Software Evolution: A Graph Based Model

Noel Vizcaino and M. Manjunathaiah

Abstract—A model based on graph isomorphisms is used to formalize software evolution. Step by step we narrow the search space by an informed selection of the attributes based on the current state-of-the-art in software engineering and generate a seed solution. We then traverse the resulting space using graph isomorphisms and other set operations over the vertex sets. The new solutions will preserve the desired attributes. The goal of defining an isomorphism based search mechanism is to construct predictors of evolution that can facilitate the automation of 'software factory' paradigm. The model allows for automation via software tools implementing the concepts.

Index Terms—Software evolution, software engineering, software architecture, software process.

I. INTRODUCTION

We are living in the age of the perpetual beta for a reason: software evolves. Software needs to be constantly updated due to technological, organisational, unforeseen circumstances and fast changes in the existing environment [1]. The relevance of software evolution has only amplified in recent times since the ubiquity of software in diverse application areas, and their complexity, has increased dramatically [2].

Software evolution is a multi-faceted problem domain. Methods for software evolution vary and are approached from several perspectives: understanding, modelling, predicting, controlling, automating, visualizing, improving etc., [3]. Many approaches are semi-formal taking domain expertise into account and propose frameworks that address a particular facet of evolution. Even with this single facet scenario cross-cutting concerns may imply that multiple elements of a complex software system may need to be modified in order to evolve the software to new requirements. Two main problems are faced in evolution: 1) combinatorial search space of possible solutions and 2) risk of introducing inconsistencies if done manually for lack of automated tool support. To address these two main concerns, constructing rigorous software evolution model has gained significant attention in recent research efforts and is recognised as a significant research challenge [4].

Graph based analysis of software systems to aid software evolution is one of the rigorous approaches that has seen a resurgence. Topological analysis of graphs has been applied for analysing complex systems in many areas and such analysis is seen to be relevant to capture useful properties to aid software evolution. For instance in a recent work graph based techniques are used to infer structural changes and to also predict defects in releases [5]. Other works such as apply

program dependence graph techniques [6], or use hyper-graphs as the basis for the formalisation to model transmission of attributes in each evolutionary step [7].

In our work we are aiming to build a general framework to capture evolution in emerging trends such as software factory paradigm [8] where the artefact under consideration is not just a source code or a single software instance but a collection of assets and a family of software products that can be considered in a systematic way. We are also interested in 'software that evolves organically' paradigm, which we loosely define to describe the kind of evolution that happens in an open collaborative distributed team projects (as in open source projects). For both these scenarios we are interested in constructing rigorous evolution techniques that can automate the exploration of evolutionary paths by constructing predictors. Here we suggest using Isomorphisms as attribute preserving operations to find viable solutions for a family of software systems that arises in an evolving activity. We can evolve software using defined operations to explore the combinatorial solution space in a structured manner. The advantage is that the search could be partially automated and engineers will be only required to provide a grand strategy and/or preserve key attributes. Our graph model is based on the hypothesis that solutions attributes will be preserved by isomorphisms. Therefore the task of the engineer is to find a seed solution that will satisfy the requirements and then use a predictor that will use this model to suggest a family of valid alternative solutions from which the designer can make a choice based on some qualitative judgements. We are empowering the practitioners to find novel solutions by automation but not to the detriment of the individual's creativity.

The paper is organized as follows. In Section II we provide software evolution scenarios and identify the graph formalism that is feasible in these contexts. In Section III we present our graph based formalism for modelling evolution. In Section IV we highlight some application scenarios where we can utilise our model to capture evolution.

II. SOFTWARE EVOLUTION CONTEXTS

Software and software evolution is more than the source code and its changes. The architecture, documentation, requirements, constraints, development history are part of a multidimensional view of software [9].

Software factory paradigm:

The software factory paradigm aims to automate the construction of a family of products by bridging the creativity of designer along with a rigorous methodology [8], [10]. Using aschema, families of products that incorporate both commonality and variability according to product instance needs are represented collectively. A software factory schema captures the attributes of a product line in terms of

Manuscript received March 13, 2014; revised August 4, 2014.

The authors are with the University of Reading, Reading, Berkshire RG6 6AH, United Kingdom (e-mail: n.vizcaino@pgr.reading.ac.uk, m.manjunathaiah@reading.ac.uk).

assets. The relationship between the assets is captured in terms of a grid that describes various levels of abstractions as shown in Fig. 1.

	Business	Information	Application	Technology
Conceptual	<ul style="list-style-type: none"> Use Cases And Scenarios Business Goals And Objectives 	<ul style="list-style-type: none"> Business Entities And Relationships 	<ul style="list-style-type: none"> Business Processes Service Factoring 	<ul style="list-style-type: none"> Service Distribution Quality Of Service Strategy
Logical	<ul style="list-style-type: none"> Workflow Models Role Definitions 	<ul style="list-style-type: none"> Message Schemas And Document Specifications 	<ul style="list-style-type: none"> Service Interactions Service Definitions Object Models 	<ul style="list-style-type: none"> Logical Server Types Service Mappings
Physical	<ul style="list-style-type: none"> Process Specification 	<ul style="list-style-type: none"> Database Schemas Data Access Strategy 	<ul style="list-style-type: none"> Detailed Design Technology Dependent Design 	<ul style="list-style-type: none"> Physical Servers Software Installed Network Layout

Fig. 1. Software attributes and views [11].

A software factory schema [8], [11] is better represented as a (directed) graph whose nodes are viewpoints and whose edges are computable relationships between viewpoints called mappings. Using a graph for modelling enables us to represent mappings that would not be possible on a grid. We are using this abstraction to model the space of all possible products. However, instead of using the schemas as the basis for software templates, we will use it to describe stages of software evolution.

III. PROPOSED MATHEMATICAL MODEL

As described above the different parts corresponding to a software system or product are its assets [8]. Some software assets have a relation with other software assets and we are required to maintain these relationships for consistency of a software system. Two grand strategies can be incorporated as attributes to be preserved, so we can make solutions to be isomorphic to a given instance of a software. Engineers choose attributes and explore the domainspace using isomorphisms and set operations. The attribute choices should encode these facts or aid to search the missing ones.

Let the set of assets be A . Let the relation S be a particular subset of the Cartesian product of the assets sets:

$$S \subset A \times A$$

$A \times A$ defines the set of all potential asset combinations, valid or not. If the set A has n elements then the set of all possible relations C contains 2^{n^2} asset combinations. A product family is a subset S drawn from this asset relation such that $S \subset C$ comprises of instances where requirements are met and consistency of relationship is satisfied.

Our goal is to automate the search of this solution space S and so we need a computational model of our product family S . Graphs provide the required mathematical model since there is a one-to-one correspondence between relations and graphs. Graphs also provide us with computational notions of Isomorphism that are necessary in designing a software evolution framework.

A. Graph Model of Software Assets

Software assets shall be modelled as the set V of vertices of a graph G . Similarly, the relationships between the assets is the edge set E . Software assets are made of other software assets, therefore we can adjust the assets set cardinality to a particular size.

B. Evolution by Isomorphism

Isomorphism and sub-graph Isomorphism gives two operations with which we can formalise a predictor framework for evolution. Isomorphism gives the ability to architect equivalent products, establish consistency across abstractions. Sub-graph isomorphism can be used to capture differences in product lines where some parts of the new design is isomorphic to evolved graph. We detail the product and check if it is isomorphic (abstract away, fine grain define) with a valid product. Likewise, we abstract further and check if the result is isomorphic against our current product graph. This isomorphic preserving operations guarantee that the resulting product will preserve the desired properties. However, primitive operations allow for the product to be evolved without the isomorphic checks. Fig. 2 shows an illustration of searching over the solution space resulting in an isomorphic product.

Fig. 2. Graph analysis framework [12] using sub-graph isomorphism detection [13].

1) Graph isomorphism

The subset S contains all the possible solutions. We need a seed (initial) solution that complies with the requirements to be represented as a graph G . We then need a mechanism to evolve from the seed an equivalent solution set. We model the evolution of equivalent products as *Isomorphism* of graph.

Definition 1 (Isomorphism): Let G, H be two graphs. A mapping $f: G \rightarrow H$ is an isomorphism, where f is bijective, with the property that if a, b are adjacent in G then $f(a), f(b)$ are adjacent in H . Graphs G and H are then considered to be isomorphic.

If we start with the seed solution G we can then traverse the solutions space using operation f to find other solutions. Following this reasoning, we define a *traverse* operator, always moving within some level of abstraction:

$$\forall g \in G \exists h \in H \ t: V(G) \rightarrow V(H)$$

C. Primitive Operations

We can construct the expand operator by changing a vertex

node to a more detailed graph. Similarly by abstracting away a graph into a vertex node of higher level (abstraction) we construct the collapse operator. These will adjust the level of detail of the seed solution S .

$$\begin{aligned} \text{expand: } V &\rightarrow V' = V \cup V_{\text{subproblem}} \\ \text{collapse: } V' &\rightarrow V \end{aligned}$$

IV. APPLICATIONS OF SOFTWARE EVOLUTION USING OUR GRAPH MODEL

A. Model-Driven Production Line

The outcome of this model has strong parallelisms with the model-driven software production line multi-model. The transition from domain engineering to application engineering can be achieved with the set of operations devised. It may be plausible for the engineers to establish what kind of outcome is needed and then let the model discard (or find a valid region, depending on the point of view) from the solution space. In practice this operations will evolve models in this multi-model framework actually turning general systems into customized versions adapted to the client's needs. It is possible to use the model to preserve properties (for strategic reuse, for instance) as showcased by the Carnegie Mellon University's Software Engineering Institute software product line catalogue [14]. Families of isomorphisms located in the solution space could point to solutions akin to software families. It is certainly possible to optimise the relationships of different (software) assets in such a way that similar improvements are feasible.

Let f be an isomorphism between *Domain Engineering* (DE) and *Application Engineering* (AE) Cartesian product of their assets set:

$$\begin{aligned} \forall d \in DE \exists a \in AE \\ f: G(DE) \rightarrow H(AE), f(d) = a \end{aligned}$$

Fig. 3. Transition from domain engineering to application engineering [3].

We can evolve using isomorphisms from any Domain engineering view to a particular Application Engineering a view, and thus, define the solution space of a product line of software solutions based on the graphs defining the dependencies between the two views. This is done by using the graphs G, H and finding isomorphisms which comply with this view shift as illustrated in Fig. 3.

B. Architectures Isomorphic to Cloud Systems

Cloud software is clearly defining the functions of the

server and the client in two stacks communicating by HTTP. This whole development mimics the Model-View-Controller (MVC) architecture. With the advent of the smartphone all clients are collapsing to a single client stack, merging traditional desktop development, mobile development and web development. The client is concentrating on the presentation layer. The whole stack is shaping to communicate with the client and therefore to supply the information in optimal ways. When needed, a whole layer is bypassed communicating directly to the relevant client layer. A remarkable development is that the OS is being virtualized and thus treated as yet another layer in the stack. This allows for the server stack to provide variety of different server-side development frameworks.

Consider a scenario where a software system for a desktop needs to be evolved to a system that is compatible for a cloud system. We can define the asset relationship for the two systems as follows:

$$\begin{aligned} A &= \{\text{server}, \text{database}, \text{AJAX}, \text{UX}\} \\ UX &= \{\text{J2SE}, \text{gtk+}, \text{dotnetAPI}, \text{win32API}\} \end{aligned}$$

and the asset relation is $A \times A$, from which we can obtain a solution G represented as a di-graph.

The asset set in a cloud context is:

$$B = \{\text{database}, \text{server}, \text{AJAX}, \text{browserbased}\}$$

and a corresponding asset relation $B \times B$, whose equivalent di-graph of a solution is denoted by H .

We consider G to be the seed solution from which we want to evolve into H . We can cast the problem of a desktop product to a cloud product in terms of an isomorphism. Let f be the isomorphism:

$$f: G(A) \rightarrow H(B)$$

with the following mapping (example isomorphic graph as shown in Fig. 4):

Fig. 4. Graph isomorphism example [15].

We can delete A_{V3} and replace it with the new property preserving (isomorphism preserving) B_{V3} . In a desktop to cloud context that could make the Cartesian pair $\{\text{server}, \text{UX}\}$ be replaced by the pair $\{\text{server}, \text{browserbased}\}$.

The preservation of property of the rest of the software is equivalent to identity mappings for the rest of the vertex (assets) as follow:

$$f(A_{V1}) = B_{V1}$$

$$f(A_{V_2}) = B_{V_2}$$

$$f(A_{V_4}) = B_{V_4}$$

Likewise, regulatory compliance could make us shift documentation to document intensive ISO-9000. However, we may not desire to change the isomorphic graph but just the topology albeit being different products. Similarly, other assets, like a particular software logical view, could be preserved in this fashion. We can also grow the graph and search for the next isomorphic graph, since we want to add new properties.

Validating every single pair may not be possible or desirable. We can establish what we consider as valid pairs using a digraph. This enables us to discard undesired combinations and it does not constraint the creative options of the practitioners. For simple examples it could be trivial to validate, but If the product is complex and thus the graph, there could be a significant amount of properties to check and therefore the utility of this model based search approach is justified.

Given the definition of isomorphism, there must be a path of adjacent vertices in isomorphic graphs. Therefore, two isomorphic graphs guarantee a dependency path between one product (desktop solution in this case) and its equivalent Cloud implementation and vice-versa. This way, key properties are preserved by the isomorphism. However, we know that the system is different so we can argue the merits of one product against the other. To add another level of complexity we can make both solutions isomorphic to a logical or architectural view to preserve properties of a higher level of abstraction. We can also create a taxonomy of solutions based on these higher level isomorphism checks.

This taxonomy will enable us to identify families of interest of products. We can conjecture how this could be classed as a sort of architectural-time programming.

V. CONCLUSION

We have presented a graph model as a rigorous approach to software evolution. Our key idea is to allow the exploration of a set of well defined operations to architect equivalent solutions. We showed that isomorphism enables exploration of a solution space in a property preserving manner. The architectural changes can be introduced and recorded in a systematic way. Using isomorphisms of known solution structures to model new software could leverage the acquired previous knowledge and even find relationship insights beyond the art of software development. By freeing the engineers with relevant automation we actually get more engineering, and paradoxically, we further the art aspect by empowering their creative choices.

Our future work will consider the computational complexity of operations on graph and its effect on constructing practical predictor frameworks. One solution is to consider meta-heuristic techniques, from soft computing paradigm, i.e. genetic algorithms, to devise operations beyond plain operations on the graphs. Other good candidate for a meta-heuristics would be (to train) a neural network for recognition. This could work a global scale or at fine level depending on what is what we are looking for. These meta-heuristic can be used to discard solutions that do not meet

some criteria and therefore highlight some pockets or clustering of solutions. Other possibility is to consider polynomial time algorithms for graph operations that are feasible under certain conditions [16].

REFERENCES

- [1] K. Bennett, M. Munro, N. Gold, P. Layzell, D. Budgen, and P. Brereton, "An architectural model for service-based software with ultra rapid evolution," in *Proc. IEEE International Conference on Software Maintenance*, 2001, pp. 292–300.
- [2] J. Visser. (2009). Change is the constant. [Online]. Available: <http://ercim-news.ercim.eu/en88/keynote/change-is-the-constant>
- [3] ERCIM news 88. [Online]. Available: <http://ercim-news.ercim.eu/en88>
- [4] T. Mens, "The ERCIM working group on software evolution: The past and the future," in *Proc. the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, 2009, pp. 1–4.
- [5] P. Bhattacharya, M. Iliofotou, I. Neamtii, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proc. the 2012 International Conference on Software Engineering*, 2012, pp. 419–429.
- [6] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proc. the 14th International Conference on Software Engineering*, 1992, pp. 392–411.
- [7] M. Harn, V. Berzins, Luqi, and A. Mori, "Software evolution process via a relational hypergraph model," in *Proc. 1999 IEEE/IEEE/ISAI International Conference on Intelligent Transportation Systems*, 1999, pp. 599–604.
- [8] J. Greenfield and K. Short, "Software factories: Assembling applications with patterns, models, frameworks and tools," in *Proc. Companion of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003, pp. 16–27.
- [9] S. Reiss, "Constraining software evolution," in *Proc. International Conference on Software Maintenance*, 2002, pp. 162–171.
- [10] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, 1 edition., Wiley, 2004.
- [11] Software factories: Assembling applications with patterns, models, frameworks, and tools. (Apr. 2013). [Online]. Available: <http://msdn.microsoft.com/en-us/library/ms954811.aspx>
- [12] Graph_tool.topology assessing graph topology- graph-tool 2.2.28 documentation. [Online]. Available: <http://graph-tool.skewed.de/static/doc/topology.html>
- [13] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976.
- [14] Catalog of Software Product Lines. [Online]. Available: <http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm>
- [15] Biodatamining figure. [Online]. Available: <http://www.biodatamining.org/content/4/1/10/figure/F3?highres=y>
- [16] Approaches to solving the graph isomorphism problem. (Nov. 2012). [Online]. Available: <http://smartech.gatech.edu/handle/1853/6494>

Noel Vizcaino is a doctoral researcher in computer science at the University of Reading (UK). He has a BSc in computing from the University of Abertay Dundee (UK) and a MSc in applied computing technologies from University of Northumbria (UK). His professional career includes the development of fleet management systems and M2M embedded systems for British companies.

M. Manjunathaiah is a lecturer in computer science and informatics at the University of Reading in UK. His research interests are in massively parallel computing models, scalable parallel computing architectures, techniques for practical parallel programming and formal methods for hardware software designs. He received a PhD degree from the University of Southampton, UK, for his thesis on compilation techniques for parallelization and parallel program verification. After his doctoral studies he held research fellowships at the University of Reading, Oxford University Computing Laboratory and University of Manchester all in the U.K., researching into Polyhedral model, co-design in the CSP model and dynamic parallelism.