliance checking methods using a visualisation tool. They evaluate the effectiveness of reflexion models, relation conformance rules and component access rules in detecting architecture erosion using an air-traffic control system. A similar comparison of reflexion models, dependency structure matrices and source code query languages is described in a contribution by Passos et al. [33].

Dobrica and Niemela [34] present a survey of architecture analysis methods developed over the years along with a simple framework for comparing, contrasting and evaluating these methods. Finally, O'Brien et al. [35] survey architecture reconstruction approaches used in the industry but do not classify these methods.

A common factor among most of these surveys is that they do not offer a classification of the techniques, which is useful for analysing their effectiveness and applicability for particular scenarios. A number of existing works also have a narrow focus with their discussions centred around specific strategies. In contrast, our survey provides a wider coverage in terms of different approaches available for controlling erosion and number of techniques within a given approach. We also use our classification framework to discuss the relative benefits and drawbacks as well as potential applicability of the various approaches included in the survey.

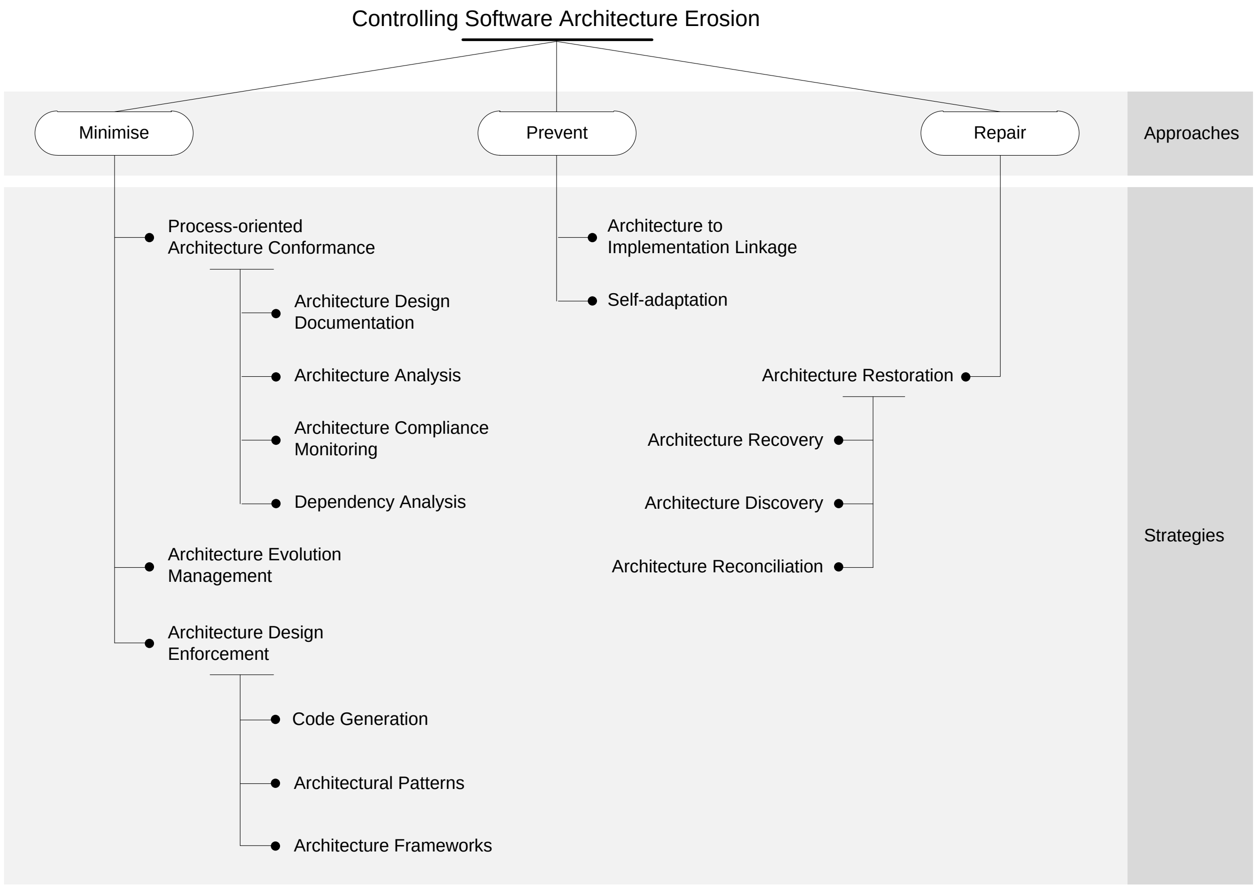
3. Classification

As the first step of our survey, we classify existing approaches for controlling architecture erosion into three broad categories depending on whether they attempt to *minimise*, *prevent* or *repair* erosion. Each of these categories contains one or more sub-categories based on the high-level strategies used to realise its goal. Some of these sub-categories are further divided indicating the specific strategy adopted. Figure 1 illustrates this classification framework through which the results of the survey are presented.

At the top-most level, the *minimise* approach contains those strategies which are targeted towards limiting the occurrence and impact of architecture erosion but may not be effective in eliminating it. On the other hand, strategies that fall into the *prevent* category aim to completely eradicate erosion. The third set of strategies attempt to *repair* the damage caused by erosion and reconcile the implementation with its architecture.

The second level of the classification framework lists specific strategies under each of the top level categories. These are briefly explained below.

- MINIMISE
  - ◇ **Process-oriented Architecture Conformance** includes software engineering processes that ensure architecture conformance during system development and maintenance activities.
  - ◇ **Architecture Evolution Management** covers methods available for managing the evolution of architectural specifications in parallel with implementation artefacts.
  - ◇ **Architecture Design Enforcement** incorporates methods and tools available for transforming architectural models into implementation.
- PREVENT
  - ◇ **Architecture to Implementation Linkage** includes mechanisms that associate or embed architectural models within source code and support the ability to monitor architectural compliance at runtime.
  - ◇ **Self-adaptation** technologies enable systems to reconfigure themselves to align with their architectures after a change has been made to either the implementation or the runtime state.
- REPAIR



Controlling Software Architecture Erosion
Minimise
Prevent
Repair
Approaches
Process-oriented Architecture Conformance
Architecture Design Documentation
Architecture Analysis
Architecture Compliance Monitoring
Dependency Analysis
Architecture Evolution Management
Architecture Design Enforcement
Code Generation
Architectural Patterns
Architecture Frameworks
Architecture to Implementation Linkage
Self-adaptation
Architecture Restoration
Architecture Recovery
Architecture Discovery
Architecture Reconciliation
Strategies

Figure 1: *Classification framework of existing methods for controlling architecture erosion*

◇ **Architecture Restoration** covers methods available for recovering and reconciling an eroded architecture with its intended architecture.

The next six sections present the survey results for each of the above strategies. We conclude each section by discussing the adoption, efficacy and cost-benefit analysis of its class of strategies.

4. Process-oriented Architecture Conformance

Architecture conformance, which is vital for minimising architecture erosion, is generally achieved through process-centric activities during software development. Literature identifies a number of reasons for architecture erosion that relate directly to human and organisational factors. Parnas [1] highlights inadequate design documentation, misunderstood design principles and poor developer training as key triggers of erosion. Similarly, Eick et al. [12] argue that vague requirements specifications, poor architectural designs and programmer variability, among other factors, lead to architecture erosion.

The ability to ensure architecture conformance during development and maintenance is built into most formal software development processes such as the Rationale Unified Process [36, 37] and the Open Unified Process (part of the Eclipse Process Framework) [38]. These processes incorporate architecture reviews to ensure that the architecture meets user requirements, design reviews to confirm that designs adhere to architecture guidelines and code reviews to check that architectural principles are not violated in program code. Similarly, change requests are reviewed and approved by architects to ensure maintenance updates are agreeable with the intended architecture.

To address the issue of programmer variability, software processes often include some form of skill-gap analysis to identify the training needs of new members inducted to a project team. Junior team members are also paired with senior developers as a scalable mentoring and review mechanism in large teams. Some of these process activities are usually supplemented by process automation tools in order to increase productivity and reduce human-induced errors.

Software engineering processes incorporate the following strategies to enhance the effectiveness of controlling architecture erosion:

- Architecture Design Documentation,
- Architecture Analysis,
- Architecture Compliance Monitoring, and
- Dependency Analysis

We discuss the survey results under the above categories in sections 4.1 to 4.4.

4.1. Architecture Design Documentation

Addressing Parnas' concern that poor design documentation can lead to architecture erosion, mature software processes prescribe uniform and rigorous documenting of architecture specifications. In order to be useful in understanding the architecture, documented specifications have to be both sufficiently detailed for analysis and adequately abstract for understanding [39]. It should be noted that architecture documentation does not necessarily require a process framework. However, this activity is often made part of a software engineering process, which is useful in ensuring that the documentation artefacts are up to date and well catalogued.

Popular architecture documentation techniques focus primarily on recording system structure, its interactions with the operating environment and its transactions with the organisation. They use a combination of views, viewpoints (i.e. a perspective of a view) and mappings among views to capture different aspects of an architecture. The 4+1 View Model [40] and Views and Beyond [39] provide a strong foundation for view-based documentation of architecture while Bachmann et al. [41] describe a technique for documenting system behaviour using a number of different notations such as state charts, ROOMcharts and collaboration diagrams.

Software architectures are often specified using informal notations that lack precise definition. This informality may cause misinterpretation of the architecture that, in turn, could lead to erosion. An architecture description language (ADL) [42] attempts to overcome this problem by providing a formal and precise notation with well-defined semantics for describing an architecture. In addition, ADLs serve as a platform for architectural visualisation, analysis and validation as well as generating implementation. A number of ADLs have been developed over the years, each with its own characteristics and capabilities geared towards addressing certain architectural aspects. However, most ADLs are conceptually based on the structural architecture primitives of components, connectors, interfaces and configurations [42]. Examples of well-known ADLs include Acme [43], Darwin [44] and xADL [45] all of which model structural properties while Rapide [46] was developed to model dynamic properties of systems. Further discussions covering numerous ADLs can be found in a survey by Clements [47] and in a work describing a comprehensive classification framework for ADLs by Medvidovic and Taylor [42].

The IEEE 1471-2000 standard (also known as ISO 42010), titled the *Recommended Practice for Architectural Description of Software-Intensive Systems* [48], provides a comprehensive set of guidelines, as recommendations, for describing software architectures. This standard prescribes *how* an architecture should be documented and *what* should ideally be included in such a description. However, it was not the aim of IEEE 1471 to suggest an ideal architecture or a notation for specifying architectures. The conceptual framework defined in IEEE 1471 requires an architecture to be described primarily in terms of stakeholders, their concerns, views and viewpoints.

Besides architecture specifications, documentation of architectural design decisions is not widely practised though a few techniques have been developed for this purpose. Tyree and Akerman [49] propose an approach that uses document templates to systematically capture architecture design decisions and their rationale. A post-design approach for capturing design decisions is discussed by Harrison et al. [50] where patterns in the architecture are analysed to capture the rationale that motivated them.

4.2. Architecture Analysis

Architecture analysis methods formalise architecture evaluation and review. The Architecture Trade-off Analysis Method (ATAM) [51] allows architects to evaluate the intended architecture with respect to user requirements, quality attributes, design decisions and design trade-offs, and associate priorities and risk levels with important scenarios. The outcome of an ATAM analysis is a utility tree that presents a comprehensible view of the risks, sensitivity points and non-risks in the architecture. Industrial case studies (e.g. the Linux case study by Bowman et al. [52] and a study of the Space Station operations control software by Leitch and Strouli [53]) indicate that risks and sensitivity points in software architectures are more susceptible to erosion. In particular, developers may attempt to correct system issues related to quality attributes (e.g. poor performance or inadequate security) using code fixes without realising that these issues could well be related to design flaws in the architecture.

A number of other architecture analysis methods have been developed including the Scenario-based Architecture Analysis Method [54] and the Software Architecture Evaluation Model [55]. Complementary to scenario-driven analysis, a concern-based analysis method is introduced by Tekinerdogan et al. [56]. In this work, the authors rely on dependency structure matrices (DSMs) of the architecture to derive concerns important to stakeholders using a mapping scheme. The extracted concerns form the basis for analysis and refactoring. Lindvall et al. [57] and Tvedt et al. [58] describe a process for evaluating the quality of an architecture from a maintainability perspective. Finally, an analysis technique using Coloured Petri Nets for validating architecture quality attributes is proposed by Fukuzawa and Saeki [59].

4.3. Architecture Compliance Monitoring

An important aspect of process-oriented architecture conformance is checking whether the implemented architecture complies with the intended architecture in order to identify potential erosion problems. A software process that includes regular compliance monitoring activities can keep the implementation faithful to the intended architecture as the system develops.

The reflexion models technique [60, 61] compares an extracted model of the implemented architecture and a hypothetical model of the intended architecture using an intermediary mapping defined by a human evaluator. The hypothetical architecture, in the absence of documented architecture specifications, is usually modelled by observing external system behaviour. The computed outcome is the reflexion model that identifies places in the implemented architecture where there are deviations or omissions from the hypothetical design. The comparison process is mostly a human task although tool support is available for refining mappings and visualising reflexion models. This technique depends on other tools such as call graph analysers or dependency checkers to build a model representative of the implemented architecture.

Rosik et al. [62] discuss an adaptation of reflexion models called inverted reflexion models. In this method the intended architecture is presumed available and therefore not hypothesised. A model of the implemented architecture is repeatedly refined as development progresses, and compared against the intended architecture. Thus, inverted reflexion models appear well-suited for compliance checking while generic reflexion models are useful for architecture recovery.

Postma [63] presents a method similar to reflexion models, but uses relationship constraints among architecturally significant modules to verify conformance between implementation and its intended architecture. The implemented architecture is derived from source code, but the mappings and constraints are automatically generated by tools using formal models of the intended architecture and input from architects. The use of source code query languages [64] to validate architecture conformance is briefly described by Passos et al. [33] with the help of the Semmle .QL language [65]. In this method the constraints in the architecture are specified as a set of .QL predicates that are then used to infer whether these constraints have been violated in the source code.

4.4. Dependency Analysis

A growing body of work, particularly as commercially available tools, exists for checking dependency among modules and classes using implementation artefacts [66, 67, 68, 69, 70, 71, 72, 73]. Dependency analysis can expose violations of certain architectural constraints such as inter-module communication rules in a layered architecture, thus identifying potential instances of erosion. Therefore, similar to compliance monitoring approaches, these tools play an important role in software development processes by providing a means for automating architecture conformance assurance.

A method based on DSMs is proposed [74] to allow dependencies between software modules (or subsystems) to be modelled using a matrix where columns of the matrix specify dependents and rows contain modules that are depended upon. An interesting capability of this technique is that simple design rules like "module X should call module Y" can be applied to the matrix and automatically validated against the implementation. The dependency chart in the matrix can be re-organised using *partitioning* algorithms to expose the "most used" modules and "most provided" modules which are often useful in understanding the layering constraints of an architecture. The DSM technology is available as part of a commercial product suite from Lattix [66].

Another commercial product, SonarJ [67], uses XML specifications of the intended architecture to detect dependency violations during and after development. A sister product named Sotograph [68] performs architecture conformance validation using a repository of implementation data and monitors the structural evolution of architectural models between versions of a software system. Structure101 [69] provides a visual representation of module dependencies at various levels of abstractions of an implemented architecture allowing architects to identify where undesirable dependencies may have occurred.

A number of other tools such as Axivion Bauhaus [70], Klocwork Architect [71], Coverity [72] and JDepend [73] provide various forms of dependency analysis, quality metrics and visualisation support that are all useful in architecture conformance assurance. Finally, Huynh et al. [75] present a technique for automatic conformance checking using a DSM representation of the source code obtained from the Lattix tool and a DSM model of the intended architecture. This technique is targeted towards effectively checking violations of modularity principles in architectural designs.

4.5. Discussion

Table 1 highlights the contribution of each strategy in the process-oriented category towards controlling architecture erosion.

Adoption: Only a few of the aforementioned process-oriented techniques have been reasonably well adopted in the industry. In particular, architecture design documentation (except for ADLs, which are not in widespread use) is included in most software processes due to its applicability across a wide range of software projects. While documentation alone will not stop architecture erosion, process activities such as design and code reviews are able to identify architectural non-conformance. There is also evidence of a drive towards promoting architecture analysis within mainstream software development processes [76], though they may not always be the formal approaches discussed earlier. However, despite the availability of a large number dependency/static analysis tools, recent industry surveys indicate these are not extensively used in projects for checking architectural conformance at code level [77]. Developers still rely on manual reviews for this purpose which do not scale well in most situations. Similarly, we do not find adequate evidence in the literature to indicate the widespread use of compliance monitoring methods in industry.

Efficacy: Strategies in this group are widely applicable means for minimising architecture erosion. They are particularly effective in compliance monitoring of certain non-functional attributes in architectures (e.g. adherence to standards or time-to-market). These methods are also useful in identifying early stages of architecture erosion where minor violations of architectural principles take place. This allows the implemented architecture to be kept in step with the intended architecture during initial system development. The effectiveness of most process strategies, however, is largely dependent on human factors. Process guidelines are often ignored and process checks bypassed in favour of achieving project deadlines or cutting costs. In such instances process strategies cannot be relied upon to control architecture erosion.

Cost-Benefit Analysis: Process-oriented strategies are associated with a moderate cost/benefit ratio. The effort of implementing and supporting them over a long period of time during system evolution is fairly high. At the same time their efficacy is highly susceptible to human negligence. The biggest advantage these techniques provide over other strategies is simplicity. An or-

Strategy	Contribution towards controlling erosion
Architecture Design Documentation	Records architecture design and rationale with the intent of disseminating architectural knowledge to a wider audience and provides a point of reference for developers throughout system evolution.
Architecture Analysis	Uncovers weaknesses in the intended architecture, in particular, sensitive points which can be easily violated in the implementation.
Architecture Compliance Monitoring	Establishes the means to validate whether the implementation is faithful to the intended architecture during both the development and subsequent maintenance phases of a system.
Dependency Analysis	Exposes possibly the most common form of architectural violations by detecting dependencies among components in the implementation that break constraints in the intended architecture.

Table 1: *Controlling architecture erosion with process-oriented strategies*

ganisation can use existing resources to implement most of these techniques.

Evaluation: Using processes and tools together is a viable and effective solution to architecture erosion. Besides addressing erosion, the engineering rigour introduced by this class of techniques provides a comprehensive set of supplementary benefits such as reducing defects, increasing predictability and improving reliability of the software. Additionally, these engineering processes align well with project management practices used for tracking progress, identifying risks, etc. However, there is an inherent resistance from programmers to heavy engineering processes and process guidelines are often violated to satisfy project constraints. Scaling process tasks with respect to architecture conformance assurance can be a challenge as core architectural knowledge is often retained within a small group of individuals. Rigorous process activity is also not cost effective during the maintenance phase of some systems when only a small team is retained. Therefore, we believe that while processes are necessary for the effective execution of software projects, they alone may not provide an adequate solution to control architecture erosion.

5. Architecture Evolution Management

Architecture erosion can be controlled by carefully managing the evolution of the architecture and the software system as a whole. Often evolution management is performed with the help of software configuration management (SCM) tools [78], also known as version control systems. A number of methods have been developed that exploit the capabilities of SCMs to specifically control, trace and record changes made to models of an intended architecture and its corresponding implementation. It is important to note that systems that are truly capable of managing architecture evolution do not merely perform versioning of architectural artefacts but rather can interpret the architecture models at a semantic level. There are two classes of tools found in this domain that are of interest. In the first class, the SCM plays the central role by facilitating the specification of architecture within itself. This is called an *SCM-centred architecture* [79]. In more elaborate evolution management systems the SCM becomes part of the architecture itself. This second class is termed an *architecture-centred SCM* [79]. Specific techniques presented in the remainder of the section fall under one of these two classes.

An architecture-centred SCM technique introduced by van der Hoek et al. [80, 18] is an architecture evolution environment called Mae. It combines SCM principles and architectural concepts in a single model so that changes made to an architectural model are semantically interpreted prior to the application of SCM processes. Mae uses a type system to model architectural primitives and version control is performed over type revisions and variants of a given type. Mae can also be coupled with an ADL to specify architecture models and is currently part of the ArchStudio [81] visual architecture modelling toolset. An important part of the architecture erosion problem that has been addressed by this tool is enabling the architecture itself to be updated in addition to managing changes to the implementation. Ensuring that the intended architecture and implementation are consistent with each other during system evolution is imperative to constraining architecture erosion.

ArchEvol [82] is a SCM-centred tool that enables monitoring the co-evolution of architecture specification and its corresponding implementation. A key aspect of this approach is its use of existing tools with tight integration among them. Although ArchEvol was implemented using ArchStudio, Eclipse [83] and Subversion [84], its conceptual model is independent of any specific architecture modelling, programming and SCM environments. ArchEvol allows architects and programmers to work in parallel on their respective domains of the system and takes responsibility for maintaining consistency between architectural design artefacts and program code. Architecture specifications and Java code are linked through two specifically developed plug-ins in Eclipse and ArchStudio. The system uses WebDAV as the medium to connect Eclipse and ArchStudio with Subversion. While the practicality of this approach appears promising, its treatment of architecture as a structure consisting of only components and connectors might limit its usefulness in controlling erosion. Other types of architectural elements such as ports and interfaces may also be points of erosion.

ArchAngel [85] is part of a lightweight method for minimising architecture erosion. This SCM-centred tool monitors the architectural conformance of the source code against a dependency model defined by the architect. It parses Java code using the ANTLR [86] tool to identify package relationships and compares these relationships to the dependency model of the intended architecture. Any relationship that does not conform to the model triggers a fault. The SCM (CVS [87] in this case) plays the role of informing the ArchAngel tool when source files are edited. Although this tool could be effective in detecting dependency violations, it does not have the capability of reducing erosion arising from other types of architecture violations or tracking the evolution of an architecture model with semantic interpretation.

Finally, the Molhado [88] tool is somewhat similar to ArchEvol but can be considered an architecture-centred SCM tool with additional support for interface elements. This system combines SCM concepts such as version control with architectural principles into a unified model to help track “unplanned changes” to a system’s architecture or implementation.

Discussion

Techniques in this group attempt to minimise architecture erosion by managing changes to both the implementation and the architecture in parallel while ensuring they remain faithful to each other during the evolutionary process.

Adoption: There is no evidence of widespread adoption of these strategies in industry. However, a number of published academic works exist that either extend the techniques discussed earlier (e.g. MolhadoArch [89]) or explore new avenues with the use of SCMs [90].

Efficacy: The effectiveness of these strategies in controlling architecture erosion is expected to be quite high. Their ability to monitor and control changes to an architecture specification and its implementation artefacts addresses a root cause of architecture erosion. By validating the implementation against the architecture after a change has been made to either artefact, evolution management techniques leave little room for the implementation to diverge from the intended architecture.

Cost-Benefit Analysis: The effort required for implementing an evolution management strategy is rather high because of the complex nature of most architecture specifications. The SCM tools in these systems should be able to interpret architectural models if they are to effectively compare architecture with implementation. Thus the SCM is most likely to add another level complexity to the architecture modelling process. However, the potential benefits may outweigh this extra complexity.

Evaluation: Evolution management techniques are not widespread in the industry although they would appear to be an effective strategy for controlling architecture erosion as almost every non-trivial software project uses some form of SCM. A possible reason for the lack of adoption could be the advanced capabilities required by SCM tools to meaningfully monitor the evolution of architectural models. While this capability itself is not hard to implement, the myriad of languages and other methods employed to specify architectures makes evolution management practically ineffective. Furthermore, software systems using these techniques have to rely on an external entity, which is the SCM tool, to ensure the implementation and the intended architecture do not diverge. In this arrangement the SCM tool has to be made aware of architectural rules, which creates a binding between the two, adding to the complexity of the evolution process. We suggest that an architectural specification and its implementation should instead have a conformance mechanism that does not require support from a third party.

6. Architecture Design Enforcement

Architecture design enforcement entails guiding the design and programming activities using formal or semi-formal architectural models of the target system. This approach, as exemplified by the much discussed Model Driven Architecture (MDA) [91] paradigm, is another strategy for minimising erosion. Typically, architecture enforcement is established at the beginning of the implementation process where the architectural model is transformed directly into source code or Unified Modelling Language (UML) models that form the basis for further development. Alternatively, the intended architecture of the system is refined using architecture patterns or frameworks that impose certain constraints on the implementation.

We identify the following sub-categories within this class.

- Code Generation
- Architecture Patterns
- Architecture Frameworks

These approaches are discussed within the context of the survey in sections 6.1 to 6.3.

6.1. Code Generation

Code generation tools, also known as generative technologies [92], are common in most integrated development environments (IDEs) available today. Generators use a specification of the intended architecture to produce code stubs, or skeleton classes in the case of an object-oriented language, that are representative of the intended architecture. However, code generated using architectural models does not produce useful services without additional work by a human programmer, although more detailed design specifications such as UML models can, in theory, produce code for complete implementation. Furthermore, the generated code could subsequently diverge from the architectural models originally prescribed. To overcome this *round-tripping problem* [92], the different tools used for architecture modelling, code generation and source code editing require tight integration and possibly a shared protocol for communicating architectural properties with one another.

A few ADLs have associated tools to generate implementation code from architectural specifications. The ArchStudio [81] environment contains a code generator that produces template code for xADL [93] interface specifications and other architectural constructs which can be edited by a programmer. The tool has the capability to detect whether the edited code deviates from the architecture specification. Similarly, the C2 ADL toolkit [94] produces class hierarchies in Java, C++ and Ada, while Aesop [95] generates the same in only C++. The generated class hierarchies form the basis for concrete implementation [96], though unlike ArchStudio, C2 and Aesop do not address the round tripping problem.

In a more recent work, Stavrou and Papadopoulos [97] describe a methodology for generating executable code from architecture models represented in UML 2.0 notations. The technique capitalises on enhancements in UML 2.0 to model dynamic and structural aspects as architectural level abstractions that can be transformed into code. Component diagrams are suggested for modelling static structures and sequence diagrams for modelling behaviour. An accompanying tool generates these UML models into Manifold source code. Manifold is a coordinating language for managing interactions between independent concurrent processes in complex systems [98]. It should be feasible to adopt this method to support mainstream software development languages such as Java, C++ or C$^{\#}$, although it is not discussed in the publication.

6.2. Architecture Patterns

Similar to design patterns, architecture patterns [99] are well-established and documented solutions to recurring architectural design problems. Besides the solution, architecture patterns record the design decisions and their rationale along with the context in which the pattern is applicable. Patterns are also named and catalogued, allowing architecture knowledge to be communicated and shared with minimal ambiguity. An architecture designed incorporating patterns has several key advantages in terms of minimising architecture erosion. Firstly, patterns are robust solutions that have been tried and tested. The opportunities for a correctly applied architectural pattern to violate design principles are low, thereby reducing the possibility of erosion. Secondly, patterns communicate design decisions at a level that is appropriate to programmers implementing the code. This communication is vital during system evolution, as a common cause of erosion is unavailable or misunderstood design decisions.

A limitation of architecture patterns is that their rules can be easily broken, particularly when they are configured upon application. Conflicting system requirements such as performance and maintainability often encourage programmers to make sub-optimal design decisions that disregard the constraints of an already applied architectural pattern.

Examples of common architecture patterns include the Model-View-Controller [100], Pipe and Filter [99], and the Blackboard model [101].

6.3. Architecture Frameworks

Architecture frameworks [102] (also known as architecture implementation frameworks [92]) provide a comprehensive set of tools to systematically guide the design, implementation and evolution of software systems. Most architecture frameworks provide some core functionality that can be reused or extended to build applications with the support of accompanying tools, guidelines, testing procedures and management processes. More advanced frameworks provide ready-to-use generic services such as logging or communication middleware, while some others have been designed for building domain specific applications like e-commerce. Despite this variance, all architecture frameworks guide software engineering activities using well-defined rules and contain mechanisms to detect violations of these rules. These frameworks encompass mature architectural styles and best practices that encourage good design, enforce programming consistency, minimise programmer variability, form the basis for architectural analysis and enable traceability to requirements. Thus, architecture frameworks can play an important role in managing the evolution of software and minimising architecture erosion.

Similar to architecture patterns, architecture frameworks require significant amounts of configuration when used in real-world software development projects. The generic nature of these frameworks leave room for misconfiguration which can become a source of architecture erosion.

A number of architecture frameworks have been used in industrial systems and academic research. Popular architecture frameworks such as Spring [103] and

Strategy	Contribution towards controlling erosion
Code Generation	Transforms architectural design into program stubs that accurately capture the structural properties of the intended architecture, thereby avoiding mistakes that can be made by humans when programming to an architecture.
Architecture Patterns	Provide a robust, well-understood path for implementing the architecture while communicating key design decisions to lower level abstractions with minimum overhead, thus making programmers aware of architectural properties.
Architecture Frameworks	Provide a comprehensive set of services to guide the implementation according to the intended architecture and may contain some capabilities to detect when the implementation diverges from the architecture.

Table 2: *Controlling architecture erosion with design enforcement strategies*

Struts [104] are called *application frameworks* since they support a significant number of core services required to build industry-strength applications. Both of these frameworks provide extensive support for commercial web-based software systems. Examples of enterprise class architecture frameworks that cover organisational and business factors include the Open Group Architecture Framework (TOGAF) [105], the US Department of Defense Architecture Framework (DoDAF) [106] and the Zachman Framework for Enterprise Architecture [107].

6.4. Discussion

The key strengths of the three design enforcement strategies presented with respect to reducing architecture erosion are summarised in Table 2.

Adoption: The adoption of some of the techniques surveyed under this section is quite widespread in the industry, possibly due to their simplicity and applicability to architectures of mainstream software systems. In particular, architecture patterns and frameworks are used extensively in commercial software development [108]. This is evident from the strong emphasis given to patterns and reference architectures by the Java and .NET developer communities [109, 110]. On the contrary, code generation from architecture specifications does not appear to be widely taken up, possibly due to ADLs and MDA not having significant usage in industry. Most IDEs are capable of generating code from intermediate design artefacts such as UML models, though these models usually do not capture a sufficient degree of architecturally meaningful information.

Efficacy: The effectiveness of design enforcement strategies in controlling architecture erosion depends largely on how precisely an architectural design can be transformed into a model representative of the implementation. If the transformed model is inaccurate, then at this point the implemented architecture has already diverged from the intended architecture. Therefore, architecture patterns and frameworks must be carefully selected to complement other system requirements during architectural design. Similarly, code generation techniques can be effective only if the target language has sufficient constructs to support architectural abstractions.

Cost-Benefit Analysis: Although the cost of employing these strategies is fairly low, their effectiveness in controlling erosion is not high as these techniques will not address the problem of erosion in its entirety. Architecture patterns and frameworks, however, provide the added benefit of software reuse, which contributes towards increased productivity and better quality software.

Evaluation: Propagation and enforcement of architectural properties at source code level are fundamental in ensuring that the implementation does not deviate from the architecture. In this respect, architecture design enforcement techniques present an opportunity to address the problem of erosion effectively. However, a key practical issue faced by enforcement techniques is retaining the initial fidelity between the architecture and the implementation when either entity evolves. Of particular importance is adapting the implementation to changes made in the architecture itself, which may require undoing, for instance, some previously generated code or architecture patterns.

We also see limited scope in these techniques as not all architectural properties or principles can be enforced by the current state-of-the-art. For example, communication integrity between tiers in a layered architecture can be enforced with generated code, but mainstream programming languages are not equipped to block violations of these communication rules if programmers make changes to the code. Industrial case studies [27, 111, 112, 113] show that these limitations in patterns and frameworks are intentionally exploited to satisfy competing architectural demands such as increasing performance or reducing time-to-market. One such common exploit, typically for the sake of performance, is making direct calls to the database from the presentation layer, bypassing the data-access layer which isolates the presentation logic from intricacies of data storage. Such practices negate the contributions made by architecture styles and frameworks towards controlling erosion. Therefore, unless programming languages are radically changed to model architectural concepts effectively, design enforcement strategies will not be completely effective in controlling erosion.

7. Architecture to Implementation Linkage

Significant research has been carried out with the goal of preventing architecture erosion by linking specifications of an intended architecture to the code that reifies that architecture. A key difference between architecture to implementation linkage and architecture design enforcement techniques like code generation is that linkage approaches attempt to establish a continuous correlation between the architecture and the implementation. The association is often bi-directional, enabling both the architecture and the implementation to evolve while ensuring conformance rules are not broken in the process. Therefore, opportunities for the implementation to deviate from its architecture are effectively eliminated.

ArchJava [19], developed by Aldrich et al., unifies a formal architecture specification and its implementation into a single entity. Software architectures specified using the formal notation of an ADL typically do not contain implementation details. ArchJava departs from this practice by allowing program code to reside within an architectural specification. An ArchJava specification can be thought of as an *executable architecture* due to embedded executable code. ArchJava is intended to emphasise the enforcement of *communication integrity* so that the implementation adheres to communication constraints specified among modules in the architecture. The ArchJava syntax, which is similar to that of Java, allows structural and some dynamic aspects of software architecture to be modelled. Standard Java code is included within an ArchJava specification to implement the classes and methods from the architecture description. The ArchJava compiler interprets the ArchJava specific syntax elements in the source files in order to validate communication integrity and other architectural constraints imposed by the language. The standard Java compiler then compiles the embedded Java code in the conformant ArchJava specification. ArchC# [114] is an adaptation of ArchJava that supports distributed architectures.

The Archium platform [115] consists of a compiler that processes an extension to Java, and a framework that checks runtime architectural properties. Archium was primarily designed to explicitly model architectural design decisions as part of an architecture specification and to ensure that the implementation is conformant with these decisions. The Archium strategy treats software architecture as a composition of design decisions that could possibly "vaporise" if they were not adequately captured, leading to architecture erosion. Similar to ArchJava, the Archium compiler translates source specifications to Java code before invoking the Java compiler. The compiled binaries are executed together with the Archium framework that monitors the consistency of the runtime architecture of the program.

ArchWare [116, 117] provides a complete architecture-centric platform for developing evolvable software systems. The platform consists of an ADL for specifying executable architectures, an associated set of tools, a runtime monitoring infrastructure and a generic model for architecture-driven software processes. ArchWare aims to support *active architectures*, whereby the architectural model is the system implementation. This unification enables an executing system to be dynamically evolved at the architectural and implementation levels simultaneously. The ArchWare platform is further enhanced by an architecture analysis language for validating dynamic and structural properties, and an architecture refinement language to minimise the gap between high level architectural models and the implementation. Altogether, the ArchWare suite accounts for a rigorous process for enacting architecture into implementation and subsequent evolution with the aim of preventing architecture erosion.

Researchers have long been investigating runtime conformance checking and runtime evolution management as possible approaches for stopping architecture erosion. Early work on architecture-driven runtime software evolution was carried out by Oreizy et al. [118]. Although their approach primarily intends to address the issue of evolving executing systems that are too critical for maintenance shutdowns, the authors emphasise "controlling change to preserve system integrity". A runtime platform called the Architecture Evolution

Manager (AEM) plays the central role of propagating changes in the architectural model to its corresponding implementation. The AEM ensures that architectural constraints are not violated in this process. This approach relies heavily on software connectors and implicit knowledge of the C2 architectural style, allowing components to be added, removed and updated without affecting the global runtime state of the system. The state-of-art of runtime evolution management and its future direction are discussed in a follow-up paper a decade later [119].

Discussion

Techniques that link architecture with implementation provide a strong mechanism to prevent architecture erosion by bonding the implementation to the architecture.

Adoption: Besides a few experimental applications [120, 121], we have not found evidence of widespread adoption of these techniques in the industry.

Efficacy: These technologies have the potential to be highly effective in preventing architecture erosion. The very fact that there are continual and explicit links between elements of the intended architecture and the code that implements that architecture enables both entities to evolve independently and still remain faithful to each other when the adaptation is complete. A change in the implementation that breaks an architectural rule will trigger an immediate reaction from the monitoring system. Similarly, an update to the architecture that breaks the implemented code will be easily detected. In the case of Archium, avoiding neglect of architectural design decisions can further strengthen the bond between architecture and implementation.

Cost-Benefit Analysis: Available architecture to implementation linkage techniques appear fairly costly in terms of modelling complexity and the additional runtime infrastructure they impose on software systems. These techniques are also disruptive, in the sense that they assume fundamental shifts in mainstream software engineering practices. However, this cost is worthwhile as these techniques provide possibly the best mechanism to control architecture erosion.

Evaluation: Unlike design enforcement techniques, methods that link architecture to implementation usually do not suffer from the round-trip problem. Runtime environments and the supporting tools that accompany these approaches continuously monitor associations between the architecture and its implementation. Despite these benefits, we can only conjecture a numbers of possibilities for the lack of adoption of linkage techniques.

An essential criterion for almost every strategy in this group is that a fairly complete architecture specification must be produced beforehand. Although upfront architectural design is a cornerstone in MDA, it conflicts with the more pragmatic iterative development techniques popular with practitioners. Ideally, programmers would like to begin work with a fairly accurate but incomplete architecture, while the architect continues to refine the architecture based on feedback from the developers who implement the code. Therefore, an architectural model should support this continuous refinement loop which may be difficult to reconcile with linkage strategies. These very arguments are highlighted in a paper that presents a case study of re-engineering a Java project into ArchJava [120].

Yet another weakness in techniques such as ArchJava and ArchWare is that they make it harder to differentiate the architecture specification from the implementation. This can cause difficulty in evolving the architecture itself. ArchJava mingles structural descriptions of the architecture with Java program code while ArchWare does not distinguish at all between architectural and implementation constructs. We believe that separating program code from architecture specification is important for bringing these strategies into mainstream use.

The lack of adoption can also be attributed to the fact that these linkage and dynamic conformance checking approaches require the support of an augmented runtime environment. ArchWare, Archium and AEM all require some form of a runtime infrastructure for monitoring static and dynamic conformance. This will, however, add another layer of complexity to most non-trivial software systems that already depend on such entities as middleware, application servers and persistence frameworks. Therefore, both developers and users alike will resist the extra burden in integrating, testing and deploying additional runtime environments.

8. Self-adaptation

A technology similar to runtime conformance checking is a self-organising and self-healing system that is capable of adapting to changing requirements and operating conditions [122]. As noted in section 2, software architecture erosion is a consequence of regular system maintenance performed by humans without strict adherence to architectural guidelines. Thus, it can be hypothesised that erosion could be eliminated if human intervention required to maintain the system is minimised and self-adaption is programmed correctly with respect to architecture specifications. Self-adaptive systems are typically based on a closed feedback loop consisting of sensors (or probes) that detect changes in the system's environment or its internal state, comparators (or gauges) that evaluate the sensor output against predefined policies, and actuators that initiate a sequence of changes in the system.

Rainbow [123] is a self-adaptation framework that enables an executing system to be monitored against an architectural model and drives automated repairs to the system if constraints in the architecture are violated. The monitoring data returned by probes in the executing system are transformed using gauges into information that can be related to the model, while the outcome from constraint evaluation is fed to a repair engine that initiates the necessary corrective measures in the live system.

Georgiadis et al. [124] propose another approach to self-organising software architectures. This work is primarily geared towards distributed component-based systems where removing or adding components to a system can cause the system to break its architectural properties. It makes use of a mechanism in which components have a built-in configuration view of the architecture and a component manager updates this view based on a set of hardwired structural constraints. A component that enters or leaves the system causes other components to re-evaluate their bindings to satisfy these constraints. The system is supported by a runtime communication infrastructure allowing components to interact with one another. In a later work, Kramer and Magee [122] extend and generalise Georgiadis' component model to build an architectural framework for self-managed systems and propose a tiered architectural style inspired by robotics for building autonomous software systems.

Vassev et al. [125] propose an approach for building runtime adaptive architectures by extending the Autonomic System Specification Language (ASSL) framework [126] developed by the same researchers. This framework enables the specification of autonomic systems using the ASSL language and provides a set of actions to facilitate self-modifiability of their architectures. The extended work propagates architectural changes resulting from a self-initiated action to implementation level through a runtime code generator and a code manager. It is hypothesised that this technique helps an ASSL-generated system to be hot-plugged with automatically generated code that complies with its architecture.

Discussion

The contribution of self-adaptation strategies towards preventing architecture erosion relies on minimising human interference in routine maintenance activities. These systems have a built-in knowledge base of the intended architecture to guide changes made to the implementation.

Adoption: These technologies have not been widely adopted in real-world systems, except in highly specialised applications such as unmanned space exploration vehicles where they become an important part of autonomous onboard systems maintenance processes [127]. There is, however, evidence of continuing academic research in the area of self-adaptation [128].

Efficacy: Self-adaptive techniques are quite effective in controlling erosion in certain classes of systems that have regular updates which can be automated. Therefore, if the majority of the changes in the target software system requires human intervention, the benefits of self-adaptation is drastically reduced. Furthermore, their effectiveness depends largely on the extent to which designers can anticipate the possible changes a deployed system will go through during its lifetime.

Cost-Benefit Analysis: If applied to an architecture that can benefit from a self-adaptation strategy, the cost-benefit ratio is likely to be quite high. These techniques, like architecture linkage strategies, also employ a runtime platform that must be factored in when evaluating the potential benefits.

Evaluation: Similar to linkage strategies presented in section 7, self-adaptation technologies require the target system to carry the burden of a runtime infrastructure that enables their autonomic capabilities. However, from the work that has been carried out in this area, it appears that self-adaptation is more suitable for specific architectures such as distributed systems than for a broader range of application. We also believe that a self-adaptation strategy works best for long-running software processes where system updates are performed while the software is executing and the architecture remains fairly constant during system evolution.

9. Architecture Restoration

The approaches for controlling architecture erosion discussed in the previous categories are aimed at reduction and prevention. However, preventing erosion completely is a difficult task and may not be feasible.

Studies of industry practices [35] indicate that a majority of the attempts at minimising the effects of erosion are focused on repairing the damage. The process of re-engineering software involves identifying erosion, recovering the implemented architecture from source artefacts, repairing the recovered architecture to conform to intended architecture and reconciling the implementation with this architecture. In addition, some repair techniques introduce an architecture discovery activity to establish a possible intended architecture from system requirements and use cases. Considering these different activities in architecture restoration, we have categorised the survey results under architecture recovery, discovery and reconciliation. In practice, however, these three techniques are used in very close corporation with one another.

The following sections present the survey under each of these categories.

9.1. Architecture Recovery

A large number of tools have been developed and adopted successfully in the industry for recovering the implemented architecture using static and runtime analysis of software elements. Recovery is a necessary precursor to any systematic approach that restores an eroded architecture to its intended state. An early approach for recovering useful architectural structure from source code can be found in reflexion models [60, 61]. As described in section 4.3, this technique progressively refines a hypothetical model of the intended architecture with results from a static analysis of the source code. Reflexion models are often refined by a clustering technique [129] that groups related software entities such as classes and packages together to form a higher level of abstraction, for instance a sub-system. Complementary to clustering is filtering that removes entities considered too noisy for an architectural level abstraction.

Graphical visualisation tools often help the process of recovering an architecture using reflexion models. Furthermore, algorithms and tools have been developed to automate clustering and filtering tasks. The Bunch tool [130] performs automatic clustering using dependency and call graphs generated from source code to produce architectural level views. Bunch uses a genetic algorithm and a hill-climbing algorithm to attack the search space while considering factors such as coupling and cohesiveness between modules to determine the optimal strategy to partition the dependency graph into modules. Later work in this area includes clustering techniques that use more optimised partitioning algorithms [131, 132].

The Architecture Reconstruction Method (ARM) [133] is a semi-automatic method built on top of the Dali Workbench environment [134] that enables architecture recovery from source artefacts. ARM employs pattern-matching techniques to identify a given set of concrete patterns in an extracted model of the implemented architecture. The concrete patterns are specified by a human operator with some knowledge of the system and may include generic design patterns as well. The source model is queried for pattern instances and the results are evaluated manually. Detected pattern instances are then used to reconstruct the architecture. The Rigi [135] tool is used to visually organise the recovered patterns into a model of the implemented architecture.

Niere et al. [136] suggest a similar pattern-based recovery approach. In this method an abstract syntax graph (ASG) representation of a source file is searched for design patterns that have been formally defined using graph transformation rules. There are two phases to the search algorithm executed over the ASG. In the first phase, the graph is searched bottom-up, annotating the nodes that match certain sub-patterns that occur (e.g. associations or inheritance) in design patterns. The top down search in the second pass uses this annotated information to determine the design pattern from one or more sub-patterns. The automated process is supported by the FUJABA environment but requires user interaction during analysis to filter out false positives.

Kamran [137] describes another scheme for recovering an architectural model from source code. This technique uses a pattern matching language called Architecture Query Language (AQL). A hypothesised model of the intended architecture specified using AQL is used for discovering patterns in an abstract graph representing the source model. An approach that uses string matching to detect pattern *hot-spots* in source code and progressively abstract out patterns to recover the architecture is proposed in a paper by Pinzger and Gall [138].

A process-oriented methodology for recovering architectural models and corresponding views is proposed by van Deursen et al. in their Symphony approach [139]. This method generalises architecture recovery and is not intended to reach a predetermined goal as in the case with other approaches. The recovery process focuses on building architectural views from source artefacts using any available technology. The views are constrained by viewpoints, which are defined for the problem to be addressed and reflect the source model of the system. In the second phase of this process, the source views are transformed into target architectural views using a mapping scheme. These architectural views then guide the erosion analysis and reconstruction tasks performed on the implementation.

Huang et al. [140] present a technique for architecture recovery by analysing the runtime state of a system. They propose using a reflective runtime infrastructure, in their case the PKUAS application server, to enable querying the runtime state of a system to determine its dynamic architecture. The runtime reflective platform consists of meta-objects that reflect properties of system objects, allowing the implemented architecture to be recovered. Although an objective of this strategy is to make it possible to modify the behaviour of an executing system by propagating architectural changes at runtime, the extent to which architectural changes can be made to an executing software system is not clear.

Abi-Antoun [141] proposes a method for recovering the runtime architecture of object-oriented systems using annotated type instances. The annotations are placed in the code by the developer to clarify design intent. These annotations provide the means for building a hierarchical object graph representing the runtime state of the system. The soundness of this technique is argued on the basis that it captures every runtime object instance and every relationship between these objects.

A number of other techniques for recovering architectures are given in literature. Stringfellow et al. [13] introduce a method that uses change reports and history records obtained from revision control systems. This approach derives an architecture by "lifting" file level changes to component level and coupling this analysis with relationships among source files, in particular the *#include* relationships in C/C++ source code. In addition, there have been efforts to explore the possibility of recovering useful architectural structure from source code file names [142]. The researchers of this technique propose identifying the task of a source file by interpreting the meaning of its file name. They contend that this method could prove effective in recovering the decomposition views of large legacy software systems. In a method that takes root in machine learning, Maqbool and Babri [20] present the use of a Naïve Bayes Classifier for architecture recovery. The classifier is trained with known elements in the target architecture and then directed to extract the complete architecture using its acquired knowledge base.

Jansen et al. [143] focus on recovering architectural design decisions with their ADDRA approach. Here the detailed designs from selected versions of the implementation are recovered to generate architectural views for each version. Differences between these views are then analysed to establish the architectural design decisions. This approach is distinct from other approaches that recover only the structural and dynamic properties of an architecture with little or no consideration of design rationale. Often rationale is deduced informally from other recovered architectural elements. This survey, however, has not found any formal techniques capable of recovering architecture together with rationale from implementation artefacts.

Finally, there are a large number of reverse engineering tools that aid architecture recovery. Besides those highlighted in the process-oriented architecture conformance section, almost all modern software development tools have some form of reverse engineering capability to extract architectural structures from implementation.

9.2. Architecture Discovery

Architecture discovery becomes necessary when a documented specification of the intended architecture is unavailable. The discovery process builds the intended architecture from system requirements and possibly by studying system use cases. However, architecture discovery alone is not sufficient to repair erosion, and this activity often supplements architecture recovery and reconciliation. Most existing processes for discovering the intended architecture are adaptations of the reflexion model technique explained in section 4.3.

Medvidovic et al. [144] present a method that combines discovery, recovery and architectural styles to combat architecture erosion. In this three-step process, the intended architecture is first discovered through a process known as Component-Bus-System-Property [145] that provides a transitional model to build an architectural design from requirements. The technique uses four generic architectural primitives, namely components, connectors, configurations and properties, which can effectively capture early architectural design notions. In the second step, standard tools are used to generate class diagrams from the source code, which then go through an iterative clustering process until an architecturally significant model is extracted. In the last step, the two different architectural models are integrated by identifying one or more architectural styles common to both models.

Besides the above, our survey has not identified architecture discovery methods that are specifically geared towards addressing architecture erosion. However, a large body of work exists on methods for transforming requirements into architectural design models. If the requirements are well-documented, these techniques could be used effectively to discover the intended architecture.

9.3. Architecture Reconciliation

Architecture reconciliation is the process by which the implemented architecture is mended to conform to the intended architecture obtained from either a recovery or discovery activity. A common approach for reconciling an implementation with its architecture is code refactoring [146]. In this approach, the source code is systematically restructured to follow architectural design principles without altering the externally visible behaviour of the system. Similar to reverse engineering, code refactoring is well supported by commercial software engineering tools. However, the emphasis of these tools is generally on refactoring code to meet a specific design goal like a design pattern rather than to conform to an architectural property. Work in the area of architecture-driven refactoring is scarce and therefore, would possibly become an active research area in future. Bourqun and Keller [147] present an approach based on an industrial case study called "high-impact refactoring" that refactors a system at an architecture level. An architecture recovery tool and a set of informally specified architectural constraints are used to uncover violations of the architecture. These violations are then addressed using best practices in architectural design.

Tran and Holt [29] discuss a more rigorous approach to reconciling the intended and implemented architectures. They describe a two-pronged method, consisting of forward architecture repair to overhaul the implementation to match the intended architecture and reverse architecture repair to synchronise the intended architecture with the implementation. Forward repairing involves correcting invalid dependencies amongst modules while ensuring these corrections do not impact functionality. This process is partially automated using Grok, a language for dealing with binary relational algebra [148]. Grok scripts describe the physical and logical dependency relations in the architecture. In reverse architecture repairing, architecture elements that do not contribute to the implementation (i.e. gratuity) are removed in a process that does not affect the source code or system functionality.

In an approach aimed at refactoring architectural models, Ivkovic and Kontiogiannis [149] use UML 2.0 profiles and semantic annotations to drive the refactoring process based on quality goals (or quality attributes). The quality goals are mapped to a graph in which quality metrics associated with each node. This graph forms the basis for applying refactoring transformations to the architecture modelled using a UML profile. Refactoring is repeated for all the primary quality goals in the graph. The availability of tool support for this process is not clear from the literature.

9.4. Discussion

Table 3 summarises the contribution of each restoration technique towards controlling erosion.

Adoption: Despite the large number of tools and techniques currently available, architecture restoration does not appear to have a broad adoption base in industry or academia. There are some documented case studies such as reconstructing the architecture of the VANISH tool using Dali Workbench [150] and recovering the architecture of the Apache web server [151], but they do not point to wider application. Although architecture recovery, in particular, is often a necessary step in the maintenance of mature software systems, industry practices used to recover architecture are less rigorous than those techniques discussed here. The usual method is to reverse engineer the source code to extract UML models that, together with available design documentation, form the basis for manually reconstructing the architecture. The derived architecture is then used as guidance for refactoring eroded parts of the implementation. Sim-

Strategy	Contribution towards controlling erosion
Architecture Recovery	Establishes the necessary foundation for the detection, evaluation and repairing of erosion by extracting the implemented architecture from source code and other artefacts.
Architecture Discovery	Provides the means for building a model of the intended architecture in situations where a reliable specification is unavailable. The intended architecture is a necessary prerequisite for repairing an eroded system.
Architecture Reconciliation	Enables rectifying a system with an eroded architecture by aligning the implementation with the intended architecture. Reconciliation often requires the support of architecture recovery and discovery strategies.

Table 3: *Controlling architecture erosion with architecture restoration strategies*

ilarly, we do not find noticeable application of architecture discovery methods in industry.

Efficacy: Architecture restoration is an effective tool for controlling architecture erosion and extending the useful lifetime of software. Planned restoration work can be made part of an overall maintenance strategy. However, restoration may not always work well if the system is badly deteriorated. Therefore, for best results, restoration should be coupled with one of the preventive mechanisms presented earlier.

Cost-Benefit Analysis: The cost-benefit factor of architecture recovery is directly related to the extent of erosion. The more eroded an architecture is, the costlier it is to recover and less reliable is the outcome. Therefore, with a system that has significant architectural damage, it is prudent to evaluate whether a partial recovery is more sensible. Such an analysis should compare the cost of recovery against the value the system will generate during its extended life. Architecture recovery is also attractive to organisations that do not wish to invest in preventive measures earlier in the system life cycle until the system has proved successful.

Evaluation: Although architecture restoration is an after-the-fact activity, our survey indicates that a significant number of approaches and tools developed for controlling architecture erosion fall under this category. Therefore, it appears that architecture erosion is an inevitable consequence of the software engineering practices of today. However, despite the large number of techniques available, repairing the implemented architecture to comply with the intended architecture can be an expensive and tedious task in complex software systems. More importantly, current architecture restoration techniques could recover only the structural and behavioural properties of an architecture and not architecture rationale. At best, the rationale can only be deduced from the recovered properties.

10. Choice of Erosion Control Strategies

The different erosion control strategies classified in our framework will be applicable to different scenarios in software development projects. Their inherent characteristics will determine the suitability of each strategy under various circumstances. While a detailed study of all possible scenarios and their combinations is beyond the scope of this survey, in Table 4 we provide examples of criteria that can guide the selection of an appropriate strategy for a given situation. These criteria have been derived based on our study of the characteristics and requirements of different control techniques as well as the circumstances in which erosion can occur. In the table, rows contain the criteria and some sample values while the columns show the erosion control strategies. The intersections between each criterion and strategy show the level of influence that a criterion has over selecting a strategy. We use a three-point scale to show the level of influence where (●) denotes high, (◐) denotes medium and (○) denotes low or no influence.

For instance, the stage of development of a software system can play an important role in choosing the most appropriate erosion control strategy for it. As shown in Table 4, design enforcement, architecture to implementation linkage, self-adaptation and restoration strategies are all strongly influenced by a system's current stage of development. While it makes sense to apply restoration only to a mature system exhibiting visible erosion, the other three strategies must be employed when the system is being designed. On the other hand, selection of an evolution management technique may not be influenced by the development stage. They use tools that are external to the system being developed and hence should be applicable at any point in time during development. Similarly, process-oriented strategies should be applicable throughout the lifecycle of a system. But process rigour tends to die out during the maintenance phase of a system making it less attractive for controlling erosion at this stage.

It is also possible combine different strategies together to control architecture erosion. Often process-oriented techniques reinforce other approaches and restoration methods such as refactoring are included in software processes for correcting minor erosion issues. Table 5 shows strategies that can be coupled to increase the effectiveness erosion control (marked ✓). Here we show that evolution management methods can complement design enforcement techniques well because enforced design constraints in generated code, architectural patterns and frameworks get broken mostly after the initial development of the software. Similarly, a linkage approach such as ArchJava can be used to specify architectural patterns or even full-blown frameworks thereby helping to enforce their constraints at code level. It is possible that architecture restoration can work together with some self-adaptation techniques to periodically correct parts of the implementation that may have diverged from the architecture. In this case, the a post-restoration view of the architecture should be fed back to self-adaptation mechanism to continue its adaptation activities.

Table 5 additionally shows combinations that may

	Process-oriented Architecture Conformance	Architecture Evolution Management	Architecture Design Enforcement	Architecture to Implementation Linkage	Self-adaptation	Architecture Restoration
Stage of development e.g. - new build - new implementation of existing architecture or design - deployed system consistent with architecture - deployed system with erosion	◐	○	●	●	●	●
Availability and nature of architecture documentation and models e.g. - no architecture documentation - informal architecture specification - ADL specification - other formal notations	○	◐	●	●	●	○
Architectural topology or type of target system e.g. - distributed architecture - service oriented architecture	○	○	◐	◐	●	○
Dynamicity and frequency of changes in the target system e.g. - static architecture and implementation - static architecture and occasional changes to implementation - static architecture and frequent changes to implementation - dynamic architecture and implementation	○	○	○	○	○	○
Organisational constraints on platform, technology, skills etc. e.g. - company policy restricting use the of 3rd party libraries and tools - shortage of experienced personnel with technical skills	◐	●	◐	●	●	◐
Existence of development and maintenance processes e.g. - no established processes but resources available to initiate them - no established processes or resources - established processes but not architecture-centric - established, architecture-centric processes	●	◐	○	○	○	○
Performance and other non-functional requirements of the target system e.g. - performance critical real-time system - resource constrained embedded system	○	○	◐	●	●	◐

Table 4: *Possible influence factors for choosing an erosion control strategy (high - ●, medium - ◐, low/none - ○)*

	Process-oriented Architecture Conformance	Architecture Evolution Management	Architecture Design Enforcement	Architecture to Implementation Linkage	Self-adaptation	Architecture Restoration
Process-oriented Architecture Conformance	–	✓	✓	✓	?	✓
Architecture Evolution Management	✓	–	✓	?	?	✓
Architecture Design Enforcement	✓	✓	–	✓	✗	✓
Architecture to Implementation Linkage	✓	?	✓	–	✗	✓
Self-adaptation	?	?	✗	✗	–	?
Architecture Restoration	✓	✓	✓	✓	?	–

Table 5: *Combinations that are – possible (✓), possible with certain specific techniques (?) and not possible (✗).*

work together only with some specific techniques (marked **?**) and those which are in conflict (marked ✗). As shown, some process-oriented strategies such as architecture analysis, dependency analysis and compliance monitoring may not work well with self-adaptive architecture. A self-adaptation technology determines an optimal architecture using a set of rules and input from the system environment. These architectures are therefore fairly dynamic. Performing static analysis on them may not be very useful. However, design documentation, a process-oriented strategy, can still be used to describe the core architecture of the self-adaptive system. Similar issues arise when combining evolution management or restoration strategies with self-adaptation. We also conjecture that linkage technologies can possibly work well with SCM-centred evolution management techniques but not with architecture-centred SCM approaches. The latter synchronises the implementation with the architecture which is similar to what a linkage approach does. Combining them will cause inconsistencies as they update the code to suit their own architectural views.

Considering strategies that are not compatible, self-adaptation is likely to conflict with design enforcement methods as the former requires an easily reconfigurable architecture which is not possible with the latter. A similar constraint exists between self-adaptation and linkage strategies.

11. Conclusions

This survey provides an overview of current approaches for dealing with the problem of architecture erosion. We have presented a lightweight classification framework to categorise these methods primarily for easier analysis of their efficacy. The vast number of diverse techniques, in both academic research and industrial practice, indicates that sustainability of software systems is an important problem in computer science and reinforces our belief that investments in software assets need to be protected. However, as discussed under each classification, none of the available methods singly provides an effective and comprehensive solution for controlling architecture erosion.

Enforcing architectural conformance through software engineering processes lacks scalability, does not resonate well with agile practices and is prone to human neglect. Managing the evolution of architectures through SCM tools could in theory prove very useful, but requires standardised architectural specification models and SCM systems that are capable of interpreting these models.

Design enforcement techniques, on the other hand, are effective in guiding the initial implementation phase to conform to the architecture, but have the tendency to fall short during maintenance unless supported by conformance assurance checks of a rigorous software engineering process.

Approaches that link architectural models to implementation possibly make the strongest claim for con-

	Process-oriented Architecture Conformance				Architecture Evolution Management	Architecture Design Enforcement		
	Architecture design documentation	Architecture analysis	Architecture compliance monitoring	Dependency analysis		Code generation	Architecture patterns	Architecture frameworks
When/Where applicable	Most commercial software packages				Architecture driven development	Architecture driven development (e.g. MDD)		
Pre-requisites	None	Architecture	Architecture and implementation		formal architecture specification and a SCM tool capable of understanding this architecture model	Architecture	None	None
Tools availability	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Adoption – academia	Medium	Medium	Medium	High	Medium	Low	Low	Low
Adoption – industry	High	Medium	Medium	High	Low	High	High	High
Key strengths	- Effective in curtailing erosion early in the life cycle - Integrates well with project management practices				- Ensures architecture and implementation are in sync with each other while allowing both to evolve independently	- Simple and practical for most industry applications - Patterns and frameworks reuse design best practices		
Key weaknesses	- Prone to human neglect - Lacks rigour during the maintenance phase - Some process activities are difficult to scale (e.g. architecture reviews)				- Requires standardised ADLs and SCM tools that understand these ADLs	- Round-trip problem in code generation - Weak or no mechanisms to stop violations of architectural properties in evolving systems		

Table 6a: *Summarised characteristics of erosion control strategies (part 1 of 2)*

	Architecture to Implementation Linkage	Self-adaptation	Architecture Restoration		
			Recovery	Discovery	Reconciliation
When/Where applicable	Suites both architecture driven and agile development	Specific types of architectures such as distributed systems or systems with long running software processes	Mature software systems or those which are in the maintenance phase		
Pre-requisites	Runtime environment	Runtime environment	Source code, Change records, etc.	Requirements or knowledge of system functionality	Recovered and/or discovered architecture
Tools availability	Yes	N/A	Yes	Yes	Yes
Adoption – academia	High	High	High	Medium	High
Adoption – industry	Low	Low	High	Medium	High
Key strengths	- Ensures a continuous association between architecture and its implementation	- Enables the implementation to change by it self to comply with the architecture	- Widely adopted in the industry as a practical approach to extend the life of software systems - Complements prevention techniques		
Key weaknesses	- Requires a runtime infrastructure - Architecture and implementation usually form a single entity	- Requires a runtime infrastructure - Assumes that the architecture remains fairly constant during system evolution	- Does not curtail or prevent erosion - The original intended architecture may not always be recoverable - Unable to extract architecture rationale		

Table 6b: *Summarised characteristics of erosion control strategies (part 2 of 2)*

straining architecture erosion. The bi-directional associations between architecture and implementation enable these systems to continually enforce and synchronise architectural properties in evolving systems. But these techniques as well as self-adaptation technologies impose a heavy penalty on the system in the form of a runtime environment, and therefore have not been adopted for mainstream use.

Lastly, the recover-and-repair techniques are the most prevalent in practice because poorly maintained software systems eventually succumb to architecture erosion. Although restoration approaches help extend the lifetime of software, the cost of regaining the original state of the intended architecture can be economically and technically prohibitive.

Tables 6a and 6b present a summary of our analysis of the large number of methods available for controlling architecture erosion. The information in this summary is presented under the framework described earlier. A more detailed comparative analysis is not feasible because data, such as the adoption rate of a specific technique, needed for this purpose is not available in most cases. The relative adoption indicators *High*, *Medium* and *Low* for academia were derived by inspecting the number of published work related to a given class of techniques. Similarly, adoption in industry was based on a study of available commercial products and software development processes.

Besides the existence of a gamut of technologies, fundamental causes of architecture erosion such as poor expressiveness in programming languages, lack of standardised architecture specification languages and insufficient software visualisation tools should be addressed in order to effectively control erosion. Finding a broad solution to architecture erosion may in fact require revisiting the basics of software engineering. We identify two possible strategies to direct future research in this area.

- **New programming models capable of specifying and retaining architectural properties.** Current programming languages are intended for building software elements that are at a lower level of abstraction than software architectures. Therefore, these languages are not capable of expressing architectural properties within program code. A paradigm shift is required, similar to the shift that occurred from procedural programming to object-oriented programming, which places software architectures at the focal point of all programming activities.

- **Architecture execution environments.** A generic execution environment that is instantiated by the intended architecture of a system, and thus encompasses all its properties, can provide a virtual environment in which the system executes. We envisage that such an environment will monitor the structural and behavioural aspects of the implementation at development as well as execution time. The virtual environment will act as an architectural "mould" for system execution. However, a software system designed to work within its architectural realm should also be able to execute outside this environment. Such a strategy should encourage wider adoption as the burden of an extra runtime infrastructure can be removed if required. In this case, the task of validating runtime architecture conformance should become a regular activity in the maintenance and testing processes of an evolving system.

As future work, we plan to evaluate the feasibility of both these directions. Our goal is to develop an effective and usable framework for controlling architecture erosion in the next generation of complex software systems.

Acknowledgment

This work was made possible through a PhD prize studentship awarded to the first author jointly by the Scottish Informatics and Computer Science Alliance (SICSA) and the University of St Andrews, Scotland. The studentship provides funding for 3.5 years of PhD research for finding an effective solution to the problem of architecture erosion in complex software systems.

References

[1] D. L. Parnas, Software aging, in: Proceedings of the 16th International Conference on Software Engineering, IEEE, 1992, pp. 279–287.

[2] Y. Huang, C. Kintala, N. Kolettis, N. D. Fulton, Software rejuvenation: Analysis, module and applications, in: Proceedings of International Symposium on Fault-tolerant Computing, IEEE, 1995, pp. 381–390.

[3] M. Grottke, R. Matias Jr., K. S. Trivedi, The fundamentals of software aging, in: Proceedings of 1st International Workshop of Software Aging and Rejuvenation, IEEE, 2008, pp. 1–6.

[4] D. E. Perry, A. L. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes 17 (1992) 40–52.

[5] M. Shaw, D. Garlan, Software Architecture: Perspective of an Emerging Discipline, Prentice Hall, 1996.

[6] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Second Edition, Addison Wesley, 2003.

[7] L. Hochstein, M. Lindvall, Combating architectural degeneration: A survey, Information and Software Technology 47 (2005) 643–656.

[8] M. Dalgarno, When good architecture goes bad, Methods and Tools 17 (2009) 27–34.

[9] J. van Gurp, J. Bosch, Design erosion: Problems and causes, Journal of Systems and Software 61 (2002) 105–119.

[10] M. Riaz, M. Sulayman, H. Naqvi, Architectural decay during continuous software evolution and impact of 'Design for Change' on software architecture, in: Proceedings of the International Conference on Advanced Software Engineering and Its Applications, Springer, 2009, pp. 119–126.

[11] C. Izurieta, J. Bieman, How software designs decay: A pilot study of pattern evolution, in: Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement, IEEE, 2007, pp. 449–451.

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

[13] C. Stringfellow, C. Amory, D. Potnuri, A. Andrews, M. Geor, Comparison of software architecture reverse engineering methods, Information and Software Technology 48 (2006) 484–497.

[14] I. Jacobson, Object-oriented software engineering, Addison Wesley, 1992.

[15] D. R. Harris, H. B. Reubenstein, A. S. Yeh, Reverse engineering to the architectural level, in: Proceedings of the 17th International Conference on Software Engineering, ACM, 1995, pp. 186–195.

[16] B. Bellay, H. Gall, A comparison of four reverse engineering tools, in: Proceedings of the 4th Working Conference on Reverse Engineering, IEEE, 1997, pp. 2–11.

[17] G. C. Gannod, B. H. C. Cheng, A framework for classifying and comparing software reverse engineering and design recovery techniques, in: Proceedings of the 6th Working Conference on Reverse Engineering, IEEE, 1999, pp. 77–88.

[18] R. Roshandel, A. van der Hoek, M. Mikic-Rakic, N. Medvidovic, Mae – A system model and environment for managing architectural evolution, ACM Transactions on Software Engineering and Methodology 13 (2004) 240–276.

[19] J. Aldrich, C. Chambers, D. Notkin, ArchJava: Connecting software architecture to implementation, in: Proceedings of the 24th International Conference on Software Engineering, ACM, 2002, pp. 187–197.

[20] O. Maqbool, H. Babri, Bayesian learning for software architecture recovery, in: Proceedings of the International Conference on Electrical Engineering, IEEE, 2007, pp. 1–6.

[21] P. Naur, B. Randell (Eds.), NATO Software Engineering Conference 1968, NATO Science Committee, 1969.

[22] E. W. Dijkstra, ACM Turing Lecture 1972, `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD340.html`, 1972. Accessed April 2011.

[23] M. M. Lehman, Laws of software evolution revisited, in: Proceedings of the 5th European Workshop on Software Process Technology, Springer, 1996, pp. 108–124.

[24] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch: Why reuse is so hard, IEEE Software 12 (1995) 17–26.

[25] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch: Why reuse is still so hard, IEEE Software 26 (2009) 66–69.

[26] M. W. Godfrey, E. H. S. Lee, Secrets from the monster: Extracting Mozilla's software architecture, in: Proceedings of the International Symposium on Constructing Software Engineering Tools, pp. 15–23.

[27] J. van Gurp, S. Brinkkemper, J. Bosch, Design preservation over subsequent releases of a software product: A case study of Baan ERP, Journal of Software Maintenance and Evolution: Research and Practice 17 (2005) 277–306.

[28] I. Sutton, Software erosion in pictures – FindBugs, `http://www.headwaysoftware.com/blog/2008/11/software-erosion-findbugs/`, 2008. Accessed April 2011.

[29] J. B. Tran, R. C. Holt, Forward and reverse repair of software architecture, in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, IBM, 1999, pp. 12–20.

[30] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, H. Verjus, Towards a process-oriented software architecture reconstruction taxonomy, in: Proceedings of the 11th European Conference on Software Maintenance and Re-engineering, IEEE, 2007, pp. 137–148.

[31] S. Ducasse, D. Pollet, Software architecture reconstruction: A process-oriented taxonomy, IEEE Transactions on Software Engineering 35 (2009) 573–591.

[32] J. Knodel, D. Popescu, A comparison of static architecture compliance checking approaches, in: Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture, IEEE, 2007, pp. 12–21.

[33] L. Passos, R. Terra, M. T. Valente, R. Diniz, N. das Chagas Mendonca, Static architecture-conformance checking: An illustrative overview, IEEE Software 27 (2010) 82–89.

[34] L. Dobrica, E. Niemela, A survey on software architecture analysis methods, IEEE Transactions on Software Engineering 28 (2002) 638–653.

[35] L. O'Brien, C. Stoermer, C. Verhoef, Software Architecture Reconstruction: Practice Needs and Current Approaches, Technical Report CMU/SEI-2002-TR-024, Software Engineering Institute, 2002.

[36] P. Kruchten, The Rational Unified Process: An Introduction, Addison Wesley, 2003.

[37] IBM, IBM Rational Unified Process (RUP), `http://www-01.ibm.com/software/awdtools/rup/`, 2011. Accessed April 2011.

[38] The Eclipse Foundation, Eclipse Process Framework project, `http://www.eclipse.org/epf/`, 2011. Accessed April 2011.

[39] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, Addison Wesley, 2003.

[40] P. Kruchten, The 4+1 View Model of architecture, IEEE Software 12 (1995) 42–50.

[41] F. Bachmann, L. Bass, P. Clements, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, Documenting Software Architecture: Documenting Behavior, Technical Report CMU/SEI-2002-TN-001, Software Engineering Institute, 2002.

[42] N. Medvidovic, R. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 26 (2000) 70–93.

[43] D. Garlan, R. Monroe, D. Wile, Acme: An architecture description interchange language, in: Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research, IBM, 1997, pp. 169–183.

[44] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying distributed software architectures, in: Proceedings of the 5th European Software Engineering Conference, Springer, 1995, pp. 137–153.

[45] E. M. Dashofy, A. van der Hoek, R. N. Taylor, A comprehensive approach for the development of modular software architecture description languages, ACM Transactions on Software Engineering and Methodology 14 (2005) 199–245.

[46] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann, Specification and analysis of system architecture using Rapide, IEEE Transactions on Software Engi-

neering 21 (1995) 336–355.

[47] P. C. Clements, A survey of architecture description languages, in: Proceedings of the 8th International Workshop on Software Specification and Design, IEEE, 1996, pp. 16–25.

[48] IEEE, IEEE recommended practice for architectural description of software-intensive systems, IEEE Std 1471-2000 (2000) 1–23.

[49] J. Tyree, A. Akerman, Architecture decisions: Demystifying architecture, IEEE Software 22 (2005) 19–27.

[50] N. B. Harrison, P. Avgeriou, U. Zdun, Using patterns to capture architectural decisions, IEEE Software 24 (2007) 38–45.

[51] R. Kazman, M. H. Klein, P. C. Clements, ATAM: Method for Architecture Evaluation, Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, 2000.

[52] I. T. Bowman, R. C. Holt, N. V. Brewster, Linux as a case study: Its extracted software architecture, in: Proceedings of the 21st International Conference on Software Engineering, ACM, 1999, pp. 555–563.

[53] R. Leitch, E. Stroulia, The space station operations control software: A case study in architecture maintenance, in: Proceedings of the 34th Hawaii International Conference on System Sciences, IEEE, 2001, p. 10.

[54] R. Kazman, G. Abowd, L. Bass, P. Clements, Scenario-based analysis of software architecture, IEEE Software 13 (1996) 47–55.

[55] J. C. Dueñas, W. L. de Oliveira, J. A. de la Puente, A software architecture evaluation model, in: Proceedings of the 2nd International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families, Springer, 1998, pp. 148–157.

[56] B. Tekinerdogan, F. Scholten, C. Hofmann, M. Aksit, Concern-oriented analysis and refactoring of software architectures using dependency structure matrices, in: Proceedings of the 15th Workshop on Early Aspects, ACM, 2009, pp. 13–18.

[57] M. Lindvall, R. Tesoriero, P. Costa, Avoiding architectural degeneration: An evaluation process for software architecture, in: Proceedings of the 8th International Symposium on Software Metrics, IEEE, 2002, pp. 77–86.

[58] R. Tvedt, P. Costa, M. Lindvall, Does the code match the design? A process for architecture evaluation, in: Proceedings of the IEEE International Conference on Software Maintenance, IEEE, 2002, pp. 393–401.

[59] K. Fukuzawa, M. Saeki, Evaluating software architectures by coloured Petri nets, in: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, ACM, 2002, pp. 263–270.

[60] G. C. Murphy, D. Notkin, K. J. Sullivan, Software reflexion models: Bridging the gap between design and implementation, IEEE Transactions on Software Engineering 27 (2001) 364–380.

[61] R. Koschke, D. Simon, Hierarchical reflexion models, in: Proceedings of the 10th Working Conference on Reverse Engineering, IEEE, 2003, pp. 36–45.

[62] J. Rosik, A. Le Gear, J. Buckley, M. Ali Babar, An industrial case study of architecture conformance, in: Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, 2008, pp. 80–89.

[63] A. Postma, A method for module architecture verification and its application on a large component-based system, Information and Software Technology 45 (2003) 171–194.

[64] M. Verbaere, M. W. Godfrey, T. Girba, Query technologies and applications for program comprehension, in: Proceedings of the 16th IEEE International Conference on Program Comprehension, IEEE, 2008, pp. 285–288.

[65] Semmle, Semmle .QL language documentation, `http://semmle.com/semmlecode/documentation/ql/`, 2009. Accessed April 2011.

[66] Lattix, The Lattix architecture management system, `http://www.lattix.com/products`, 2011. Accessed April 2011.

[67] Hello2Morrow, SonarJ overview, `http://www.hello2morrow.com/products/sonarj`, 2011. Accessed April 2011.

[68] Hello2Morrow, Sotograph overview, `http://www.hello2morrow.com/products/sotograph`, 2011. Accessed April 2011.

[69] Structure101, Software architecture management, `http://www.headwaysoftware.com/`, 2011. Accessed April 2011.

[70] Axivion, Axivion Bauhaus Suite, `http://www.axivion.com/products.html`, 2009. Accessed April 2011.

[71] Klocwork, Klockwork Architect, `http://www.klocwork.com/products/insight/architect-code-visualization/`, 2011. Accessed April 2011.

[72] Coverity, Architecture analysis, `http://www.coverity.com/products/architecture-analysis.html`, 2011. Accessed April 2011.

[73] Clarkware Consulting, JDepend, `http://clarkware.com/software/JDepend.html`, 2009. Accessed April 2011.

[74] N. Sangal, E. Jordan, V. Sinha, D. Jackson, Using dependency models to manage complex software architecture, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages and Applications, ACM, 2005, pp. 167–176.

[75] S. Huynh, Y. Cai, Y. Song, K. Sullivan, Automatic modularity conformance checking, in: Proceedings of the 30th International Conference on Software Engineering, ACM, 2008, pp. 411–420.

[76] R. Kazman, P. Kruchten, R. L. Nord, J. E. Tomayko, Integrating Software-Architecture-Centric Methods into the Rational Unified Process, Technical Report CMU/SEI-2004-TR-011, Software Engineering Institute, 2004.

[77] Forrester Research, The value and importance of code reviews, Technical Report, Klocwork, 2010.

[78] E. H. Bersoff, Elements of software configuration management, IEEE Transactions on Software Engineering 10 (1984) 79–87.

[79] B. Westfechtel, R. Conradi, Software architecture and software configuration management, in: Proceedings of the Software Configuration Management: ICSE Workshops SCM 2001 and SCM 2003, Springer, 2003, pp. 24–39.

[80] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, N. Medvidovic, Taming architectural evolution, in: Proceedings of the 8th European Software Engineering Conference, ACM, 2001, pp. 1–10.

[81] University of California Irvine, ArchStudio 4, `http://www.isr.uci.edu/projects/archstudio/`, 2006. Accessed April 2011.

[82] E. C. Nistor, J. R. Erenkrantz, S. A. Hendrickson, A. van der Hoek, ArchEvol: Versioning architectural–implementation relationships, in: Proceedings of the 12th International Workshop on Software Configuration Management, ACM, 2005, pp. 99–111.

[83] The Eclipse Foundation, Eclipse Language IDE, `http://www.eclipse.org/home/categories/index.php?category=ide`, 2011. Accessed April 2011.

[84] The Apache Software Foundation, Apache subversion, `http://subversion.apache.org/`, 2011. Accessed April 2011.

[85] C. O'Reilly, P. Morrow, D. Bustard, Lightweight prevention of architectural erosion, in: Proceedings of the 6th International Workshop on Principles of Software Evolution, IEEE, 2003,

pp. 59–64.

[86] T. Parr, ANTLR Parser Generator, http://www.antlr.org/, Undated. Accessed April 2011.

[87] Free Software Foundation, Inc., CVS: Concurrent Versions System, http://www.nongnu.org/cvs/, 2006. Accessed April 2011.

[88] T. N. Nguyen, E. V. Munson, J. T. Boyland, C. Thao, Architectural software configuration management in Molhado, in: Proceedings of the 20th International Conference on Software Maintenance, IEEE, 2004, pp. 296–305.

[89] T. N. Nguyen, Managing software architectural evolution at multiple levels of abstraction, Journal of Software 3 (2008) 60–70.

[90] L. G. P. Murta, A. van der Hoek, C. M. L. Werner, ArchTrace: policy-based support for managing evolving architecture-to-implementation traceability links, in: Proceedings of the 21st IEEE International Conference on Automated Software Engineering, IEEE, 2006, pp. 135–144.

[91] OMG, Model driven architecture, http://www.omg.org/mda/, 2011. Accessed April 2011.

[92] R. Taylor, N. Medvidovic, E. Dashofy, Software Architecture: Foundations, Theory, and Practice, Wiley, 2009.

[93] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, R. Taylor, xADL: Enabling architecture-centric tool integration with XML, in: Proceedings of the 34th Hawaii International Conference on System Sciences, IEEE, 2001, p. 9.

[94] N. Medvidovic, D. S. Rosenblum, R. N. Taylor, A language and environment for architecture-based software development and evolution, in: Proceedings of the 21st International Conference on Software Engineering, ACM, 1999, pp. 44–53.

[95] D. Garlan, An introduction to the Aesop System, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/able/ftp/aesop-overview.pdf, 1995. Accessed April 2011.

[96] N. Medvidovic, D. S. Rosenblum, Domains of concern in software architectures and architecture description languages, in: Proceedings of the Conference on Domain-specific Languages, USENIX, 1997, pp. 16–29.

[97] A. Stavrou, G. A. Papadopoulos, Automatic generation of executable code from software architecture models, in: Information Systems Development, Springer, 2009, pp. 447–458.

[98] F. Arbab, I. Herman, Manifold, Future Generation Computer Systems 10 (1994) 273–277.

[99] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-oriented Software Architecture Volume 1: A System of Patterns, Wiley, 1996.

[100] G. E. Krasner, S. T. Pope, A cookbook for using the Model-View Controller user interface paradigm in Smalltalk-80, Journal of Object-oriented Programming 1 (1988) 26–49.

[101] P. Nii, The Blackboard Model of problem solving, AI Magazine 7 (1986) 38–53.

[102] A. Tang, J. Han, P. Chen, A comparative analysis of architecture frameworks, in: Proceedings of the 11th Asia-Pacific Software Engineering Conference, IEEE, 2004, pp. 640–647.

[103] Spring Source, About Spring, http://www.springsource.org/about, 2011. Accessed April 2011.

[104] Apache Software Foundation, Apache Struts, http://struts.apache.org/, 2011. Accessed April 2011.

[105] The Open Group, TOGAF version 9, http://www.opengroup.org/togaf/, 2011. Accessed April 2011.

[106] Department of Defense, DoDAF architecture framework version 2.0, http://cio-nii.defense.gov/sites/dodaf20/, 2010. Accessed April 2011.

[107] J. A. Zachman, A framework for information systems architecture, IBM Systems Journal 26 (1987) 276–292.

[108] M. Shaw, P. Clements, The golden age of software architecture, IEEE Software 23 (2006) 31–39.

[109] Oracle, Java BluePrints: Guidelines, patterns, and code for end-to-end applications, http://www.oracle.com/technetwork/java/javaee/blueprints/index.html, 2011. Accessed April 2011.

[110] Microsoft, Patterns & practices: Developer Center Home, http://msdn.microsoft.com/en-us/practices/bb190332, 2011. Accessed April 2011.

[111] S. Sarkar, G. M. Rama, R. Shubha, A method for detecting and measuring architectural layering violations in source code, in: Proceedings of the 13th Asia Pacific Software Engineering Conference, IEEE, 2006, pp. 165–172.

[112] M. Lindvall, D. Muthig, Bridging the software architecture gap, IEEE Computer 41 (2008) 98–101.

[113] R. Terra, M. T. Valente, A dependency constraint language to manage object-oriented software architectures, Software Practice and Experience 39 (2009) 1073–1094.

[114] S. Parsa, G. Safi, ArchC#: A new architecture description language for distributed systems, in: International Symposium on Fundamentals of Software Engineering, Springer, 2007, pp. 432–439.

[115] A. Jansen, J. Bosch, Software architecture as a set of architectural design decisions, in: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, IEEE, 2005, pp. 109–120.

[116] R. Morrison, G. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cimpan, B. Warboys, B. Snowdon, R. M. Greenwood, Support for evolving software architectures in the ArchWare ADL, in: Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture, IEEE, 2004, pp. 69–78.

[117] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, C. Occhipinti, ArchWare: Architecting evolvable software, in: Proceedings of the 1st European Workshop on Software Architecture, Springer, 2004, pp. 257–271.

[118] P. Oreizy, N. Medvidovic, R. N. Taylor, Architecture-based runtime software evolution, in: Proceedings of the 20th International Conference on Software Engineering, IEEE, 1998, pp. 177–186.

[119] P. Oreizy, N. Medvidovic, R. N. Taylor, Runtime software adaptation: Framework, approaches, and styles, in: Companion of the 30th International Conference on Software Engineering, ACM, 2008, pp. 899–910.

[120] M. Abi-Antoun, W. Coelho, A case study in incremental architecture-based re-engineering of a legacy application, in: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, IEEE, 2005, pp. 159–168.

[121] S. Cîmpan, H. Verjus, I. Alloui, Dynamic architecture based evolution of enterprise information systems, in: Proceedings of the 9th International Conference on Enterprise Information Systems, pp. 221–229.

[122] J. Kramer, J. Magee, Self-managed systems: An architectural challenge, in: Future of Software Engineering 2007, IEEE, 2007, pp. 259–268.

[123] D. Garlan, B. Schmerl, Model-based adaptation for self-healing systems, in: Proceedings of the 1st Workshop on Self-healing Systems, ACM, 2002, pp. 27–32.

[124] I. Georgiadis, J. Magee, J. Kramer, Self-organising software architectures for distributed systems, in: Proceedings of the 1st Workshop on Self-healing Systems, ACM, 2002, pp. 33–38.

[125] E. Vassev, M. Hinchey, A. Quigley, A self-adaptive architecture for autonomic systems developed with ASSL, in: Proceedings of the 4th International Conference on Software and

Data Technologies, INSTICC, 2009, pp. 163–168.

[126] E. Vassev, M. Hinchey, ASSL: A software engineering approach to autonomic computing, IEEE Computer 42 (2009) 90–93.

[127] D. Bernard, R. Doyle, E. Riedel, N. Rouquette, J. Wyatt, M. Lowry, P. Nayak, Autonomy and software technology on NASA's Deep Space One, Intelligent Systems and their Applications 14 (1999) 10–15.

[128] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Transactions on Autonomous and Adaptive Systems 4 (2009) 1–42.

[129] C. Lung, Software architecture recovery and restructuring through clustering techniques, in: Proceedings of the 3rd International Workshop on Software Architecture, ACM, 1998, pp. 101–104.

[130] S. Mancoridis, B. S. Mitchell, Y. Chen, E. R. Gansner, Bunch: A clustering tool for the recovery and maintenance of software system structures, in: Proceedings of the IEEE International Conference on Software Maintenance, IEEE, 1999, pp. 50–59.

[131] O. Maqbool, H. A. Babri, The weighted combined algorithm: A linkage algorithm for software clustering, in: Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering, IEEE, 2004, pp. 15–24.

[132] P. Andritsos, V. Tzerpos, Information-theoretic software clustering, IEEE Transactions on Software Engineering 31 (2005) 150–165.

[133] G. Y. Guo, J. M. Atlee, R. Kazman, A software architecture reconstruction method, in: Proceedings of the 1st Working IFIP Conference on Software Architecture, Kluwer, 1999, pp. 15–34.

[134] R. Kazman, S. J. Carrière, Playing detective: Reconstructing software architecture from available evidence, Automated Software Engineering 6 (1999) 107–138.

[135] Rigi Team, Rigi, `http://rigi.uvic.ca/`, Undated. Accessed April 2011.

[136] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, J. Welsh, Towards pattern-based design recovery, in: Proceedings of the 24th International Conference on Software Engineering, ACM, 2002, pp. 338–348.

[137] K. Sartipi, Software architecture recovery based on pattern matching, in: Proceedings of the IEEE International Conference on Software Maintenance, IEEE, 2003, pp. 293–296.

[138] M. Pinzger, H. Gall, Pattern-supported architecture recovery, in: Proceedings of the 10th International Workshop on Program Comprehension, IEEE, 2002, pp. 53–61.

[139] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, C. Riva, Symphony: View-driven software architecture reconstruction, in: Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture, IEEE, 2004, pp. 122–132.

[140] G. Huang, H. Mei, F.-Q. Yang, Runtime recovery and manipulation of software architecture of component-based systems, Automated Software Engineering 13 (2006) 257–281.

[141] M. Abi-Antoun, Static extraction and conformance checking of the runtime architecture of object-oriented systems, in: Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications, ACM, 2008, pp. 911–912.

[142] N. Anquetil, T. C. Lethbridge, Recovering software architecture from the names of source files, Journal of Software Maintenance 11 (1999) 38–45.

[143] A. Jansen, J. Bosch, P. Avgeriou, Documenting after the tact: Recovering architectural design decisions, Journal of Systems and Software 81 (2008) 536–557.

[144] N. Medvidovic, A. Egyed, P. Gruenbacher, Stemming architectural erosion by coupling architectural discovery and recovery, in: Proceedings of the 2nd International Software Requirements to Architectures Workshop, IEEE, 2003, pp. 61–68.

[145] P. Grünbacher, A. Egyed, N. Medvidovic, Reconciling software requirements and architectures: The CBSP approach, in: Proceedings of the 5th IEEE International Symposium on Requirements Engineering, IEEE, 2001, pp. 202–211.

[146] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.

[147] F. Bourqun, R. K. Keller, High-impact refactoring based on architecture violations, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, IEEE, 2007, pp. 149–158.

[148] R. C. Holt, Introduction to the Grok Language, `http://plg.uwaterloo.ca/~holt/papers/grok-intro.html`, 2002. Accessed April 2011.

[149] I. Ivkovic, K. Kontogiannis, A framework for software architecture refactoring using model transformations and semantic annotations, in: Proceedings of the Conference on Software Maintenance and Reengineering, IEEE, 2006, pp. 135–144.

[150] L. O'Brien, C. Stoermer, Architecture Reconstruction Case Study, Technical Report CMU/SEI-2003-TN-008, Software Engineering Institute, 2003.

[151] B. Gröne, A. Knöpfel, R. Kugel, Architecture recovery of apache 1.3 – A case study, in: Proceedings of the International Conference on Software Engineering Research and Practice, 2002, p. 7.