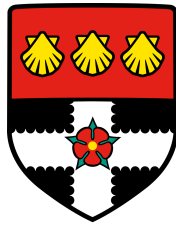


UNIVERSITY OF READING

Department of Computer Science
School of Mathematical, Physical and Computational
Sciences



Software Evolution:

*Hypergraph based model of solution space and
meta-search*

by

Noel VIZCAINO

Thesis Advisor: Dr. M. MANJUNATHAIAH

Thesis submitted for the degree of Doctor of Philosophy

Sept, 2016

Acknowledgements

I would like to thank my advisor Dr. M. Manjunathaiah. First, for his fruitful advice. Secondly, for his efforts. And finally, for his patience.

To my family.

Abstract:

A *hypergraph* based model of software evolution is proposed. The model uses software *assets*, and any other higher order patterns, as reusable components. We will use software product lines and software factories concepts as the engineering state-of-the-art framework to model evolution.

Using those concepts, the solution space is sliced into sub-spaces using *equivalence classes* and their corresponding *isomorphism*. Any valid graph expansions will be required to retain information by being sub-graph isomorphic, forming a chain to a solution. We are also able to traverse the resulting modelled space. A characteristic set of operators and operands is used to find solutions that would be compatible. The result is in a structured manner to explore the combinatorial solution space, classifying solutions as part of families hierarchies.

Using a software engineering interpretation a viable prototype implementation of the model has been created. It uses configuration files that are used as design-time instruments analogous to software factory schemas. These form configuration layers we call fragments. These fragments convert to graph node metadata to later allow complex graph queries. A profusion of examples of the modelling and its visualisation options are provided for better understanding. An example of automated generation of a configuration, using current Google Cloud assets, has been generated and added to the prototype. It illustrates automation possibilities by using harvested web data, and later, creating a custom isomorphic relation as a configuration.

The feasibility of the model is thus demonstrated. The formalisation adds the rigour needed to further facilitate automation of software craftsmanship.

Based on the model operation, we propose a concept of organic growth based on evolution. Evolution events are modelled after incremental change messages. This is communication efficient and it is shown to adhere to the Representational State Transfer architectural style. Finally, The Cloud is presented as an evolved solution part of a family, from the original concept of The Web.

Contents

1	Introduction	1
1.1	Introduction	1
1.1.1	Software Evolution as modelling target	2
1.1.2	Higher level software components	3
1.1.3	Automation	3
1.1.4	Graph based modelling	4
1.1.5	Graph modelling	6
1.1.6	Chapter summary	6
1.2	Research methodology	7
1.3	Software Evolution	8
1.4	Existing evolution related Tools	9
1.5	Overview of the model	9
1.5.1	Asset and their relationships	10
1.5.2	Solutions views (perspectives)	10
1.5.3	Equivalence class	10
1.5.4	Hypergraph and artefacts	11
1.5.5	Hypergraph expansions and software families	11
1.5.6	Organic growth	12
1.5.7	Examples and case studies	13
1.6	The basis for Automation	13
1.7	Conclusion	15
2	Software Evolution	17
2.1	Software Evolution	18
2.1.1	Introduction	18
2.1.2	Evolving Software	19
2.1.3	Evolution-ready or <i>evolvable</i> software	21
2.1.4	Conclusions	22
2.2	Analysis and Visualization tools	24
2.2.1	Introduction	24
2.2.2	Holistic Software Evolution: CodeCity	24

2.2.3	An Environment for dedicated Software Analysis tools: Moose	24
2.2.4	Recovering Software Architecture with Softwrenaut	25
2.2.5	Process Mining Software Repositories	28
2.3	Software tools for Architecture and Design	28
2.3.1	Automated Synthesis of CONNECTors to support Software Evolution	28
2.3.2	Emergent Middleware: Starlink	29
2.3.3	Pat-Evol: Pattern-driven Reuse in Architecture based Evo- lution for Service Software	29
2.3.4	CAPucine: Context-Aware Service-Oriented Product Line for Mobile Apps	30
2.3.5	MoDisco Framework	30
2.3.6	Rascal metaprogramming	31
2.3.7	Evolving Software for Molecular Modelling	31
2.4	Evaluation of the techniques	31
2.5	Software evolution approaches	35
2.5.1	Lower level: code and modules	35
2.5.2	Higher level: Architecture based	35
2.5.3	Graph based	35
2.5.4	Model-driven (SPLs)	37
2.6	Relational model overview	37
2.6.1	Modelling asset relationships	40
2.6.2	Evolution guided by Isomorphism	41
2.7	Example: Architectures isomorphic to cloud systems	41
2.8	Conclusions	44
3	A Relational Hypergraph based Model	46
3.1	Introduction	47
3.2	Preliminaries	47
3.2.1	Relations	48
3.2.2	Graph of Asset relation	49
3.2.3	Graph Isomorphism as a structure preserving operation	49
3.2.4	Sub-graph Isomorphism to identify related structures	50
3.2.5	Existence of Primitive operations	50
3.3	Relational Hypergraph Model	50

3.3.1	Evolution using Hypergraphs	53
3.3.2	Single <i>Asset</i> evolution	53
3.3.3	Hierarchical evolution of multiple <i>Assets</i>	55
3.3.4	Existence of Product lines	55
3.4	Encoding families using a Relational Hypergraph Model	58
3.4.1	Software Product lines: Single asset	58
3.4.2	Coupler enablers or the assets where to grow from	64
3.5	Multiple asset based product lines : Case studies	64
3.5.1	Case Study I: Evolution of Browser Technology	64
3.5.2	Case Study II: Cloud compute engine	67
3.5.3	Case Study III: Regulatory or Legal Constraints	69
3.6	Conclusion	70
4	Engineering The Elements of Evolution	72
4.1	Introduction	72
4.1.1	Software design insights	73
4.2	Engineering the model	75
4.3	Artefacts as complex operands	76
4.3.1	Assets as abstract building sub-blocks	77
4.3.2	Artefacts as a hypergraph based software factory schema	78
4.4	Evolution and Configurations	79
4.4.1	The model as the basis of a component model for evolution	81
4.4.2	Property transmission, recording and tracking	81
4.5	Evolution Operations	82
4.5.1	Core operations	83
4.5.2	Designing a seed artefact	84
4.5.3	Designing a coupler	84
4.5.4	Shift to Solution view	85
4.5.5	Shift to compatible Coupler	85
4.5.6	Evolution step: <i>Evolvè</i> with Δ under \simeq	85
4.6	Conclusions	87
5	Evolution Automation Feasibility	89
5.1	Introduction	90
5.1.1	The model in contrast	90
5.1.2	Python prototyping	91

5.1.3	The model	92
5.1.4	Cloud solution	93
5.1.5	ReST microservice	93
5.1.6	Use cases	93
5.2	Architectural Overview	96
5.2.1	Jupyter server	97
5.3	Configuration Context files	98
5.3.1	Encoding Assets	100
5.3.2	Encoding Artefacts	101
5.4	The Model as a whole	101
5.4.1	Artefact custom initialisation	103
5.4.2	Applying Configuration Contexts	104
5.4.3	Instance metadata	106
5.4.4	Configuration Contexts Automation	107
5.4.5	Evolution operations Δ	111
5.4.6	Evaluating and tracking desired properties	111
5.4.7	Searching for other isomorphic solution views	111
5.5	I/O and Visualisation	116
5.5.1	Displaying static graphs	117
5.5.2	Displaying dynamic graphs	118
5.6	Modelling: Examples of scripting use	120
5.6.1	Single asset evolution	122
5.6.2	Multiple assets evolution	126
5.7	ReSTful microservice	127
5.7.1	Selectable Test scenario (sandbox function)	129
5.7.2	ReST Resources	129
5.7.3	How to implement ReSTful operations	129
5.7.4	A test scenario to showcase the model basics	131
5.8	Notebook deployment and Jupyter access	132
5.9	Microservice deployment	134
5.10	Conclusions	136
5.10.1	Evolution as documentation	137
5.10.2	The model should be the deepest module	138
5.10.3	Splitting Artefact Class	139
5.10.4	Asynchronous operation upgrade	139

5.10.5	Future scaling	140
6	Software Evolution: organic growth	141
6.1	Introduction	141
6.2	Organic growth	142
6.2.1	Generative models	144
6.2.2	Predictors: Key search properties	144
6.2.3	Scanning using predictors	145
6.3	The Cloud Family: Branch evolution	145
6.3.1	Starting stage: The birth of The Web	145
6.3.2	The Client family	146
6.3.3	The Server family	147
6.3.4	The Cloud family evolution	148
6.3.5	The Cloud family: Analysis	148
6.4	Conclusions	149
6.4.1	Evolution process of growth	149
6.4.2	Families	149
6.4.3	The Cloud family	150
7	Conclusions	151
7.1	Introduction	152
7.2	Formal modelling considerations	152
7.2.1	Queries using graph based metadata	152
7.2.2	Automating solution (family) search	153
7.2.3	Observations and findings	154
7.3	A Finite State Machine generator	154
7.3.1	Solution family instance detection	154
7.3.2	Operations as data efficient messaging events	154
7.4	A (Scale-Free) network of assets assembler	155
7.4.1	Unlimited relationships	156
7.4.2	Existence of Simpler Solution families	156
7.4.3	The need for complex families	156
7.5	Model Implications and Re-Interpretations	157
7.5.1	The coupler as a solution subspace	157
7.5.2	Solution sub-space iterator	158
7.5.3	Emerging fractal spaces	159

7.5.4	[] _≈ as a Closure	159
7.6	Critical appraisal	159
7.6.1	Terminology and concepts	159
7.6.2	Models can be dangerous	160
7.6.3	The need for empirical evidence	160
7.6.4	Prototype implementation	160
7.6.5	Automation of craftsmanship	161
7.7	Recommendations for Future Research	162
7.7.1	Visual Modelling	162
7.7.2	The research of other implementations	163
7.7.3	Automatic stop using conditional traversal	163
7.7.4	Complex Data Configurations	163
	Bibliography	166
	Appendices	183
A	Appendices	183
A.1	Prototype notebook examples	183
A.2	Python files	183
A.2.1	csv2assets.py	183
A.2.2	launcherharvest.py	185
A.2.3	microservice.py	188
A.2.4	ioutils.py	195
A.2.5	model.py	201
A.2.6	artefact_views_test.py	208
A.3	Produced Support data	208
A.3.1	The tile view Jinja2 template	208
A.3.2	The single view Jinja2 template	209
A.3.3	Single view javascript data sample	212
A.3.4	Sample JSON Artefact graph format	212
A.4	Python stdout(console) prints	213
A.5	XML configuration files	215
A.5.1	Artefact XML sets master configuration schema to validate master	215
A.5.2	Artefact XML assets master configuration example	215

A.5.3	Artefact XML asset extra configuration fragment example . .	219
A.5.4	Google Cloud Launcher configuration fragment(layer)	221
A.6	Large images/captures	226

List of Figures

1.1	Software product lines development phases propagation [1] [2]	5
2.1	Differences between agile and heavyweight approach to software development. By Khan and Balbo [3].	21
2.2	ArgoUML source code map, generated using Codecity [4]. This tool is under an academic non-commercial licence.	25
2.3	Moose data visualisation with Mondrian [5] [6] (CC BY 4.0)	26
2.4	Softwrenaut [7] [6] (CC BY 4.0)	27
2.5	FASR+ProM [8] [6] (CC BY 4.0)	28
2.6	Emergent Middleware model: Starlink [9] [6] (CC BY 4.0)	29
2.7	Transition from domain engineering to application engineering. "The Multimodel in the Software Product Line development process". ISSI Research Group (Polytechnic University of Valencia). (CC BY 4.0) [10] [6]	38
2.8	GPLv3 Graph-Tool analysis framework [11] using subgraph isomorphism detection [12]	41
2.9	Graph isomorphism example. (CC BY-SA 3.0) by AAAS (adaptation). Wikimedia. [13]	43
3.1	Different Hypergraph views as described by Stell [14]. Left side: a Hypergraph $H = (V, E)$ where the edge set is $E = a, b, c, d, e, f$ and the vertex set is $V = s, t, u, v, w, x, y, z$. The right: the corresponding relation φ on $V \cup E$	51
3.2	A directed hypergraph example view [15].	51
3.3	Growing current graph produced through couplers.	53
3.4	Evolving equivalence classes: many views of a compatible solution (for illustration purposes)	56
3.5	Evolving families of products using coupler sets with cardinality k	57
3.6	Higher level $\{g_i\} \cup \{c_j\}$ gives rise to a (property preserved) new family of products	58
3.7	Firefox based architectural dependencies. Coupler graph via extension for augmented browsing or custom web post-processing capabilities (relaxed security sandbox may be available)	59

3.8	Architectural dependencies converging on coupler graph via GreaseMonkey for custom web post-processing	62
3.9	Different solutions from the same equivalence class	63
3.10	$FX \cup \{wasm_1, wasm_2\}$	66
4.1	Artefacts virtual connection	78
4.2	Artefacts virtual connection, artefact added	79
4.3	Practitioners define and affect artefacts through various means	83
4.4	Sequence of events defining a FSM [16] and also an evolution path to a higher solution G_n	86
5.1	View of the starting folder of the running Jupyter server using a web browser as the client. Existing notebooks with .ipynb extension.	94
5.2	Getting the SHA-1 hashed ID of a resource stored as <i>seed.json</i>	95
5.3	The prototype as a rudimentary graph imaging tool. Blue transparent circles added with LibreOffice Draw	95
5.4	Placement of custom Jupyter notebook files in a Google Cloud Datalab VM instance. [17]	96
5.5	Current prototype logical access as implemented. Green circles are web clients. Blue rectangles are server related and the white one is an interface. The purple trapezoid is an I/O function. Any other I/O is omitted as it is of general access (utility). Dotted lines are optional uses.	98
5.6	A Jupyter notebook editing cell featuring sample prototype code	99
5.7	Custom asset configuration fragment layer loaded into the graph as metadata with tag (label) <i>licence</i>	106
5.8	Google Cloud Launcher asset configuration fragment loaded into the graph as metadata.	110
5.9	Prototype notebook outputting current count of metadata labelled <i>property</i>	112
5.10	Prototype notebook outputting current pre-filtered valid isomorphisms as dictionaries.	114
5.11	Sample static graph output using matplotlib library	118
5.12	Isomorphic solution tile of some sample static views as generated and displayed.	119
5.13	Sample output using D3.js based GPL component [18, 19]	120

5.14	Generated compact tiled-view of expanded $[view] \simeq$ specimens (other views). It can be loaded within a Jupyter notebook or via a web browser. Jinja2 template version <i>iframe</i> sizes are bigger so the there is more room for each graphs.	121
5.15	Google Cloud Platform sample Artefact	124
5.16	Notebook featuring single asset source code for loading of JSON artefacts. Graph output depicted over.	125
5.17	Adding two couplers using assets such they require further isomorphic relationship.	126
5.18	Multiple asset notebook example.	128
5.19	Test scenario evolution.	130
5.20	Artefacts with graph on disk can be evolved using ReSTful operation handling. The page updates to the resulting base graph.	131
5.21	Small sample of the expected outcome of the test scenario. Tiled view of thick Δ evolved seed with many specimen views all belonging to the new $[solution\ graph]_{\simeq}$. $[seed\ graph]_{\simeq}$ belonging views are sub-graph isomorphic to all of them.	133
5.22	Loading JSON artefacts and the embedding of graph views. Graph output depicted over.	135
6.1	Logistic curve with various parameters. Wikimedia Commons. [20]	143
6.2	Evolution of the client affects the system as a whole.	147
7.1	1 family. 2 levels. 4 ordered sequences of Equivalent solutions operations. 4 paths.	158
A.1	Equivalence class generation of sample known static views.	227
A.2	All views evolved with the same artefact using the <i>evolve_delta()</i> thick Δ method.	228
A.3	Notebook text output. Includes data before and after evolution and configuration layering.	229
A.4	Sample test interactive output as a Jupyter notebook: "multiple simultaneous static graphs.ipynb"	230

List of source code samples

1	McIlroy's solution	74
2	Example of asset configuration fragment layering new metadata in design-time	80
3	Sample asset configuration file entry. Database technologies description in Google Datalab collection [17]	100
4	Basic Artefact class initialisation. However we need more data for it to be usable.	102
5	Custom class initialisations. <i>cls</i> is a convention for the class to be customised	104
6	We keep track of the added context fragments. It overrides metadata on tag name collision. Not a bug but part of corresponding metadata update.	105
7	Sample asset entries from the context fragment layer <i>views.xml</i> . Tags used are <i>licence</i> and <i>softwareview</i> . Any tag can use watching out for semantic <i>word clash</i> with any other used by metadata. This means they will capture the tag or label and later override related metadata.	105
8	Python property using <code>@property</code> in lieu of getter/setters: <i>properties</i> as a dictionary based node metadata	107
9	Generic choice of node metadata using existing tags (labels)	107
10	Creating a subset of the Cartesian product of assets in a <i>xml.etree.ElementTree</i> tree data structure. It is called inside of a loop with various categories. The loop should be moved inside for clarity and efficiency.	108
11	Evolution by graph expansion (thin Δ) using a one base graph (a key view in itself) and a coupler artefact instance.	113
12	Evolution by multiple graph expansion (thick Δ) using all stored views and a coupler artefact instance. All resulting solution guaranteed to be analogous. Their metadata fingerprint could be different.	115
13	Tagged metadata occurrence count. Also useful for solution view assessment and sorting	115
14	Checking for sub-graph isomorphism existence.	116

15	Using loaded node metadata to filter and find existing isomorphisms.	116
16	Defining asset names as node identifiers. These relationships model valid Cartesian pairs to form a relation.	123
17	Load JSON graph data to create an Artefact instance. This replaces the need to define node relationships as Cartesian pairs.	123
18	Using the evolve method (thin Δ) in succession.	123
19	Google Cloud Platform sample artefact modelling.	124
20	Reformatted data as JavaScript (initially used for testing)	126
21	Google Cloud Platform sample multiple asset based artefact generation.	127
22	Test scenario function	129
23	Operation processing by URL <i>/api</i> entry using HTTP verb POST . .	130
24	function test_scenario activated by http://127.0.0.1:5000/test_scenario	132
25	The model generates all offline data needed to embed within a notebook	134
26	Flask server launcher	136

List of Abbreviations, acronyms and nomenclature

- .NET Microsoft .NET Framework [21].
- ADL Architecture Description Language [22].
- ADM Architecture Driven Modernisation [22].
- AE Application engineering [23] [24].
- AJAX Asynchronous JavaScript and XML [25].
- Anaconda A Scientific Python distribution [26].
- API Application Program Interface.
- asm.js Low-level subset of JavaScript Mozilla specification (cross-browser) [27].
- AST Abstract Syntax Tree [28].
- Azure Microsoft Cloud services [29].
- Bokeh A canvas-based charting library for the web. [30].
- bs4 An error-tolerant HTML parser Python library [31].
- C# A general purpose programming language. [32].
- CASE Computer Aided Software Engineering.
- CMS Content Management System [33].
- CORBA Common Object Request Broker Architecture [22].
- CRUD Create, Read Update and Delete operation set [34] [22].
- CSS Cascading Style Sheets [35].
- CSV Comma-Separated Values data format standard [35][22].
- curl A command line networking tool [36].

- cygwin A command line unix environment layer for MS Windows [37].
- D3 Short for D3.js library [38].
- D3.js D3 is a JavaScript library for visualizing data using web technologies [38].
- DE Domain Engineering [23] [24].
- Debian A type of linux distribution [39].
- DOM Document Object Model [35].
- DSL Domain-Specific Language [22].
- EMF Eclipse Modelling Framework [22].
- Emscripten Emscripten is an LLVM to JavaScript compiler [40].
- Flask Python server for network microservices [41].
- FSM Finite State Machine [42].
- GA Genetic Algorithm.
- GCP Google Cloud Platform [43].
- Git A decentralised source control system [44].
- GML Graph Markup Language [45].
- Google Datalab. Google Cloud services [17].
- GPL GNU Public Licence [46].
- GPLv3 GNU Public Licence version 3 [46].
- GreaseMonkey Firefox extension allowing custom scripts [47].
- GUI Graphical User Interface
- HATEOAS Hypermedia as the engine of application state [48].
- HTML5 HTML version 5 [35].
- HTTP Hypertext Transfer Protocol [35].
- IaaS Infrastructure as a Service [22].

-
- ID Unique identifier.
- IDE Integrated Development Environment.
- IPython Python-exclusive Jupyter predecessor. Incorporated in Jupyter as a Python engine. [49].
- J2EE Java 2 Platform Enterprise Edition [22].
- J2SE Java 2 Platform Second Edition [22].
- Jinja2 A web templating system [50].
- jQuery Javascript based declarative library for DOM manipulation [51].
- JS Javascript
- JSON JavaScript Object Notation [52].
- Jupyter A multi-language system for client-server programming and deployment. [53] [54].
- Linux Free and open source unix-like operating system [55].
- LLVM Low Level Virtual Machine [56].
- Matplotlib A Python library for the generation of static graphics and charts [57].
- MDE Model Driven Engineering [58] [24] [22].
- MDPL Model Driven Production Line [6].
- MDRE Model Driven Reverse Engineering [59] [6].
- MS Microsoft
- MVC Model-View-Controller [60].
- NetworkX A Python network(graph) library [61].
- OMG Object Management Group [22].
- OO Object Orientated
- OS Operating System.
- p2p peer-to-peer

- PaaS Platform as a Service [22].
- PNG Portable Network Graphics. Image filetype [35].
- POSIX Standard Unix APIs [62].
- ReST Representational State Transfer [48] [22].
- ReSTful (adjective) It adheres to the ReST architectural style principles.
- SaaS Software as a Service [22].
- Scipy Scientific library for Python [63].
- SCM Software Configuration Management. Broader concept than VCS [64].
- SE Software engineering.
- SHA-1 Secure Hash Algorithm 1 [65].
- SOA Service-oriented architecture [22].
- SpiderMonkey Firefox rendering engine [66].
- SPL Software Product Lines [67].
- SQL Structured Query Language [68].
- Subversion Centralised source control software from Apache Foundation [69].
- SVG Scalable Vector Graphics [35]
- SVN Apache SVN. A source control system [69].
- TamperMonkey Multi-browser extension allowing custom scripts [70].
- URI Uniform Resource Identifier [35].
- URL Uniform Resource Locator. The web URI [35].
- V8 Google's Open source Javascript engine. Part of Chrome browser. [71].
- VCS Version Control System. Part of SCM. [44].
- VF2 Algorithm for fast sub-isomorphic graph testing [72].
- VM Virtual Machine [27].

W3C World Wide Web Consortium [35].

wasm Short for WebAssembly binary [27].

WebAssembly The binary version of asm.js code [27].

WebGL Web Graphics Library [66].

wget A command line networking tool [73].

Wiki Software for collaborative web content creation [44].

win32api Standard API access expected in a Microsoft Windows 32 bit environment.

WinForms GUI library part of Microsoft .NET Framework.

XML eXtensible Markup Language [35] [22].

XML-RPC XML based Remote Procedure Calls (RPC) [22].

XPCOM Cross Platform Component Object Model [66].

XSLT eXtensible Stylesheet Language Transformations [35].

XSS Cross-Platform Scripting [74]

XUL XML User Interface Language [66].

YAML YAML Ain't Markup Language [75].

Introduction

Contents

1.1 Introduction	1
1.1.1 Software Evolution as modelling target	2
1.1.2 Higher level software components	3
1.1.3 Automation	3
1.1.4 Graph based modelling	4
1.1.5 Graph modelling	6
1.1.6 Chapter summary	6
1.2 Research methodology	7
1.3 Software Evolution	8
1.4 Existing evolution related Tools	9
1.5 Overview of the model	9
1.5.1 Asset and their relationships	10
1.5.2 Solutions views (perspectives)	10
1.5.3 Equivalence class	10
1.5.4 Hypergraph and artefacts	11
1.5.5 Hypergraph expansions and software families	11
1.5.6 Organic growth	12
1.5.7 Examples and case studies	13
1.6 The basis for Automation	13
1.7 Conclusion	15

1.1 Introduction

There is a wealth of existing software *assets* that needs to be reused. There is a demand of ever more complex software. We see software as an evolving system

bound by time and its corresponding changing environment. The future is uncertain so we need evolution-ready or *evolvable* software. Software evolution also represents a development process lifecycle in itself.

This research is based on these pillars:

1. The software is made interrelated (software) assets. The practitioners can establish which of those relationships make sense within an engineering context.
2. To model the software evolution paths we shall use these higher level artefacts as software asset-based components.
3. The resulting model is feasible to be implemented using current technologies.
4. The model based tool shows a degree of automation is possible.

1.1.1 Software Evolution as modelling target

Models are representation of the target system, a part of the reality of the world. A target system can be governed by a theory, with laws and axioms, and then be interpreted as the model [76]. Software evolution can be interpreted as a process of step-wise changes in a feedback-loop to adjust to the environment [77]. The changes are the change in the relationships among its software constituents. Therefore, we need to encode the constituents, their relationships and their step-wise changes. The model is based on software assets and their relationships. A relational model.

A desired software solution should exhibit properties leading to strategic reuse and also to be able to evolve by assemblage or composition with finer elements. Further *evolvable software* should be also an outcome of evolution [77]. The scope will be limited to architectural or higher (software) abstraction level. However, other models operate at any level of abstraction and they will be considered for study. Our approach will not depend on source code level minute changes but to always work at the highest level of abstraction possible.

1.1.2 Higher level software components

Software evolution, in the software product lines context, is an emerging area. It aims to define new development frameworks with higher order patterns of reuse based on software *assets*. Novel software development paradigms such as software factories have been proposed to integrate the innovative capabilities of the practitioner using a systematic methodology [78] [79].

New models of evolution based on higher order patterns of reuse, as in the software factory paradigm [78] [79], are currently shaping the landscape of model driven approaches. Our interest is the generation of software akin to SPLs and their related reconfigurability (also their evolution) [80]. Evolution has to cope with the complexity of software evolution in a continuous evolution context. Traditional approaches to software evolution (as a model driven approach to transform an abstract model to an implementation) are being replaced with similar abstractions. These include the identification of reusable forms in the form of invariants for a family of related applications. Such new developments are even impacting on the laws of software evolution itself in proposing revisions [81] [82] [83]. We present novel graph-based modelling frameworks for emerging reusable forms. We can formalise the structural properties of evolutionary paths and a structured means to traverse a combinatorial evolution as the solution space.

1.1.3 Automation

The model enables automation of any engineering aspect. These include the search for any engineering insight, including but not limited to, desirable evolution paths. The outcome is intended to be an aid or tool designed to empower practitioners not replace them. Complex and tedious tasks are automated and thus free the practitioners in favour of more creative or higher level tasks, *automation of craftsmanship*.

The critical issue is the realisation of this grand goal of *automation of craftsmanship*. It is convenient to design it as a rigorous evolution framework. This is required to provide a higher level of abstraction that captures reuse and thereby reduces the complexity of the evolution in a combinatorial solution space. This is due to the explosion of possible element relationships to consider in a complex

software project.

1.1.4 Graph based modelling

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. Such analysis is valuable to discover useful properties to aid software evolution. For instance in a recent work, graph based techniques are used to infer structural changes and also to predict defects in releases [84]. Other works such as [85] apply program dependence graph techniques to track the desired software attributes or use hypergraphs as alternative models for evolution [86].

In our work we are aiming to build a general framework to capture evolution in emerging trends such as software factories in particular and SPL in general. The artefacts under consideration are not just source code or a single software instance but a collection of assets [78] [79]. Software factories yield families or classes of software products that can be considered as the result of structuring evolution systematically as software product lines [67]. We are interested in ‘software that evolves organically’ phenomenon. We loosely define it so as to describe the kind of configurations’ evolution that happens in an open collaborative distributed team projects, as in open source projects or open frameworks. Another interesting development is compute engines on the cloud. They enable higher order patterns of reuse in the form of compute engines. Their basic assets provides new ways of architecting software solutions that can lead to an ever larger family of product lines in a Cloud Systems context [87]. In both these scenarios, the traditional notion of evolution is replaced by higher order abstractions and patterns of reuse. Thus, it affects software evolution dynamics in different ways. For both these scenarios, we are interested in constructing rigorous evolution techniques that can automate the exploration of evolutionary paths by constructing attribute *predictors*.

A software product family is a collection of software-intensive systems having common assets and sharing architectural properties. These are derived from the relations among assets [67]. This collective asset arising in this paradigm is

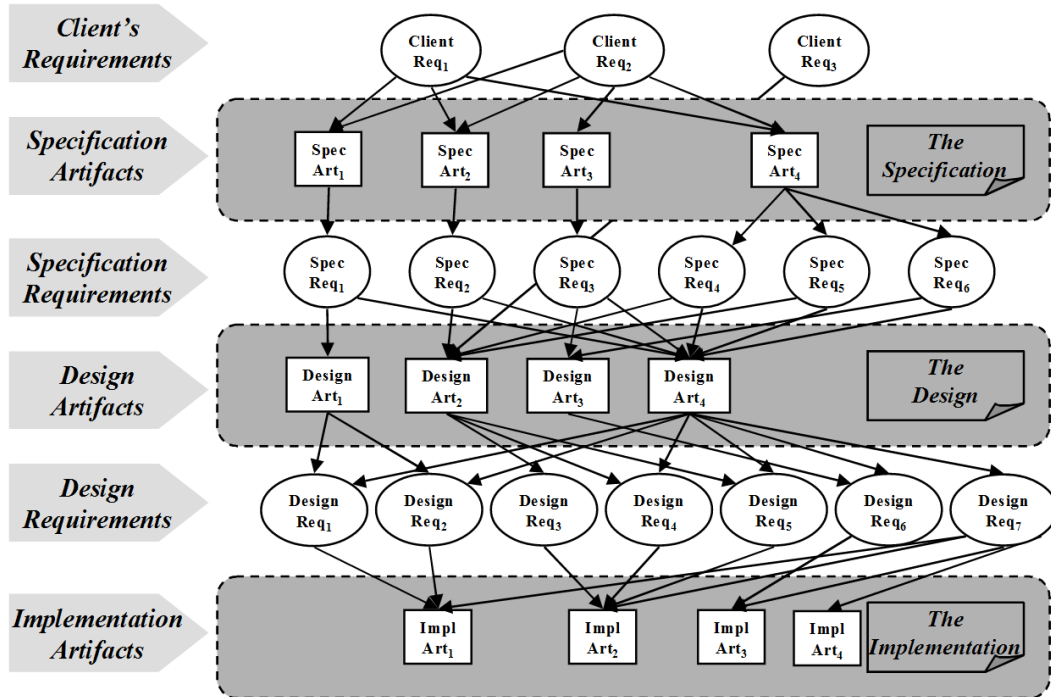


Figure 1.1: Software product lines development phases propagation [1] [2]

shown in Figure 1.1.4. The software architecture for a product family captures the variabilities and commonalities of the product line. New products in the family are instantiated from the same reference architecture and simultaneously variations are introduced to differentiate the products. The evolution in this context consists of a managed set of variation points. These enable a controlled diversification of the product family. Managing variabilities and commonalities of a product family is a challenging task. Therefore, methods to automate the (evolution) process are an integral part of a product line framework [78] [79].

In our framework we propose using graph theoretical notions of *isomorphisms* as attribute preserving operations. The model has to find commonalities and viable solutions for a family of software systems as it evolves. We evolve software systems to introduce variabilities using defined operations on these graph structures to explore the vast combinatorial *solution space* in a more structured manner.

A characteristic feature of this product line evolution paradigm is that the products are built from common assets in a prescribed way. We can capture the common foundations shared by a collection of software systems in terms of *meta* graph models and graph transformations such as isomorphism. Software factory template consisting of reusable artefacts that include assets beyond just code (such as samples, custom tools needed to build the members of the product family, guidelines and others). All can be uniformly treated as higher order patterns of reuse. Our graph model is based on the hypothesis that solution-wide attributes will be preserved by isomorphisms. Therefore, the task of the practitioner is to find a seed solution that will satisfy the requirements and then use a *predictor* properties. They use the underlying model to suggest a family of valid alternative solutions. The practitioner can then make a choice based on some qualitative judgements. The advantage is that the search could be automated to some degree. Practitioners will be only required to devise a grand strategy and/or to preserve key attributes. This is analogous to the concepts explored in [88]. We are empowering the practitioners to find novel solutions by automation of the detailed (and error prone) aspects of evolution and thus empower the practitioner creativity in exploring higher level design attributes.

1.1.5 Graph modelling

The basic software engineering elements to model evolving relationships are abstract objects representing software pieces. We shall call them *assets*. To address their concern and context, a graph structure has been selected as the foundation of the model. Design is restricted to high level software pieces (assets). Thus, we limit the scope of the relationships considered but it does not limit the model potential at a later stage. These relationships are valid pairs within a Cartesian product of known asset sets. The graph helps to simplify the modelling as many pairs will not make sense from an engineering perspective. We will use a Digraph to account for this fact (ordered pairs) but in general the order of the pair might not be relevant if the relationship is really abstract. It is based on conventions. Sometimes the relationships are that of *interdependence*.

1.1.6 Chapter summary

The topics researched use the following logical chapter order:

1. Chapter 2 Evolution. It covers concepts of software evolution and an analysis of existing tools within an evolution context by researching their material when available.
2. Chapter 3 Hypergraph Model defines the hypergraph model in detail providing examples of its use in Software Engineering problems.
3. Chapter 4 Engineering the Elements of Evolution. A detailed design of the model linked to software engineering topics. The underlying operation is thoroughly explained. Operations and operands are defined. It lays the basis for an implementation.
4. Chapter 5 Evolution Automation Feasibility. Demonstrates through the creation of a working prototype that the concepts and the design researched are viable and that they can be implemented.
5. Chapter 6 Organic Growth and Evolution deals with the links between our definition of organic growth as a by-product of the model operation as designed.
6. Chapter 7 Conclusions. Research findings and recommendations for future work. A critical appraisal of the research process and outcomes.

A detailed introduction to the concepts as they arise (by chapter order) follows. Chapter 6 is explained last not to break the order or said concepts.

1.2 Research methodology

There are many research methodologies, using the scientific method, which vary among disciplines. The research method used is "Constructive Research" (Based on Ontological Realism) [89]. There is an objective reality we are trying to understand with our limited means. The main source of information is the Software Engineering body of knowledge. This research combines theory and practice.

The theory also applies existing mathematics knowledge. A formal model provides rigour. The main building block shall be "the relationships of objects". There are some software engineering pre-conditions setting the context from where to build a model. Models are an oversimplification of reality (ontological foundation viewpoint) however they also narrow the scope of the problems and help abstract away unnecessary concerns.

The model implementation has to be feasible. In other words, it needs to be actually implementable using the currently available state-of-the-art. There could be many possible implementations but one is to be provided as evidence of feasibility. This implementation shows it can be (empirically) reproducible. Also, different implementations will be equivalent from the modelling point of view.

1.3 Software Evolution

Software evolves to adapt to a changing environment. With new information appearing the system needs to be updated. New requirements reflect changing needs. This new information also includes the discovery of bugs and other general maintenance may be needed. All these adjustments were explored by Lehman [90] as a whole set of 8 laws. This maintenance orientated research origin is represented as The Lehman Laws of Software Evolution. These laws are a good starting point to discuss the scope of the problem and a descriptive list of challenges for tackling.

Revised Lehman's Laws of Software Evolution [91] [92] with some clarifying comments(our interpretation):

1. Continuing Change: Program must be adapted or becomes unfit.
2. Increasing Complexity: Complexity increases with evolution unless deliberately dealt with.
3. Self-Regulation: Evolution is self regulated so the development process and outcome match the normal distribution.
4. Conservation of Organisational Stability (invariant work rate): The effort on evolving software will be constant thorough its lifetime.
5. Conservation of Familiarity (growth rate decrease): A critical level is reached, for same behaviour, therefore growth slows for that level.
6. Continuing Growth: Software *crystallises* to fit future needs and to fix bugs.
7. Declining Quality: The increasing feeling of unfitnes for non-evolving software.

8. Feedback System: Multi-level closed feedback loop is of essence to understand and evolve software.

There are other views of the concept of software evolution. Grossly unplanned development under heavy uncertainty is one of the motivators of viewing software evolution as a process. Also, such process should lead to *evolvable* software as an outcome. It should be designed for change and adaptation so uncertainty for future conditions are better tolerated [77].

1.4 Existing evolution related Tools

Software evolution tools have been explored and studied to gain insights on state-of-the-art approaches. The tools are diverse, they use many techniques and technologies. Some aspects have been useful for the purpose of exploring future tool design options. There are two different grand categories in our opinion. One is the tools that deal with analysis and visualisation category. These are centred in providing a snapshot of the software attributes state at present, the past evolution and future directed evolution prescription. Code analysis and code mining techniques are quite popular. The other category comprises the tools centred in architecture and design concerns. The practitioner is required to learn how to use the frameworks. These latter tools enhance the evolution-readiness properties we previously discussed. In summary, these criteria were used to assess the relevance of the tools and thus, their inclusion. This classification analysis is finally synthesized in a comparison table, the type of tool, techniques and technologies involved, and the tools advantages and disadvantages were evaluated.

1.5 Overview of the model

A model for software evolution has been researched. The model is formalised for rigour and to facilitate the automation at later stage. A strategic and planed approach to reuse was required. We borrow the concept of asset from **Software Product Lines** (SPLs) [93]. Configuration schemas to generate SPLs targeting different views was studied in detail by Greenfield [79] [78]. We can use both as conceptual foundations for the model to target. It should be noted that SPLs, and related software factories, are an ideal and convenient target to illustrate the model **but not necessarily the only target**.

1.5.1 Asset and their relationships

Definition 1 - Asset:

Architecture and higher level of abstraction constituents will be called assets.

The assets are related. These relationships will be pairs of assets and can be valid(possible) or not. Engineers, scientists and software architects are the *practitioners* who will validate the configurations describing these relationships.

Let the set of assets be \mathcal{A} . The set $\mathcal{A} \times \mathcal{A}$ defines the set of all potential asset relationships combinations, valid or not. If the set \mathcal{A} has n elements then the set of all possible relations C contains 2^{n^2} asset combinations. A relation R is a Cartesian product subset. Relations of assets can be represented as graphs. We will explore further relational concepts in the model chapter.

1.5.2 Solutions views (perspectives)

The model encodes the evolution steps towards a solution. The evolution steps create evolutions paths. They can be recorded, from start to end, allowing their study. Traversals of the different branches representing evolution is feasible in various ways. The solution space is comprised by all the different solutions views. Solutions views are perspectives to a problem using asset relationships. Using the term perspective sound even more abstract so we will stick to *software view*.

Definition 2 - Solution view:

Set of valid asset relationships. This is a subset of all possible valid relationships. It can be represented or identified by its resulting directed graph.

Isomorphism

A solution view can be encoded as a graph. Isomorphism is a structure preserving operations that can be performed on graphs. Equivalent solutions are considered isomorphic and their graphs will be isomorphic. New solutions can be found by searching for isomorphic graphs or sub-graphs.

1.5.3 Equivalence class

We found that the generality of the structure gives further freedom to encode relationships. This serves to pack all the solutions views into a solution or per-

spective solving a concrete engineering problem. This represents a slice of the solutions space. Since we chose graphs to encode the solutions, views the slicing is an equivalence class.

Definition 3 - Equivalence class:

Disjoint partitioning of the solution space in sets made of equivalent solutions views. Denoted by using square brackets: $[]$.

1.5.4 Hypergraph and artefacts

The incidence structure chosen to encode the relationships of the many solution views is the Hypergraph. This represents relationships of objects among themselves on a many-to-many basis. The hypergraph links equivalence classes together in two ways. If configuration knowledge is available, an equivalence class can be defined. This equivalence class can be expanded by all theoretical graph isomorphisms that can be found. This governing relationship among equivalence classes will be denoted as \simeq .

Definition 4 - Artefact:

Denotes a software solution made of related assets and their configuration (properties) data. It is modelled as an instance of a hypergraph modelling a solution view belonging to $[solution] \simeq$. It can be represented or identified by its base graph.

An artefact is an engineering output in the classic software engineering terminology. This is also an engineering process output. The term also comes from SPLs engineering by-products.

Besides belonging to $[solution] \simeq$ it has to be fully initialised with available configuration information. A coupler graph can upgrade to artefact and thus identify compatible (equivalent) views. A solution will usually mean any compatible solution and a solution view is the specific solution. We use the graph of a solution view to represent it unequivocally. The model aims to systematically assemble solutions encoded as artefacts from searchable compatible sets to form a new artefact.

1.5.5 Hypergraph expansions and software families

We grow from an equivalence class $[solution] \simeq$ by expanding its base graph. We could discover new valid vertices and therefore establish new relationships. This

will enlarge the known equivalence class. We can also add a coupler graph and compose a grander hypergraph. This is allowed under \simeq . However, this will break the bijection of relations which make the base graph. The new evolved equivalence class will be governed by other isomorphism. This way we can create a chain of solutions. ($[solution]_{\simeq_1} = S_1, [solution]_{\simeq_2} = S_2 \dots [solution]_{\simeq_n} = S_n$) represent different solution sub-spaces or solution *families*.

These families of software are grown by evolution steps we will denote as Δ . Every Δ step has a new associated governing \simeq , thus creating a hierarchies and families.

Δ is performed under \simeq to grow a seed hypergraph with a coupler graph made of assets relationships. Thus, a path with many branches is created. This will characterise hierarchies of families. **Single asset evolution** will occur in a thin path. **Multiple asset evolution** represent the general case where the path is thick and evolution models a whole family branch. The way this path is created is the basis to consider this type of growth as organic growth.

1.5.6 Organic growth

There are many loosely related definitions of organic growth, like from the business and biology domains, notwithstanding others. The rationale for having one, in our case, is that it simplifies the discussion about a complex set of interrelated concepts. The power of growing comes from within the definition of the structure operations and operands. We will discuss how the emerging dynamics can help understand complex constructs like Cloud systems. These systems span many layers of interrelated and contrasting relationships. These cross software definition boundaries and sometimes model evolution from hardware to software. This is illustrated by the physical server on a server farm being replaced by a virtual machine instance. We are able to model higher level asset relationships like these. The resulting expansion is also organic as growth from within (the initial architecture is not abandoned). This is due to the sub-graph isomorphic chains binding the expansions.

Definition 5 - Organic growth:

Step-wise closed feedback-loop of directed compositions or expansions. Uses specific selections from devised collections of simple structures to incrementally create a more complex one. There will be a starting state and a way to internally

direct the growth to a desired future state.

These collections are limited by a hierarchy of $[solution]_{\simeq_k}$. Being $solution = selection_k$ for a particular isomorphic level. The result is a changed or evolved *itself* in resonance with the environment. Other sets of constraints and rules apply for operators and operands. Exponential growth will be constrained by resources. The *emergence* of new properties is expected.

The attachment point where to expand from must be selected. This selection has degrees of freedom of utmost importance for growth dynamics like a scale-free network [94]. These nodes bring over new potential attachment points candidates. In our model this is performed by explicitly sharing a common node both in the solution view and the coupler. This indeed obviates the need for extra parameters when designing the evolution operators.

1.5.7 Examples and case studies

Software engineering based examples of concrete solutions are discussed to clarify how the model operates. These include augmented browsing and cloud computing example contexts. These were chosen for didactic and illustration purposes as we assume a degree of familiarity with related software architectures and solutions.

1.6 The basis for Automation

Leon Osterweil in 1987 claimed "Software Processes are software too" [95]. Fundamentally, software development processes as well as software reuse can be pre-programmed. In this view, we are execution actors of such processes. The research of new architectures was also highlighted by Osterweil. We are able to define a software development process to be somehow partially automatable. A formal model can facilitate the automation of many aspects of the development process.

The software engineering interpretation of the operations and operands defined by the model are discussed in detail in the engineering chapter and then implemented in a sample prototype as evidence of feasibility. This features a class to store instance state. The instance will be the graph representing a solution view. $[graph]_{\simeq}$ is the set of all possible (compatible) solutions views and can

be populated by adding compatible nodes tagged as *isonodes* entries to the assets configuration file. These configuration files can be fragmented in context lists and loaded overwriting the metadata held by the graph nodes. This allows for design time configurations.

Extensible configurations

What is valid has to be configured. The configurations can be part of a Software Factory schema [79, 78]. Software Factories automate the production of software product lines (SPLs) with these configurations. The model supports information extensible configuration schemas are akin to a software factory configuration. These can be used to automate the production of SPLs. Resulting SPLs will comply with measurable requirements and properties specified in the configurations. Configurations are open to refinement and further extension.

Evolution Automation Feasibility describes a working prototype as one of the possible implementations of the model. It is provided as a demonstration that the model can be implemented. It allows to illustrate the model and to gain further understanding. This is not necessarily the only one interpretation, the correct or most suited one.

Prototype highlights:

- Implemented in Python 3 using the free Anaconda scientific distribution [96].
- Artefacts represented as a graph of asset relationships data encoded as JSON files.
- XML configuration asset master file with mandatory entries (*isonode* and *property*).
- XML custom configuration fragments associated to assets to set custom entries.
- It can be deployed as part of Jupyter Notebooks for interactive web execution. These can be deployed in supporting Cloud services.
- It generates graphs using Matplotlib and D3.js based components for static and dynamic display of graphs.

- Sample ReST interfacing using access to views as web resources. A first approach to the model as ReSTful to gain further insights.

1.7 Conclusion

Software evolution has two meanings. Firstly, it is a step-wise process. Secondly, it is a closed feedback loop of self-changes. It follows a growth or development which by itself is a procedure, an algorithm, to re-adapt to a changing the environment. It is the properties within the software itself that make it more prone to be easily *evolvable*, adaptable to change.

Model directions:

- Strategic planned reuse of assets and their relationships.
- Organic growth feasible as defined.
- Automation implementation feasible.
- Adaptable to other software designs because not only it is formalised but uses Software Engineering general concepts.

These objectives have been accomplished. Each chapter will put more emphasis in explaining the distinct topics and the research and techniques involved.

Proposed model highlights

- Small set of operations: *Shifting*(select) and Δ (*advancing*).
- Inner sub-operations: **Gather** views to shift to by *exploring* or *traversing*.
- **Shifting or steering**: shift to view and shift to compatible coupler (another view).
- **Advancing or Δ 's**: single asset Δ , multiple asset Δ (general case). These also can be viewed as thin branches or trunk branches depending how much solution subspace they explicitly consider. A thin branch can represent a thicker branch. Thus we will usually talk about single asset thin Δ .
- Advancing produces growth by creating a new graph expansion by coupler attachment under a new \simeq .

- Growth happens by thickening the branch, that is, expanding $[graph]_{\simeq}$ sub-space by adding isomorphism compatible nodes.
- Rich data operands based on hypergraphs encoding of asset relationships and their configurations. These operands are created by the equivalence class $[]$ **under** \simeq isomorphic operator.
- Emerging **software families** based on solution space partitioning.

If we think about this at a higher level of abstraction we are in position to deduce that a model based on software evolution should reflect the processes which exhibit organic growth. We can also deduce that the properties which make the software evolution-ready should be preserved.

Graph theory is an appropriate basis to model the relationships that need to be analysed in software evolution. Since it is familiar and there is plenty of theoretical knowledge, implemented tools and libraries based on such knowledge. We utilise these to develop a general framework.

Papers

Chapter 3 is the published paper [97]: Vizcaino, N. and Manjunathaiah, M., 2015. "Software evolution: a graph based model". *Lecture Notes on Software Engineering*, 3(3), p.164.

Presented at ICSTE 2014, 6th International Conference on Software Technology and Engineering. It combines literature review, the proposed problem scope and offers a solution. It also suggests possible applications.

For re-submission:

- "*Software Evolution formalisation with Graph Isomorphism*" submitted to *Automated Software Engineering* (Springer)
- "*A Relational Hypergraph Model for Software Evolution*" submitted to *IEEE Transactions on Software Engineering* with the more general concept of multiple asset evolution.

These gave the pointers to improve the chapter 3 further.

Software Evolution

Contents

2.1	Software Evolution	18
2.1.1	Introduction	18
2.1.2	Evolving Software	19
2.1.3	Evolution-ready or <i>evolvable</i> software	21
2.1.4	Conclusions	22
2.2	Analysis and Visualization tools	24
2.2.1	Introduction	24
2.2.2	Holistic Software Evolution: CodeCity	24
2.2.3	An Environment for dedicated Software Analysis tools: Moose	24
2.2.4	Recovering Software Architecture with Softwareaut	25
2.2.5	Process Mining Software Repositories	28
2.3	Software tools for Architecture and Design	28
2.3.1	Automated Synthesis of CONNECTors to support Software Evolution	28
2.3.2	Emergent Middleware: Starlink	29
2.3.3	Pat-Evol: Pattern-driven Reuse in Architecture based Evolu- tion for Service Software	29
2.3.4	CAPucine: Context-Aware Service-Oriented Product Line for Mobile Apps	30
2.3.5	MoDisco Framework	30
2.3.6	Rascal metaprogramming	31
2.3.7	Evolving Software for Molecular Modelling	31
2.4	Evaluation of the techniques	31
2.5	Software evolution approaches	35
2.5.1	Lower level: code and modules	35
2.5.2	Higher level: Architecture based	35

2.5.3	Graph based	35
2.5.4	Model-driven (SPLs)	37
2.6	Relational model overview	37
2.6.1	Modelling asset relationships	40
2.6.2	Evolution guided by Isomorphism	41
2.7	Example: Architectures isomorphic to cloud systems	41
2.8	Conclusions	44

2.1 Software Evolution

2.1.1 Introduction

There are many methodologies to develop software [98]. The available tools and the advent of mass collaboration are shaping the way we develop software [99]. There are grand architectural decisions that make sense but for the most part software ought to be built bottom up [100]. A more organic incremental way of developing is actually possible.

There should be a hierarchy of software assets to be modified or deployed at any abstraction level. The engineer's work would be integrated in a collaborative and incremental way [99] [101]. The problem arises when this combinatorial explosion of potential solutions become too complicated to be dealt with just experience or intuition. There must be a subset of solutions belonging to the solution space that can be detected following a particular criterion. Also, equivalent solutions may or may not be desirable depending of which traits they exhibit. The problem may not be linear in complexity and large scale project could become unnecessarily difficult if a wrong combination of software assets is used.

Software actually evolves as an incremental work-bite changes in a closed feedback loop. Actually evolved by human input using several tool-chains (which include IDEs and VCS systems). To gain insights about the nature of software we need to study existing software and its development process. Software needs to be constantly updated due to technological, organisational, unforeseen circumstances and fast changes in the existing environment [102]. Software engineering

is not like other traditional engineering fields [103] and the developing process must reflect this fact. We need to encourage incremental progress. Directed trial and error techniques, prototyping, re-factoring and so forth. Emergent properties will appear to showcase the process as an evolving complex network [104]. Software projects are among the largest complex engineering projects today and therefore will require extensive tooling and automation if we are to keep up. We can wonder what is the philosophical equivalent of software in Nature. There are analogies to be considered and good solutions to imitate from natural systems. For instance, treating them as individuals (which age) as part of species (which evolve) [105]. Here species can be interpreted as a software family of solutions. However, from the microscopic point of view could be seen as (stem cells, evolves, let us say specialises) cell building-type blocks [106]. We would like to distinguish between the process of *evolving software* and the readiness for the software to *evolve*, even autonomously [77]. We will put the emphasis on the semi-autonomous type since the aim is to empower the practitioner to make informed decisions. However, a level of automation can be expected at lower levels of abstraction. The scope of the evolution tends to go even higher than architecture level towards software classification as software product lines or software families.

2.1.2 Evolving Software

Evolutionary software development can be facilitated if the right architectures and development strategies are put in place. The software is developed with little anticipation of future conditions (the development process). However, this software is designed for change and adaptation so uncertainty for future conditions are better tolerated [77]. Software evolution was originally rooted in software maintenance concepts [90] [91] [92] when the waterfall model was nascent.

Revised Lehman's Laws of Software Evolution(1997):

1. Continuing Change*
2. Increasing Complexity*
3. Self-Regulation*
4. Conservation of Organisational Stability (invariant work rate)
5. Conservation of Familiarity (growth rate decrease)

6. Continuing Growth*
7. Declining Quality
8. Feedback System

2.1.2.1 Software that grows *organically*

The fact that software evolves and growth mimicking organic growth is not only rooted in the process but is heavily enhanced depending on the methodology used.

An empirical study [107] demonstrates that open source software complies with laws 1, 2, 3 and 6 of Lehman's Laws of Software Evolution. Open source software grows organically out of the distributed effort of a network of participant who organised themselves ad-hoc. Open source exploded with the use of the Internet. These developments enabled ad-hoc teams of various sizes, with certain work-flows (like Wikis [108] or Git [44]), also yield good results showcasing organic growth.

Software development can be approached in many ways. It is interesting to contrast agile methods against heavy or traditional methods. Heavyweight Methods are rooted in military and aerospace projects with planning spanning several years. Khan and Balbo [3] contrasted heavyweight methodologies against Agile Methods as depicted in 2.1. They are documentation intensive and based on traditional engineering procedures of long detailed planning. ISO-9000 documentation level is considered the quality standard for many industrial software projects. This methodology may not be suited to certain projects. In contrast, Agile methodologies, cater to the fast-paced continuous change and uncertainty that surround many projects.

Keenan [109] formulates the hypothesis that each software project should have a process tailored to its needs and circumstances. The fact is unlike under the old models software is built upon, these agile development paradigms software is literally evolved [110]. The difference of the approach is clear. Requirements are often too volatile and keeping the initial requirements constant along the life of the project is a mistake. Under agile methodologies software is grown from a

	Agile Methods	Heavy Methods
Approach	Adaptive	Predictive
Success Measurement	Business Value	Conformation to plan
Project Size	Small	Large
Management Style	Decentralized	Autocratic
Perspective to Change	Change Adaptability	Change Sustainability
Culture	Leadership-Collaboration	Command-Control
Documentation	Low	Heavy
Emphasis	People-Oriented	Process-Oriented
Cycles	Numerous	Limited
Domain	Unpredictable/Exploratory	Predictable
Team Size	Small/Creative	Large
Upfront Planning	Minimal	Comprehensive
Return on Investment	Early in the project	End of the project

Figure 2.1: Differences between agile and heavyweight approach to software development. By Khan and Balbo [3].

small set of requirements and features to be stopped at some point in the future after several cycles of requirements validation. The fact is there are enough safe stop points so the client can request an early stop due to market conditions or any other circumstances. The life cycle of the development is an incremental and adaptive process as Abrahamsom et al. [111] explains. This process takes into account and accommodates changes, finally adapting to them.

They are many tools, artefacts, code and abstractions that could be used also in an integrated and collaborative way [112] [113]. Moreover, users and programmers, in their respective stage of abstraction, should be involved to the maximum extent possible. The implication is that software can be composed of abstractions and go higher or lower depending on development needs.

2.1.3 Evolution-ready or *evolvable* software

Software could exhibit different levels of readiness to evolve. This software readiness enhancements are also considered of interest within the software evolution moniker [77]. A good line of research would be to tune readiness properties to maximise some strategic feature.

Attributes of *evolvable* software

The dynamics of operating with artefacts should enable the practitioners to design software solutions that are:

- **Productive:** Fast ad-hoc solution with minimal time to market
- **Part of Strategic or planned reuse :** Not reinventing the wheel: Reuse of existing assets
- **Time tested or trustworthy:** Existing possible bugs in assets remain, but we do not create new ones
- **Easy to modify by assemblage:** Ample choice or alternatives and maintenance or adaptation with no programming involved.
- **Modular:** Less coupling than done programmatically
- **Sufficient or satisfactory:** Less customisable than done programmatically from scratch but solves the current problem as defined
- **Antifragile:** Challenges or constraints develop into enhancing features despite hostile and hazardous fast changing environment [114]
- **Evolvable:** Output ready or prone to be further evolved
- **Traceable :** Output ready or prone to be further evolved
- **Organic:** Undirected growth, with self-organising tendencies and by bottom-up development, in a self-reinforcing loop [115]

2.1.4 Conclusions

We can conclude that we can choose to evolve fast with software pieces and lose full customisation, as in this model, or evolve slowly with the fullest customisation (source code) at the price of potential bugs (also may be by starting from scratch). This is the difference between evolving with high level pieces of software or evolving by changing source code instead. This fast evolution can be also reinterpreted as a consequence of producing *evolvable output*. We want the software solutions to be intrinsically *evolvable* or evolution-ready, as a premeditated engineering strategy [77] [116].

Software Evolution is a field with an immense amount of different approaches to be researched. Evolution seems to bring the process of software development and the resulting software properties and their mutual dynamics together. Bio-mimicry or bio-inspired software solutions and processes are common. But the presence of incremental growth in a closed-loop feedback underlying system seems to be key. The question is how to bring this to a feasible model without abstracting important features away from reality.

2.2 Analysis and Visualization tools

2.2.1 Introduction

In this section we will explore selected software evolution related tools [6]. The core techniques are highlighted. Each of them brings a solution to the problems brought forward by software evolution. Different aspects and patterns of the tools can be extrapolated to other tools. Finally, a table with the different features will state the various aspects to consider as analysed.

2.2.2 Holistic Software Evolution: CodeCity

Codecity [4] is a tool to piece together various sources of software evolution. As part of Software Configuration Management (SCM) toolset, source control tools, like Subversion [69] or Git [44] allow for a snapshot of the current state of the software project keeping a multidimensional record(history) all various activities. The REVEAL [117] group at the University of Lugano approaches the research in a holistically trying to unify sources of information and to cover the gaps in the information source. A highlighted tool is CodeCity by Richard Wettel where source code can be modelled as city district (Figure 2.2) using various software metrics as parameters [4] [6]. Classes are represented by buildings, packages as districts and the number of methods is mapped as the height of the buildings. The intensity of blue represent number of lines of code. There is also colour coded design problem detection.

We believe there are parallels to a potential visualisation similar metrics even if is encoded as a graph. This approach can also be applied for source code re-factoring.

2.2.3 An Environment for dedicated Software Analysis tools: Moose

Moose [5] [6] is an open source project started in 1996, part of the FAMOOS European project to study object-oriented systems. Software and data analysis can be customised from the various raw data imported by Moose. Mondrian [118] is a tool for visualisation feeding on Moose analytical data. Includes a scripting engine and allows for several visual representation of software components such as packages, classes, methods and their dependencies (Figure 2.3). Metrics are represented by size and colour. For instance, the darker the colour, the longer

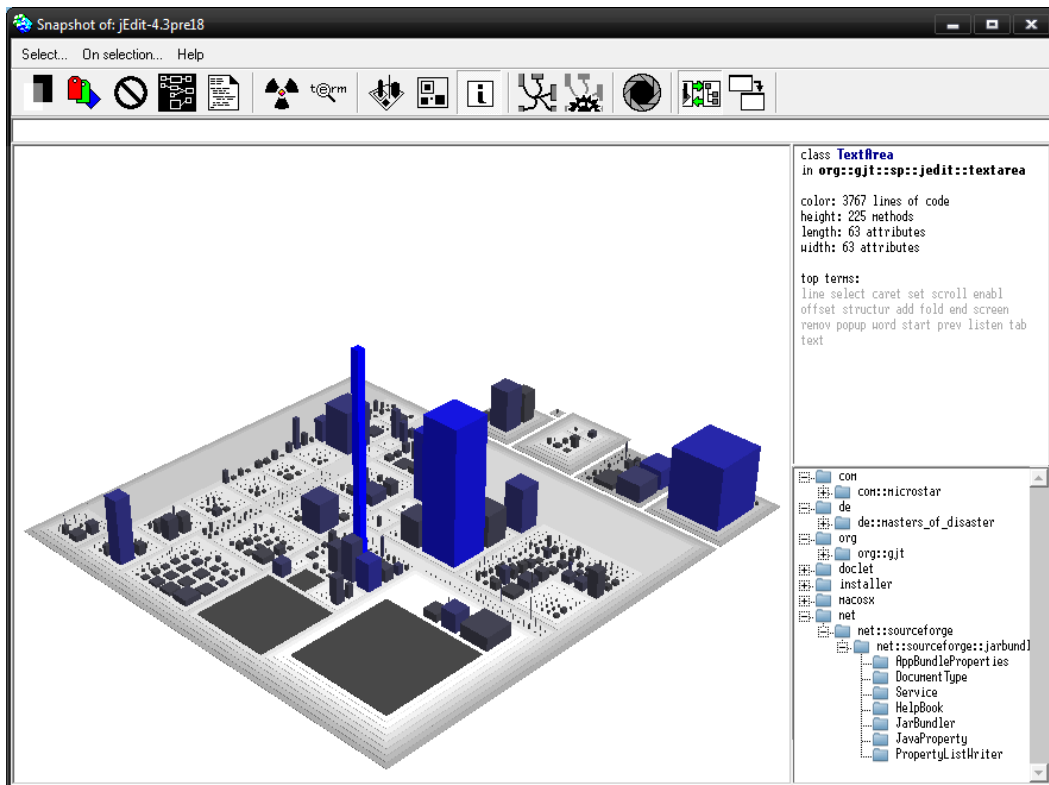


Figure 2.2: ArgoUML source code map, generated using Codecity [4]. This tool is under an academic non-commercial licence.

the method persisted in the source. We could see the chronological evolution of the module (its state) as a 2D treemap where new classes change from yellow in gradient towards blue as the number of changes increase.

2.2.4 Recovering Software Architecture with Softwareaut

An interactive and collaborative architectural manipulation tool [7] to mitigate the degradation of software as it evolves in time. The modules are viewed as treemaps (2.4) with their size proportional to the lines of code. The width of the relationships represents the number of invocations between modules. Practitioners can programmatically filter the whole representation to highlight modules and dependencies based on a set of engineering criteria. Aside from said filters, inspectors provide insights about the evolution of a module.

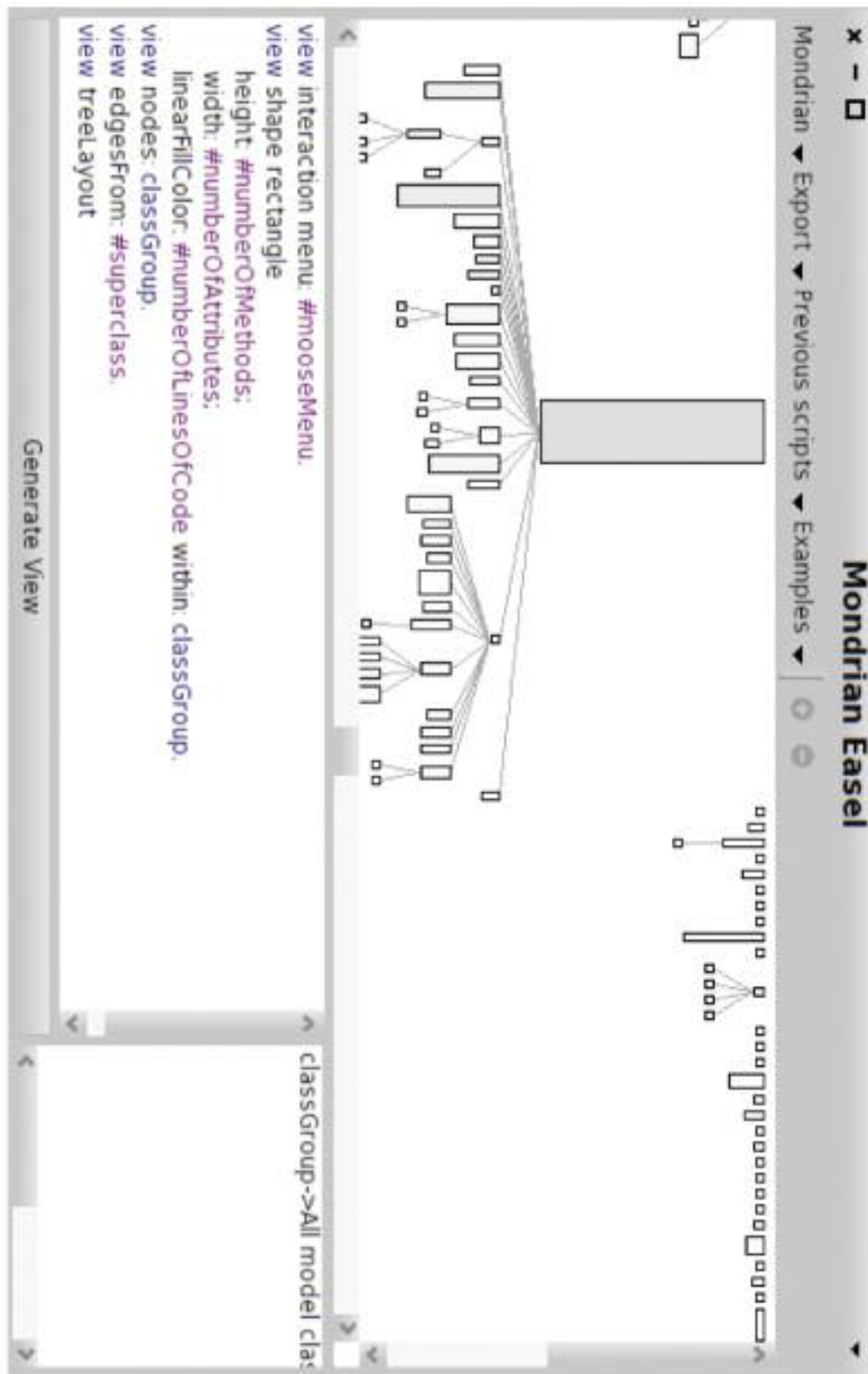


Figure 2.3: Moose data visualisation with Mondrian [5] [6] (CC BY 4.0)

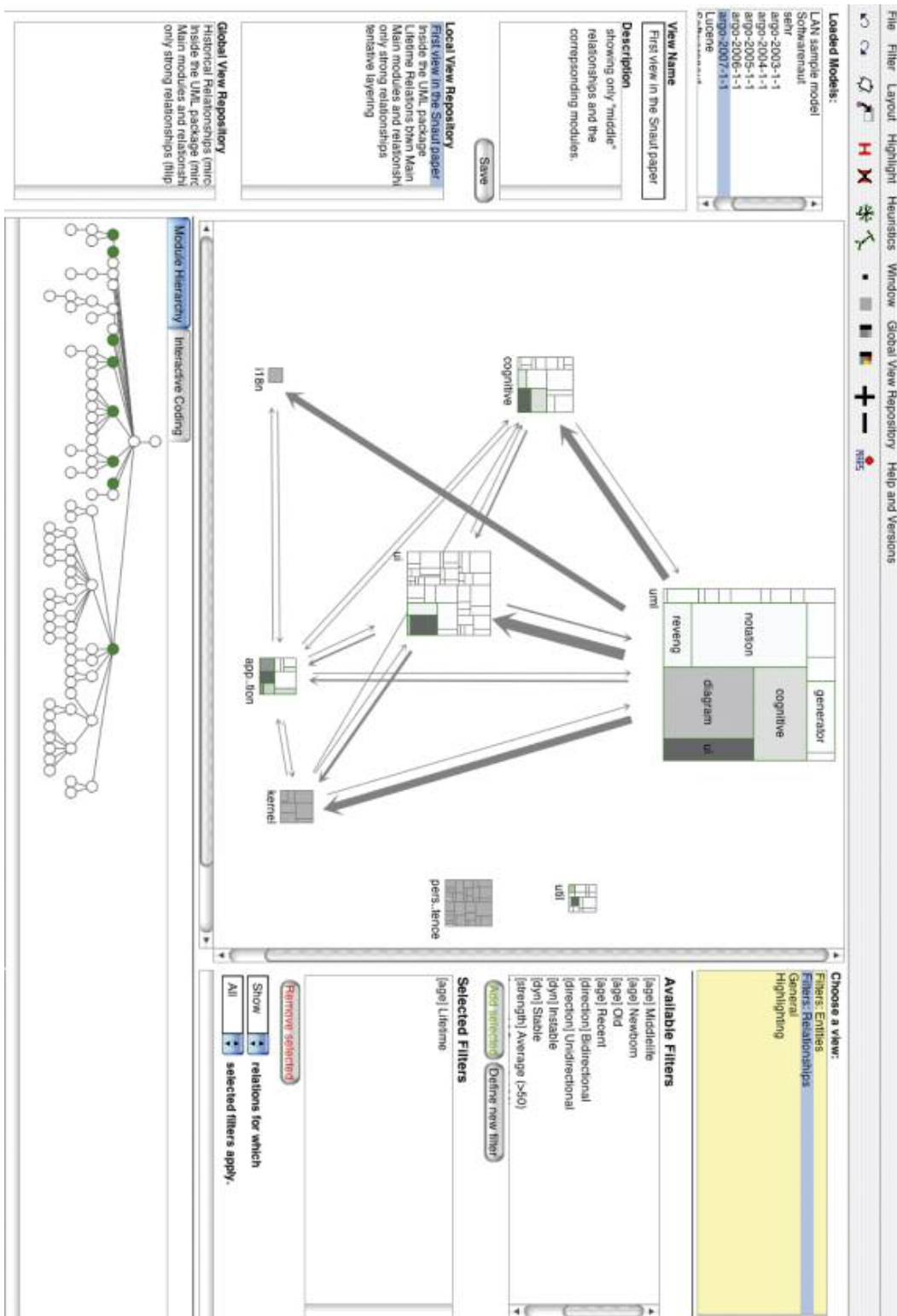


Figure 2.4: SoftwareNaut [7] [6] (CC BY 4.0)

2.2.5 Process Mining Software Repositories

FRASR [8] [6] is a tool for the analysis of data repositories from various data sources (log preprocessing step), depicted in Figure 2.5. The output will be input into ProM (the process mining step). ProM [119] is an open source framework from implementing process mining tools. Reuse can be easily visualised, as depicted in Figure 2.5. The first triangle represents the initial prototype being leveraged as the step for the next iteration (big triangle), showcasing reuse. The researchers believe this mining allows for a good degree of prediction based analysis.

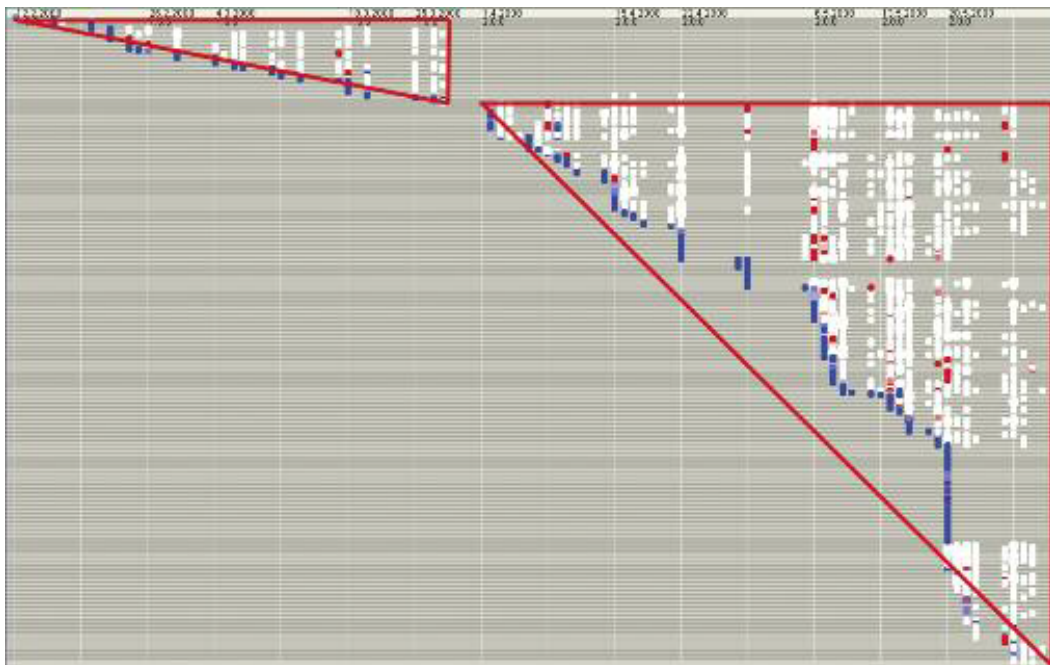


Figure 2.5: FASR+ProM [8] [6] (CC BY 4.0)

2.3 Software tools for Architecture and Design

2.3.1 Automated Synthesis of CONNECTors to support Software Evolution

Automated synthesis of CONNECTors enabling continuous composition. The CONNECTors are software pieces that emerge from the understanding of dis-

parate software artefacts like protocols, actions, data models. They effectively interact automatically once generated. The researchers developed a theory of CONNECTors to synthesise an application layer for automatic interoperation [6].

2.3.2 Emergent Middleware: Starlink

Starlink [9] [6] is an open source project that has been successful at dynamically generating middleware for CORBA to XML-RPC protocols to understand each other and also XML-RPC interoperating with the Picasa REST API. This tool uses the CONNECTors previously described [120]. Emergent middleware chart shown in Figure 2.6.

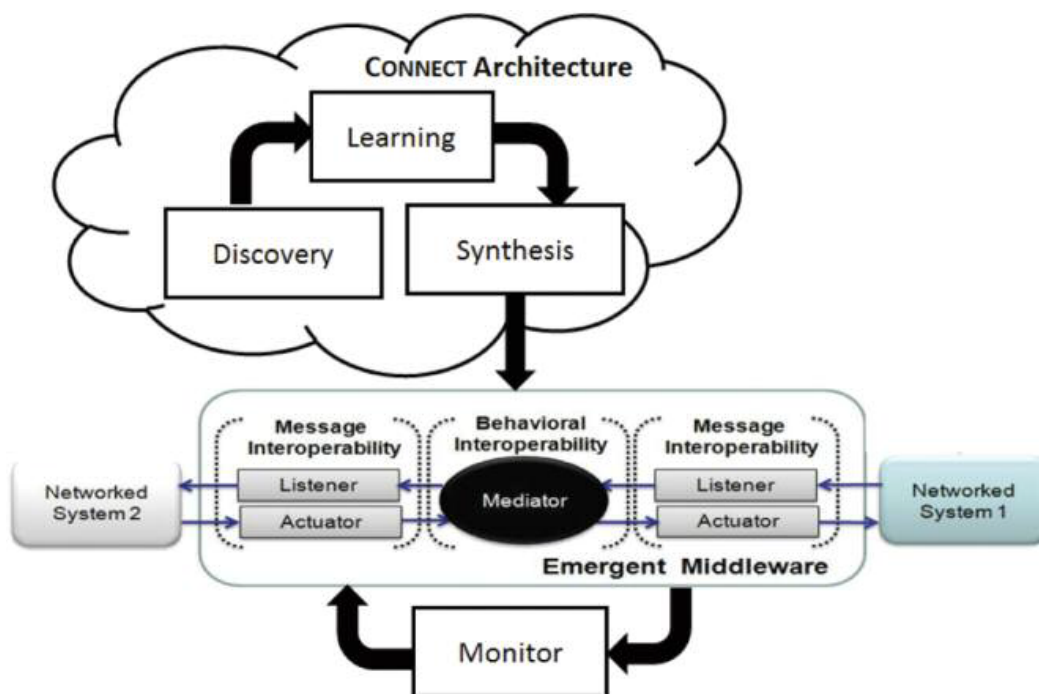


Figure 2.6: Emergent Middleware model: Starlink [9] [6] (CC BY 4.0)

2.3.3 Pat-Evol: Pattern-driven Reuse in Architecture based Evolution for Service Software

Off-the-shelf architecture evolution based on a constructive architecture-based evolution process for service software (SOA). Automated identification of evolution patterns from the architecture change log. A pattern library acts as a reposi-

tory to enable, using operations, pattern-driven change execution. The latest one allows for reusable change execution. The metamodel uses a typed attributed graph in which said graph encodes the configuration for the components and their connectors. The modelling encodes the evolution operations. These are based on graph transformation rules. This model enables the structural evolution of architectures. The AI algorithms use the pattern library database to generate a starting graph (in a markup language) that will be transformed using a XSLT (XML template transformation technology) to a target graph [6].

2.3.4 CAPucine: Context-Aware Service-Oriented Product Line for Mobile Apps

This project creates Dynamic Software Product Line (SPL) adapted to mobile heterogeneous systems. This is achieved using previously developed and tested assets. Asset in this context means any software artefact used to develop an application. In the SPL model, reuse of software assets among software families is straightforward since the similarity and difference levels among the applications were identified. Therefore, various configurations are possible. The interaction between a variability model and an aspect model to be adapted even at runtime (using a technique called Runtime weaving which allows for reconfiguration on execution) [6].

2.3.5 MoDisco Framework

MoDisco is a Model-Driven reverse Engineering framework to update legacy systems (actually tested on such systems) [59]. It uses Model Driven Engineering (MDE) [58] technologies provided in Eclipse and the Eclipse Modelling Framework (EMF) [22]. It has been open sourced and also modularised. J2SE [22], J2EE [22] and XML [35] technologies are supported and has a plethora of model discoveries, transformations and code generators among other artefacts. The model is considered by the OMG Architecture Driven Modernisation (ADM) Task Force as the reference provider for real implementations of several of its standards [22]. This could have implications for other enterprise class tools [6].

2.3.6 Rascal metaprogramming

Rascal [121] [6] is a Domain-Specific Language (DSL) for analysis, transformation and visualisation of existing source code. It is written in Java and integrates with the Eclipse IDE. This tool can be used in source to source transformations to see where competing software patterns make more sense. The researchers expect this tool to be of benefit of tree-centric object-oriented type of software. Software metrics, static analysis, code transformations and code generations and further DSL have been applied using Rascal.

2.3.7 Evolving Software for Molecular Modelling

Group of tools (programs and scripts) that simulate various levels of physical (quantum, molecular) phenomena or interactions. Each level of abstraction corresponds with a tool. Each tool communicates to the next higher level (tool). This keeps the simulations at the most appropriate level of abstraction. The tools are named *WOLF₂PACK*, *GROW* and *ESPreso++*, ordered by abstraction level. These tools were developed by the Fraunhofer Institute for Algorithms and Scientific Computing (SCAI), Germany [6].

2.4 Evaluation of the techniques

There are some common techniques that appear recurrently on the examples. Repository mining is used quite widely. Not only analyses source code (and its chronological changes) but, in some cases, also analyses other items, like source control commits (from the SCM tool used) annotations. There are some time-tested libraries to do this mining.

When code is analysed, modules and/or packages and classes are mapped (along with their methods, attributes and properties). Generally this is performed by using any type introspection provided by the programming language used. Once we read all the structure with its classes, objects and object behaviour we can build an image of the system at runtime. This could be implemented ad-hoc if no better alternative is available. Below there is a table that summarises various aspects of the tools discussed and their strengths (and also applicability to future evolution software tools).

Tool	Type	Techniques, libraries and frameworks	Advantages (↑) and Disadvantages (↓)
Codecity	SCM mining [64]	Smalltalk [122], Moose [5]	<ul style="list-style-type: none"> • ↑ Visual feedback on patterns • ↑ Analyses Many languages • ↓ Framework monoculture (Moose)
Moose [5]	SCM mining	Smalltalk [122]	Time-tested mining framework used in many projects.
Softwareaut	SCM mining	Smalltalk [122], Moose [5], ArgoUML [123]	<ul style="list-style-type: none"> • ↑ Time-tested open source framework. • ↑ It analyses Many languages • ↓ Framework monoculture (Moose)
FRASR+ProM	SCM mining	XML [35] based, FRASR [8], ProM [119]	<ul style="list-style-type: none"> • ↑ Log preprocessing and process mining separated • ↑ Novel approach to repository mining • ↓ Less examples of use

Tool	Type	Techniques, Libraries and frameworks	Advantages (↑) and Disadvantages (↓)
Starlink	Protocols interoperation	Java [124], XML [35], custom DSLs [22], automata [16], CONNECTors (AI) [6]	<ul style="list-style-type: none"> • ↑ Autonomous • ↑ Based on computer science principles: Uses automata, AI • ↑ It is been tested. • ↓ Unforeseen situation could come up at any time
Pat-Evol	Pattern mining	SOA [22], XML (XSLT) [35], graphs (math)	<ul style="list-style-type: none"> • ↑ It uses standard techs like SOA and XSLT • ↑ Based on computer science principles: Uses automata, AI • ↑ It generates evolutions operations. Evolution traceable • ↓ It depends on the quality of the patterns found • ↓ It depends on the quality of the patterns repository
CAPucine Framework	SPL engineering [67]	Model-driven SPL [67] [6], MDE [58] [24] [22], code generation	<ul style="list-style-type: none"> • ↑ It is just a framework for models • ↑ Code generation possible • ↓ It may be hard to go from theory to practice

Tool	Type	Techniques, Libraries and frameworks	Advantages (↑) and Disadvantages (↓)
MoDisco	Model-Driven Reverse Engineering (MDRE) [59] [6]	Java [124], XML [35], Eclipse (IDE) centric [125]	<ul style="list-style-type: none"> • ↑ Eclipse (IDE) centric • ↑ It uses standard techs: Java, XML • ↓ Eclipse only
Evolving Molecular modeling	Simulation engine	C++ [126], Python [127], POSIX [62]	<ul style="list-style-type: none"> • ↑ It uses popular languages • ↑ It uses popular standards like POSIX • ↑ Parallel programming • ↑ Techniques could be applied to other domains • ↓ Custom made tool, not general purpose • ↓ Simulation oriented (good for simulation though)
Rascal	DSL based [22]	Java [124], Eclipse (IDE) Meta-Tooling Platform [125]	<ul style="list-style-type: none"> • ↑ Popular techs: Java , Eclipse • ↑ metaprogramming is flexible • ↓ The actual metaprogramming could be challenging and depends on the skill of the practitioners: no guarantees

2.5 Software evolution approaches

2.5.1 Lower level: code and modules

Bhattacharya et al. [128] proposed building graphs with the relationships of the changes in the source code or modules. The developer collaboration dynamics (VCS commits), bugs, static function calls (finer source level relationships) and modules communication (coarser source level relationships) are also encoded as graphs. All these events capture the software evolution process and can be thus mined and tracked.

2.5.2 Higher level: Architecture based

Benett et al. [129] proposed a serviced based architectural model for software evolution. It identified the software lifecycle as evolution milestones. Importantly, it states that the speeds at which the various software components evolve may differ. Cook et al. [116] addressed our concern of the software as a software process and its properties as being *evolvable* (evolvability). This can be pre-designed or pre-meditated. It also points to the advantage of dealing **first** with the architecture (higher level of abstraction) evolvability properties using ADLs. These ADLs are supposed to supplant any lesser formal framework. Architectures have traceable and connected evolution paths. Also, there are architectural styles, even across domains, of evolution depending on the relationship between architectures, their paths and their evolution stages based on Barnes et al. research [130]. Furthermore, in the higher level category, we could include SPLs configuration evolution as the higher order components to evolve with [80] [23].

2.5.3 Graph based

2.5.3.1 Pattern based

From an evolution approach point of view, Pat-evol [6] is an example of graph based evolution with item relationships implemented as GML graphs. It uses XSLT graph transformations to replicate CRUD primitive changes. The repository of changes patterns is mined for reusable configurations. It also establishes the relationship between the GML graphs as FSM sources for pattern identification purposes. These patterns get stored in a library or repository. The outcome is that we have a match between the evolution changes and their correspond-

ing graphs. This is stored as a XSLT transformation. Based on pre and post-conditions transformations can be pulled from the repository based on existing pattern changes.

2.5.3.2 Evolution paths (traceability)

Maletic et al. [131] proposed a graph based encoding for mode-to-model mapping. The models are graph based. There are traceability links for this inter-model relationships. In the case of this research, XML is suggested to encode the data regardless of the origin. Other XML related technologies like XSLT can be used to encode transformations. To encode links (relationships) and their associated metadata XPath [35] is suggested. All these techniques are the basis of the traceability between inter-model graphs.

2.5.3.3 Evolution tree or lifecycle based model

Schach and Tomer researched the evolution of software where lifecycle phases are encoded as trees [2]. This tree follows the development trail or axis. The maintenance changes are in the maintenance axis and feature tree mappings tracking the changes (evolution). This gets upgraded to a graph when we take into account feedback loops on the whole process. For instance, a requirement update will certainly create this loop. The result is a propagation graph encoding both the lifecycle and all subsequent changes. We can see any maintenance as of software evolution. There is the case of using the SPL artefacts as inputs to the model as they later researched [1].

2.5.3.4 Hypergraph based

Harn et al. [85] [86] proposed a model based on hypergraphs. Here, waterfall model milestones or software development lifecycle milestones are encoded in the hyperedges. The outputs or product of the engineering process (including those referred also as documentation artefacts) are the vertices or nodes. With this overarching concept of component, practically any object is a possible node. These objects include:

Criticisms, issues, requirements, specifications, modules, programs, and optimizations.

Hyperedges are created with action-steps or events:

i.e.: Software prototype demo step, issue analysis step, requirement analysis step, specification design step, module implementation step, program integration step, software product demo step, and software product implementation step. The flexibility of this relational hypergraph model adapts to the multidimensional nature of the different software evolution aspects being encoded.

2.5.4 Model-driven (SPLs)

A model-driven example is the model-driven software production line multi-model [10]. The transition from domain engineering (DE) to application engineering (AE) [24] [23] [6] can be achieved with a set of operations. It may be plausible for the engineers to establish what kind of outcome is needed and then let the model discard unwanted solutions from the solution space thereby finding a valid solution. In practice these 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 [67]. 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 f between *Domain Engineering* (DE) and *Application Engineering* (AE) Cartesian product of their asset set with themselves.

$$\forall d \in DE \exists a \in AE \text{ such that } f : G(DE) \rightarrow H(AE), f(d) = a$$

See figure 2.7 which illustrates the view shift.

We can evolve using isomorphisms from any Domain engineering d 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 complying with this view shift.

2.6 Relational model overview

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, un-

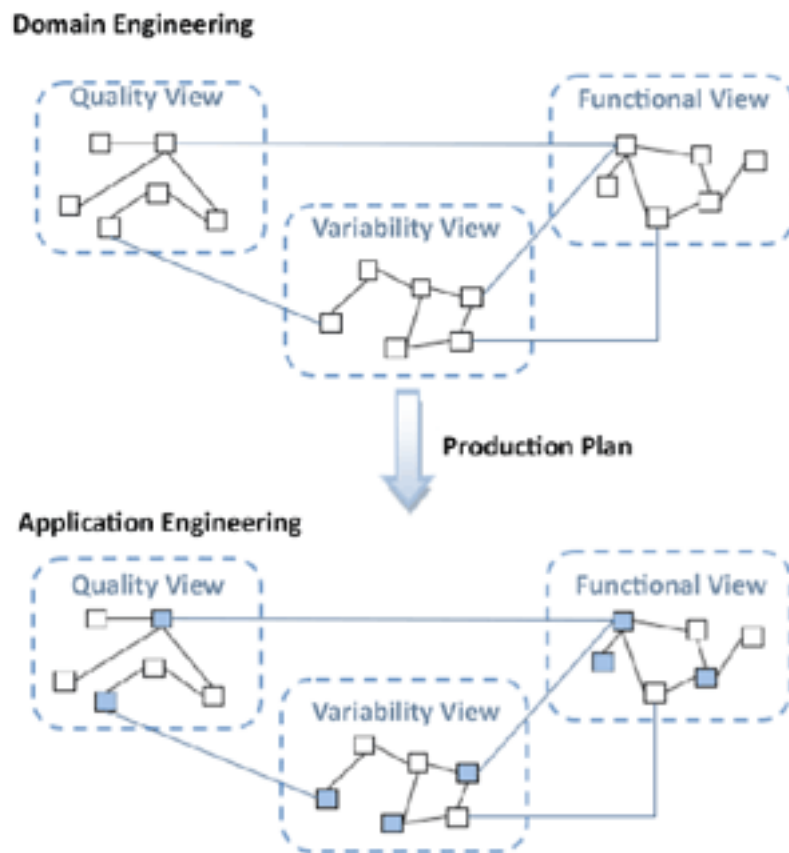


Figure 2.7: Transition from domain engineering to application engineering. "The Multimodel in the Software Product Line development process". ISSI Research Group (Polytechnic University of Valencia). (CC BY 4.0) [10] [6]

foreseen circumstances and fast changes in the existing environment [102]. 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 [132].

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., [6]. 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) combinato-

rial 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 [132].

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 [84]. Other works such as [85] apply program dependency graph techniques to study the transmission of attributes in each evolutionary step, in addition to the use hypergraphs as the basis for the formalisation [86].

In our work we are aiming to build a general framework to capture evolution in emerging trends such as software factory paradigm [79] where the artefact under consideration is not just the source code or a single software instance but a collection of assets and a family of software products that can be considered systematically. We are also interested in the 'software that evolves organically' paradigm. It describes the kind of evolution dynamics 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 arise 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. As previously said, our graph model preserves solution attributes by leveraging graph isomorphisms. Therefore, the task of the engineer is to find a seed solution that will satisfy the initial requirements and then use *predictors* that will use this model to suggest or point to a family of valid alternative solutions from which the designer can make a choice. This will be based on some qualitative judgements. We are empowering the practitioners to find novel solutions by automating this task and this is not meant to the detriment

of the individual's creativity.

The next sections are organised as follows. We provide software evolution scenarios and identify the graph formalism that are feasible in these contexts. We also present our graph based formalism for modelling evolution. Finally, we highlight some application scenarios.

2.6.1 Modelling asset relationships

As described above the different parts corresponding to a software system or product are its assets [79]. Some software assets have a relation with other software assets and we are required to maintain these relationship 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 software. Engineers choose attributes and explore the domain space 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 \mathcal{A} . Let the relation S be a particular subset of the Cartesian product of the assets sets:

$$S \subset \mathcal{A} \times \mathcal{A}$$

$\mathcal{A} \times \mathcal{A}$ defines the set of all potential asset combinations, valid or not. If the set \mathcal{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 $S \subset C$ consists 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 provides us with computational notions of *Isomorphism* that are necessary in designing a software evolution framework.

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

2.6.2 Evolution guided by Isomorphism

Isomorphism and sub-graph Isomorphism gives two operations formalising 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 an evolved graph. We detail the product and check if it is isomorphic (at the right abstraction level) with a valid product. Likewise, we abstract further and check if the result is isomorphic against our current product graph. This isomorphism preserving operation guarantees that the resulting product will preserve the desired properties. However, primitive operations allow for the product to be evolved without the isomorphism checks. Figure 2.8 shows an illustration of the searching operation of the graph 2.8a over the solution space graph 2.8b resulting in a sub-isomorphic relationship or binding.

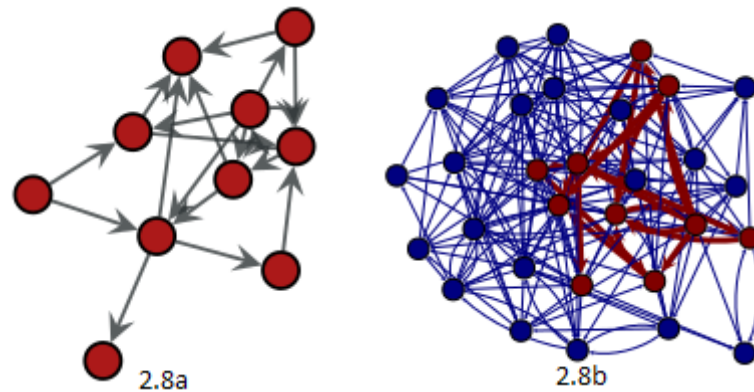


Figure 2.8: GPLv3 Graph-Tool analysis framework [11] using subgraph isomorphism detection [12]

2.7 Example: Architectures isomorphic to cloud systems

Cloud software [87] is clearly defining the functions of the server and the client in two stacks communicating by HTTP [35]. This whole development mimics the Model-View-Controller (MVC) architecture [60]. 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 con-

centrating 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 virtualised and thus treated as yet another layer in the stack. This allow for the server stack to provide a variety of different server-side development frameworks. The characteristics of cloud solutions are described in the taxonomy by Rimal et al. [87].

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 using \mathcal{A} and \mathcal{B} as the assets sets.

The asset set in a desktop context (offline):

$$\mathcal{A} = \{server, database, AJAX, UI\}$$

Being UI a collapsed view of desktop UX/GUI technologies.

and the asset relation is $\mathcal{A} \times \mathcal{A}$, whose equivalent di-graph representation can be denoted by H

The practitioner has a starting asset set of nodes corresponding previous desktop context:

$$\mathcal{B} = \{database, server, AJAX, browserbasedUI\}$$

and a corresponding asset relation $\mathcal{B} \times \mathcal{B}$, whose equivalent di-graph is denoted by G . We consider G to be the *seed* solution to evolve from to H . We can formulate this as an isomorphism between graphs G and H .

We can study the problem of a desktop product to a cloud product:

$$f : G \rightarrow H$$

Let f be the isomorphism such as $f : G(A) \rightarrow H(B)$ and $A = \mathcal{A} \times \mathcal{A}$, $B = \mathcal{B} \times \mathcal{B}$

using the isomorphic graphs in Figure 2.9. Graph G 2.9a and graph H 2.9b.

$$f(G_a) = H_1, f(G_b) = H_6, \dots, f(G_j) = H_7 \text{ matching pairs.}$$

We can delete G_c and replace it with the new property preserving (isomorphism preserving) H_8 . In a desktop to cloud context that could make the Cartesian pair $(server, UI)$ be replaced by the pair $(server, browserbased)$. Likewise, regulatory compliance could make us shift documentation to document

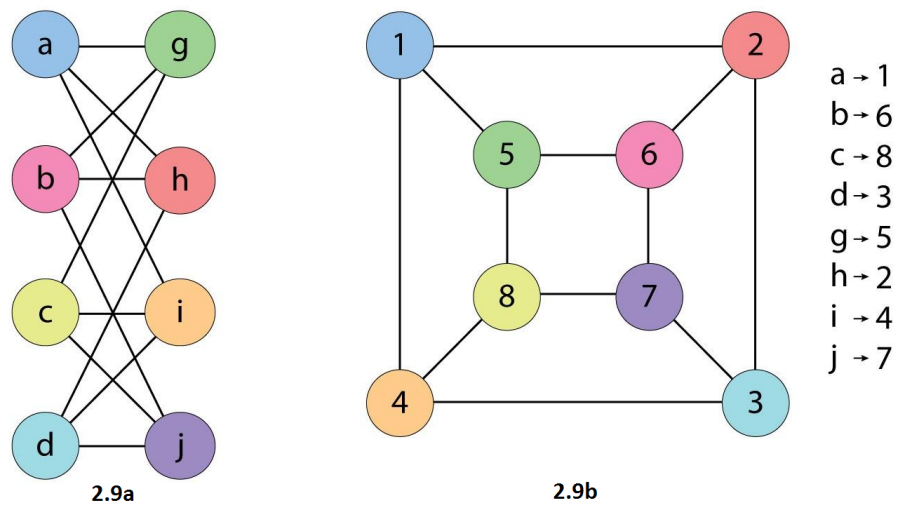


Figure 2.9: Graph isomorphism example. (CC BY-SA 3.0) by AAAS (adaptation). Wikimedia. [13]

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 valid pairs using a digraph. Therefore, a digraph could be the input we need for validity. This enables us to discard undesired combinations and it does not constrain the creative options of the practitioners. For simple examples it could be trivial although it might point to some overlooked property. If the product is really complex and thus the graph, there could be a significant amount of properties to track and therefore such graph encoding 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 systems are 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 this higher level isomorphism check. This taxonomy will enable us to identify families of products. We can conjecture how this could be classed as a sort of *architecture-time* programming.

2.8 Conclusions

We have studied the aspects of software evolution. These show it is a process with start and end states with ongoing outcome-states or stages. This process is bound by Lehman's Laws of Software Evolution. Moreover, it is a software development process with a lifecycle. All these concerns should be considered to understand the evolution phenomenon in a software engineering context.

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 can represent the design or architecting of equivalent solutions in a software evolution context. The architectural changes can be introduced and recorded in a systematic way. This is already happening at source code level by collaboration of engineers and/or in open source on a massive scale. We further this trend providing a model to evolve a system. 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.

Future work should 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 a Soft-Computing domain, such as genetic algorithms, to devise operations beyond plain operations on the graphs. Other good candidates for a meta-heuristic would be (to train) a neural network (operation) to recognise specific patterns in the whole model. This could work at 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. The isomorphism check could be done, for instance, using some library implementation like the *VF2 algorithm for NetworkX (Python)* [72]. Another possibility is to consider polynomial

time algorithms for graph operations that are feasible under certain conditions [133]

Regarding the tools sampled, to create a software evolution tool, we may need software configuration management tools (SCM) mining. Either with ad-hoc methods or using a framework like Moose to do the heavy lifting. Using Moose is safer but is a less novel approach, as many tools use it. The source code can certainly be statically analysed using industry standard libraries. The Starlink /CONNECTors [6] approach is really interesting because it is bottom up and autonomous. However, it may be hard to predict its behaviour. The approach has been proven to work and the protocols do communicate autonomously. In the same category is Pat-Evol [6] as “the system comes about” with the solution automatically. Pattern matching is a well studied subject in Computer Science. It is a remarkable fact that Pat-Evol uses operations to replicate evolution and transform graphs of different software patterns into each other. Software Product Lines (SPLs) are widely used to isolate commonalities and allow for customisation of families of software. These families evolve in time and a lot of research follows this direction. Existing tools based on Eclipse also surfaced to do reverse engineering and metaprogramming among other various aspects related to software evolution. Code generation is also a recurrent feature in many systems as well as the use of widespread open source technologies.

A Relational Hypergraph based Model

Contents

3.1	Introduction	47
3.2	Preliminaries	47
3.2.1	Relations	48
3.2.2	Graph of Asset relation	49
3.2.3	Graph Isomorphism as a structure preserving operation	49
3.2.4	Sub-graph Isomorphism to identify related structures	50
3.2.5	Existence of Primitive operations	50
3.3	Relational Hypergraph Model	50
3.3.1	Evolution using Hypergraphs	53
3.3.2	Single <i>Asset</i> evolution	53
3.3.3	Hierarchical evolution of multiple <i>Assets</i>	55
3.3.4	Existence of Product lines	55
3.4	Encoding families using a Relational Hypergraph Model	58
3.4.1	Software Product lines: Single asset	58
3.4.2	Coupler enablers or the assets where to grow from	64
3.5	Multiple asset based product lines : Case studies	64
3.5.1	Case Study I: Evolution of Browser Technology	64
3.5.2	Case Study II: Cloud compute engine	67
3.5.3	Case Study III: Regulatory or Legal Constraints	69
3.6	Conclusion	70

3.1 Introduction

In this chapter we introduce *meta graph* models to formalise software evolution for this emerging paradigm of higher order reuse. We make the following contributions:

- We introduce a relational hypergraph model to capture evolution in a product line context.
- We formalise evolutionary paths in this hypergraph model by expansion of the graph through *isomorphic* and *coupling* transformations. Paths in this hypergraph enable traceability in software evolution and enables a practitioner to systematically explore a solution space.
- We identify equivalence classes that relates the structural properties of hypergraphs to product families.
- We provide a quantitative measure for existence of product lines.

We use a *hypergraph* as a *meta*-representation to traverse the solution space where edges are labelled by isomorphic mappings. Our relational hypergraph model is therefore different from those presented in other works [85] [86] since the paths in our graph model are constructed from *isomorphic* operations and our model relates to product lines.

The chapter is organised as follows. In section 3.2 we recall some preliminary properties on relations and graphs. In section 3.3 we present our graph based formalism for modelling evolution. In section 3.5 we evaluate the novel modelling capability for emerging application scenarios.

3.2 Preliminaries

Given a collection of assets, software product lines can be created. There is a hierarchy of abstractions where each level is defined by interlinked configurations for layers of artefacts belonging to the development cycle. Within each of the viewpoints a software factory template associates reusable artefacts that collectively define the attributes that lead to differentiable product lines. Relationship within a viewpoint and across viewpoints define semantic links and finally a factory schema for architecting products from a common set of assets. A similar strategy

is to be followed by software product lines emerging from the organic growth of an extensible set of assets as we illustrate in the case study section.

The different parts corresponding to a software system or product are its assets. 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 system: we explore this in the form of single asset and multiple asset evolutions in the next section. Practitioners choose attributes and explore the domain space using isomorphisms and set operations. The attribute choices should encode these facts or aid in the search for the missing ones. Isomorphism and sub-graph isomorphism are two operations with which we can formalise a predictor framework for evolution. Isomorphism gives the ability to architect equivalent products, establish consistency across abstractions or provide a basis for evolving organically. Architectural patterns of varied complexity will emerge eventually from the process to be strategically assessed as noted in [134]. Sub-graph isomorphism can be used to capture differences in product lines where some parts of the new design is isomorphic to an evolved graph.

3.2.1 Relations

Let the set of assets be \mathcal{A} . The relation $\mathcal{A} \times \mathcal{A}$ defines the set of all potential asset combinations, valid or not. If the set \mathcal{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. We will need the following two types of definitions in our formalisation.

Definition 6 - Partial Order:

A relation R on a set \mathcal{A} is called an *Partial Order* if it is reflexive, anti-symmetric and transitive [135].

An element a is the *least upper bound* of a partially ordered set (*poset*) (*l.u.b.* of $(\wp(\mathcal{A}), \leq)$) if and only if:

- a is an upper bound of $(\wp(\mathcal{A}), \leq)$
- for every upper bound b of $(\wp(\mathcal{A}), \leq)$, $a \leq b$

Definition 7 - Equivalence Relation:

A relation R on a set \mathcal{A} is called an *equivalence relation* if it is reflexive, symmetric and transitive.

Two elements a and b that are related by an equivalence relation are called *equivalent* and are denoted $a \simeq b$. An equivalence relation decomposes the set of elements A into *equivalence classes*. The equivalence class of a with respect to \simeq is denoted $[a]_{\simeq}$ [136].

3.2.2 Graph of Asset relation

Our goal is to automate the search of the evolution solution space around the seed solution set \mathcal{S} and so we need a computational model of our product family \mathcal{S} . Graphs provide the required mathematical model since there is a one-to-one correspondence between relations and graphs. Graphs also provides us with computational notions of *isomorphism* that are necessary in designing a software evolution framework. 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.

3.2.3 Graph Isomorphism as a structure preserving operation

The subset S contains all the possible initial 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 8 - 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:

$$\forall g \in G \exists h \in H \text{ such that } t : g \rightarrow h$$

3.2.4 Sub-graph Isomorphism to identify related structures

Definition 9 - Sub-graph Isomorphism:

Let G, H be two graphs and $S \subset H$ be a subgraph. A mapping $g : G \rightarrow S$ is a *sub-graph isomorphism*, where g is injective, with the property that if a, b are adjacent in G then $f(a), f(b)$ are adjacent in H .

Sub-graph isomorphism can be used to evolve a given seed solution G into a family of solution (super) set H such that the evolved design satisfies the original requirements and any additional constraints that are put in defining a new product line.

3.2.5 Existence of Primitive operations

We 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 .

$$\text{expand} : V \rightarrow V' = V \cup V_{\text{subproblem}} \quad (3.1)$$

$$\text{collapse} : V' \rightarrow V \quad (3.2)$$

3.3 Relational Hypergraph Model

In order to capture evolutionary aspects of product lines we need a representation to relate a solution with its equivalent solution set space. Since our seed solution itself is a graph of the asset relation on the assets \mathcal{A} we need a more flexible generalisation of a graph but still an incidence structure — a *hypergraph* where each edge is a set of vertices and can be related to other vertices. (many to many relationships) [137] [14].

Definition 10 - Hypergraph:

A *hypergraph* is a directed graph denoted as $H = (V, E)$ where

- V is the set of vertices or nodes.
- E is the set of edges or hyperedges, a set of subsets of V .

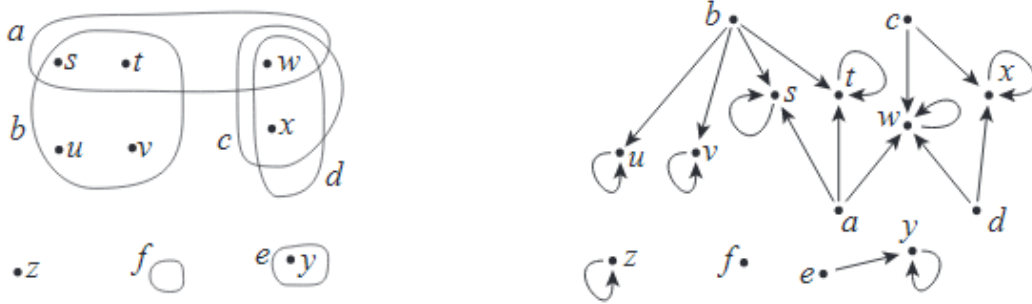


Figure 3.1: Different Hypergraph views as described by Stell [14]. Left side: a Hypergraph $H = (V, E)$ where the edge set is $E = a, b, c, d, e, f$ and the vertex set is $V = s, t, u, v, w, x, y, z$. The right: the corresponding relation φ on $V \cup E$.

A hypergraph can also have many views or interpretations as it is going to be illustrated. In Figure 3.1, the left side represent hyperedges as sets. On the right it can be viewed as a relation φ over the union of both sets.

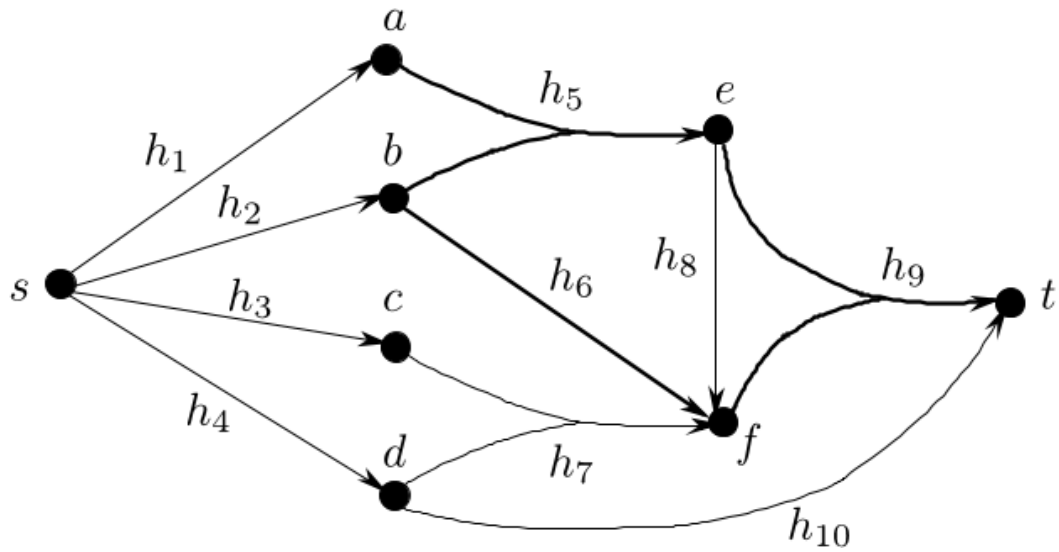


Figure 3.2: A directed hypergraph example view [15].

Another view shown in Figure 3.2 depicts a directed graph. We should notice the nodes drawn origin and destination.

The model requires the edges to represent an isomorphism and thus be mapped to an isomorphic operation \simeq . Let $L : E \rightarrow \simeq$ be a set of labels or identi-

fiers that maps edges to an isomorphic operation \simeq .

Each vertex in this hypergraph corresponds to an instance of a solution from S constructed from the asset relationship graph G . In evolving a seed solution, additional vertices are introduced into this hypergraph and these vertices also represent a (sub) graph from another view. There are two kinds of new vertex sets:

- Those corresponding to equivalent seed solution that are derived from isomorphic operations.
- Those corresponding to a new set of asset relationship, which we term as *coupling* graphs to evolve a seed graph into a new solution.

An edge of a hypergraph represents one of the two kinds of relationships. An edge labelled with the isomorphic operation \simeq correspond to equivalent solutions from the seed solution G . Edges without any labels link G with coupling graphs \mathcal{K} .

Figure 3.3 shows an example of a hypergraph with the above characteristics to model an evolution step. The yellow graph G represent the seed solution and the added coupler green graph belong to the solution family $[G]_{\simeq}$ by the expansion operator \simeq . All resulting graphs that are isomorphic from seed to expanded graph constitute the equivalence class $[G]_{\simeq}$. If f and h are isomorphic operations then they belong to the same equivalence class. g could be a subsequent expansion under a new isomorphism. The diagram depicts how commonalities and variabilities can be systematically introducing within the framework of isomorphism.

A hypergraph with the defined structure above models the basic components in an evolution framework through vertices and edges. The paths in this graph represent evolutionary history and the evolutionary paths of a particular product line are governed by some basic feasibility factors that we established in previous sections. In general, it is possible to automate the construction of evolutionary paths by transformation of hypergraphs. The decision to follow a particular path and shape the evolutionary outcome is determined by the strategy that a practitioner applies based on some objective criteria that is pertinent to a product line.

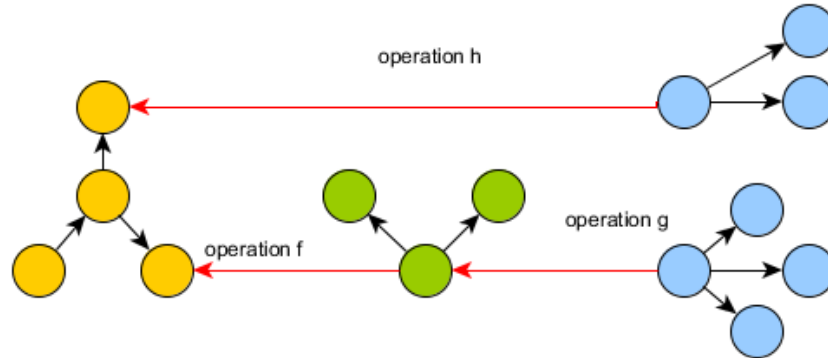


Figure 3.3: Growing current graph produced through couplers.

3.3.1 Evolution using Hypergraphs

Given a basic structure of a model for evolution in terms of hypergraphs we can now construct two main methods for an evolution framework. The first applies to the case of a single asset situation and shows how product line evolution can be cast as a hypergraph expansion. This can then be extended to the more general case of multiple assets. The first can be viewed as a local expansion with transformations of only a few nodes introducing variability. The second can be viewed as a global expansion from a coupling of local expansions that enables a systematic exploration of a wider evolution space.

3.3.2 Single Asset evolution

We traverse a seed hypergraph G using isomorphic operations \simeq to explore the solution space. Such transformations are useful to evolve a product with only a few attribute changes, since the intention here is to preserve much of the core assets shared by a product line and enable the variations to be introduced. This need stems from the customisation for a specific requirement or to enhance a product to suit technological advances.

The exploration of this solution space is not arbitrary but will be governed by the following structural property. Let $\hat{G} = \{G_1, G_2, \dots, G_n\}$ denote a set of graphs that are related by isomorphism \simeq . The relational aspect of this hypergraph model comes from the fact that the relation \simeq on $\hat{G} \times \hat{G}$ is an equivalence relation as follows:

- every G_i is isomorphic to itself $G_i \simeq G_i$ under an identity function hence it

is reflexive.

- if $G_i \simeq G_j$ then $G_j \simeq G_i$ and hence is symmetric.
- if $G_i \simeq G_j$ and $G_j \simeq G_k$ then $G_i \simeq G_k$ hence transitive.

From this structural property we can deduce that a set of equivalence classes for a graph G , ($[G]_{\simeq_1} = S_1, [G]_{\simeq_2} = S_2 \dots [G]_{\simeq_n} = S_n$) represent different product evolution sub-spaces or, in other words, product *families*.

Figure 3.3 shows an example of evolution of an asset graph through single couplers. A sub-graph in the resulting expanded graph should be isomorphic to the seed graph G .

Given a $[G]_{\simeq}$, the equivalence class of isomorphic graphs, we now consider expansion of the graph using couplers \mathcal{K} . Every graph $\hat{g} \in [\hat{G}]_{\simeq}$ is a potential candidate for evolving into another hypergraph through the introduction of a coupler node k . This gives rise to a family of related products in which the common features are contained in the equivalence class of asset graphs $[\hat{G}]_{\simeq}$. These structural aspects of the graphs are higher order *patterns* that express common contexts, i.e., a product line, and problem-solution pairs and provide a rigorous practical approach. We can use such patterns as aggregates in a software evolution framework.

Figure 3.3 shows an example of evolution of an isomorphic graph with coupler graphs. Every path in this expanded graph is a feasible product line and the software evolution process is governed by existence of such paths. From the structural property of the expansion graph, we get the following property of a product line.

Proposition 1 - Product family class:

A sub-system of a software product line forms an equivalence class $[\hat{G}]_{\simeq}$ under isomorphism \simeq .

In practical terms we can interpret the above proposition as follows: there exists attribute preserving transformations that a practitioner can apply to systematically evolve a product line by incorporating a small set of *coupling assets* that meets an enhancement criteria. Using the evolutionary paths, it is possible to traverse the solution space while preserving the original higher order constraints that the seed solution satisfied in the definition of the base solution.

3.3.3 Hierarchical evolution of multiple Assets

The method presented for single asset evolution above provides an incremental approach to make a *one step* transition within the boundaries of a particular equivalence class. We now consider the general case when a collection of sub-products can be evolved into an integrated evolved product by aggregation of multiple assets through its equivalence classes $[\hat{G}]_{\simeq}$ in a structured manner. To accomplish this general framework, we will define mechanism to compose collection of equivalence classes to form larger hypergraphs. The expansion is not arbitrary but is again constrained by requirements on the initial seed solutions which translate into structural properties on the composed hypergraph. In particular, we will establish an existence criteria for product line evolvability based on this structural property.

Each such (sub) product will be represented by its equivalence class \hat{G}_i and will be sub-graph isomorphic in the evolved hypergraph. Evolvability in this case relies on the ability to compose a given set of graphs through a small set of coupling graphs (nodes). Let $\{\hat{G}_1, \dots, \hat{G}_n\}$ be distinct equivalence classes that satisfy distinct requirements/constraints corresponding to (sub) products. Through our traverse operation (isomorphic check) we can identify candidate graphs set $\{g_1, \dots, g_n\}$. We then evolve (Figure 3.4) these candidate graphs by inserting new *coupling* graphs $\{c_1, \dots, c_k\}$, that can be used as coupling in the evolution task. To compose graphs, we need the following join operation:

Definition 11 - Evolve:

a structure preserving map $\Delta : \hat{G} \rightarrow \hat{H}$ is an expansion of a hypergraph \hat{G}_i such that $\{g_1, \dots, g_i\} \cup \{c_1, \dots, c_j\}$ is constructed by inserting of appropriate edges connecting a seed g with an appropriate coupler c resulting in a hypergraph \hat{H} .

With this expansion we can define a general evolution framework as it provides a larger solution space encompassing multiple feature sets and reveals an underlying evolution structure that is not directly apparent to a practitioner because of the combinatorial expansion.

3.3.4 Existence of Product lines

Let $\hat{M} = \{\hat{G}_1, \dots, \hat{G}_n, \hat{H}_1, \dots, \hat{H}_k\}$ be the expanded set of hypergraphs constructed by traversing the solution space through an \simeq operation as defined above. Since

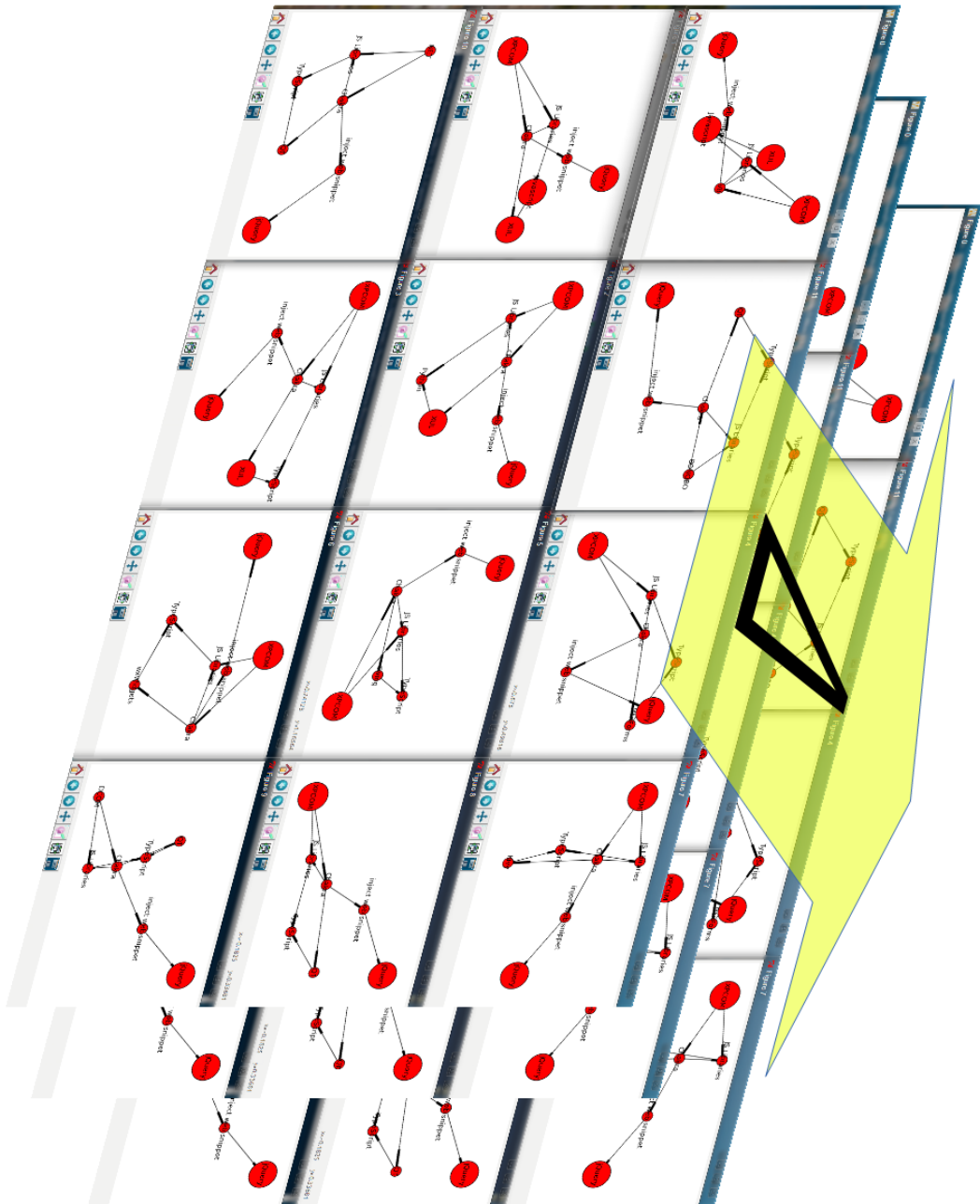


Figure 3.4: Evolving equivalence classes: many views of a compatible solution (for illustration purposes)

each graph g_i and c_i satisfy a distinct requirement, the set $\wp(\hat{M})$ under the \simeq relation is a partial order as follows:

- every $\hat{G}_i \Delta \hat{G}_i$ evolves itself and hence is reflexive.
- if $\hat{G}_i \Delta \hat{H}_j$ and $\hat{H}_j \Delta \hat{H}_k$ then $\hat{G}_i \Delta \hat{H}_k$ hence transitive.
- if $\hat{G}_i \Delta \hat{H}_j$ and $\hat{H}_j \Delta \hat{G}_i$ implies $\hat{H}_j = \hat{G}_i$ hence anti-symmetric.

From the above property we get the following proposition about product lines.

Proposition 2 - Existence of Product lines:

A product line exist if there are k least upper bounds corresponding to a coupling set of size k .

Figures 3.5 and also (simplified) in 3.6 illustrate an hierarchical evolution and existence of k least upper bounds.

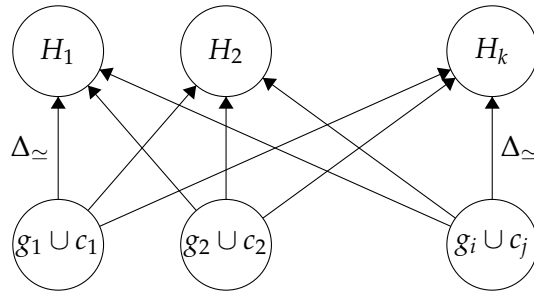


Figure 3.5: Evolving families of products using coupler sets with cardinality k

In practical terms, we can interpret the above proposition as follows: the design of a product family $[G]_{\simeq}$ is parametrised by its constraints and its evolution will also be dependent on a small set of additional requirements. In order to evolve consistently there can be up to k couplings that relates to valid product lines. From the assets point of view, the existence proposition states that a practitioner with a set of assets $\{A_1, A_2, \dots, A_n\}$ at their disposal can evolve a product line with up to k feasible solutions in one exploration step.

The graphs belonging to equivalence classes meaningfully grow i.e., *evolve* with the Δ operation. Any possible subset of the assets represented in a equivalence class $g_i \subseteq G$ can potentially be evolved using any possible assets subset provided by the coupler $c_i \subseteq C$ set. Therefore, the resulting $g_i \cup c_j$ that is isomorphic to the seed will be part of the equivalence class $H_i \in H$ up to H_k potential solutions, which will belong to a family of products.

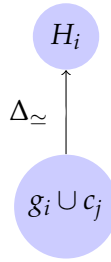


Figure 3.6: Higher level $\{g_i\} \cup \{c_j\}$ gives rise to a (property preserved) new family of products

This expansion technique can be further explored hierarchically to construct a larger solution space as shown in Figure 3.5. H_k is the upper bound for the total solutions available for the operation Δ over the union of the graph set and the coupler graph set. The metric k is parametric as it is dependent on the profile of the asset set. It is possible to quantify this as it relates to the cardinality of the equivalence classes; we have not fully explored this aspect yet.

3.4 Encoding families using a Relational Hypergraph Model

Software product lines (SPL) and software factories have common features. The levels of similitude, as pertaining our asset driven model, enables us to reuse many features. It is possible to say the transition from domain engineering to application engineering can be achieved sharing these features with the operations devised in our model. It may be plausible for the practitioner 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 these operations will evolve models in this multi-model framework actually turning general systems into customised versions adapted to the client's needs as depicted in Figure 3.7.

3.4.1 Software Product lines: Single asset

Here we present a case study of applying our single asset formalism to illustrate the modelling of software product lines based on web technologies. A baseline product corresponding to a seed solution will be defined around which extensions

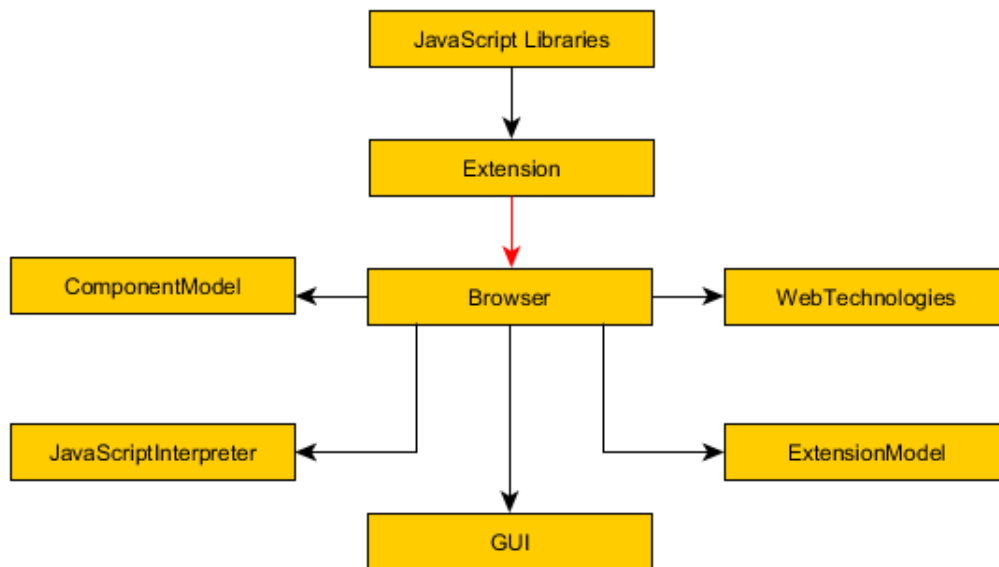


Figure 3.7: Firefox based architectural dependencies. Coupler graph via extension for augmented browsing or custom web post-processing capabilities (relaxed security sandbox may be available)

can be viewed as evolution of the system resulting in product lines. This can be an effective way of creating new useful software by using the browser as a development platform [138] [139]. The extensibility feature of web browsers gives rise to a family of product lines depending on the starting seed solution (See also Figure 3.7).

$$BrowserAssets = \{GUI, ComponentModel, JavaScriptInterpreter, ExtensionModel, WebTechnologies\}$$

For instance, in the concrete case of Firefox:

- XPCOM as the component model.
- XUL as GUI (mainly for user interface)
- SpiderMonkey as the Javascript interpreter.
- An extension model.
- W3C web technologies are expected to be supported by the browsers.

These are technologies part Mozilla technologies [66] and also part of Mozilla Platform [140]. These would constitute the core assets for the seed solution. We gather which are the required browser services would be needed to construct a compatible solution. We can label "abstract dependency" or "logical dependency", not necessarily a software or source code dependency the relationships between assets. (NOTE: To logically elaborate on the concepts, examples of browser internal assets associations are used, for didactic and illustrative purposes. It is not a claim on the appropriateness of such associations).

A corresponding seed solution is given by the following relation where each pair represents a relation:

$$f : [BrowserAssets]_{\simeq_1} \rightarrow [FX]_{\simeq_1}$$

with the following mappings (if needed for functionality):

$$f(JavaScriptInterpreter) = SpiderMonkey$$

$$f(GUI) = XUL$$

$$f(XPCOM) = .NET$$

$$f(ExtensionModel) = FirefoxExtensionModel$$

A set of equivalence classes for a graph of the core asset FX with the edges encoding architectural dependencies and the nodes being part of the FX core asset $\{[FX]_{\simeq_1} = C_1, [FX]_{\simeq_2} = C_2 \dots [FX]_{\simeq_n} = C_n\}$ represents different product evolution sub-spaces belonging to Firefox product *families* depending on the choice of a seed solution for evolution.

Let $[FX]_{\simeq_1} = C_1$ denote a product line corresponding to a seed solution of FX . Using this as the base solution we can develop product lines in different ways as follows which corresponds to evolving a single asset using isomorphic mappings and coupling transformations.

3.4.1.1 Augmented Browsing: Custom Scripts

We could extend the functionality of the previous Firefox solution by adding a new XPCOM component or by adding a new functionality leveraging XPCOM [66] like, for instance, cookie support. We consider the family label not changed as long as components remain the same in any case. However, we can create a GreaseMonkey [47] custom script which allows the seed to be evolved into a new solution. The requirement for Firefox and the script is for GreaseMonkey

to be present as enabler. GreaseMonkey is one node attachment enabler. It allows Javascript based scripts as for augmented browsing or web page custom processing [141]. It is an alternative route to implement custom extensions within a browser extension model. Custom Javascript programming (importing extra Javascript Libraries, DOM and CSS manipulation, etc), before or after all rendering is done, is now allowed. Extra customisation, like for instance, personalised business views, is possible now [47]. Just as a side note, GreaseMonkey scripts can be converted to Firefox extensions obviating the need for GreaseMonkey itself. This could be considered an equivalent solution.

Denote the GreaseMonkey script by the following assets and a corresponding di-graph as a coupler k :

$$\text{GreaseMonkeyScript} = \{\text{CustomScript}, \text{jQuery}\}$$

Our evolved system \hat{FX} can then be represented by a coupled graph which extends our base graph FX transmitting relevant properties. \hat{FX} represents a solution in the larger space which is isomorphic to the seed solution and can be defined as follows:

$$\hat{FX} = [FX]_{\simeq_1} \cup \{(\text{CustomScript}, \text{jQuery})\} \cup \{(\text{CustomScript}, \text{GreaseMonkey})\}$$

In this evolution the following sub-graph isomorphism holds:

$$f : [FX]_{\simeq_1} \rightarrow S$$

where $S \subset \hat{FX}$

3.4.1.2 Another view from the same family

We can consider a different family with similar functional attributes and hence would be isomorphic to a Firefox product line. What we want to show here is that it is possible to start at one point in the solution space and move to another point in the solution space where we realise the same functionality but evolving the system to a different point with an underlying component set that provides a new family. Using the following assets we can develop a new family $[\text{ChromeNET}]_{\simeq}$ as shown in Figure 3.8.

$$\text{ChromeNET} = \{\text{Javacript}, \text{V8}, \text{JSLibraries}, \text{.NET}, \text{WinForms}, \text{TamperMonkey}\}$$

TamperMonkey is the Google Chrome equivalent to GreaseMonkey.

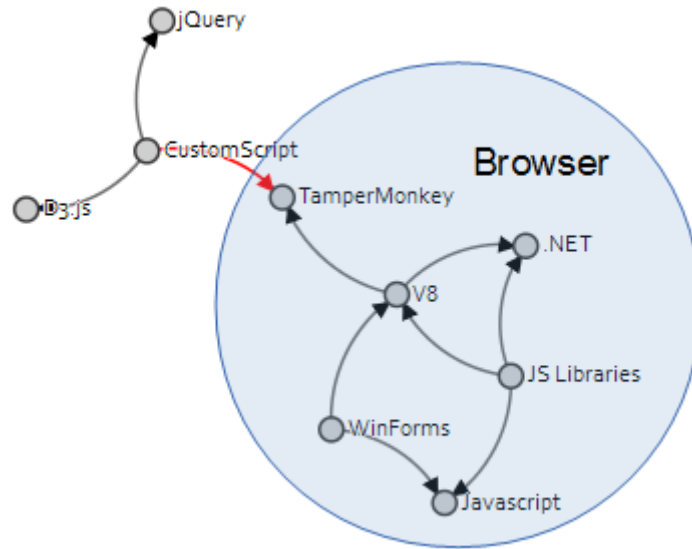


Figure 3.8: Architectural dependencies converging on coupler graph via GreaseMonkey for custom web post-processing

$[ChromeNET]_{\simeq}$ can be considered isomorphic since we can apply the traverse operation and find an isomorphic transformation (among other possible) as follows.

$$f : [FX]_{\simeq_1} \rightarrow [ChromeNET]_{\simeq}$$

with the following mappings (if needed for functionality):

$$f(\text{SpiderMonkey}) = V8$$

$$f(XUL) = WinForms$$

$$f(XPCOM) = .NET$$

$$f(\text{GreaseMonkey}) = TamperMonkey$$

In this mapping, the relationships are preserved while the variations in the product lines are realised by replacing specific components which in an evolution context amounts to choosing a different path in the solution space. This solution is a mere view of all the architecturally compatible solutions represented by the hypergraph (Figure 3.9). This example is easy but there could be cases where the mapping is not one to one and the assets need to be abstracted in for the resulting graphs to be considered isomorphic.

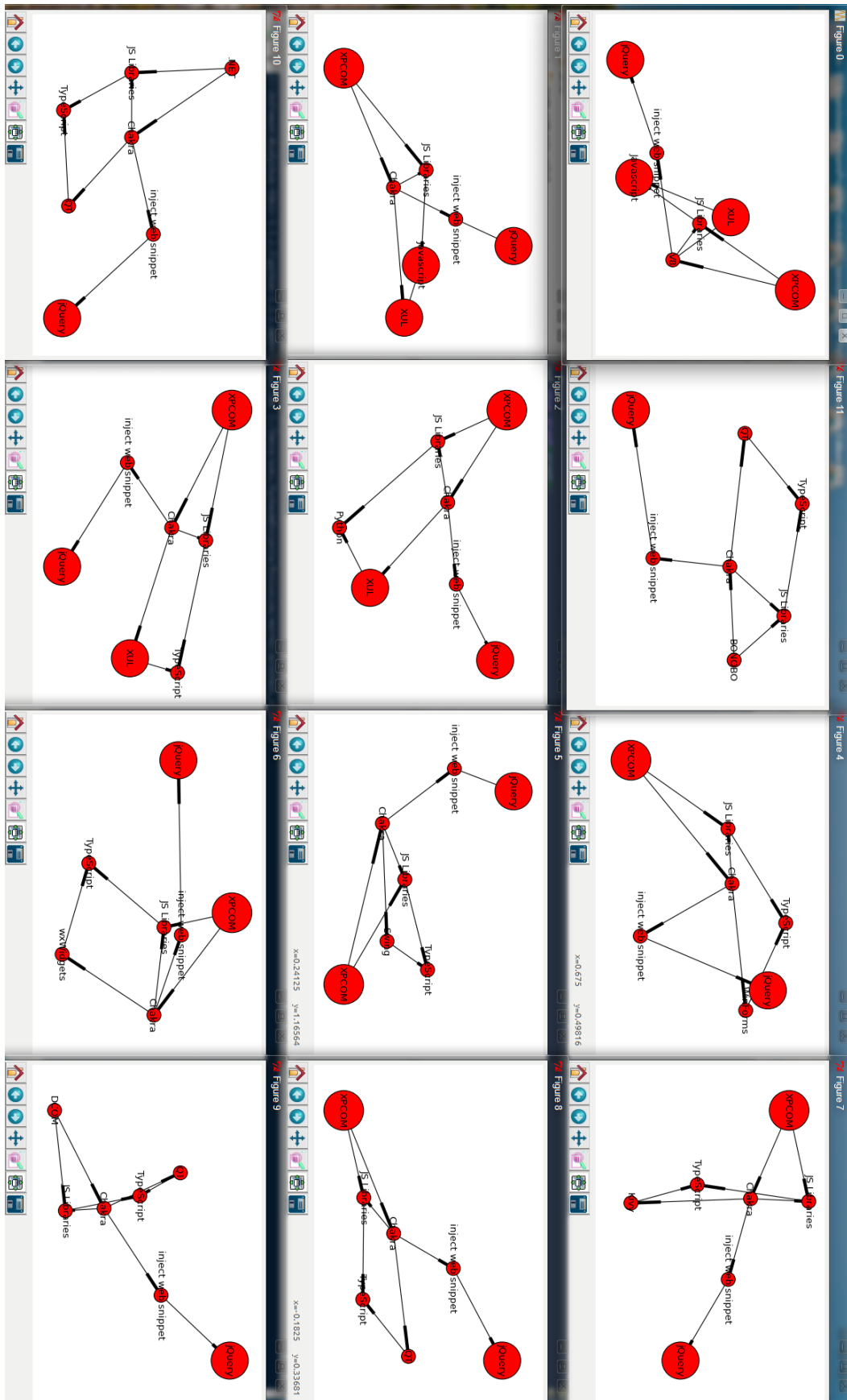


Figure 3.9: Different solutions from the same equivalence class

3.4.2 Coupler enablers or the assets where to grow from

Let a SpiderMonkey [66] instance be created by a python [127] script. Subsequently, a Javascript library could be loaded to provide some extra services. The SpiderMonkey component further enables customisation by loading Javascript libraries and this provides the means to evolve the base functionality in numerous ways. This indeed constitutes a new family of products whether depending exclusively on Firefox technologies or based on other equivalents. However, the point is all these families are related since their dependencies from some point of view make them functionally compatible but also extensible-compatible.

Firefox supports asm.js, a highly efficient Javascript subset. This code is a compilation target (via LLVM [56] compiler and Emscripten [40]) allowing for C++ [126] compilation to target the browser with asm.js [27] as virtual machine code. Therefore, Firefox, considered as an asset, enables growth via asm.js. The only limitation is some OS facilities are blocked by the security sandbox.

Every time access is gained a whole new wealth of potential growth arises. Some assets enable the realisation of new evolution paths. Therefore, we could label them as *coupler enablers*. It is convenient to explicitly add them to the graph for consideration. This means richer graphs make exploration more meaningful and ultimately useful.

3.5 Multiple asset based product lines : Case studies

Here we consider two applications that exhibit the phenomena of structured evolution that can be modelled in terms of single asset and multiple asset formulation developed in the previous sections. As we will note, the higher order patterns of reuse in the form of invariants of the graph leads to a new interpretation of the traditional laws of evolution relating to "continuing growth" and "increasing complexity" for E-type systems evolution [81].

3.5.1 Case Study I: Evolution of Browser Technology

The Mozilla Firefox is an exemplar of an E-type system which has evolved continuously resulting in nearly fifty versions. At the core is a basic functionality of a web browser and variations have been introduced which differentiates the different versions. The trunk-branch development structure has resulted in the

evolution of Firefox as variants of a common ‘product’ which can be modelled as a software product line.

The evolution of Firefox has involved an eco-system of multi-language integration through cross-compilation into java script, in particular, to asm.js subset. Performance requirements for mobile and other emerging high performance requirements is providing new evolutionary pathways through *wasm* [27], an *AST* intermediate representation. Here we illustrate how these evolutionary trends can be modelled in the multiple asset formalism.

XPCOM, XUL (mainly for user interface) and the SpiderMonkey Javascript engine are technologies part of Mozilla Firefox architecture [66, 140]. We consider these to constitute the core assets \mathcal{A}_1 for the seed solution. The chosen assets are a simplification of software engineering elements for elucidating our formalism. They include not just code but higher level abstraction components of the architectural viewpoint of a software product line framework. For instance, the asset *XUL* is composed of the assets $\{XULRequirements, XULDesign, XULImplementation, XULDocumentation\}$ to include all the documentation needed for *XUL* to be implemented successfully.

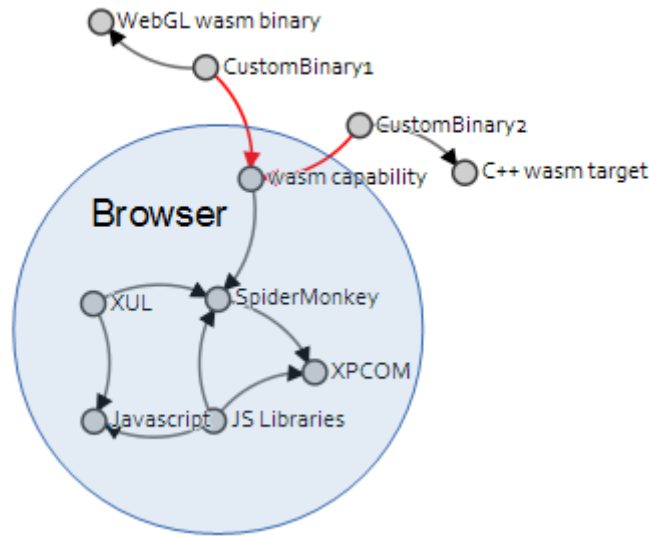
$$\mathcal{A}_1 = \{XUL, Javascript, SpiderMonkey, JSLibs, XPCOM\}$$

A corresponding seed solution is given by the following relation:

$$FX = \{(SpiderMonkey, XUL), (SpiderMonkey, Javascript), (XUL, Javascript), (JSLibs, Javascript), (XPCOM, Javascript), (XPCOM, SpiderMonkey), (SpiderMonkey, JSLibs), (XPCOM, JSLibs)\}$$

Let $[FX]_{\simeq}$ denote a product line corresponding to a seed solution *FX* shown in Figure 3.10. Proposition 1 states that structure preserving transformations can be introduced to systematically evolve a product line. Using *FX* as the base solution we can develop product lines in different ways as follows which corresponds to evolving a multiple asset using isomorphic mappings and coupling transformations.

With the plethora of platform and devices constantly emerging the need for a common runtime for the web has been recognized in recent developments. For efficiency a *AST* (Abstract Syntax Tree) binary based on a Javascript subset (asm.js) known as *WebAssembly*, called *wasm* for short overcomes the overheads associated with parsing asm.js. Let *wasm* be the set containing all possible (validated)

Figure 3.10: $FX \cup \{wasm_1, wasm_2\}$

pieces of software that can be compiled to this binary. This can include any cross-compiled instances although currently C [142], $C++$ [126] are primary targeted languages.

Addition of a $wasm$ component evolves the baseline FX with new functionality. In this instance $wasm$ is used as an AST target for custom java script. As shown in figure 3.10, a coupler node corresponding to $wasm_1$ can be defined with the following relation:

$$wasm_1 = \{(WebGLJsProgram, wasmbinary1), (jQuery, wasmbinary1), (Documentation1, wasmbinary1)\}$$

The evolved system \hat{FX} is a coupled graph $\{FX \cup wasm_1\}$ which extends the base graph FX which can be recorded as a structure preserving map in the evolutionary process:

$$\Delta : FX \rightarrow \hat{FX}$$

We continue to evolve with another component $wasm_2$ to incorporate new functionality. This can be specified with a coupling graph as follows:

$$wasm_2 = \{(C++, wasmbinary2), (Documentation2, wasmbinary2)\}$$

These evolutionary steps using hypergraphs gives rise to a family of products in which the common features are contained in an equivalence class

$$\Delta_{FX} : FX \rightarrow [FX]_{\simeq}$$

Thus, the higher order patterns of reuse are captured as structural properties of hypergraphs ($[FX]_{\simeq}$) and commonalities of product line preserved through isomorphism.

3.5.2 Case Study II: Cloud compute engine

The emerging *software as a service* paradigm provides a new framework for higher order patterns of reuse and new models of evolution that goes beyond traditional software evolution due to the *organic* nature of product evolution in such open framework contexts.

Consider the cloud compute engine where organic growth arises from the possibility to architect products from an open platform of software packages that can be executed seamlessly on cloud platforms [143]. Here we have a set of components, i.e, the packages and engines, that can be deemed as higher order assets. Software products can be architected at a higher level by composition of components. Since the set of available assets can be in thousands, the set of possible products resides in a large evolution space with a multitude of product lines.

We illustrate how our multiple asset formalism can be applied to model the evolution of product lines. Consider the example of a web-based thin client design to create a virtual desktop infrastructure consisting of a browser, an OS and any server side related software pertinent for a specific product.

We use partial functionalities of products by using **Virtual Machines (VMs)**. Open frameworks over the cloud deliver the advantage of using loosely coupled architecture of the web to naturally separate business logic from presentation. The servers (or The Cloud) keeps the *business logic* and the client deals mostly with the presentation. These servers are today load-balanced farms where the server side code or software run in **VM containers** on top of the **host OS**. A **VM guest instance** runs the software needed for any particular client depending on the service layer required. In cloud architecture the Operating System layer is considered the *infrastructure layer*. This business could run the strategic elements tied to the OS inside the container (guest VM) and use any available mechanism to communicate with other software running on the host OS.

Organic growth can occur by simple aggregation of the assets relevant to the side we are working with. We can acknowledge the overall aggregate evolution

as two instances of multiple asset evolution pertaining to server side and client side as follows.

Server Side

On the server side the practitioner can select the necessary assets from the Google Compute Engine [144] (IaaS [22]) collection to find a particular software solution. One solution (requirements) could be asynchronous web server, key-value pair database all running under a stable Linux OS [55] with Debian [39] distribution package compatibility.

A seed solution can be modelled with the following relation:

$$SS = \{(Debian7_OS, Redis), (Debian7_OS, webservers), (Debian7_OS, Node.js)\}$$

We traverse this hypergraph and evolve this seed solution using isomorphic or coupler transformations. Substitution of Debian7 (Debian [39] version 7) for another linux OS corresponds to isomorphic transformation and addition of a custom solution on the server to incorporating coupler graph to evolve the system. In both cases we introduce few attribute changes and the commonalities of the product line will be maintained in the equivalence class.

$$\Delta_{SS} : SS \rightarrow H_i$$

where $H_i = [SS \cup ServerSideCustomSoftware]_{\approx_1}$

Global evolution

On the client side we can evolve the web browser to customize it to provide a new presentation for the thin client that is pertinent to the server side product line. As in the previous section, we can achieve this customization through a *wasm* coupler. The evolution of this sub-product can be parametrised in terms of its equivalence class as defined earlier:

$$\Delta_{FX} : \hat{FX} \rightarrow H_j$$

where $H_j = [\hat{FX} \cup wasm]_{\approx_2}$

The complete system can be modelled as a hierarchical evolution of multiple assets where the server side and client side sub-products are evolved to form a composed product. The equivalence classes of each product are aggregated into a larger hypergraph through insertion of coupler graphs c_i

$$\Delta_{SYS} : \{H_i \cup H_j \cup c_i\} \rightarrow H_k$$

This global evolution amounts to evolving products into families of a related software ecosystem by virtue of creating a family $[H_k]_{\simeq}$ that parametrises the solution space in terms of commonalities preserved through sub-graph isomorphism and variabilities in terms of couplers. Thus, we can evolve systems in a structured manner through higher order patterns of reuse that are related to the structural properties of our meta graph.

In this example it is important to note that the method illustrated above applies to an instance of a multitude of product lines that are feasible from the vast set of assets in the Google compute engine. The example shown here applies to one instance of an organic growth, but the modelling framework is general and is therefore applicable to every instance of composition of a client-server or a distributed solution from higher order reuse of assets. We can therefore view this organic growth in terms of a set of equivalence classes ($[H]_{\simeq_1} = S_1, [H]_{\simeq_2} = S_2 \dots [H]_{\simeq_n} = S_n$) representing different product evolution sub-spaces corresponding to different product families.

3.5.3 Case Study III: Regulatory or Legal Constraints

Evolution of software systems is constrained by artefacts that are not necessarily code or architectural models. In large software systems regulatory compliance, and associated assets in the form of documentation, constrain how sub-systems and entire systems can evolve.

We may need the software (code) to be compatible to a particular licence, open source or commercial. There could be any other **legal or domain specific requirements**, including, for instance, to be validated for use in a pharmaceutical industry related solution. This latter example is also a good example of a *daisy chain of approval* required for all the software elements involved. This may encompass multiple components and tracking them may need to be automated or systematically checked. Also, compatibility at all levels must be enforced. This compliance can be interpreted as a constraint and also classifies the software as families depending on the licences or constraints on the assets we can actually use.

The evolution of this software can be modelled uniformly by insertion of coupler graph that constrains the evolution of the sub-graph with these additional

requirement. For example, the above global evolution can be evolved in two ways depending on the type of constraint as follows:

$$\begin{aligned}\Delta_{SYS_i} &: \{H_k \cup GPL_i\} \rightarrow M_i \\ \Delta_{SYS_j} &: \{H_k \cup PharmaL_j\} \rightarrow M_j\end{aligned}$$

3.6 Conclusion

Higher order patterns of reuse in the form of disjoint assets sets are being proposed to capture evolution in which a family of related products can be evolved in a structured manner. The open frameworks such as cloud engines offers higher order patterns of structured evolution of classes of product lines. We have presented a *meta* graph model as a rigorous approach to software evolution. Our key idea is to enable the exploration of the solution space aided by a structure preserving transformation. To this end, we constructed a general mechanism of expanding equivalence classes of product line graphs. This general framework provides a larger solution space using multiple feature sets. It reveals an underlying evolution branching possibility. This is not directly apparent to a practitioner because of the combinatorial explosion.

A set of well defined operations can represent architectural realisations of equivalent solutions in software evolution. The architectural changes can be introduced and recorded systematically. This is already happening at source code level by collaboration of engineers and/or in open source on a massive scale. We contribute to this trend by providing a model to evolve a system. Using isomorphisms of known solution structures we can short-cut the path to model new software. It could leverage the acquired previous knowledge and even find relationship insights beyond the actual level of software development practice. By freeing the engineers with relevant automation assistance we could actually get more engineering, and paradoxically, we also further the art aspect by empowering their creative choices.

The higher order patterns of reuse in our meta-graph model has implications on how systems evolve and the new metrics that may be necessary to quantify evolution. E-type systems to which Leeman's law were pertinent have been shown to be inadequate for modelling systems that evolve growing organically as per open source examples [82, 83]. In our modelling framework we are able to treat sub-systems as a sub-graph through the coupler transformation. The expan-

sion of Firefox functionalities through *wasm* coupling enables evolution at large scales. Therefore, the system growth can exhibit super-linear growth as in other cases of open source examples shown recently [82]. Simple metrics such as lines of codes to define complexity are therefore not suitable for these meta models of evolution.

In the case study *II* we showed that the evolution space can be large with a multitude of product families. Automating the exploration of this evolution space will require the design of predictors that can use some form of generative models to construct solution subspaces autonomously. Designing predictors will take a step closer to the *automation of craftsmanship* goal and towards a rigorous evolution framework. This is another area of future work we are currently pursuing.

Engineering The Elements of Evolution

Contents

4.1 Introduction	72
4.1.1 Software design insights	73
4.2 Engineering the model	75
4.3 Artefacts as complex operands	76
4.3.1 Assets as abstract building sub-blocks	77
4.3.2 Artefacts as a hypergraph based software factory schema	78
4.4 Evolution and Configurations	79
4.4.1 The model as the basis of a component model for evolution	81
4.4.2 Property transmission, recording and tracking	81
4.5 Evolution Operations	82
4.5.1 Core operations	83
4.5.2 Designing a seed artefact	84
4.5.3 Designing a coupler	84
4.5.4 Shift to Solution view	85
4.5.5 Shift to compatible Coupler	85
4.5.6 Evolution step: <i>Evolve</i> with Δ under \simeq	85
4.6 Conclusions	87

4.1 Introduction

There is a wealth of existing software solutions to many problems. These solutions can be of any level of abstraction. Software available for production needs to be taken into consideration. The relevant pieces of software we are able

work with will be called *assets* following the naming convention for Software Product Lines (SPLs henceforth) reusable assets [145]. The challenge is to create meaningful engineering relations between distinct reusable pieces of software. The result will be an engineering solution to a concrete problem. To further narrow the scope of research only architecture a higher level is considered. Of course, today source code can operate at that level. A browser can be instantiated in the source code and give access to all its features programmatically. Since we are intending to model at high level of abstraction, we can ignore some details in favour of the more abstract higher ones.

In this chapter we will detail the operators and operands from a software engineering point of view. To achieve this we will discuss the rationale behind the design of the operators and operands. There are some design choices that marked the development of key successful software. We use them as guidance to look for time-tested design choices.

4.1.1 Software design insights

Imitating the successful is a common heuristic [146]. However, "It has worked before" is a known cognitive bias. We should imitate some aspects as long as we fully understand the reasons why they worked and in what context. The basis of the design is to have few concrete cohesive operations over complex rich operands. There are some instances of successful software development supporting this approach. Some of these are the ReST architectural style, Unix philosophy, CRUD [34] [22] and SQL [68] as the implementation of Relational Calculus in the Relational Model for Databases.

The ReST architectural style

The Representational State Transfer (**ReST**) is a reinterpretation *ex post* of how the architecture of the web emerged and how and why it works [48]. ReST style architecture performs http operations over resources. These are accessed through the URIs. This is information-on-demand of any kind. It could be text, audio, video, any software or **code on demand** like JavaScript libraries. These resources can be viewed as software assets as part of a higher software solution. We propose that Software Evolution dynamics should exhibit properties similar to systems like the ReST architecture. These dynamics also include other systems

exhibiting scalability in its development process considering all abstraction levels. For instance, in a Web based system, resources are retrieved on demand. These resources can be data or programs. The kind of resource is specified (MIME Types) and code on demand is possible but not required. However, the fact that can be use is a seamless way to extend functionality. This allows for the **right piece of software** to be selected. This point of view is interesting as we also are modelling extension or growth as based on composition or assemblage. Resources are other way of viewing/interpreting our definition of assets (or even artefacts). The fact that they solve a concrete problem is the *valid relation*.

Unix philosophy

Unix philosophy is an example of reuse, composition and modularity. Everything is represented by a file is a main metaphor and text is the universal interface. The standard command tools follow these principles and can be chained to perform complex tasks. Unix pipes allow for composition of tools using text input and output as the (universal) interface. For instance, this was perfectly captured by the famous challenge in Knuth vs McIlroy [147] where the simplified description of the challenge reads (verbatim):

"Given a text file and an integer K, you are to print the K most common words in the file (and the number of their occurrences) in decreasing frequency."

Knuth solutions was a literal programming exhibition exercise written in various pages of elaborate Pascal [148]. McIlroy solution use the power of Unix pipes to assemble a solution using reusable Unix command line tools.

Listing 1 McIlroy's solution

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed \${1}q
```

This used the power of a reusable toolset in a functional programming style. The data is processed until is fit for purpose.

CRUD as a small set of core operations

The (S)CRUD (Create, Read, Update, Delete, and later also Search) set of operations were firstly coined by Martin [34] as basic database actions. Later Fielding reinterpreted these as a mapping to The Web verbs used by the HTTP protocol (post, put, post, delete) [48]. This simplifies the design as the design of Universal Resource Identifiers (URIs, URLs on the Web) requires the resources to be accessible by URI. Since the the verbs are The Web verbs, the resources have to be nouns. URIs facilitate location of resources and are considered part of the search operation.

SQL/Relational Calculus

SQL is a language to manipulate linked tables (rows and columns of cells). The relational model specifies a mode of operation to operate on these structures (using a declarative paradigm) using Relational Calculus [149]. Without delving too much into this topic we can say that some operations are interesting as they steer towards a desired data sub-space just using some parameters. Such operations could be seen as a way to cherry pick and filter data pockets.

4.2 Engineering the model

Previously, we let the set of those relevant software pieces called to assets represented as \mathcal{A} . As per the established theoretical framework the relation $\mathcal{A} \times \mathcal{A}$ defines the set of all potential asset combinations, valid or not.

However due to the combinatorial explosion caused by the Cartesian product of the assets, a systematic and effective system to discard unwanted options needs to be devised. The most discriminate option is for the practitioner(s) to approve valid asset relations.

We have to find a particular software solution using just a subset of all the valid pairs of assets. Let us call it relation f . Still, we could have many valid similar relations. Everyone of them would meet the requirements in various degrees. They also need to be studied and evaluated. Let us call these directional

edges *dependencies*. They could be labelled architectural, logical or design dependencies. What matters is they will be required for the isomorphic check to make sense. Edges meaning will remain through evolution.

This valid relation f can be represented as a graph. The model proposes to use the isomorphism as a structure preserving operation to create new relations isomorphic to f . These allow a coupler graph to expand the original or any and preserve key properties. Therefore, these couplers expand an equivalence class $[\text{artefactgraph}]_{\simeq}$ which represents the collection of all these related graphs under \simeq isomorphic operator (guarantor of validity).

Engineering links of differing models can be configured using XML files as discussed on Maletic et al. [131]. Mainly two ideas are of direct relevance:

- Differing (graph) models can be connected (linked).
- Different configuration XML files can be used for this purpose.

It is clear we can encode graphs in a very abstract way and similarly operate on them. Let us agree isolated nodes are to be called **assets** and they are only elevated to **artefact** status by being part of a **graph** encoding a software solution.

The model can be viewed as the basis of as a software factory to produce SPLs based solutions. This is being achieved with only key operations over configured artefacts as opposed to lower lever source code programming.

4.3 Artefacts as complex operands

Software is also a technical artefact produced by the intellect. Based on this commonly used terminology we can specify further and denote an *artefact* a (software) resource with engineering meaning, a solution template, a building block [150]. The standard software engineering definition defines it as the software process technical outputs. Similarly from a SPL based project documentation point of view [151]. Let an *artefact* be a solution with the engineering information necessary to support informed decision making.

Based on our model an artefact is represented by its base graph as representing a **solution view** belonging to $[artefact\ graph] \simeq$. Therefore, it makes sense to use **just one graph** as the artefact identifier for the whole equivalence class under \simeq . Artefacts also contain the nodes where they can be attached to or attach from. By encapsulating this information as part of the operand we reduce the amount of information needed by the operators.

We will seek to explicitly add the extra information relevant to the solution context. This would be the task of the practitioners through a set of configuration files. This fits well with the concept of software factories we discussed in other chapters. Thus, we can pack all the engineering information in one single entity. This will facilitate later reconfigurability and operation.

In our model, the operands are the relevant software artefacts, the building blocs. These artefacts can interact in a finite number of ways and they should be ready for that. To do this, they need to carry the necessary information for any devised operation to be successful. Different Artefacts create a hierarchy of operation abstraction.

There is no impediment for the lower level of artefacts to give rise to higher level of artefacts. It is important for any structure to recursively allow for the emergence of higher level structures and interactions. It is also desirable to allow for these to allow other practitioners to come up with solutions not foreseen by the initial designers of the system. Although it poses challenges, the strategic advantages outweigh the disadvantages, like innovative evolution with novel artefacts.

The goal is to enable the software artefacts to also contain the information for the eventual purpose of achieving software development automation. Programming is increasingly replaced by just configuring the necessary items.

4.3.1 Assets as abstract building sub-blocks

Artefacts are resources encapsulated in a minimally meaningful structure. The chosen structure is a graph encoding the relationships of the assets. Assets themselves could upgrade to artefact by gaining such structure (graph) if needed. This

allows us to focus the level of abstraction that better captures the engineering specifics of interest. Although for generating SPL high level assets are used other abstract items like key information (like a supported **standard**), data or **documentation**. Documentation *is* software and should be able to be included into account.

4.3.2 Artefacts as a hypergraph based software factory schema

Lets engineer a software solution, and represent it as a graph, based on the assets available. This is one solution but there could be many. We could come up with another solution functionally isomorphic to the previous one. Similarly, this make both graphs isomorphic as depicted in Fig. 4.1. Both solutions solve the same engineering problem and they are equivalent in such sense. However, they are different since they will have some disjoint features. For the purpose of evolution these features may be important. Exploring is part of the discovery of novel solutions. To illustrate lets use a generic example:

In the Figure 4.1 below we have two artefacts.

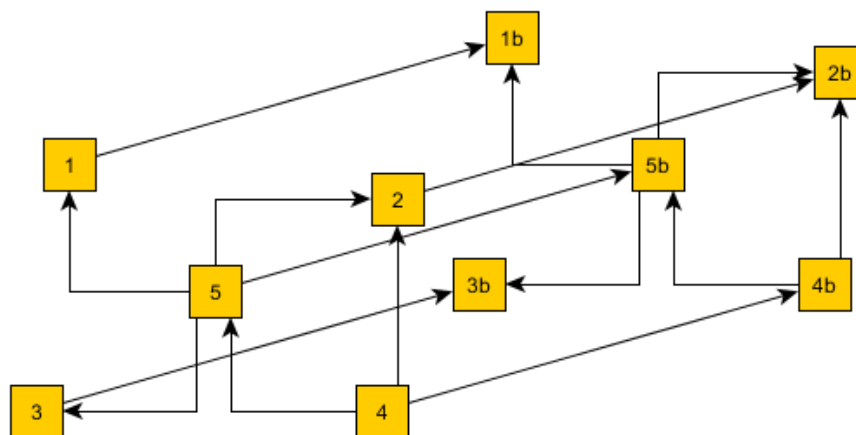


Figure 4.1: Artefacts virtual connection

The assets are the vertices or nodes and the edges are the orthogonal edges. **Node 1** is, for the purpose of the project, functionally equivalent to **node 1b**. Same logic apply to subsequent nodes. There are connections, albeit virtual, which links both graphs. These connections are the oblique edges4.1. A hypergraph is a graph whose edges are a set of nodes. The orthogonal graph edges have the same meaning as long as they go from and to the same node (or nodes if applicable).

Said differently, the numbers are the same and therefore graph isomorphism is preserved. However the virtual connection packs nodes in sets. We need to encode all this information into an artefact.

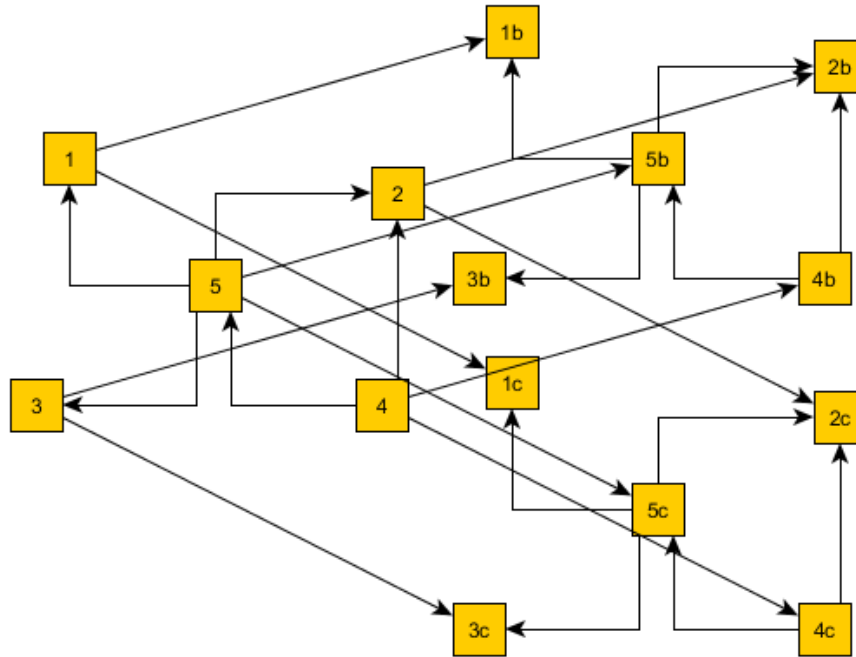


Figure 4.2: Artefacts virtual connection, artefact added

We continue adding isomorphic graphs if possible as shown in Figure 4.2. They all belong to the same family or *equivalence class*. Thus, valid software is modelled. The implication is that families defined by evolution steps can help classify software into taxonomies. We can use just one artefact as the representative of its equivalence class, $[\text{artefact graph}]_{\simeq}$. Therefore, there is one way of expanding an equivalence class $[\text{artefact graph}]_{\simeq}$. This is performed by adding a known valid node relationship. We will denote this node as *isonode* for further reference.

4.4 Evolution and Configurations

Like in a Software factory the software will be composed with available assets with known **configurations schemas**. This composition causes the artefact to gain or loose *properties*. These properties can be predictors as studied in previous

chapters. The predictors should be included as part of the artefact configuration. These configurations represent contexts affecting the evolution.

The evolution has to be modulated by the required views at every Δ step. The configurations take into account multiple **semantic aspects** regarding the software engineering of the solutions space. Any **engineering data** should be included in configuration schemas. These configurations can be encoded in a relevant data format.

Listing 2 Example of asset configuration fragment layering new metadata in design-time

```
<asset name="node.js" version="4.5.0">
    <server> asynchronous </server>
    <feature> event-driven </feature>
    <feature> non-blocking I/O </feature>
    <licence> MIT </licence>
    <language> JavaScript </language>
    <owner> John Smith </owner>
</asset>
```

As previously said, existing software assets can be leveraged as reuse elements to create **diverse solutions**. The intention in designing these configurations is also to minimise the expense of customising combinations of configurations. The result of this mode of operation furthers the **strategic reuse** of these software assets. Practitioners need to discover how to configure seed artefacts *to grow them* into final products. It can also be enhanced by operating on **standardised software repositories**.

It is worth highlighting that, as part of the factory, configurations can be tailored to describe and record the evolution of **inter-model relationships** or model view-shifts [131]. Thus, to actually model the isomorphic graphs (oblique relations) is feasible as different valid *solution-options* can be linked. This could be one element of configuration to take into consideration. Nevertheless, it solves more, as such XML configuration file could encode even more detailed aspects of the modelling. The practitioner deliberately decides which properties are of interest

at any moment.

4.4.1 The model as the basis of a component model for evolution

One property of the ReST architectural style is the **resource on demand** feature. In an assemblage of functionality perspective is of particular interest the *code on demand* feature. This can be seen as a basic component model characterised by being very loosely coupled. It also has the feature of being mostly source code based (we can ignore obfuscated code here). This is a flexible approach to model for full customisation abilities. This is due to the fact that composing in this case only requires URIs (like URLs). It is remarkable how simple and easy is to extend the functionality of a JavaScript file just by linking with other files. **Unrestricted linking** is precondition to create scale-free networks [94].

4.4.2 Property transmission, recording and tracking

The Firefox extension model is similarly extendible as explored in the previous case studies showcasing of the use of the model. The Firefox extension model is based on standard web technologies. This causes the extensions model *to inherit* some of their engineering properties. The **transfer and tracking** of properties as defined in the artefact configuration must be included as part of the artefact as **engineering information**. We encapsulate the **engineering value** into the artefacts as fully configured complex components. Emerging dynamics relating to the engineering data can be analysed by operating on artefacts as **views of complex components**. Indeed, software should be able to upgrade to a different configuration by only **swapping assets** creating a new software view part of $[artefact\ graph]_{\simeq}$. Some properties remain unchanged and some are changed or upgraded. The evolve operator Δ preserve encapsulated properties but in every step the evolution record is kept. This way property tracking is enabled within the artefact as it grows.

The value of properties as metadata

To illustrate what properties can be lets use a practical engineering example: Network programming to deal with multiple concurrent clients required multi-

threading programming. Each thread handles each client. Programming thread-safe software is an engineering challenge. Multi-threading is handled by the web server which enables clients to make concurrent requests via standard protocols. However, as this functionality is provided by the server, there is no need to program it as any other server from the ground up if the threaded web server is the interface (with all the complexity it entails). Therefore, this complexity can be bypassed by choosing to use a server with this capability. Thus, property transmission occurs on element aggregation. Such element, a software piece, aggregates its properties to the system. There would be no need for such properties to be added separately by interfacing with this software piece. Therefore, knowing any properties of a piece of software provides a potential engineering advantage. As later was discovered, a side effect of using threads this way is I/O blocks affecting the performance of the servers. To fix these, asynchronous servers were created which place client handling in a single event loop, a single thread. Client handling is therefore more efficient. Just by replacing the web server a new software solution handles this specific problem. This is easy as **The Web** is a loosely coupled software solution. This new property could be encoded as an XML text (tagged) entry as implemented in Listing 2. Tracking a few properties is intuitively easy. However, analysing and tracking of a vast network of thousands of interlocked properties, from possibly unfamiliar pieces of custom software, is a challenge indeed. This requires configurations that will enable the necessary automation level for decision making support.

4.5 Evolution Operations

Based on the model theory previously researched Equivalence class represented is by relations that are isomorphic. The artefact represents an equivalence class $[artefact\ graph]_{\simeq}$. The resulting artefact from the expansion of the graph by a coupler will create a new different equivalence class $[evolved\ artefact\ graph]_{\simeq}$. All of them are related if the relation (\simeq) of this expansion (isomorphism) can be done using different couplers on the same equivalence class (the evolution). This means that all artefacts governed internally by the same isomorphism can be **grown or expanded simultaneously**(Δ) to a larger artefact **using exactly the same couplers**. We want to be able to make an intelligent analysis over which expansion is more suitable based on the specific engineering requirements for the problem. The

previous equivalence class evolution step will always be sub-graph isomorphic to the next step. This chain of steps defines families of solutions. A kernel of fundamental operations to operate on artefacts through various means. These include SCRUD basic implicit operations, evolution Δ under \simeq and exploring or fanning out the corresponding equivalence class $[\textit{artefact graph}]_{\simeq}$ to select or shift to a **solution view**. For an overall view see Figure 4.3.

4.5.1 Core operations

CRUD based core operation affecting artefacts:

- Create: Artefacts or assets definition and configurations.
- Read: Artefacts or asset and configuration reading.
- Update: Artefacts or assets and configuration amending.
- Delete: Artefact or assets and configuration Deletion.
- Search: Enabled by artefact and its graph as an identifier.

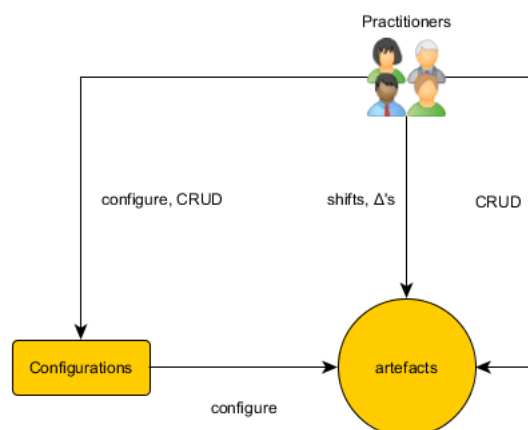


Figure 4.3: Practitioners define and affect artefacts through various means

The staging and execution of the **evolution operation** Δ comprises several steps in various stages. The following is a description of them as connected to the model.

4.5.2 Designing a seed artefact

1. Select relevant assets from available assets set
2. Make a **seed artefact** (a graph from valid asset pairs)
3. Select the same assets or other definition of assets from other practitioner(s)
4. To be isomorphic under \simeq other artefacts encoded must be isomorphic. This is implicitly highlighted by the oblique relations among the artefacts (isonodes). They are composed of valid pairs (orthogonal edges). This makes them expand their $[seed_artefact\ graph]_{\simeq}$ further
5. Stage $[seed_artefact\ graph]_{\simeq}$ defined.
6. The configured artefact is the representative or Id of its equivalence class.

Now a solution view must be selected from all available in $[seed_artefact\ graph]_{\simeq}$ by shifting in that subspace.

4.5.3 Designing a coupler

A coupler graph can be viewed as an artefact if needed (It is a solution too).

1. coupler: Select same assets or other definition of assets from other practitioner(s)
2. The node where to evolve from must be identified and has to be present in the coupler graph
3. Make coupler artefact
4. Stage $[coupler\ graph]_{\simeq}$ defined

Now a coupler view must be selected from all available in $[coupler_artefact\ graph]_{\simeq}$ by shifting in that subspace. These operations limit each other and the solution view shift should happen first but it is not mandatory.

4.5.4 Shift to Solution view

Branch out or shift by selecting a solution view from equivalence class. These steps are performed by the practitioner but can be partially automated. Both the seed and the coupler artefacts base graphs must contain a common node. The coupler artefact carries the information about its coupling options.

1. Evaluate $[seed_artefact\ graph]_{\simeq}$ properties and other configured engineering data
2. Based on the evaluation, shift to solution view from $[seed_artefact\ graph]_{\simeq}$

4.5.5 Shift to compatible Coupler

Branch out or shift to a node where to attach. Any of these steps are, as before, performed by the practitioner but can be partially automated. Both the seed and the coupler artefacts base graphs must contain a common node. The coupler artefact carries the information about its coupling options.

1. Evaluate coupler graphs properties (perhaps modelled as an artefact) and other configured engineering data
2. Based on the evaluation, shift to the appropriate coupler

4.5.6 Evolution step: *Evolve with Δ under \simeq*

Advancing or Δ 's: single asset Δ (using one solution and one coupler), multiple asset Δ (general case using sets). These also can be viewed as thin branches or thick branches depending how much solution subspace they explicitly consider. A thin branch can represent a thicker branch. Thus, we will usually talk about single asset thin Δ as the *evolve* step. Advancing can produce growth by creating a new graph expansion by coupler attachment under a new \simeq .

With previous coupler or another suitable one this will advance and grow into a new solution subspace:

1. Δ : After the two shifts, **assemble or compose** the new resulting **validated** artefact. This expands the graph and therefore the hypergraph.
2. The resulting artefact is the representative or Id of its new higher level equivalence class.

3. Final stage $[evolved_artefact\ graph]_{\simeq}$ defined.
4. Their multiple property profiles can now be re-evaluated or expanded.

To model multiple assets also several single asset evolution steps can take place. This occurs if and only if the result of evolving those artefacts result in incompatible $[evolved_artefact\ graph]_{\simeq}$ from a future evolution point of view. Notice that configuration contexts within artefacts also play a role in the Δ operator. Family hierarchies thick branches can be divided in thin branches pertaining smaller related subspaces.

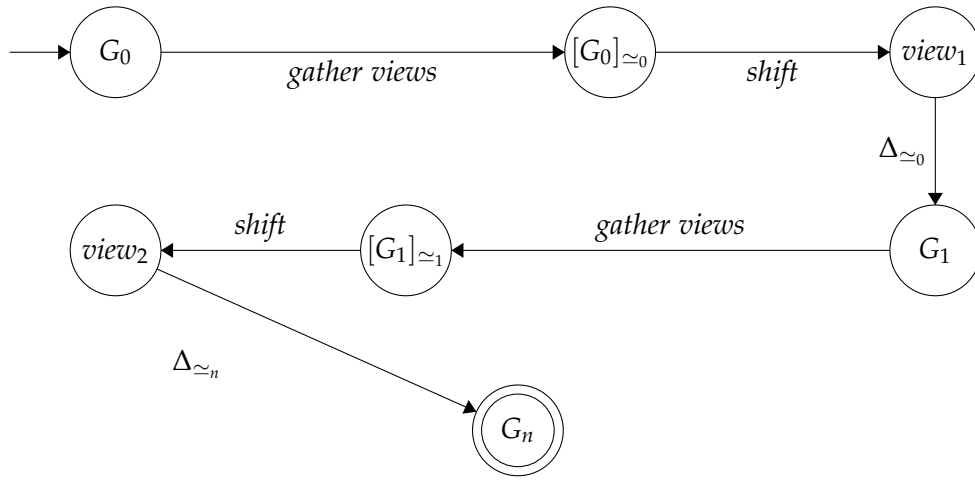


Figure 4.4: Sequence of events defining a FSM [16] and also an evolution path to a higher solution G_n .

Definition 12 - Traverse:

Search for isomorphic graphs (or sub-graph) to a graph related to $[graph]_{\simeq}$.

Definition 13 - Gather:

Compiles the known solution by **searching** and **adding** views belonging to an equivalence class under isomorphism.

There two ways of gathering the views:

1. *Explore* $[seed\ graph]_{\simeq}$ isomorphic node relationships and **add** the views.
2. *Traverse* to find **sub-graph isomorphisms** in graphs built with other relationships and **add** the hits as views.

Both ways enlarge the equivalence class. *Explore* uses known data and *traverse* is a deeper exploration that may require building a relationship collection repository. We could have a name for short traversals like probe or scan.

4.6 Conclusions

A few key operations manipulating rich complex data in the form of artefacts. These encapsulate all the engineering information including relevant contexts. These contexts could serve as the basis for software factory schemas. The model as designed is characterised by being able to:

- Encode assets and their information
- Encode artefacts as instances of $[artefact\ graph]_{\simeq}$
- Encode the expansion of the $[artefact\ graph]_{\simeq}$ by adding an isonode.
- Encode the hypergraph as part of the operands and operator
- Enable exploration for the $[artefact\ graph]_{\simeq}$ (with known configuration data)
- Enable discovery of new graph or subgraphs as part of $[artefact\ graph]_{\simeq}$ (thus expanding it)
- Enable evolution by expanding the hypergraph using artefacts via the Δ operator
- The resulting structure should be *evolvable* or evolution-ready in the same fashion (the feedback loop).
- Enable organic growth as previously defined: a step-wise and bottom up ordered and scalable hypegraph expansion.

The evolution over the potential solution space results in process that can be modelled as 3 basic events. The shiftings (internal, first and in any order) and one Δ . Both shiftings need to occur before a Δ can take place. This generates a new FSM that models the events as an evolution path to reach a particular solution view.

The branching or shifting can be done by the practitioners based on their engineering talent. These shifts could be automated, up to level for certain cases, as we advance (evolve) if we have known stop conditions.

- Solution branching: Solution view wise shift. **Shift to** seed artefact solution view or perspective from the equivalence class $[seed\ graph]_{\simeq}$. This selects solution sub-space. Moreover, it limits coupler candidates.
- Coupler branching: Coupler wise shift. **Shift to** (artefact with) coupler graph. There could be many solution view compatible graphs each possibly featuring a different attachment node.
- Advance: **Evolve** Δ using compatible coupler allowed by \simeq

It makes more sense to shift first to the solution view but the other cannot be ruled out as a selection preference to limit solution views subspaces as possible candidates for attachment. Every advance increment represented by a $\simeq_{counter}$ governing a hypergraph expansion and therefore it creates a new isomorphism relation and increases that **counter**. The chain of events creates branches or paths through the solution space. Moreover, operations altogether can be used to scan or probe (by traversal) the potential solution space from a starting point in δ_{\simeq} increments. This process uses the two-dimensional selection using view-coupler branching pairs as the means of steering or shifting to the desired direction. This obviates the need for input in every step as all the branches are traversed and assessed. We could break the solution isomorphism using metadata encoded into the artefact thus discarding unwanted exemplars. Therefore, we are able to simplify the search by reducing the number of options. This narrows the branch points, to continue evolving from, to more appropriate ones.

Evolution Automation Feasibility

Contents

5.1	Introduction	90
5.1.1	The model in contrast	90
5.1.2	Python prototyping	91
5.1.3	The model	92
5.1.4	Cloud solution	93
5.1.5	ReST microservice	93
5.1.6	Use cases	93
5.2	Architectural Overview	96
5.2.1	Jupyter server	97
5.3	Configuration Context files	98
5.3.1	Encoding Assets	100
5.3.2	Encoding Artefacts	101
5.4	The Model as a whole	101
5.4.1	Artefact custom initialisation	103
5.4.2	Applying Configuration Contexts	104
5.4.3	Instance metadata	106
5.4.4	Configuration Contexts Automation	107
5.4.5	Evolution operations Δ	111
5.4.6	Evaluating and tracking desired properties	111
5.4.7	Searching for other isomorphic solution views	111
5.5	I/O and Visualisation	116
5.5.1	Displaying static graphs	117
5.5.2	Displaying dynamic graphs	118
5.6	Modelling: Examples of scripting use	120
5.6.1	Single asset evolution	122
5.6.2	Multiple assets evolution	126

5.7 ReSTful microservice	127
5.7.1 Selectable Test scenario (sandbox function)	129
5.7.2 ReST Resources	129
5.7.3 How to implement ReSTful operations	129
5.7.4 A test scenario to showcase the model basics	131
5.8 Notebook deployment and Jupyter access	132
5.9 Microservice deployment	134
5.10 Conclusions	136
5.10.1 Evolution as documentation	137
5.10.2 The model should be the deepest module	138
5.10.3 Splitting Artefact Class	139
5.10.4 Asynchronous operation upgrade	139
5.10.5 Future scaling	140

5.1 Introduction

A sample implementation of the model is provided to showcase automation possibilities and the viability of the proposed model beyond theory. The main goal is to be didactic and to demonstrate that the model can be implemented using current technology. All software used is licensed as open source. The prototype maximises the amount of insights gained with respect to the model theoretical capabilities. It also serves to assess the appropriateness of the current engineering design choices.

There are many models and tool related to software evolution. Let us make a comparison with significant examples to help understand our model and implementation choices. We can make a straightforward comparative if we adhere to Harn et al. terminology [85, 86]. These terms correspond to previous engineering chapter dealing with operator and operands. Their effect over the hypergraph structure is also considered.

5.1.1 The model in contrast

Now we can compare our model and proposed prototype design to other existing approaches to gauge further understanding.

Table 5.1: Contrasted approach elements classified using Harn et al. terms.

	software evolution objects	
	<i>software evolution steps</i>	<i>software evolution components</i>
Harn et al. [85, 86]	Software prototype demo step, issue analysis step, requirement analysis step, specification design step, module implementation step, program integration step, software product demo step, and software product implementation step.	Criticisms, issues, requirements, specifications, modules, programs, and optimizations
Pat-evol [6]	XSLT graph transformations to replicate CRUD primitive changes.	GML graphs. Repository of changes patterns mining. Configurations.
Our model	solution view shift, coupler view shift, thin Δ , thick Δ , isomorphic binding each step (\simeq_i)	Artefacts, Software assets, Asset Relationships (including isomorphic ones). Asset Configurations. Isomorphic graphs and their preserving expansion.

Table 5.2: Completely different hypergraphs

	Hypergraph	
	<i>hyperedges</i>	<i>nodes</i>
Harn et al. [85, 86]	software evolution steps (software development event)	software evolution components (software outputs and process artefacts)
Pat-evol [6]	None: Simple graphs.	
Our model	Assets and their graphs. Artefact instances.	Isomorphic preserving nodes, graphs of assets. Isomorphic graphs and their expansion viewed as sub-graph isomorphism (\simeq_i).

5.1.2 Python prototyping

Python [127] excels as a text processing language. It is suitable for prototyping where changes and overhauls could be frequent and severe. There are some particularities of Python as a development language. A python file is a module and a folder is a package. From an object orientated point of view a module has its own namespace scope and behaves and can be used as the singleton pattern if needed. Namespaces are incredibly useful not only because avoid naming clashes but they allow us to use names based on the context at hand.

Functions are object and can also be loaded independently. This allows for and encourages intense modularity. A strict OO approach like in Java or C# [32] is dispensable if other appropriate options are provided. There is freedom to apply multi-paradigm development techniques where applicable within the frame of the problem.

Python libraries for science are comprehensive. They come pre-packaged in distribution bundles like Scipy [63]. They also are included in large comprehensive packages like Anaconda [26]. There is a large community behind them to support open science. These bundles simplify prototype deployment.

User interaction with the model can be handled by a Python script, the python interactive console, a Jupyter notebook (Figure 5.1) and/or via a ReST microservice. It should be noted that all code listings shown, not the original files, are justified close to 84 columns. This has been done in accordance with Python code logic (meaning it will still work). The full listings, the module documentation as well as other larger prototype outputs, are available in the appendix. The source code is commented using restructured text for Sphinx, the de facto documentation tool for python source code.

5.1.3 The model

The objective is to keep the data processing on the server as much as possible and to leave any client just as a presentation layer to interact. We use NetworkX [61] as the graph aware library for the model formalisation implementation. The relational information is loaded into a digraph data structure. The class Artefact represent any modelled compatible solution. Known solutions are deduced from any input data. The graph acts as the ID for the equivalence class. Meanwhile, The isomorphic compatibility \simeq is guaranteed at any time. The hypergraph emerges by adding any compatible solutions views found. One way that can be done it is via configuration files or by adding any subgraph isomorphic solution view. There is a mandatory configuration file currently called *assets.xml*. It defines some sample properties and isonodes. Any other tag can be added here or in other configuration file. The content of these files details engineering information that will be added to the graph nodes as a metadata payload. No payload has been added to the edges (but it could be used). The whole configuration sys-

tem works as layered where tag clashes perform data overrides. The dynamics offered by the combination of the graphs with these configurations bring engineering possibilities we will study in depth.

5.1.4 Cloud solution

A major development is the maturity of the Jupyter software package as a viable solution for client-server interactive reporting [53] [54]. The systems used to be called IPython [49] and to be restricted to the Python language. Now it supports many languages through the use of plugins called language kernels. It can be deployed in many *cloud* services providers (including major ones) as described by this small survey [53]. The notebook files with custom sample tests can be uploaded for easy presentation and to show how the implementation works. Since this is a client/server solution, the prototype is as loosely coupled as the web. It is easy to replace client and server with no issue propagation. It is possible to import Python based notebooks as modules in a regular development environment.

5.1.5 ReST microservice

The microservice interface is provided to show how the model fits the ReST architectural style (which of course includes The web). The dual purpose was to achieve a convenient way to display outputs and thus testing. It facilitates Jupyter integration as the resources are accessible in various ways (i.e. as show in Figure 5.2). ReST style interface facilitates a uniform and efficient communication system. The bulk of the data should be downloaded once in a real scenario. Any other transfer happens by incremental **change events**. The server side is intended to keep the data and its processing features while any presentation concerns are on the client side.

5.1.6 Use cases

Finally, the single asset case, multiple asset case and various cloud platform modelling cases are featured as examples. They show how to use the model with a problem. The tool was incrementally used for the whole thesis. That is why visualisations features were built up completely ad-hoc as needed. Examples of such

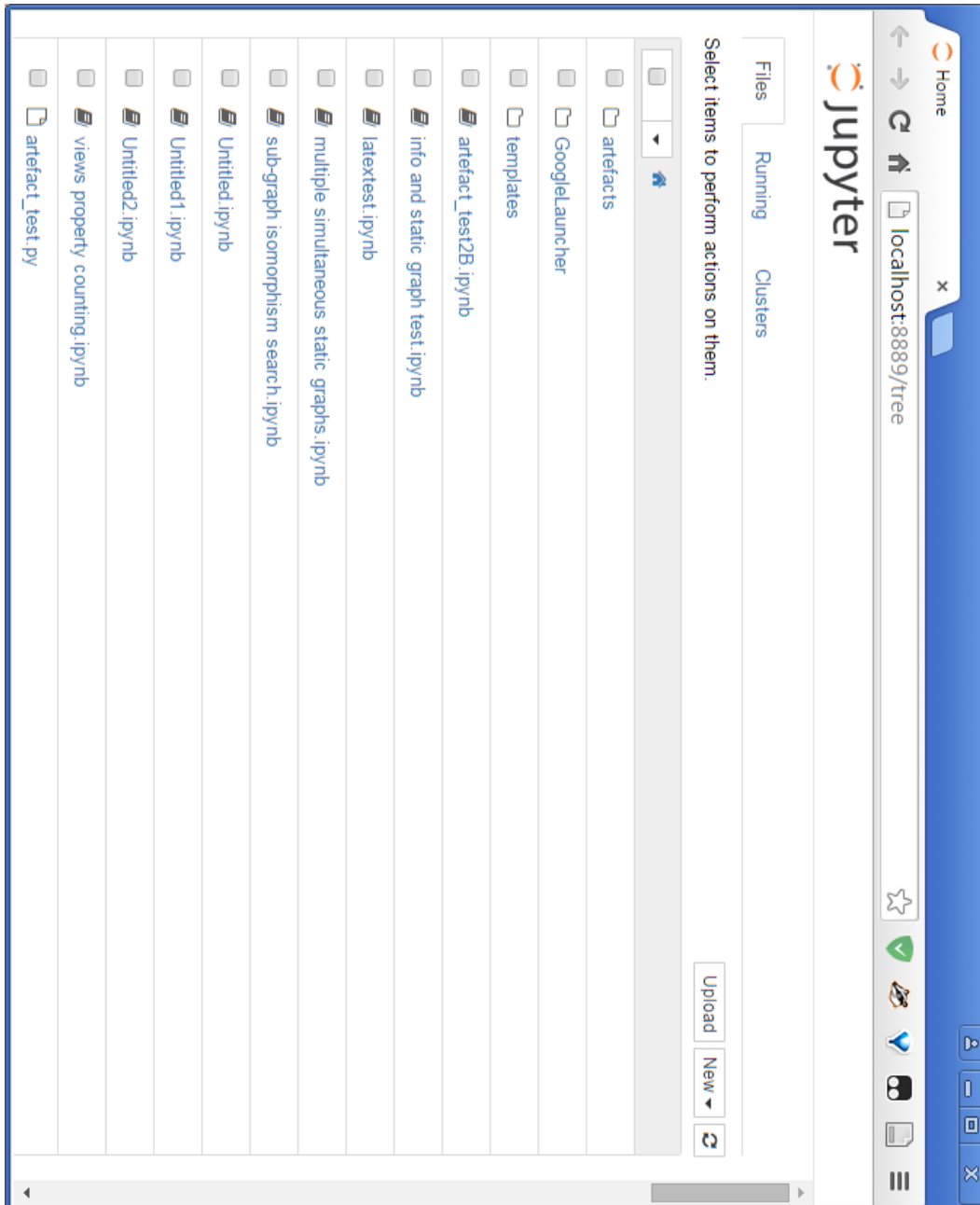


Figure 5.1: View of the starting folder of the running Jupyter server using a web browser as the client. Existing notebooks with .ipynb extension.

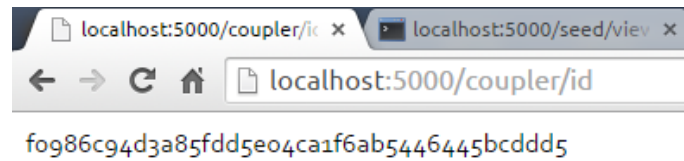


Figure 5.2: Getting the SHA-1 hashed ID of a resource stored as *seed.json*

visualisations in Figure 5.3.

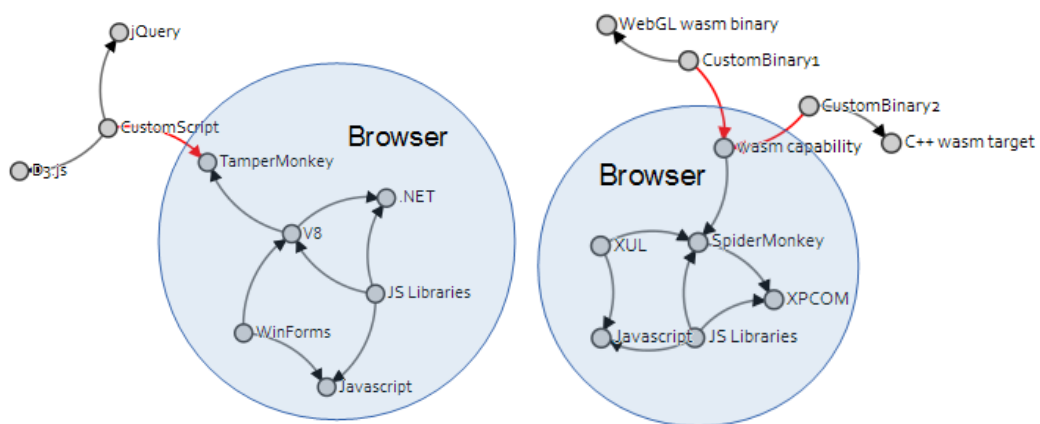


Figure 5.3: The prototype as a rudimentary graph imaging tool. Blue transparent circles added with LibreOffice Draw

5.1.6.1 Cloud integration

Deployment steps are similar albeit different for any cloud provider like Google Datalab [17] or Azure [29]. It is assumed the practitioner can install software in a personal virtual machine (VM) instance. The Jupyter notebooks along with a web browser could serve to interact with the model remotely (Figure 5.4). This is an alternative to the desktop deployment. This is the foundation of the prototype, as a tool concept, to be useful from a teamwork perspective. Currently, there is the option to run a Docker image locally with all the necessary framework [152]. For our needs the Anaconda package distribution sufficed.

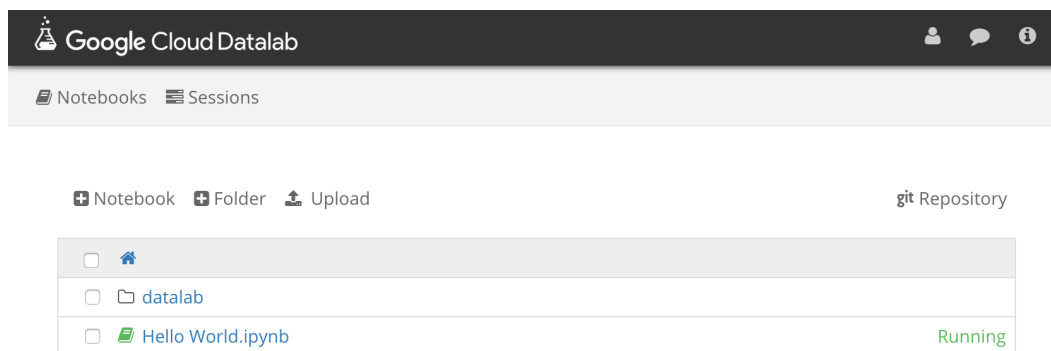


Figure 5.4: Placement of custom Jupyter notebook files in a Google Cloud Datalab VM instance. [17]

5.2 Architectural Overview

The *model* module features the implementation of the model. Printing and information functions are implemented in the *ioutils* module. This module features any I/O and related. It is a utility module as the name indicates. The prototype is implemented in Python 3.5. The interpreter and needed libraries are all included in the distribution used (Anaconda 4.0.0). The model works by using the python modules by various means provided. This includes the Jupyter server, any web browser or directly scripting in Python 3. Larger screenshots are featured in the appendix.

Notable libraries (and software) not included with a standard python 3.5 interpreter (server-side view):

- NetworkX [61] for the graphs and graph querying, including sub-graph isomorphism.
- Matplotlib [57] for the static graphs
- Jupyter [53] [54] (server) to create, configure and run Jupyter notebooks.
- Flask [41] (server) to implement a ReST based cloud microservice (HTTP interfacing).
- Jinja2 [50] template system.

Notable libraries (and software) on the client side:

- D3.js (or just D3) for dynamics graphs on the web using SVG rendering.
- A GPLv3 customised template for the D3 force graph to be later used as a Jinja2 template.

The Jinja2 templates are just normal HTML files where some external programming has been added. The production of dynamic graphs was a bit convoluted. Loading a JavaScript file with the graph data was a fair solution. However, the evaluation needs increased the need for a faster cycle of testing. That lead us to conclude a more streamlined option was needed. This was achieved by reusing existing functions as this was more a convenience that a core model need. In any case, it is easier to convey some information visually. Therefore, as the static graphs felt short the new improved dynamic graph solved various problems including positioning the graph for a screenshot. This Jinja2 solution was the shortest path to solve the problem given existing function but not the best. In the conclusions section some pointers will be given for an easy upgrade.

5.2.1 Jupyter server

The Jupyter server can access all the prototype modules since it features a python interpreter. It also provides a way to use a web browser as the client. The prototype can be deployed as VM or docker image for convenience. For an existing VM instance the folder can be added in the relevant path.

As previously said, the Jupyter server includes access to language runtimes including Python. The use of this interpreter take place in an editable and runnable

frames called **cells** (Figure 5.22). The cells can execute the text contained using the active language kernel. Outputs are subsequently displayed in the browser making current rendered web page to grow vertically. Combined with the fact that this also is a web page all the system can be accessed without leaving the current browser tab. The contents and state of these page interactions can be stored in files called *notebooks*. There are development stories as they were last left, like a last snapshot. This allows to seamlessly continue where the work had been left. The current prototype architectural high level view for key elements and possible interactions can be seen in Figure 5.5.

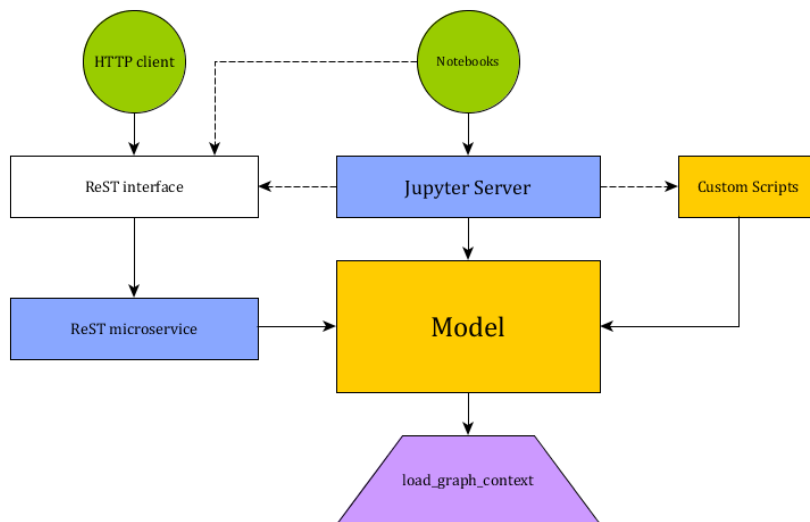
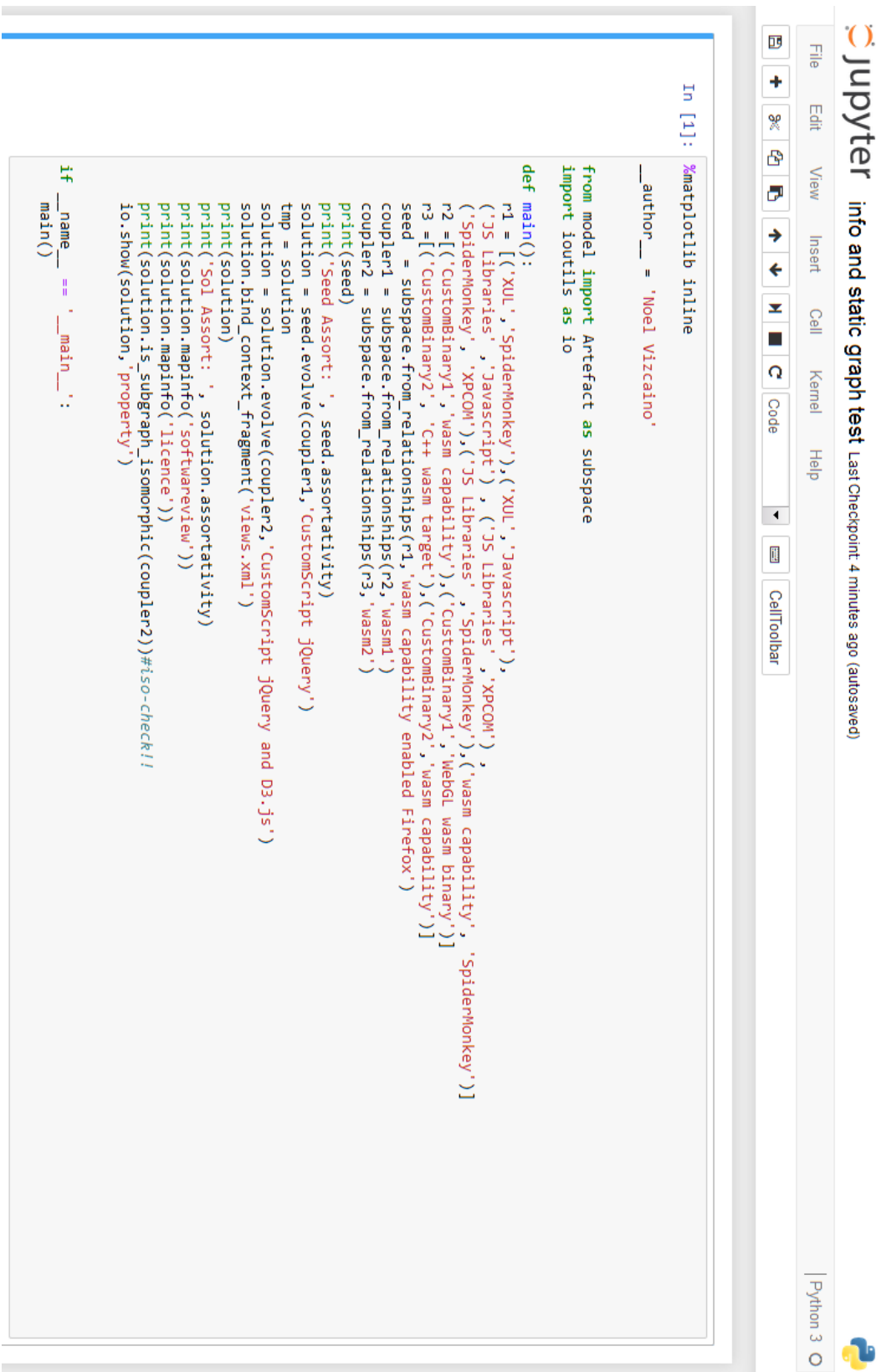


Figure 5.5: Current prototype logical access as implemented. Green circles are web clients. Blue rectangles are server related and the white one is an interface. The purple trapezoid is an I/O function. Any other I/O is omitted as it is of general access (utility). Dotted lines are optional uses.

5.3 Configuration Context files

The XML or JSON files are used to configure the model. They could be replaced by YAML files or any other convenient format. We deliberately chose them for illustration purposes. The graph if encoded in JSON can be useful for client/server new data transfer operation (thus, no conversion needed). XML schemas should be used to validate the files. One example for the current master assets XML design (file) is provided in the appendix.



The screenshot shows a Jupyter notebook interface. At the top, the Jupyter logo is on the left, and the text 'info and static graph test' is followed by 'Last Checkpoint: 4 minutes ago (autosaved)'. Below this is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', and 'Help'. On the right side of the menu bar, there are icons for file operations, a 'Code' dropdown menu, and a 'CellToolbox' button. The main area of the notebook contains a code cell with the following Python code:

```
In [1]: %matplotlib inline
__author__ = 'Noel Vizcaino'

from model import Artefact as subspace
import ioutils as io

def main():
    r1 = [('XUL', 'SpiderMonkey'), ('XUL', 'JavaScript'),
          ('JS Libraries', 'JavaScript'), ('JS Libraries', 'XPCOM'),
          ('SpiderMonkey', 'XPCOM'), ('JS Libraries', 'SpiderMonkey'), ('wasm capability', 'SpiderMonkey')]
    r2 = [('CustomBinary1', 'wasm capability'), ('CustomBinary1', 'WebGL wasm binary')]
    r3 = [('CustomBinary2', 'C++ wasm target'), ('CustomBinary2', 'wasm capability')]
    seed = subspace.from_relationships(r1, 'wasm capability enabled firefox')
    coupler1 = subspace.from_relationships(r2, 'wasm1')
    coupler2 = subspace.from_relationships(r3, 'wasm2')
    print(seed)
    print('Seed Assort: ', seed.assortativity)
    solution = seed.evolve(coupler1, 'CustomScript jQuery')
    tmp = solution
    solution = solution.evolve(coupler2, 'CustomScript jQuery and D3.js')
    solution.bind_context_fragment('views.xml')
    print(solution)
    print('Sol Assort: ', solution.assortativity)
    print(solution.mapinfo('softwareview'))
    print(solution.mapinfo('licence'))
    print(solution.is_subgraph_isomorphic(coupler2))#iso-check!!
    io.show(solution, 'property')

if __name__ == '__main__':
    main()
```

Figure 5.6: A Jupyter notebook editing cell featuring sample prototype code

5.3.1 Encoding Assets

XML files describes and configures the assets. This information will be used as additional input to the model. There are two mandatory entries expected to be in a master file, also mandatory. Any other sets of entries can go in separate files we call configuration fragments. These can be separately named and loaded.

The *property* tag and the *isonode* tag. These names could be changed later if needed. These tags and any other custom tag can be used. They belong to a semantic namespace mapped to the graph metadata dictionary namespace. These two needs to synchronised with each other.

The model implements two properties for easy access to them but changes are trivial using refactoring tools. The XML data has all the engineering information needed about the assets. Also, assets can be specified to expand the hypergraph by adding nodes we denote *isonodes*. The isonodes are **known not to break** the isomorphism relation governing the equivalence class they belong to. The importance here is to have the ability to model this as the model operation will require it. The more complete the configuration files are the more useful subgraph isomorphism and other graph analysis would be. *isonode* is also used for clarity as can be remember as short for isomorphic-node. This serves as a proof of concept ass other equivalence class expansions can be encoded similarly by using other tag and a corresponding associated graph.

Listing 3 Sample asset configuration file entry. Database technologies description in Google Datalab collection [17]

```
<asset name="Redis">
  <property>Database</property>
  <property>key-value</property>
  <property>no-sql</property>
  <isonode>PostgreSQL</isonode>
  <isonode>Cassandra</isonode>
  <isonode>MongoDB</isonode>
</asset>
```

5.3.2 Encoding Artefacts

An artefact instance carries all necessary information to evolve itself by graph expansion using composition. The choice of the working instance is the **shifting** operation. These as can be labelled as the variables they are. Artefact also carry a name and a cryptographic SHA-1 hash as ID.

To implement an artefact we use a digraph from the NetworkX library. The node IDs are the asset names. An Artefact is an instance of a valid family via isomorphism. This family encodes a subset of the Cartesian product of the assets. It is obvious that not all the pairings will be valid for a variety of reasons including sheer infeasibility or just by not being adjusted to problem constraints.

The list of items tracked per artefact instance:

- The list of couplers is stored to keep track of added couplers.
- The list of edges where evolution took place.
- The list of applied contexts.
- The list of views known to belong to the equivalence class this artefact is representative of.

These lists are ordered lists.

An Artefact base graph acts as key or ID of its equivalence class $[artefact] \simeq$. It can be persisted by saving its graph structure as a JSON file. This *graph.json* file establishes the relationship between nodes. This file can also be generated using the code stating the list of node pairs. All graph metadata contained will be written to the JSON file. It acts as a graph snapshot. This feature is also useful to make a sequence of snapshots to track hypergraph evolution.

5.4 The Model as a whole

The format chosen to encode the equivalence class expansion by node addition is to add an appropriate *<isonode>* tag entry inside the asset definition. The artefact after configuration loading can update its equivalence class. A equivalence class

Listing 4 Basic Artefact class initialisation. However we need more data for it to be usable.

```
def __init__(self, name):
    """
    Creates basic artefact with empty digraph and blanked fields.

    :param name: Name of the artefact
    """
    self.name = name
    self.id = self.generateID(self.name)
    self.graph = nx.DiGraph()
    self.couplerlist = []
    self.evolved_edges = []
    self.contexts = []
    self.views = []

def generateID(self, data):
    """
    Generates a new one way cryptographic hash using sha1 algorithm.
    The hexadecimal digest is returned as ID based on the utf8 self.name
    encoding, just for convenience.

    :param data: string to use as unicode bytes to be hashed
    :return: the hex digest
    """
    return hashlib.sha1(data.encode()).hexdigest()
```

is a list of views that must be kept updated.

In addition, any known or found sub-graph isomorphism can be added to the artefact. Any metadata information or attribute desired can be part of an isomorphic validation. These choices turn the underlying isomorphism on and off. We accept or reject nodes as part of the graph based on the metadata stored. This feature serves to explore and discover useful engineering relationships among the artefacts. These features could be really difficult to achieve without automation as the number of relationships among the artefacts views grows. Indeed, a combinatorial explosion of data will have to be analysed by algorithms of varying performance.

The equivalence class arising from the artefact views isomorphism is also connected to a boolean vector encoded as tags present in each asset configuration fragment. There is no limit in the descriptive potential of these combined configuration files.

The advantage of having these configuration files is also that the software does not need to be changed should we come with a new property or alternative or even a tag. Tags are encoded using dictionaries (associative arrays) It is implemented in a generic so adding any tag is trivial. This means that the complexity on the configuration side can easily grow without disrupting the model.

The model implements all the *Artefact* class methods operating on its instances. It should be noted these instance names are important as they label or tag the shifts done. These can be understood as the joint result of an evaluation and selection. As of this prototype those are assumed to be done by the practitioner or programmer depending on her engineering needs. However, it suffices to demonstrate further automation options feasibility. One way to help evaluate the current state is to use the *assortativity* property to assess solution evolution. This property arises from the *preferential attachment* or nodes, also called *assortative selection* This provides an *assortativity* coefficient that measures the level of nodes similarity after being selected (attached). Such selection criteria will affect the graph properties as a whole. Thus, graphs can also be classified. Moreover, evolution can be filtered using the *find_isomorphic_pairs* method to evaluate and find suitable expansions. Automation ultimately depends on how much good engineering data we have.

5.4.1 Artefact custom initialisation

The data structure to hold the data is a Digraph to acknowledge the node edges as Cartesian pairs. Therefore, we use the NetworkX *nx.Digraph()*. It is initialised in `__init__` which is not a constructor but it serves to initialise the instances. Class custom initialisation can be aided by some special type of standardised Python decorators. These use some syntactic sugar in the form of *classmethod* before the function definition. This help to initialise the instance depending on the input data. *cls* is the convention to name the class itself as parameter that needs to be

initialised itself with `__init__`.

Listing 5 Custom class initialisations. *cls* is a convention for the class to be customised

```

@classmethod
def from_relationships(cls, data, name):
    """
    To create an artefact using an assets pairs relationship list

    :param data: list of asset pairs
    :param name: name for this artefact
    :return: new artefact with a default attached context (nodes metadata)
    """
    artefact = cls(name)
    artefact.graph.add_edges_from(data)
    artefact.bind_context_fragment()
    return artefact

@classmethod
def from_graph(cls, g, name):
    """
    To create an artefact using a graph containing asset relationships

    :param g: a NetworkX graph
    :param name: name for this coupler artefact
    :return: new artefact with a default attached context (nodes metadata)
    """
    artefact = cls(name)
    artefact.graph = g
    artefact.bind_context_fragment()
    return artefact

```

The sizes of the nodes represent their corresponding the number of properties by default. Other criteria can be used. This initialises everything based on default and/or existing selected configuration files.

5.4.2 Applying Configuration Contexts

Different configuration contexts can be set. These can be sliced further in configuration fragments as multiple files. They will all be layers of metadata at runtime, being only the *master* one, with properties and isonodes, mandatory. They will **override** previous settings if there is a **tag name clash**. The method

bind_context_fragment will do just that on an instance basis. It is useful to set all information early in the evolution to be able to track the trails of graph data dynamics. The default context is the assets encoded in the master file with system path *DEFAULT_ASSETS_PATH*.

Listing 6 We keep track of the added context fragments. It overrides metadata on tag name collision. Not a bug but part of corresponding metadata update.

```
def bind_context_fragment(self, config=DEFAULT_CONTEXT_FILE):
    """
    Binds a metadata node information layer (context fragment) to
    its (base) graph node metadata

    :param config: path to node metadata file
    """
    with load_graph_context(self.graph, config) as g:
        self.graph = g
    self.contexts.append(config)
```

Listing 7 Sample asset entries from the context fragment layer *views.xml*. Tags used are *licence* and *softwareview*. Any tag can use watching out for semantic *word clash* with any other used by metadata. This means they will capture the tag or label and later override related metadata.

```
<asset name="bs4">
  <licence>MIT</licence>
  <softwareview> business logic</softwareview>
</asset>
<asset name="Kivy">
  <licence>Dual GPL</licence>
  <softwareview> UX/Presentation</softwareview>
</asset>
<asset name="jsforcegraph">
  <licence>GPLv3</licence>
  <softwareview> Presentation</softwareview>
</asset>
<asset name="jQuery">
  <licence>MIT</licence>
  <softwareview> Presentation</softwareview>
</asset>
```

The current file is *assets.xml*. Any other can be set with different names. One

option is to add practitioners contexts *assets_practitioner1.xml*. The other option is to create another file and add the entry `<owner>practitioner1</owner>` to the assets owned by the practitioner. This is how the sample *views.xml* was created. These contexts (Figure 5.7) add a level of flexibility akin to what is expected of software factories schemas. These are chosen in design time but choices during runtime to branch out or drive the evolution to new evolution solution spaces. These redirections could be programmed if some conditions are met within the current state of the evolution. This would allow for the comparison of different multiple extensible configurations.

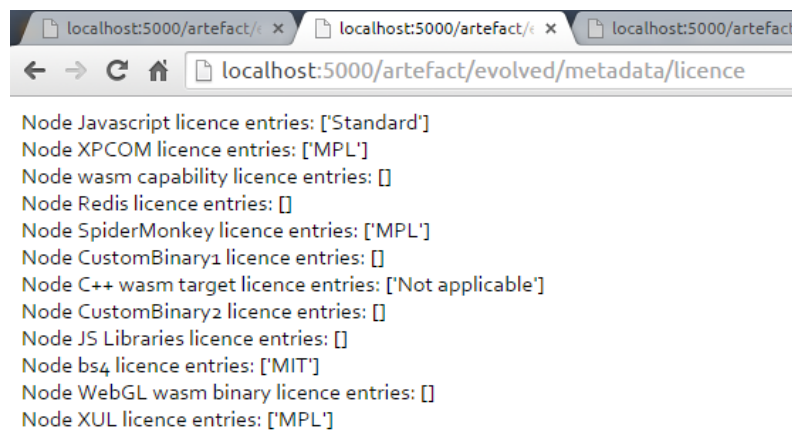


Figure 5.7: Custom asset configuration fragment layer loaded into the graph as metadata with tag (label) *licence*.

5.4.3 Instance metadata

A dictionary (associative array) is used as the basis to store and attach data to a node. It is convenient to use the node since all the dynamics with nodes will carry their data along. These attachments can be overridden by new ones. For convenience properties are created to access the dictionary related to artefact properties. This is similar in the case of the *isonode* tag.

For other custom tags a dictionary or associative array must be mapped by node. The method `_fixmiss` ensures there is no key error and blanks are applied where appropriate. This is a read operation. The modifications have to happen in the XML configuration files.

Listing 8 Python property using `@property` in lieu of getter/setters: *properties* as a dictionary based node metadata

```
@property
def properties(self):
    """
    Access to tag property node metadata as dictionary

    :return: properties dictionary, use properties[node]
    """
    ps = self.tagmap(Artifact.property_tag)
    return ps
```

Listing 9 Generic choice of node metadata using existing tags (labels)

```
def tagmap(self, tag):
    """
    Returns nodes stored tagged metadata

    :param tag: name of the metadata dictionary
    :return: a tag dictionary where nodes are the keys
    """
    assetmap = nx.get_node_attributes(self.graph, tag)
    self._fixmiss(assetmap)
    return assetmap
```

5.4.4 Configuration Contexts Automation

Since configuring enough meaningful engineering information can be a daunting task, a simple proof-of-concept example of automation has been devised.

Google Cloud Launcher has been **harvested/scrapped** into a **CSV** file to gather mock engineering data to test some of the model features. The current count of assets available is close to two hundred. They are classified into categories as also can be deduced from the string in the *href* attribute of their links. The assets are to be classified into meaningful categories that we will use to build a relation. In this relation we assume the elements belonging to a category are perfectly interchangeable from an engineering point of view. The categories that are too generic are discarded and thus the subset is reduced. An additional reduction happens as we filter further for undesired or redundant items containing the string "Standard". We proceed to write an XML file *googlelauncher_layer.xml* containing the chosen subset of assets encoded as isonode entries in the relevant assets. The

other assets will not contain the entries. The idea is to illustrate how a **subset of the Cartesian product** of the assets can be expressed by a virtual relation. This relation can be labelled and in our case the label is *subspace.expansion_relation_tag*. This label is currently "isonode" because it is short and easy to remember as isomorphic-node. Similar semantics can be invented if new relations are discovered. In fact, there could be many others. We recycle the label as the tag we will use in a XML configuration file. This tag will be the name of the dictionary (associative array) containing corresponding node metadata (Figure 5.8).


Listing 10 Creating a subset of the Cartesian product of assets in a *xml.etree.ElementTree* tree data structure. It is called inside of a loop with various categories. The loop should be moved inside for clarity and efficiency.

```
# LISTING NOTE1: Using from model import Artefact as subspace
# LISTING NOTE2: XML_RELATION_TAG = subspace.expansion_relation_tag defined as
# "isonode" for the whole Artefact class.
def relation2tag(root,category, xfilter):
    """
    Cartesian product of the asset set with itself using valid categories.
    Other more meaningful relational subsets could be created.

    :param root: root of the xml tree
    :param category: a valid category
    :param xfilter: a undesired string
    :return: amended xml tree
    """
    for assetA in root.iter('asset'):
        nameA = assetA.get('name')
        catA = assetA.get('category')
        for assetB in root.iter('asset'):
            nameB = assetB.get('name')
            catB = assetB.get('category')
            if catA in category and catB in category:
                if nameA!=nameB and xfilter not in nameA and xfilter not in nameB:
                    isonode = customTree.SubElement(assetA, XML_RELATION_TAG)
                    isonode.text= nameB
    return root
```

This is an example of how to automate the creation of complex asset entries. These will belong to custom layers of various configuration context fragments. These will be later applied (not necessarily in sequence) to build rich metadata

payloads into the nodes of all the artefact views. This view persists in the form of a JSON file. Different views may have different metadata and that is the key for graph isomorphism discrimination.



```

-----[Artifact: bs4xexpansion <- bs4coupler]-----
Node SpiderMonkey property entries: [JavaScript engine, 'asmjs', 'wasm', 'language:JavaScript', 'Mozilla application framework']
Node C++ wasm target property entries: [C++ libraries, 'NO filesystem IO', 'fastest', 'load-time-efficient']
Node XUL property entries: [GUI, 'web', 'XML', 'cross platform', 'Mozilla application framework']
Node WebGL.wasm binary property entries: []
Node JavaScript property entries: [language, 'imperative', 'Object Oriented', 'text processing', 'web']
Node CustomBinary2 property entries: []
Node CustomBinary1 property entries: []
Node Redis property entries: [Database, 'key-value', 'no-sql']
Node bs4 property entries: [HTML processing, 'HTML error tolerance', 'language:python', 'text processing', 'web']
Node XPCOM property entries: [business layer, 'component model', 'enable:plugins', 'text processing', 'cross platform', 'Mozilla application framework']
Node JS Libraries property entries: []
Node wasm capability property entries: [Node SpiderMonkey isonode entries: ['V8', 'Chakra]]
Node C++ wasm target isonode entries: []
Node XUL isonode entries: ['WinForms', 'Swing', 'wxWidgets', 'Kivy', 'QT']
Node WebGL.wasm binary isonode entries: []
Node JavaScript isonode entries: ['Python', 'TypeScript']
Node CustomBinary2 isonode entries: []
Node CustomBinary1 isonode entries: []
Node Redis isonode entries: ['Cassandra', 'EDB Postgres Enterprise', 'Aerospike', 'MySQL', 'MongoDB', 'PostgreSQL', 'Cassandra', 'Percona', 'MongoDB Multi-VM', 'CouchDB', 'DataStax Enterprise', 'ClearDB']
Node bs4 isonode entries: []
Node XPCOM isonode entries: [DCOM, '.NET', 'BONOBO]
Node JS Libraries isonode entries: []
Node wasm capability isonode entries: []
Context sequence: [assets.xml, 'views.xml', '\\GoogleLauncher\\googlelauncher_layer.xml']
Couplers so far: [Rediscoupler, bs4coupler, bs4coupler]
Evolution edges: [(CustomBinary1, 'bs4'), (CustomBinary1, 'Redis')]

```

Figure 5.8: Google Cloud Launcher asset configuration fragment loaded into the graph as metadata.

5.4.5 Evolution operations Δ

Once the initial setup is done, we can start evolution steps. The coupler graph can be interpreted as an artefact instance since it is a software solution too.

The coupler graph has to include the node where to attach. In other words, it is **important** to note that the common *enabler* node must be explicitly included in the coupler as well as in the seed artefact. The program will deduct anything else related to this step operation henceforth. In this fashion we keep data as just data without forcing the user to identify such *enabler*. This has useful consequence as we edge to a more functional programming philosophy in regard to data processing.

The key operation *evolve* will create a new Artefact based on an existing artefact using a coupler graph. This is an evolution step as part of an intelligent aggregation strategy. This works in conjunction with the existing artefact information. All relevant further and derived data will be updated or generated accordingly. This means that the knowledge of the suitability of the evolution relies on the coupler existence whereas the viability of the solution as a whole relies on the starting seed solution.

5.4.6 Evaluating and tracking desired properties

Looking at the generated family it may be obvious which solution has the most desired properties. This can be achieved by using a desired tag to describe the property sought as the **node size**. It can be accomplished, for instance, by adding the entry `<predictor-tag>data-of-interest</predictor-tag>` to the relevant asset configuration file. Of course, visually this works quite well with a small example. If the asset data were large enough it may not have been obvious. It may very well not be advisable to try to visualise it as computer resources may not be enough. However, a table and corresponding chart can be generated to discover desired viable candidates (Figure 5.9).

5.4.7 Searching for other isomorphic solution views

To check if there is no sub-graph isomorphism in the first place to avoid a pointless search refinements the function *is_subgraph_isomorphic* is provided. This can

```

In [1]:
__author__ = 'Noel Vizcaino'

from model import Artefact as subspace
import itertools as io

def main():
    r1 = [('XUL', 'SpiderMonkey'), ('XUL', 'JavaScript'),
          ('JS Libraries', 'JavaScript'), ('JS Libraries', 'XPCOM'),
          ('SpiderMonkey', 'XPCOM'), ('JS Libraries', 'SpiderMonkey'), ('wasm capability', 'SpiderMonkey')]
    r2 = [('CustomBinary1', 'wasm capability'), ('CustomBinary1', 'WebGL wasm binary')]
    r3 = [('CustomBinary2', 'C++ wasm target'), ('CustomBinary2', 'wasm capability')]
    seed = subspace.from_relationships(r1, 'wasm capability enabled Firefox')
    coupler1 = subspace.from_relationships(r2, 'wasm1')
    coupler2 = subspace.from_relationships(r3, 'wasm2')
    solution = seed.evolve(coupler1, 'CustomScript jQuery')
    solution = solution.evolve(coupler2, 'CustomScript jQuery and D3.js')
    solution.bind_context_fragment('views.xml1')
    #io.print_d3_data(solution)
    solution.update_views()
    for view in solution.views:
        print('{} Property Count: {}'.format(view.name, sum(view.count('property'))))
    #io.show_multiple_artefacts(solution.views, 'property')
    if __name__ == '__main__':
        main()

View {'XPCOM' by 'DCOM'} Property Count: 31
View {'XPCOM' by '.NET'} Property Count: 36
View {'XPCOM' by 'BONOBO'} Property Count: 36
View {'JavaScript' by 'Python', 'XPCOM' by 'BONOBO'} Property Count: 36
View {'JavaScript' by 'TypeScript', 'XPCOM' by 'BONOBO'} Property Count: 36
View {'JavaScript' by 'TypeScript', 'SpiderMonkey' by 'V8', 'XPCOM' by 'BONOBO'} Property Count: 36
View {'JavaScript' by 'TypeScript', 'SpiderMonkey' by 'Chakra', 'XPCOM' by 'BONOBO'} Property Count: 36
View {'JavaScript' by 'TypeScript', 'SpiderMonkey' by 'Chakra', 'XPCOM' by 'BONOBO', 'XUL' by 'WinForms'} Property Count:
36
View {'JavaScript' by 'TypeScript', 'SpiderMonkey' by 'Chakra', 'XPCOM' by 'BONOBO', 'XUL' by 'Swing'} Property Count: 36
View {'JavaScript' by 'TypeScript', 'SpiderMonkey' by 'Chakra', 'XPCOM' by 'BONOBO', 'XUL' by 'wxWidgets'} Property Coun
t: 36
View {'JavaScript' by 'TypeScript', 'SpiderMonkey' by 'Chakra', 'XPCOM' by 'BONOBO', 'XUL' by 'Kivy'} Property Count: 33
View {'JavaScript' by 'TypeScript', 'SpiderMonkey' by 'Chakra', 'XPCOM' by 'BONOBO', 'XUL' by 'QT'} Property Count: 36
CustomScript jQuery and D3.js Property Count: 36

```

Figure 5.9: Prototype notebook outputting current count of metadata labelled *property*.

Listing 11 Evolution by graph expansion (thin Δ) using a **one** base graph (a key view in itself) and a coupler artefact instance.

```
def evolve(self, coupler, name):
    """
    Expand, with coupler graph, current base graph carrying existing
    configurations.
    This is operator (thin)  $\Delta$  over the representative view of
     $[instance\ graph]_{\sim}$  where
    the equivalence class evolves but implicitly.

    :param coupler: the artefact with the graph to attach
    :param name: a name for the resulting artefact
    :return: expanded solution view
    """
    self.name = name
    self.id = self.generateID(self.name)
    common = set(coupler.graph) & set(self.graph)
    # print('common:', common)
    self.graph = nx.compose(self.graph, coupler.graph)
    for node in common:
        for edge in coupler.graph.edges():
            if node in edge:
                self.evolved_edges.append(edge)
    self.couplerlist.append(coupler)
    return self
```

also be used to validate the growth chain states created by any Δ 's growth steps. When we talk about isomorphism, we always refer to node based isomorphism. This is also the point of view used in the *NetworkX* library implementation. Thus, other point of view, referring to edges, will have to be implemented differently.

As featured in the "*sub-graph isomorphism search.ipynb*" notebook extra views can be discriminated and found. There can be many possible sub-graph isomorphisms. To discriminate them we can use loaded node metadata as the initial filtering criteria. This will only work if there such sub-graph isomorphism. The output is an iterator pointing to dictionary data of the found isomorphism. Data are key-value pairs, featured node by node (Figure 5.10).

```
In [1]:
__author__ = 'Noel Vizcaino'

from model import Artefact as subspace
import itertools as io

def main():
    r1 = [('XUL', 'SpiderMonkey'), ('XUL', 'Javascript'),
          ('JS Libraries', 'Javascript'), ('JS Libraries', 'XPCOM'),
          ('SpiderMonkey', 'XPCOM'), ('JS Libraries', 'SpiderMonkey'), ('wasm capability', 'SpiderMonkey')]
    r2 = [('CustomBinary1', 'wasm capability'), ('CustomBinary1', 'WebGL wasm binary')]
    r3 = [('CustomBinary2', 'C++ wasm target'), ('CustomBinary2', 'wasm capability')]
    seed = subspace.from_relationships(r1, 'wasm capability enabled Firefox')
    coupler1 = subspace.from_relationships(r2, 'wasm1')
    coupler2 = subspace.from_relationships(r3, 'wasm2')
    solution = seed.evolve(coupler1, 'CustomScript jQuery')
    solution = solution.evolve(coupler2, 'CustomScript jQuery and D3.js')
    solution.bind_context_fragment('views.xml')
    #io.print_d3_data(solution)
    solution.update_views()
    #for view in solution.views:
    #    print('{} Property Count: {}'.format(view.name, sum(view.count('property'))))
    #io.show_multiple_artefacts(solution.views, 'property')
    for iso in solution.find_isomorphic_pairs(seed, 'isnode', 'Firefox'):
        print(iso)

if __name__ == '__main__':
    main()

{'XUL': 'XUL', 'C++ wasm target': 'C++ wasm target', 'CustomBinary1': 'CustomBinary1', 'CustomBinary2': 'CustomBinary2', 'JS Li
braries': 'JS Libraries', 'SpiderMonkey': 'SpiderMonkey', 'WebGL wasm binary': 'WebGL wasm binary', 'XPCOM': 'XPCOM', 'wasm cap
ability': 'wasm capability', 'Javascript': 'Javascript'}
{'XUL': 'XUL', 'C++ wasm target': 'WebGL wasm binary', 'CustomBinary1': 'CustomBinary2', 'CustomBinary2': 'CustomBinary1', 'JS
Libraries': 'JS Libraries', 'SpiderMonkey': 'SpiderMonkey', 'WebGL wasm binary': 'C++ wasm target', 'XPCOM': 'XPCOM', 'wasm cap
ability': 'wasm capability', 'Javascript': 'Javascript'}
```

Figure 5.10: Prototype notebook outputting current pre-filtered valid isomorphisms as dictionaries.

Listing 12 Evolution by multiple graph expansion (thick Δ) using all stored views and a coupler artefact instance. All resulting solution guaranteed to be analogous. Their metadata fingerprint could be different.

```
def evolve_delta(self, coupler, name):
    """
    Explicitly evolve the known views belonging to
    :math:[instance graph]_{\|simeq} currently known.
    A trunk :math:\Delta operation.

    :param coupler: the artefact with the graph to attach to all views
    :param name: a name for the resulting artefact
    :return: expanded equivalence class
    """
    self.update_views()
    for index, view in enumerate(self.views):
        self.name=name
        self.views[index].id = self.generateID(self.name)
        self.views[index]=view.evolve(coupler, view.name+' <- '+coupler.name)
        self.views[index].couplerlist.append(coupler)
        self.views[index].evolved_edges=list(set(self.evolved_edges))
    return self
```

Listing 13 Tagged metadata occurrence count. Also useful for solution view assessment and **sorting**.

```
def count(self, tag):
    """
    Counts and returns node metadata instances as id by an existing tag context

    :param tag: an existing(initialised) tag context
    :return: metadata instances or entries count
    """
    return [len(self.tagmap(tag)[n]) for n in self.graph]
    ...
    #LISTING NOTE: This code below would print property count per view.
    for view in solution.views:
        print('{} Property Count: {}'.format(
            view.name, sum(view.count('property'))))
```

Listing 14 Checking for sub-graph isomorphism existence.

```
def is_subgraph_isomorphic(self, subartefact):
    """
    Finds if current artefact itself contains ANY other isomorphic artefact

    :param subartefact: sub-artefact with a relevant sub-graph to check with
    :return: True if found.
    """
    iso = isomorphism.DiGraphMatcher(self.graph, subartefact.graph)
    return iso.subgraph_is_isomorphic()
```

Listing 15 Using loaded node metadata to filter and find existing isomorphisms.

```
def find_isomorphic_pairs(self, subartefact, tags, values):
    """
    Check if current artefact contains other isomorphic sub-artefact considering
    metadata information as a filter.
    And provides the isomorphic result of the filter.

    :param subartefact: sub-artefact with a relevant sub-graph to check with
    :param tags: affected node metadata context tags list
    :param values: corresponding node metadata context actual datum list
    :return: Returns generator of dictionaries with found isomorphic pairs
    """

    iso = isomorphism.DiGraphMatcher(self.graph, subartefact.graph,
                                     node_match=
                                     isomorphism.categorical_node_match(tags,
                                                                           values))

    return iso.isomorphisms_iter()
```

5.5 I/O and Visualisation

As previously said, in python modules can be understood as a singleton class. It is convenient to add all I/O supporting methods in a single module, as a utility module. They can be disk access or console based. The basic information about an artefact is implemented as a `__str__` method so artefact instances know how to pretty-print themselves using the standard `print()` function. This is also helpful to output to any text accepting system, like a web browser.

List of implemented operations, briefly:

- JSON read and write
- Save PNG snapshot of current graph
- Show a static graph view
- Load all known assets data into a graph.
- Print adjacency matrix to console.
- Several other print to stdout and data output functions.
- Convenient output for extra graph generation to be used in D3.js based component offline and online by using the microservice jinja templates.
- Save JavaScript custom data output files imported offline (superseded by the microservice facilities).

5.5.1 Displaying static graphs

This graph visualisation is done using *Matplotlib* [57] (Figure 5.11). The inconvenience here is the lack of interactivity. Able to generate rich graphical output from xml and json description files (customisation). All these features had to be feasible not having to change the python source code every time.

5.5.1.1 Display of $[view] \simeq$ static graphs

The method *explore_isonodes* generates the known equivalence class based on currently applied configuration data. An *isonodes_graph* generates the partial hyper-graph for the current data using sets as destination nodes as a curiosity implemented to gain insights (Figure 5.12).

The viewing of a hyper-graph is a challenge in itself. It is out the scope of the project but it is useful to consider it. That is why *solutions views* sets were chosen. The image produced corresponds to all the artefacts views as viable candidates that can be generated based on the information provided. Each view is a view of the validated slice of the combinatorial explosion derived from the asset relationships. This could be huge as the assets number grows. Any edge could actually be a code dependency, a functional feature, or any other meaningful engineering information. This choice needs to be consistent across the equivalence class.

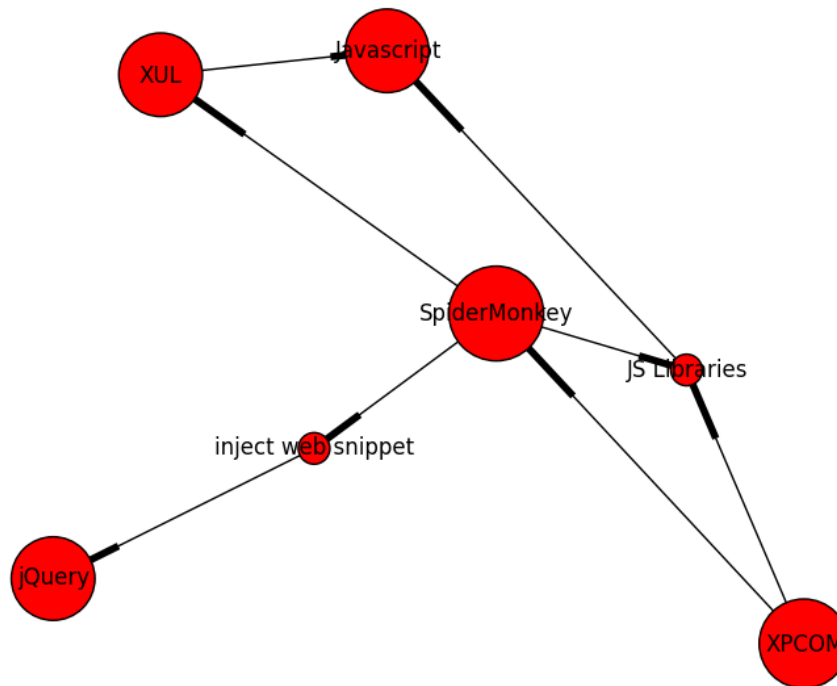


Figure 5.11: Sample static graph output using matplotlib library

5.5.2 Displaying dynamic graphs

Client side dynamic interactive charts uses a D3.js based solution [38] (Figure 5.13). However, we chose to leverage the power of an existing GPLv3 component customisation by Michel Bostock for the quick generation of web based graph dynamic output [19]. This feature is convenient to study the outputs and to gain insights on any extra information. The great advantage is that presentation customisation resides at the appropriate layer (JavaScript and CSS level). Also, this presentation model can be reused by any web based client.

D3.js allows for declarative style programming to be used similar to jQuery [51]. This also includes cascade methods. For the purpose of demonstrating capabilities a JavaScript file with the data is produced with the necessary code to be imported. This allows to produce a dynamic graph perfect for screen capturing. The D3 based GPLv3 licensed custom component, by Michel Bostock is used for as a Jinja2 [50] based template for all dynamic graphs [19].

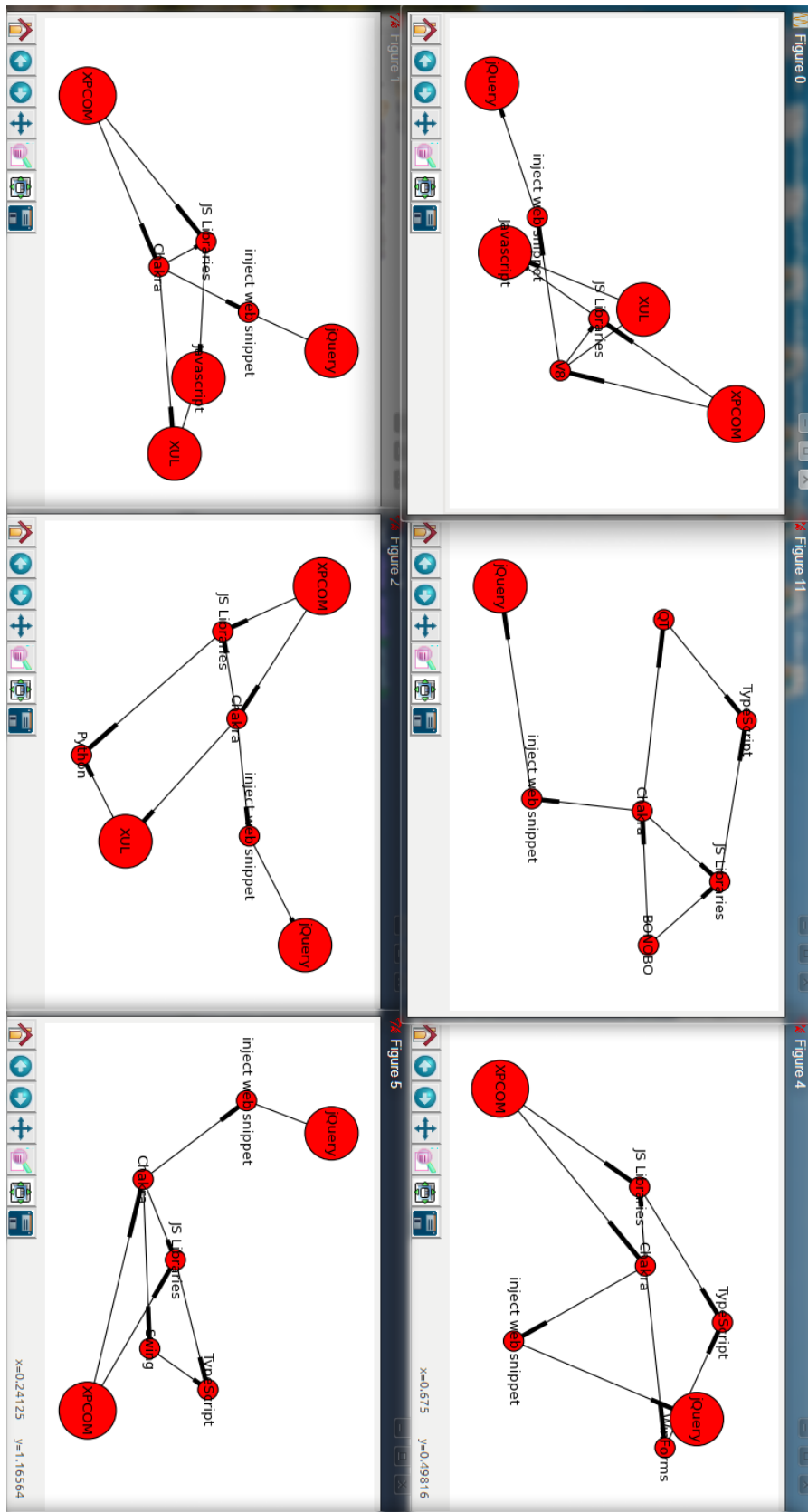


Figure 5.12: Isomorphic solution tile of some sample static views as generated and displayed.

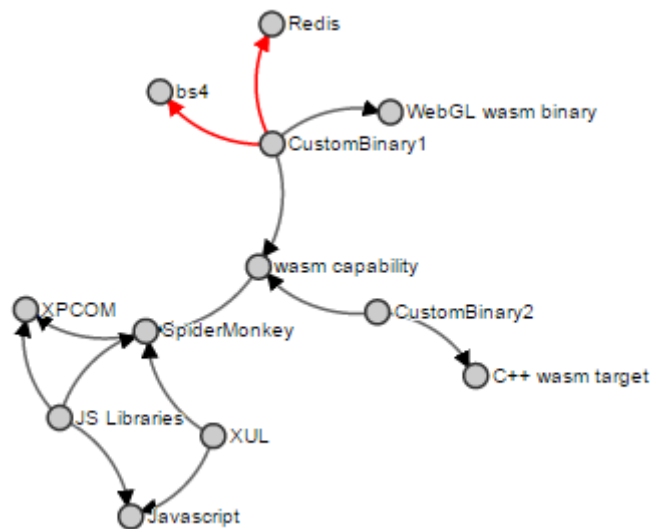


Figure 5.13: Sample output using D3.js based GPL component [18, 19]

5.5.2.1 Display of $[view] \simeq$ dynamic graphs

The evolution steps can be automated by code or interactively. The D3.js chart is embedded (Figure 5.14) in a Jupyter notebook [49, 54] along with other standard facilities. In this fashion the evolution is self-documented and it is also of great use for presentations and for non-practitioners of all levels to explore the possibilities.

5.6 Modelling: Examples of scripting use

Prototype access

Aided by the vast richness that can be added through configuration files the model can be used programmatically. This can be done through the various means be either anything with access to a Python 3 interpreter with adequate module dependencies met. There are several caveats with the current prototype. A minor one is that the microserver cannot be started within Jupyter notebook cell (Figure 5.22) but can be launched separately. For Jupiter to be able to access the modules the current working path must be the same. Starting in the same path is a solution that will suffice here but for a bigger project other arrangements may be needed. The model records evolution steps but does not persist them (to files). The model is efficient as persist just the graphs. I realised while

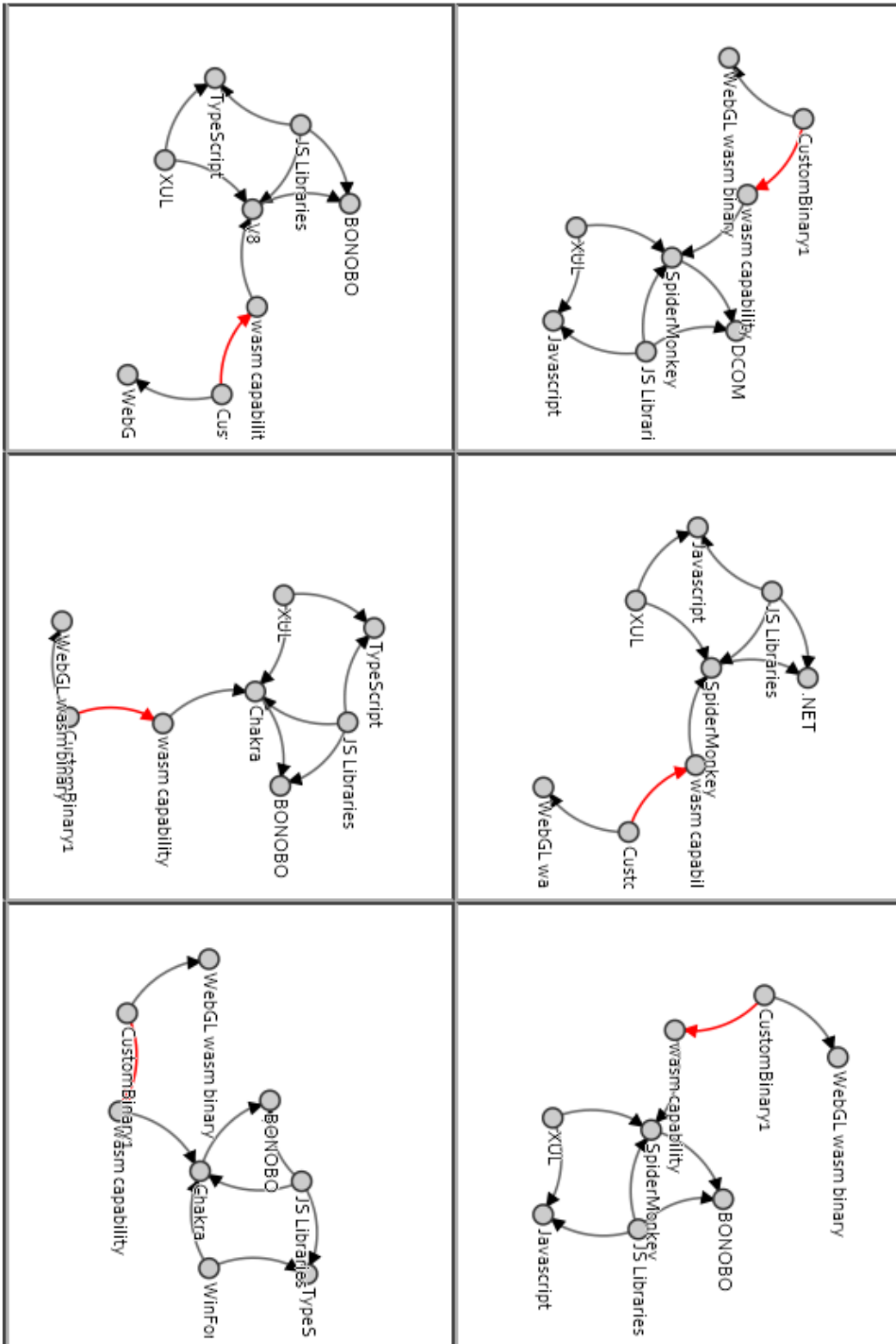


Figure 5.14: Generated compact tiled-view of expanded $[view] \approx$ specimens (other views). It can be loaded within a Jupyter notebook or via a web browser. Jinja2 template version *iframe* sizes are bigger so there is more room for each graphs.

loading specimens that the loaded one showed no evolution. This is correct as no evolution took place from the current script point of view. I could not foresee all the emerging future use cases as new functionality was added and combined.

The created scenario within the rest microservice remembers the evolution steps because it just took place (live with the system). Albeit anecdotal is worth pointing this as it may be confusing but ultimately makes sense. The red arrows therefore represent any last evolution steps that have taken place over the artefact being displayed. Only the nodes affected will get the arrows.

The meaning of the direction of the arrows represent the fact that they model Cartesian pairs. Sometimes the dependence is mutual and a frame of reference is needed. A solution makes both seed and coupler depend on an enabler node. That is because the whole solution depends on this node to be assembled. Another point of view could be that the node depends on other node to be a functioning node. This is akin to a software dependency or architectural dependency. Edges are not labelled but it could be interesting to visualise the order of the evolution steps as they are stored and tracked along with a history of added couplers.

5.6.1 Single asset evolution

Browsing Augmentation based Artefact

The example of a website post-processing script using Greasemonkey and Firefox (basic notebook example in Figure A.3). This concept is known as augmented browsing [153] [154]. This script manipulates the DOM tree, post-rendering stage, in the browser. If the data does not come from the originating URL a browser error is triggered. This is known as Cross Site Scripting (XSS) and it is currently forbidden by browsers as a security measure. This post-processing scripts use a relaxed security sandbox which allows XSS among other features. The utility of these scripts (or extensions) is remarkable (including interdisciplinary uses) as illustrated by Pafilis et al. [138].

GreaseMonkey *enables* Firefox to accept custom scripts. The relation $a R b$ applied create a graph expansion that is isomorphic to other allowed solutions. In this case is *browser R augmented_browser_enabler*. The browser could be actually

enabled without the need for external software but we can model such feature as independent. We consider a collapsed or expanded node depending on what we actually want to put the emphasis on. Equivalence class: augmented browser, Firefox compatible $[FX]_{\simeq}$. As a side note, since browsers share increasingly this feature and couplers (i.e. the extensions) the higher more abstract equivalence adds all those views as belonging to that family of solutions.

Listing 16 Defining asset names as node identifiers. These relationships model valid Cartesian pairs to form a relation.

```
r1 = [('XUL', 'SpiderMonkey'), ('XUL', 'Javascript'),
      ('JS Libraries', 'Javascript'), ('JS Libraries', 'XPCOM'),
      ('SpiderMonkey', 'XPCOM'), ('JS Libraries', 'SpiderMonkey')]
r2 = [('SpiderMonkey', 'GreaseMonkey'), ('CustomScript', 'GreaseMonkey'),
      ('CustomScript', 'jQuery')]
r3 = [('CustomScript', 'D3.js'), ('CustomScript', 'GreaseMonkey')]
```

Listing 17 Load JSON graph data to create an Artefact instance. This replaces the need to define node relationships as Cartesian pairs.

```
with io.load_graph_context(io.load_json('graphdata.json')) as g:
    test = model.Artefact.from_graph(g, 'test')
```

Listing 18 Using the evolve method (thin Δ) in succession.

```
solution = seed.evolve(coupler1, 'CustomScript jQuery')
evolved_solution = solution.evolve(coupler2, 'CustomScript jQuery and D3.js')
```

The variable names used by the artefact instances act as the labels of the artefacts chosen and name *shifts* if used. The nodes that will serve to attach the graphs needs to be mutually shared for the mechanism to work. This feature simplifies greatly the design of operation as we eliminate redundant parameters. It is also interesting to keep related data as compact as possible. In this manner, the artefact instances and their views always carry the necessary information to be expanded and thus, evolved.

Cloud Solution based Artefact

Generation of a cloud solution from Google Cloud Platform solutions as Assets. All the solutions present in the Google Cloud Launcher are considered. A custom

script for augmented browsing using the Redis [155] database and Flask [41] as web server could be one simple example. The artefact links to the previous browser based artefact with Firefox as the common node (Figure 5.15).

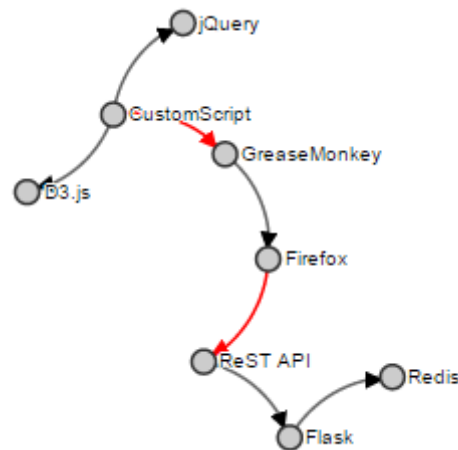


Figure 5.15: Google Cloud Platform sample Artefact

Listing 19 Google Cloud Platform sample artefact modelling.

```

r3 = [('Firefox', 'ReST API'), ('ReST API', 'Flask'), ('Flask', 'Redis')]
r1 = [('GreaseMonkey', 'Firefox')]
r2 = [('CustomScript', 'jQuery'), ('CustomScript', 'D3.js'), ('CustomScript',
'GreaseMonkey')]
seed = model.Artefact.from_relationships(r1, 'GreaseMonkey enabled Firefox')
coupler1 = model.Artefact.from_relationships(r2, 'GreaseMonkey Script')
coupler2 = model.Artefact.from_relationships(r3, 'Cloud')
solution = seed.evolve(coupler1, 'CustomScript')
solution = solution.evolve(coupler2, 'Evolved with Cloud')

```

JavaScript file output

JavaScript file output sample generated by the prototype for seamless experimentation. Important to note that *dependency* and *coupler_added* are entities named by convention as any other name could have been used. *dependency* could be a logical dependency, architecture, view or any other relevant term. The prototype could be upgraded visualise different edges meaning. The type can be fully customised. *dependency* here means any logical dependency and it could coincidentally mutual depending on the point of view.

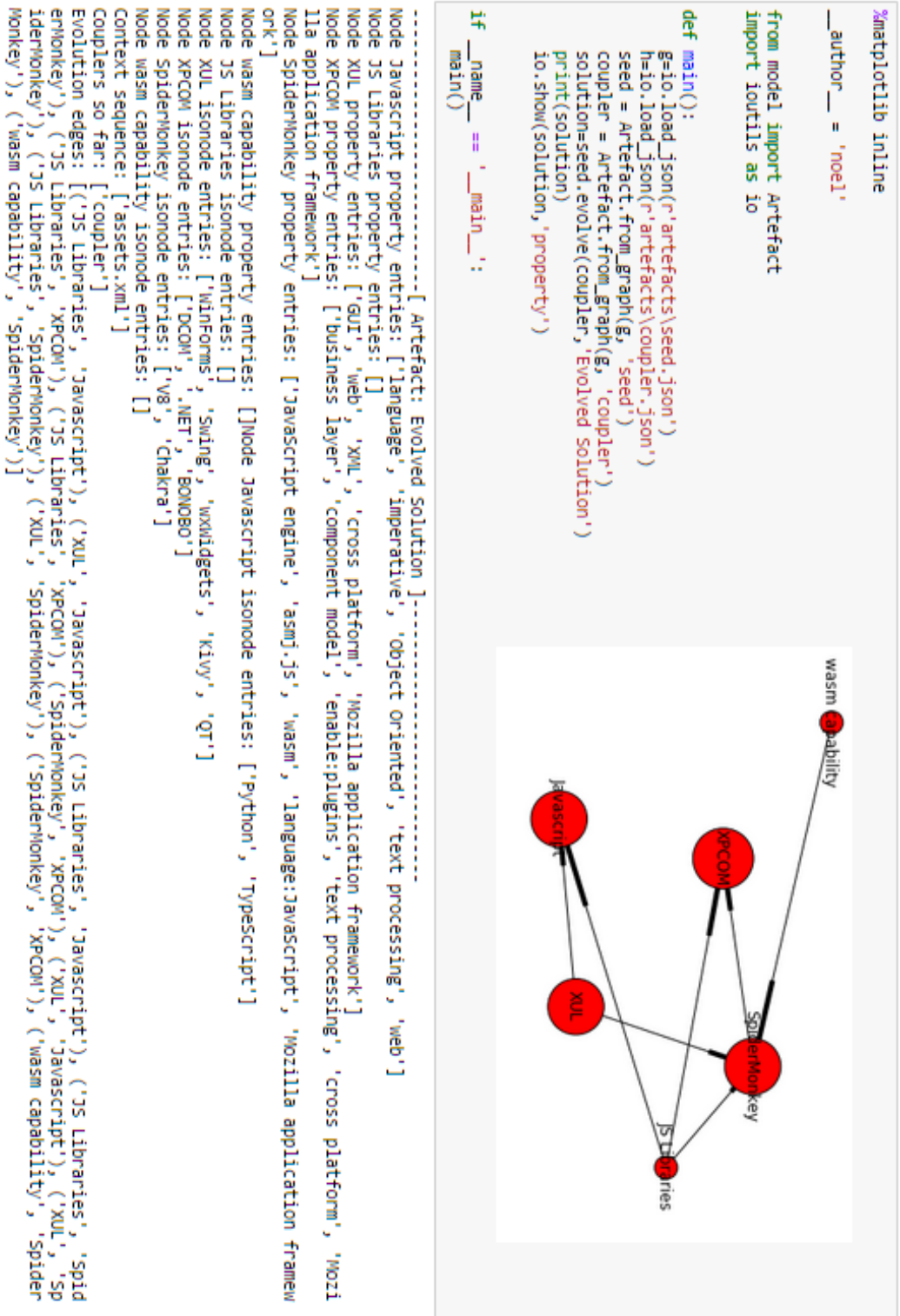


Figure 5.16: Notebook featuring single asset source code for loading of JSON artefacts. Graph output depicted over.

Listing 20 Reformatted data as JavaScript (initially used for testing)

```

var links = [{source: "CustomScript" , target:"jQuery", type: "dependency"},
{source: "CustomScript" , target:"GreaseMonkey", type: "coupler_added"},
{source: "CustomScript" , target:"D3.js", type: "dependency"},
{source: "Flask" , target:"Redis", type: "dependency"},
{source: "Firefox" , target:"ReST API", type: "coupler_added"},
{source: "ReST API" , target:"Flask", type: "dependency"},
{source: "GreaseMonkey" , target:"Firefox", type: "dependency"}];

```

5.6.2 Multiple assets evolution

When multiple assets belonging to solutions or couplers with different compatibilities we can perform a multiple evolve step (Figure 5.17 and Jupiter cell in Figure 5.18). This will create a family of solutions on its own. The previous example for the model chapter features a double coupler expansion.

We model same Firefox scenario but we assume it will be wasm capable. Using WebAssembly binaries (wasm) we create two couplers. One coupler depends on WebGL capabilities and the other is produced by compiling the C++ into wasm. This is isomorphic with wasm capable solutions but to be isomorphic with both (the expansion) it needs to have access to the same space of possibilities. WebGL and the used C++ framework, simultaneously. Sample modelling below.

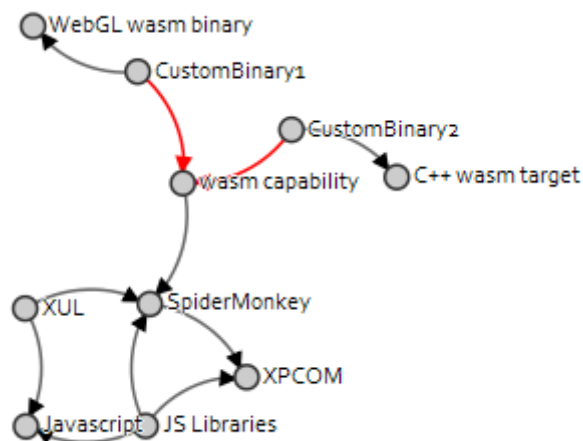


Figure 5.17: Adding two couplers using assets such they require further isomorphic relationship.

Listing 21 Google Cloud Platform sample multiple asset based artefact generation.

```

r1 = [('XUL', 'SpiderMonkey'), ('XUL', 'Javascript'),
      ('JS Libraries', 'Javascript'), ('JS Libraries', 'XPCOM'),
      ('SpiderMonkey', 'XPCOM'), ('JS Libraries', 'SpiderMonkey')]
r2 = [('wasm capability', 'SpiderMonkey'), ('CustomBinary1', 'wasm capability'),
      ('CustomBinary1', 'WebGL wasm binary')]
r3 = [('CustomBinary2', 'C++ wasm target'), ('CustomBinary2', 'wasm capability')]
seed = Artefact.create_from_relationships(r1, 'wasm capability enabled Firefox')
coupler1 = Artefact.create_from_relationships(r2, 'wasm1')
coupler2 = Artefact.create_from_relationships(r3, 'wasm2')
solution = seed.evolve(coupler1, 'wasm WebGL capable program')
solution = solution.evolve(coupler2, 'wasm targeted C++ app')

```

In practice this can be automated by using *evolve_delta* if done over the same *[view]* \simeq . If there are various different couplers in a list, we should loop through the list. Then we can call *evolve* or *evolve_delta* depending if one view or all the views bound by the same \simeq are affected. In practice the only first one is needed unless we want to hold all metadata in memory to do isomorphism related graph discriminating queries. Demonstrating such queries will require a significant amount of valid test data. The underlying mechanism is already implemented by the NetworkX library using the VF2 algorithm [72].

5.7 ReSTful microservice

There is a simple cloud microservice implemented to illustrate the convenience of the model to be implemented in a client-server context. It has been designed following the ReST architectural style. It is absolutely experimental but the proof of concept fits the model prototype design. It also serves to test the model and provides an alternative means to output text rather than just stdout (console). Besides being possible to test it over the network is also a useful cross-platform solution.

The microservice offers the chance to test the model using the browser as a feedback channel. This made basic testing seamless. The implementation was an ad-hoc reuse of existing functions. Although functional, it can be improved. Fortunately, the basic foundations have been laid to show how well the model

```
In [1]: %matplotlib inline

__author__ = 'Noel Vizcaino'

from model import Artefact as subspace
import ioutils as io

def main():
    r1 = [('XUL', 'SpiderMonkey'), ('XUL', 'Javascript'),
          ('JS Libraries', 'Javascript'), ('JS Libraries', 'XPCOM'),
          ('SpiderMonkey', 'XPCOM'), ('JS Libraries', 'SpiderMonkey'), ('wasm capability', 'SpiderMonkey')]
    r2 = [('CustomBinary1', 'wasm capability'), ('CustomBinary1', 'WebGL wasm binary')]
    r3 = [('CustomBinary2', 'C++ wasm target'), ('CustomBinary2', 'wasm capability')]
    seed = subspace.from_relationships(r1, 'wasm capability enabled Firefox')
    coupler1 = subspace.from_relationships(r2, 'wasm1')
    coupler2 = subspace.from_relationships(r3, 'wasm2')
    solution = seed.evolve(coupler1, 'CustomScript jQuery')
    solution = solution.evolve(coupler2, 'CustomScript jQuery and D3.js')
    solution.bind_context_fragment('views.xml')
    io.print_d3_data(solution)
    solution.update_views()
    io.show_multiple_artefacts(solution.views, 'property')

if __name__ == '__main__':
    main()

var links = [{source: "CustomBinary1" , target:"WebGL wasm binary", type: "dependency"},
{source: "CustomBinary1" , target:"wasm capability", type: "coupler_added"},
{source: "CustomBinary2" , target:"wasm capability", type: "coupler_added"},
{source: "CustomBinary2" , target:"C++ wasm target", type: "dependency"},
{source: "SpiderMonkey" , target:"XPCOM", type: "dependency"},
{source: "JS Libraries" , target:"XPCOM", type: "dependency"},
{source: "JS Libraries" , target:"SpiderMonkey", type: "dependency"},
{source: "JS Libraries" , target:"Javascript", type: "dependency"},
{source: "XUL" , target:"Javascript", type: "dependency"},
{source: "XUL" , target:"SpiderMonkey", type: "dependency"},
{source: "wasm capability" , target:"SpiderMonkey", type: "dependency"}];
```

Figure 5.18: Multiple asset notebook example.

design will fit within the rest style architecture. The model behaves like a generator of state machines modelling a protocol. HTTP is precisely a communication protocol. The operands are resources and the operations make use of the web verbs creating or manipulating the resources. The implementation should make this line of analysis clear. It should be noted that is possible to implement a ReST style architecture not using The Web.

Table 5.3: The model as a ReST style compatible design

	Operators	
	<i>Operands</i>	<i>nodes</i>
The Web as ReST architecture [48]	HTTP verbs POST, PUT, DELETE mapping CRUD operations	Resources, reachable by HTTP verb GET using URIs (URLs)
Our model	Solution view shift, coupler view shift, thin Δ , thick Δ , transmitted using same HTTP verbs (It could also be done using JSON data)	Artefacts and their related resources reachable by URL

5.7.1 Selectable Test scenario (sandbox function)

There is normal operation or operation based on a test pre-made scenario (Figure 5.19). Switching with:

- `http://127.0.0.1:5000/normal_scenario` (Default on initialisation, loads any `<serverartefact>` stored on disk)
- `http://127.0.0.1:5000/test_scenario` (Where a mock model situation is tested)

Listing 22 Test scenario function

```
def test_scenario(artefact_json_name):
    """
    Simple Model and API online testing.
    :return: sample artefact
    """
    seed = load_JSON_artefact(DEFAULT_SEED)
    seed.evolve(subspace.from_relationships([('CustomBinary1',
    'Redis')], 'rediscoupler'), 'redisexpansion')
    solution = seed.evolve_delta(
        subspace.from_relationships([('CustomBinary1', 'bs4')],
        'bs4coupler'), 'bs4expansion')
    # or any not included before saving in any JSON artefact
    solution.bind_context_fragment('views.xml')
    # isonodes or properties must be updated together via XML,
    # otherwise the other data persists as is not overridden
    solution.bind_context_fragment(r'.\GoogleLauncher\googlelauncher_layer.xml')
    solution.update_views()
    return solution
```

5.7.2 ReST Resources

5.7.3 How to implement ReSTful operations

There is normal operation or operation based on a test pre-made scenario. Switch with:

- Using `http://127.0.0.1:5000/operationtest` The form (Figure 5.20) used to transmit a sample Δ operation and affected operands.
- `http://127.0.0.1:5000/api` Executes any operation done via HTTP verb POST

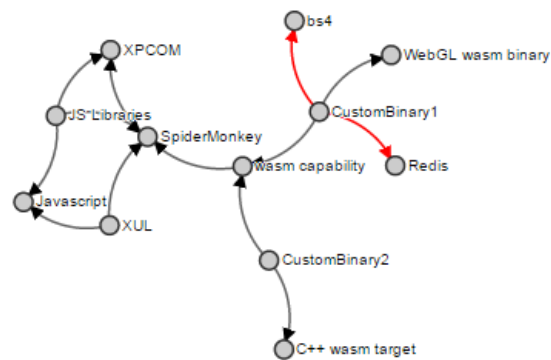
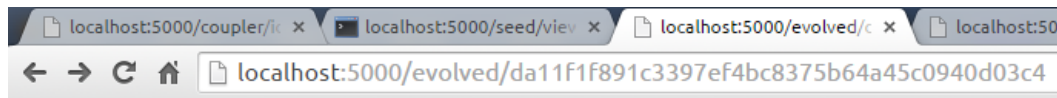


Figure 5.19: Test scenario evolution.

Listing 23 Operation processing by URL `/api` entry using HTTP verb POST

```

@app.route('/api', methods=['POST'])
def operations():
    """
    Evolve two artefacts by (JSON) name (It should use ID at later stage)

    :return: resulting dynamic graph view
    """
    seed = active_equivalence_class.use(request.form['seed'])
    coupler = active_equivalence_class.use(request.form['coupler'])
    view = seed.evolve(coupler, seed.name+'_delta_'+coupler.name)
    #return request.form['seed']+ '_delta_' +request.form['coupler']+ ' = '+ view.id
    io.dump_json(ARTEFACT_PATH+view.id+'.json',view.graph)
    # another fancy option
    return render_template('viewtemplate.html', d3data = io.d3_data(view))

```


Table 5.4: Cloud Microservice API Resource access.

URL	Resource
http://127.0.0.1:5000/<serverartefact>/id	SHA-1 hash
http://127.0.0.1:5000/<serverartefact>/<viewname>	Renders a graph of any available view including the base graph.
http://127.0.0.1:5000/<serverartefact>/<viewname>/JSON	The graph with metadata as JSON
http://127.0.0.1:5000/<serverartefact>/<viewname>/info	Basic textual information for any views
http://127.0.0.1:5000/<serverartefact>/viewlist	The artefact stored known views
http://127.0.0.1:5000/<serverartefact>/tileviews	Graphs take a time to load if there are too many. Experimental. Of course only practical for few views.
http://127.0.0.1:5000/<serverartefact>/info	Basic textual instance information from the base graph
http://127.0.0.1:5000/<serverartefact>/metadata/<tag>	Access to live metadata stored using tags as set in XML configurations.
http://127.0.0.1:5000/operationtest	Returns a basic form to test online the the basic thin Δ evolution operation.



Figure 5.20: Artefacts with graph on disk can be evolved using ReSTful operation handling. The page updates to the resulting base graph.

5.7.4 A test scenario to showcase the model basics

There are two modes of operation. Normal mode and test mode. Normal mode is the mode where an artefact is loaded from JSON data store on this using the name of the file as a form of id. The second model of operation is a mock test situation

where some test scenario has been created to create custom outputs (Figure 5.21). This test scenario will load `evolve.json` as the seed artefact to populate the equivalence class views list. This is performed last with the method `update_views` which delegates to `explore_isonodes()`. These views will contain the graph state derived from the immediate prior operations results. A thin and a thick Δ 's are performed for didactic purposes. The thin delta `evolve` method evolves one view with one coupler. Thick or wide Δ `evolve_delta` evolves the whole known equivalence class views set (implemented as a list) with the same coupler. The ability to do this last one is of paramount importance because the more graphs with metadata we have in memory the deeper the pool to do isomorphic searches and other graph analysis.

Listing 24 function `test_scenario` activated by `http://127.0.0.1:5000/test_scenario`

```
def test_scenario(artefact_json_name):
    """
    Simple Model and API online testing.

    :return: sample artefact
    """
    seed = load_JSON_artefact(DEFAULT_SEED)
    seed.evolve(subspace.from_relationships([('CustomBinary1', 'Redis')],
        'rediscoupler'), 'redisexpansion')
    solution = seed.evolve_delta(subspace.from_relationships([('CustomBinary1',
        'bs4')], 'bs4coupler'), 'bs4expansion')
    # or any not included before saving in any JSON artefact
    solution.bind_context_fragment('views.xml')
    # isonodes or properties must be updated together via XML,
    # otherwise the other data persists as is not overridden
    solution.bind_context_fragment(r'.\GoogleLauncher\googlelauncher_layer.xml')
    solution.update_views()
    return solution
```

5.8 Notebook deployment and Jupyter access

Start *Jupyter notebook* server in the command prompt. This can be done in a suitable folder where the notebooks are present or are to be created. For immediate access to the model is best to start it (or configure it) in the prototype modules folder. This will be the working directory. The server will open the

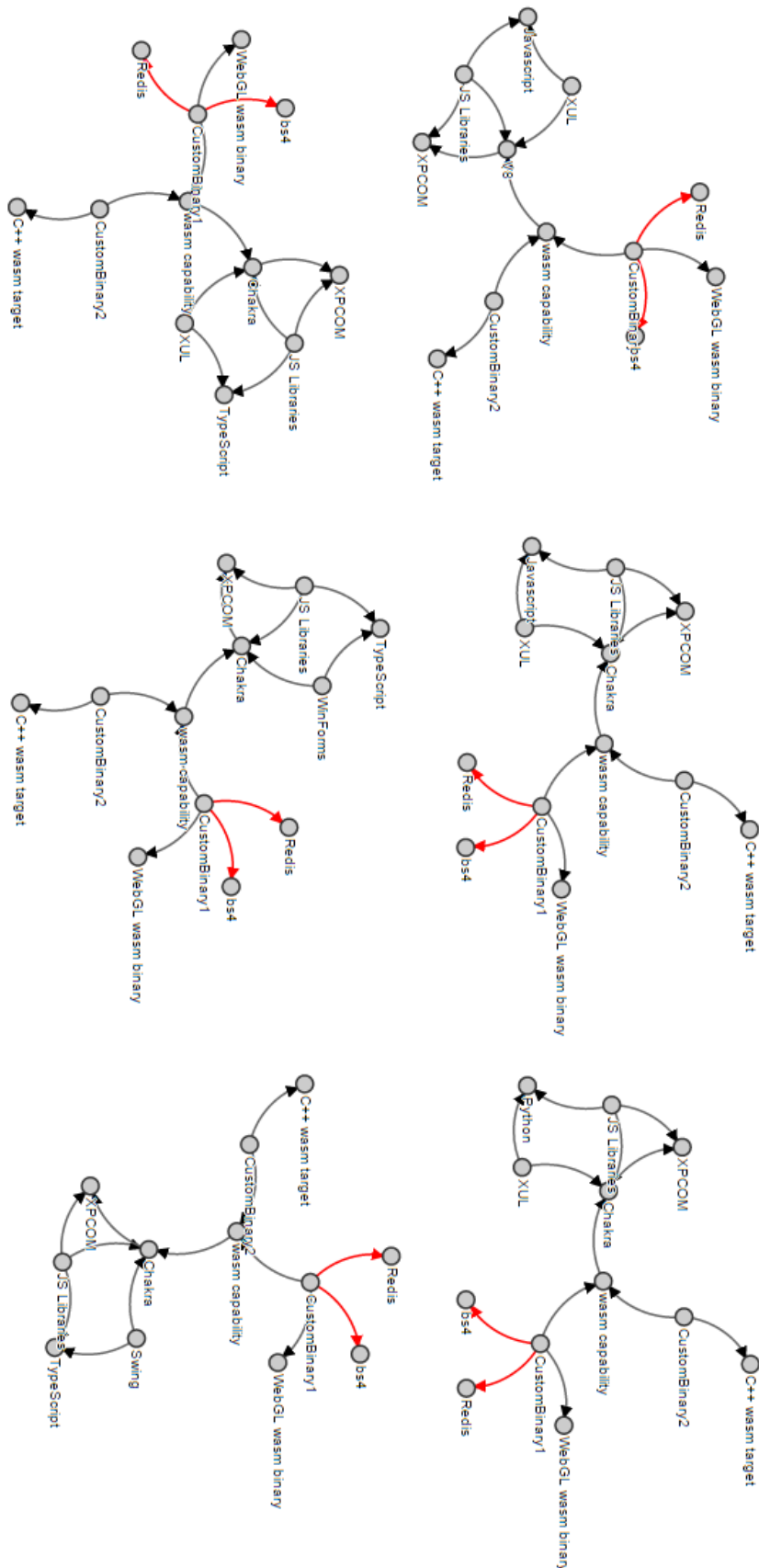


Figure 5.21: Small sample of the expected outcome of the test scenario. Tiled view of thick Δ evolved seed with many specimen views all belonging to the new $[solution\ graph]_{\approx}$. $[seed\ graph]_{\approx}$ belonging views are sub-graph isomorphic to all of them.

default browser pointing to this specific folder or any other. A suitable *localhost* port must be open being **8888** the default (it appears to hop forward to next number if blocked). Using the microservice along with the notebooks is possible and the best way to test the tool (with no IDE). The server usually is started separately but is feasible, to launching within a notebook cell if needed in the future. Elements are written and used offline to embed in the notebook (Figure 5.22). The toolset gain capabilities with extensions. Also communication between servers is possible using JSON.

Start server command:

```
$jupyter notebook
```

Browse to any notebook and click it (as it is a link) to open it.

```
http://localhost:8889/prototypepath/server/anynotebook.ipynb
```

The modules must be able to be accessed by the notebook to be used. All paths need to be adjusted accordingly so it is best to use relative path. This will also make deployment to the cloud easier.

There are several cells. Inside the code cell python code can be input. The result will appear and populate the web page increasing its size. The whole resulting live example can be saved as a notebook for easier posterior examination with no input involved. This allows for interactive tinkering and easy share of ongoing results while keeping documentation of the whole process documented.

Listing 25 The model generates all offline data needed to embed within a notebook

```
%%HTML
```

```
<iframe width="1024px" height="1024px" src="template_file">
```

```
</iframe>
```

5.9 Microservice deployment

The *microserver.py* has to be launched making sure **PORT** is open and that **DEFAULT_HOST** is reachable. On timeout, the graphs will not show due to the browser default settings, as explained before. Some browsers can be configured to deal with this but is best to leave the default settings.

```

%matplotlib inline
__author__ = 'noel'

from model import Artefact
import ioutils as io

def main():
    g=io.load_json(r'artefacts\seed.json')
    h=io.load_json(r'artefacts\coupler.json')
    seed = Artefact.from_graph(g, 'seed')
    coupler = Artefact.from_graph(h, 'coupler')
    solution=seed.evolve(coupler, 'solution')
    print(io.d3_data(solution), 'solution.js')
    io.write_d3js(solution, 'solution.js')

%%HTML
<iframe sandbox="allow-scripts" scrolling="no" width="400px" height="400px" src="viewgraph.html"></iframe>

```

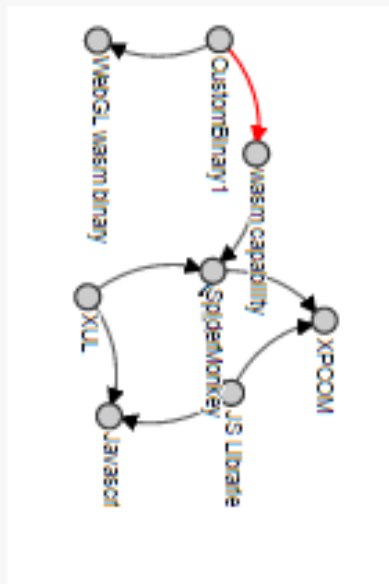


Figure 5.22: Loading JSON artefacts and the embedding of graph views. Graph output depicted over.

Listing 26 Flask server launcher

```
import os, microservice

if __name__ == "__main__":
    port = int(os.environ.get("PORT", microservice.MICROSERVICE_PORT))
    microservice.app.run(debug=True, host=microservice.DEFAULT_HOST, port=port)
```

Further testing and experimenting:

`curl` [36] is a nice networking tool similar to `wget` [73]. We do some low level HTTP response testing from the command line, using the console in Bash, `cmd.exe` or `cygwin` depending on the system in use. The Flask debugger should also be activated, for extra feedback.

```
$curl --header "Content-type: application/json"
      POST http://127.0.0.1:5000/api
      --data "@operation.json"
```

5.10 Conclusions

As seen, implementing the model is certainly possible. It has been implemented open-ended (generic) to demonstrate its expansion possibilities. By using configuration files we gain in flexibility without having to rewrite the source code. However, this does not prevent us from adding information. We choose an isomorphic family by selecting *an instance* of that family and from there generate compatible families. The addition of isomorphic view is exemplified by implementation of isomorphic check methods. The actual addition will require meaningful data so its left as explanatory enough. This works by essentially by delegating on the relevant NetworkX methods.

The automation of configuration information generation and its connection to the theory has been achieved. Through the use of existing Google Cloud Launcher data an example of **automation of configuration generation** and its amendment has been showcased. Furthermore, the prototype models isomorphic additions by expanding a solution view set by adding extra special nodes. **The relation** labelled *isonode* serves as the means to illustrate the concept. It is the basis to show how to link **graph metadata to configuration fragments**. The

XML configuration can be made more complex as well as the JSON data. Of course, the current choice of data format mix is done for didactic purposes. Different data formats could be more appropriate depending on several factors. Evaluating those factors could be an additional good future line of research for their respective projects.

When the number of items grows the amount of relationships, and thus interlinked metadata, grows exponentially. This makes hard to keep tabs on things. We can be certain that as long as the data is of validated quality, lets say a software relationship is truly part of an isomorphism, the output will be sound. But this does not require the person assessing the relationships to know all the details about the project. This is a sand-boxing of concerns into configurations and it is of great use as allows for contributions from different types of practitioners, including non-engineering domain experts.

There are plenty of ad-hoc visualisation facilities. These are incidentally needed to aid in the discussion of concepts and the operation of the model. However, they serve to illustrate the scientific value of the prototype as an aggregation of capabilities.

5.10.1 Evolution as documentation

A command log with a chain of IDs (or a Jupyter notebook for richer narratives) with evolution steps as engineering data could be added to projects. This would allow tracking not only the evolution of the project but how this evolution fits in the grander-than-architecture scheme of things. Another bonus is the extra information being implicitly tracked like asset data or any other added item of engineering interest. This helps in the sense of increasing the level of automation potential of any task we already do, a sort of engineering book-keeping. The fact that the files generated and used are all text based data makes them candidates **to be tracked with CVS tools like Git** for further data mining potential configuration additions.

Implementation Evaluation

Design choices mistakes have been made while implementing the solution. These became more prominent as understanding was gain through the prototyping stages. However, that is the point and tangible gain from prototyping. A series of critical evaluations of some aspect of the prototype sections follows along with suggested changes.

5.10.2 The model should be the deepest module

The I/O access to read the graph data and populate is unavoidable but not inside of the Artefact class. The decision to have it inside of the custom initialisation methods for the class was made to enforce it. However just after that doing an enforced views update would be useful. It could not be done as a recursive call to *Artefact.from_graph* would happen in the method *explore_nodes*. This method should not have used that and return a list of artefact instances. Instead, It should return graphs and thus we solve the recursion problem. Having done that, all I/O could be taken off the class. If done so, the initialisation enforcing *bind_context_fragment()* and possibly *update_views()* could also be taken elsewhere. They could still be done inside of the *model* module but outside of the class. If we also moved there the relevant methods from the *ioutils* module we may have that module made independent from *ioutils* too. All this combined will allow to put such I/O logic at a higher abstraction level and make the model class easier to test. This dependency inversion is done to keep business logic at the deepest level and volatile choices like I/O at the highest. Truth is that in this Python implementation the problem is not too acute. Functions and other entities can be independently loaded for modules. All the I/O functions are kept in the I/O module. The only function could be just added to the model module. In any case these concerns may point to a better redesign route. *bind_context_fragment()* should be called from a higher script, and thus, dismiss the configuration enforcement in the custom class methods. This will help to redesign the class and any derived testing.

5.10.3 Splitting Artefact Class

Another problem revealed through the use of the model is that evolution information tracked regards just the main view but not other views. This is clearly a design mistake if we want to keep the state of all the views beyond just the loaded metadata. That gives way to interesting considerations. The initial idea was to generate all the solution views from the master view (as a graph) contained in the artefact instance. Their IDs were an afterthought to replace the name but that helped with the insights. We realised how meaningful it was to call the fully qualified *model.Artefact* class as *subspace* python namespace. The $[ar\text{terfact graph}]_{\sim}$ defines such solution subspace. This is the equivalence class representing all the potential compatible views bound by isomorphism. Therefore, *Artefact* as a higher class could hold all the views as instances. This will enable tracking views state individually. This means that when we evolve a view we do an *evolve_delta* on an instance and when we evolve the artefact we do an *evolve_delta* on a set of instances. This makes the separation and meaning of these two Δ operations easier to understand at first glance. The method *evolve* will evolve view instances and *evolve_delta* *Artefact* instances holding all views. *Artefact* should be renamed as, let us say, *SolutionView* and a new class called *Artefact* or *Subspace* would hold the *evolve_delta* method. This is because any view addition, like for instance sub-graph isomorphic one, will have to be stored in that class instance. These classes can both remain in the *model* module. Thus, the design is clearly improved. Another recommendation would be to create a higher class *Family*. It could store the chains of family hierarchies emerging from Δ chained operations as they are discovered by comparing the chains ends metadata (This will require sizeable valid data).

5.10.4 Asynchronous operation upgrade

To consider all the aspects of a fully fledged asynchronous online solution is totally out the scope of this research but it is interesting to discuss how it would fit in the prototype.

In order to improve communication efficiency, the Jinja2 [50] template(s) should be transmitted just once. Any change should be implemented using asynchronous facility on the client side. D3.js works with data loaded or encoded into a SVG graph. The functionality is client-side here. AJAX style communication for

asynchronous client communication is possible using any option available (like the ones included in D3 or jQuery). The model would still run as the server side solution. The client can implement the update request of its own partial data changes. To achieve this we suggest using JSON data as the means of transmitting any change (interaction), including of course the all operations. These include **the shifts** that could actually be implemented as evaluation and selection separately. These as of now are performed implicitly by the practitioner but they could be automated. With these minute state-of-the-art changes the foundation of a complete online (asynchronous) solution to operate the model will be complete. This is notwithstanding any replacement to D3 and therefore the SVG rendering. Other libraries will have other facilities and render to different targets. This does not affect the upgrade underlying philosophy but could affect overall performance.

5.10.5 Future scaling

Given the relational nature of the model a persistent solution that scales and is cloud deployment friendly can be designed. The straightforward solution is to use a graph based database. An optional ad-hoc efficient solution is to use the key-value pair database Redis (or any other) to store relational data of any kind. This could include node-ID to metadata mappings. This could implement just the persistence aspects and leave the graph metadata querying to the NetworkX library. However, the concept of configuration files still useful as a human friendly way to input custom data. The use of a combination of these approaches makes a good case for future research.

Software Evolution: organic growth

6.1 Introduction

In the previous chapters we discussed how an evolution framework can help solve problem arising from the combinatorial explosion of asset relationships. This so far have been limited to a Software Line Product context as example of high-order software component reuse.

Organic growth is characterised by step-wise closed feedback-loop of directed compositions or expansions. Specific selections from devised collections of simple structures to incrementally create a more complex one. A starting state and a way to internally direct the growth to a desired future state. These aspects are already part of the model operation. So far we have not departed from an orthodox meaning of the expression.

However, the model can be configured to work as a generative model to find higher order patterns. These patterns are growth chains product of evolving artefacts using the rules previously discussed. We can use predictor properties to enable a search within a range. This can be moved forward to ever larger depths all of them capturing the time-ordering of evolution events. Branching events model evolution in a way analogous to Git(VCS) [44].

The artefact could encompass other higher order structures not made just by *assets*. Cloud systems feature multi-level ways or designing solutions. These designs feature the use Virtual Machines or containers as part of the architectural elements of a solution. We build evolutionary paths using such elements to reach higher order structures using our hypergraph model.

Combining our model, interpreted a generative model, with the Google Cloud Platform (GCP) builds a framework of high-order reuse. To this aim, the operations behaviour have to be adjusted as to meet the constraints of known generative models.

6.2 Organic growth

A seed starting solution state, a way of directing the growth to and a desired future state are requirements to grow organically. The model covers these but there are more.

According to Lehman's laws of software evolution, system growth will be bounded by its size. This viewpoint was further re-enforced by Lehman et al. statistical analysis [91]. System growth, measured in terms of numbers of source modules and number of modules changed, is usually sub-linear. Lehman details that there is a slowing down effect as the system gets larger and more complex. Additionally, we may also consider Free and Open Source Software (FOSS) as higher level components. Lehman's Laws of Software Evolution set limits that seem not to apply to open source.

The addition of elements is expected to follow a sigmoid or logistics cumulative curve. This resembles population growth in ecological systems and the pattern of diffusion of innovations. Also, this is a typical curve to track learning skills progress. We propose to measure this growth in software complexity terms. We should study how complexity changes and grows under different evolution branches. It is common knowledge that "logistic functions model resource limited exponential growth" [156] as depicted in Figure 6.1. They also represent accumulation of a function with a peak value, like the bell curve (normal distribution).

For instance, it has been confirmed empirically by Scacchi that open source software evolution is at least **super-linear or exponential** for FOSS large projects during the early stages [157]. Such project also feature the typical S-shaped sigmoid curve of growth. Some these projects have spawned for decades and they do not show the slow down of growth and decline in quality predicted by Lehman's Laws in spite of increased complexity. This gives new light of understanding to

Lehman's Laws of software evolution in the context open source. Based also on the findings by Bhattacharya et al. [84] we can conclude that open source exhibits super-linear growth.

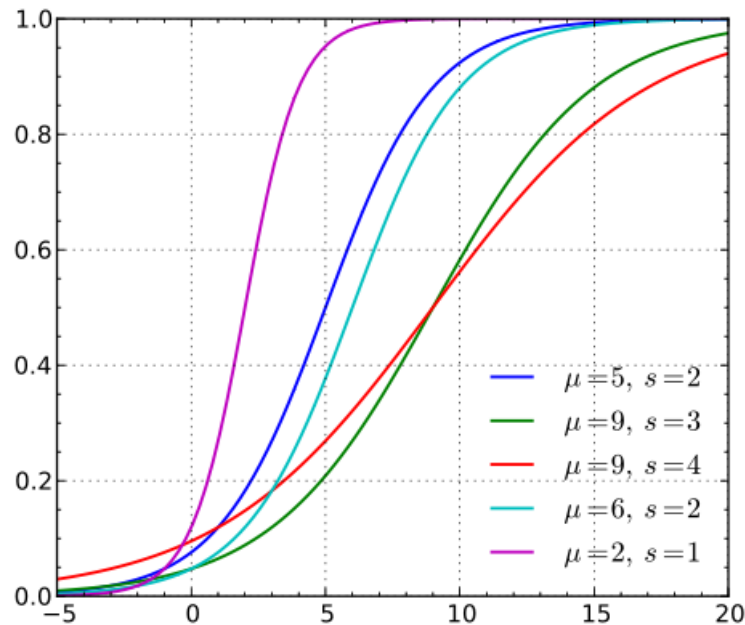


Figure 6.1: Logistic curve with various parameters. Wikimedia Commons. [20]

The super-linear growth scenarios route around previous limitations:

1. Open source: Large FOSS components with developer popularity and stability in maintenance terms.
2. Cloud Systems (Server-side): Here the evolution based on higher order 'reuse' of 'engines' allow system growth. Measures of system growth in term of code are no longer meaningful as the code size grows super-linearly.
3. Web growth (Client-side): emerging developments such as *wasm* binaries (WebAssembly) enables large scale evolution by coupling existing large software base through cross-compilation.
4. The combination of all three: ReST architecture can incorporate all three under a common access interface. Allowing for asynchronous modular growth. The system is able to evolve asymmetrically with parts in different stages of evolution.

6.2.1 Generative models

A multitude of product lines are feasible from the vast set of assets in the Google Cloud Platform (GCP). We are able to model any composition of a client-server or a distributed solution from higher order reuse of assets. We can therefore view this organic growth in terms of a set of equivalence classes ($[H]_{\simeq_1} = S_1, [H]_{\simeq_2} = S_2 \dots [H]_{\simeq_n} = S_n$) representing different product evolution sub-spaces corresponding to different product families. These emerge from the chain of graph expansions arising from Δ events. A family will comprise all the paths leading to the same solution albeit using different branches. The branches will feature a collection of states or waypoints guiding the evolution. This guidance is under a set of constraints affecting the shift operations.

Node attachment probability and Power-laws

(Object Orientated) Software designs are **scale-free** if the *probability of relationships* among classes follows a *scaling-law* as stated by Valverde and Sole [158]. This can be extrapolated to any relationships of any software construct as long as the nodes have the same properties as classes (from a graph network point of view). For Scale-Free networks it should follow the Barabasi-Albert Model [94, 159]. The attachment of couplers is not random but features preferential or assortative attachment to nodes based on their properties. Another condition is that the number of potential attachments (links or relationships) is unlimited (or not limited by resources). The concept of Node fitness can also be used to generate Scale-Free networks [160]. The Small-world effect (short inter-node distance on average) guarantees the chains of solution expansions will not be long on average. Hubs and the clustering of nodes can also be expected. This concepts have been applied to forecast future evolution in existing Java software projects [161].

6.2.2 Predictors: Key search properties

The parameters of our algorithm can be set forth as a traversal operation with a stop condition. The base algorithm is as follows:

- Randomly choose a set of *isonodes* for expansion if there is no useful configuration for them.
- Select solution view based on existing properties

- Generate a solution $([H]_{\simeq_1} = S$ using couplers based on pre-defined criteria
- iterate for a fixed number of expansions or use the metadata (as predictors) to trigger a stop condition.

6.2.3 Scanning using predictors

The algorithm to explore an area determined by the Δ and \simeq_n depth. At each step backtrack is possible, until we find a branch that is satisfactory. This is analogous to a maze search process but using certain solution sub-spaces as the open paths. The difference with the traversal search is that Δ 's are allowed. The reason is to expand the graph to look deeper into the new emerging node metadata relationships. The deeper the search into the resulting combined structured data, the greater space gets covered. The challenge is how to automate the encumbering relationship evaluations as the sub-space grows away from a seed point. To search for desirable solutions also can be interpreted as to avoid undesired ones. The practitioner eventually selects from desired options discovered if one is not automatically selected. This operation has a clear role to play regarding the automation potential of the model. Finally, the performance impact of this operation, for large enough $([H]_{\simeq_i}$, should be studied empirically.

6.3 The Cloud Family: Branch evolution

The history of the web/cloud as a sample model for multilevel (view-abstraction transformations) evolution growth. Modules division and further specialisation is happening at the functional level on both sides of communication. We assume the events discussed as common historical knowledge. The Web is one implementation of the ReST architecture. The statement can be reinterpreted into a new one: The web is a *software view* of the ReST architectural style. If the aspects modelled as relationships are seen as This is the **contextual framework of the evolution**.

6.3.1 Starting stage: The birth of The Web

The history of Cloud System from early web until today had this **starting stage** context:

1. *The Seed*: HTTP served static full text with hyperlinks (HTML) to be rendered by the browser

2. The loose coupling and high cohesion forced by the protocol allowed client and server to evolve separately.
3. Presentation and Controller (presentation events) remained client side as the browser was event based.
4. Medium (the protocols) forced the separation of concerns.

The common critical parts to evolve at both sides are efficiency of the communication, hardware access and later performance. From there on their operational context remains as the separation of concerns.

6.3.2 The Client family

A series of events we can call client events defined what the web browser is today. The client seems to specialise in those areas where it makes more sense given its context. Unsurprisingly, the client act as the presentation layer as we are the ones using it. This made it event-aware. A series of evolution steps happened each adding a feature or solving a problem. This in no way perturbed the server development. Both happen in tandem thanks to the low coupling and high cohesive nature of The Web as a software solution. This also means as part of the ReST architectural style for which *The Web* is an implementation. The [solution] \simeq here was the existing arrangement and the coupler contributed with the necessary software changes. **The choice** of the couplers(*shiftings*) was guided by the presentation nature of the client. Features added to the client by W3C are evidence of the targeted changes: structure (HTML), presentation (CSS) and semantics (tags with meaning, like <article>).

Here is an incomplete list of client-side events just for illustrative purposes:

$$ClientFamilyChain = \left\{ \begin{array}{l} \Delta \text{ Pictures and other media (plugins) added to the client} \\ \Delta \text{ DHTML: the client gains scripting} \\ \Delta \text{ The web goes asynchronous (AJAX).} \\ \Delta \text{ HTML:Code On demand using URLs} \\ \Delta \text{ Declarative data manipulation.} \\ \Delta \text{ Hardware access. OpenGL, OpenCL, codecs.} \\ \Delta \text{ HTML: Structure, presentation and semantics separated.} \\ \Delta \text{ WebAssembly: The web becomes compilation target} \end{array} \right.$$

The evolution is also depicted in Figure 6.2 as affecting the whole previous state.

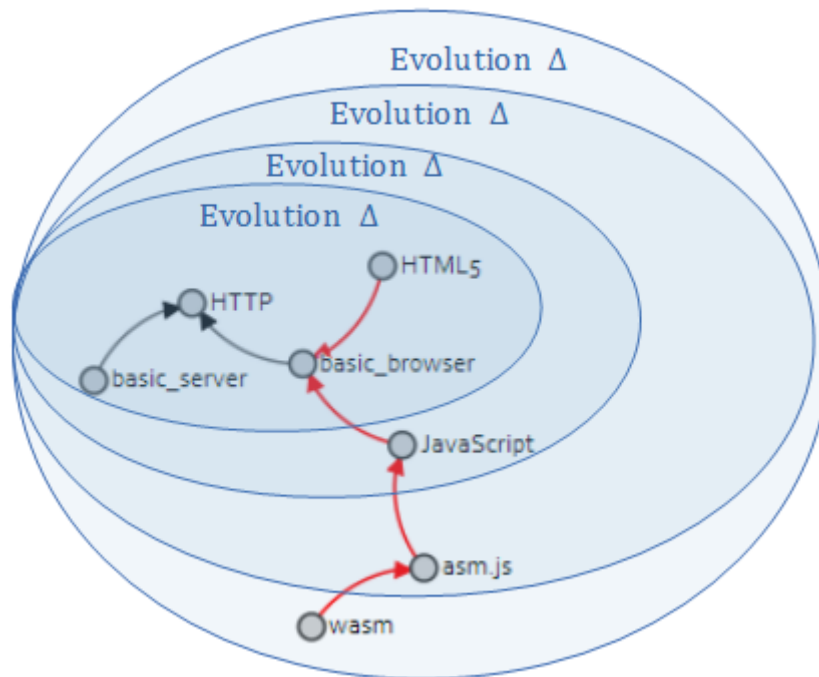


Figure 6.2: Evolution of the client affects the system as a whole.

6.3.3 The Server family

Aside from the mentioned critical parts the evolution of the server gets marked by the conversion of servers of any time to VM instances. They later specialise into containers with less overhead by abstracting OS functions. There features innovations that obviously matter only in a server context. It should be noted that in a peer-to-peer (p2p) network the client and the server would be the same. Its getting increasingly convenient to access services by VM or container instance. For instance, we may want to run specialist software without configuring it or having total control of it. We can run our own server and use The Web as the interface, may be any other. This is what the Jupyter server does by exposing currently implemented language kernels (loaded as plugins). The event leading to this The $[solution] \simeq$ ensure an **sub-graph isomorphic** solution available will links both chains to create a new family part of The Cloud family and by **transitivity** to ReST. We acknowledge a Cloud solution must include an appropriate client side to be of any use. It develops server-side but uses client evolved capabilities.

Here is an incomplete list of server-side events just for illustrative purposes:

$$ServerFamilyChain = \left\{ \begin{array}{l} \Delta \text{ Scripting; browser code(HTML) generation} \\ \Delta \text{ Database sharding.} \\ \Delta \text{ Load balancing: More hardware servers.} \\ \Delta \text{ Server goes asynchronous.} \\ \Delta \text{ Virtual instances replace servers.} \\ \Delta \text{ VM instances turn into containers(OS services abstracted)} \end{array} \right.$$

6.3.4 The Cloud family evolution

The ReST style architecture and other factors contribute to make these families isomorphic. Also, client are expected to work with the server by standard or common feature agreement compliance. Such pairings can evolve in various ways but any current Cloud will be the relation of both. This new family is created using respective last compatible families. The collection of engineering criteria will define the governing isomorphism. Based on previous description we can elaborate:

$$\begin{aligned} [ServerFamilyChain]_{\simeq_i} &= [H_s]_{\simeq_1} \Delta [H_s]_{\simeq_2} \Delta \dots \Delta [H_s]_{\simeq_i} \\ [ClientFamilyChain]_{\simeq_j} &= [H_c]_{\simeq_2} \Delta [H_c]_{\simeq_2} \Delta \dots \Delta [H_c]_{\simeq_j} \\ [CloudFamily]_{\simeq_k} &= [ServerFamily]_{\simeq_i} \Delta [ClientFamily]_{\simeq_j} \end{aligned}$$

Therefore, both chains can evolve together as a longer chain. By the means of this mathematical description it can be deduced the evolution happen in asynchronously as H_s and H_c solution combinations. Δ is an anti-symmetrical operator and seals the time-order event sequence producing the new $[H]_{\simeq}$. The order matters. Resulting state will be bound by first choice after Δ

6.3.5 The Cloud family: Analysis

Client-server compatibility is agreed upon by certain standards but poorly automated. Servers have to rely on client user-agent strings and the client does really known a priori the server capabilities. Not all services are ReST based or truly follow the design philosophy.

There are two major grand events happening in both sides of the network:

1. First a split into subunits: Further functional modularisation towards lower coupling and

2. (Those) Units Specialisation: Specialise to offer straight higher coherence solutions (thinner modules).
3. These happened within their respective operational context.

This looks like a divide and conquer heuristic happening at The Cloud system level. We also expect data portability from closer family solutions to be less convoluted

6.4 Conclusions

6.4.1 Evolution process of growth

Today even source code can be at a high level of abstraction. Components can be also be large complex software frameworks. We combine opportunistic reusing of transversal components with the planned strategic reuse of SPL assets and other higher-order structures. This latter can be extended to Cloud components. If the hypothesis defended by Musco et al. is true and there is an underlying software evolution process common to all software [162]. That hypothesis is backed by empirical data statistic analysis.

This process can use open source sub-components super-linear growth to accelerate, in aggregate, higher-order asset evolution. Independent modules can evolve separately and asynchronously thus increasing, or not slowing, the evolution constant speed. The evolution events generate a time-order reflected as a chain of expansions. These go ever deeper into the solution sub-space and this gets increasingly closer to a solution.

The information is carried forward by the evolution process, shaped by the selections (shifts, direction) and subsequent Δ . It is really important to point out the sheer amount degrees of freedom required to form long evolution chain.

6.4.2 Families

As generative model this process can create chains of solutions, with expansion stages, of various lengths. Any different chains of different steps are guaranteed to be compatible if they reach the same last state. If so, they belong to the same solution sub-space, that is the same equivalence class. The transitive property of

the equivalence relation \simeq of that state play the key role in this context. The chain will lead to different properties to be evaluated giving a characteristic property fingerprint to each family. Subspace can be scanned use predictors to evaluate and help to select which chains, within a range or depth, lead to the desired solution state (or stage).

6.4.3 The Cloud family

The Cloud family can be constructed as a series of relationships of its high-order assets. Its solutions are fruit of the independent evolution of the server-side and client-side parts helped by this divide. This divide is the foundation of *The Web*, which itself is an instance of a ReST architecture. Anything that improves the communication will make this divide more meaningful. Common interfaces proposed by ReST help in this regard. This arrangement enhances the high cohesiveness and low coupling of the modules with respect to the other side. The evolution events lead to this arrangement and the final state is The Cloud. This state is part of a hierarchy of families. Client-Server, The Web, The Cloud and then ReST are stages of evolution or their respective solution sub-spaces.

Therefore, we manage to find instances composed of interrelated higher-order assets satisfying a particular set of engineering needs. We can automate aspects of this search and empower the practitioners without hindering their creativity.

Conclusions

Contents

7.1	Introduction	152
7.2	Formal modelling considerations	152
7.2.1	Queries using graph based metadata	152
7.2.2	Automating solution (family) search	153
7.2.3	Observations and findings	154
7.3	A Finite State Machine generator	154
7.3.1	Solution family instance detection	154
7.3.2	Operations as data efficient messaging events	154
7.4	A (Scale-Free) network of assets assembler	155
7.4.1	Unlimited relationships	156
7.4.2	Existence of Simpler Solution families	156
7.4.3	The need for complex families	156
7.5	Model Implications and Re-Interpretations	157
7.5.1	The coupler as a solution subspace	157
7.5.2	Solution sub-space iterator	158
7.5.3	Emerging fractal spaces	159
7.5.4	$[\]_{\simeq}$ as a Closure	159
7.6	Critical appraisal	159
7.6.1	Terminology and concepts	159
7.6.2	Models can be dangerous	160
7.6.3	The need for empirical evidence	160
7.6.4	Prototype implementation	160
7.6.5	Automation of craftsmanship	161
7.7	Recommendations for Future Research	162
7.7.1	Visual Modelling	162
7.7.2	The research of other implementations	163

7.7.3	Automatic stop using conditional traversal	163
7.7.4	Complex Data Configurations	163

7.1 Introduction

Model Overview

All graphs are relational information glued into a hypergraph as the incidence structure of choice. This is achieved by virtue of establishing new relationships between vertices and their functional equivalents through equivalence classes. These, partition the whole solution space into subspaces ($[]_{\simeq}$) made of solution views. The underlying isomorphism \simeq enables further reach by extending these subspaces as desired. We also grow the subspace into other subspace with Δ where previous isomorphism does not apply. Let us remark that the formalisation model, in fact can be applied to any relationship based problem based on pairs of elements. Therefore, it is of general applicability.

The result is a formal model comprising:

1. A functional core (Shifts and Δ over $[]_{\simeq}$) (CRUD operations implied).
2. Operands are complex chains of relationships modelled with equivalence classes governed by a *compatibility assuring* isomorphism operator.
3. The result is a chain of hypergraph states through evolution time.

7.2 Formal modelling considerations

7.2.1 Queries using graph based metadata

The solution path can be thin or thick depending how many common branches lead to the same solution. They could be filtered out later with criteria not yet considered for some other improved choice. This can be done, for instance, by breaking the isomorphism using stored node metadata. This allow us to destroy undesired relations not matching our constraints.

Hence, there is an implicit view a graph consisting of assets with attributes (related structured data). These attributes can be turn on and off by adding entries to the relevant configuration file or at runtime on the nodes graph metadata. Data configurations are a convenient way to store node metadata in design-time. The advantage of storing data into a graph structure comes from the wealth of existing techniques for graph analysis. We filter a graph collection by ignoring relationships based on node metadata to **discover graphs**. This is also a **graph search** function if the filtering is extreme.

7.2.2 Automating solution (family) search

Software evolution is "*projection into the uncertain future*" since the environment will change and we just have some available information. This is an exploration and search, multidimensional in nature. This encompasses the evolution process and the final state of such evolution. Using reusable elements we construct complex operands. Then, using a small set of operations, we process them by using the configuration information available. The *evolve* operator Δ acts as a growth step into a possible future growth path.

It is easier to take decisions as early as possible. With each evolution step some minute change to the model occurs. This brings a little insight from an unevaluated uncertain future. We cannot evaluate all the options at any later stage but *shifting* (steering) the evolution as we operate is possible. Therefore, decisions and backtracking can happen early, enabling to search for satisfactory branching.

In other words, we can ping or probe a possible close future state by locally *traverse* to it and then evaluate if it is desirable. The distance to a desired state establishes the initial degree of probing difficulty. A collection of *probe actions* can perform a scanning of a certain subspace size. The further we are from a point the harder it will be to search and evaluate all the options. This provides a way to see into future evolution events (Δ) outcome with just the information at hand. It should be noted that taking the wrong steps could make back-tracking seriously prohibitive (albeit inevitable sometimes). We can just cover so much of the solution sub-space. Data to be assessed grow exponentially with Δ . That is why so important to make good chain-wise early assessments.

7.2.3 Observations and findings

We can enumerate some interesting findings:

1. From a *behavioural* point of view is a Finite State Machine generator. This has wide ranging implications discussed below.
2. From the way relationships are linked together forming a network (*growth*) it can be a Scale-Free network generator. There is no empirical proof, just compliance with some literature pre-conditions.

7.3 A Finite State Machine generator

The way model operates leading leads to consider it within an Automata Theory context. A Finite State Machine or a Deterministic Finite Automaton are related to graphs [42]. Each evolution path is a recipe to assemble a correct solution. It features a succession of states encoded as $[]_{\simeq}$ with an end state representing the desired solution. A generator of valid growth tracks.

7.3.1 Solution family instance detection

But also a validator of tracks. This could validate tracks (solutions paths) in the same fashion regular expressions detect text patterns. Both ways model a FSM. Regular expression are just one FSM implementation. Instead of regular expressions we can use the $[]_{\simeq}$ trail to recognise traces of this growth tracks in large (asset) relationships graphs. We could use this to search and replace long chains with short chains, thus simplifying the software system. This is performed considering one instance. In addition, since we model families, we can study all instances belonging to the same family. Guaranteed to be equivalent by the Δ_{\simeq} chaining.

7.3.2 Operations as data efficient messaging events

Furthermore, this state machine is an event-driven system, an event-driven state machine. It can be implemented as an event driven FSM. The operators are the events and the state is kept by $[]_{\simeq}$. Coincidentally an event driven FSM models a communication protocol.

The model as *modelling a protocol* explains why it is so data efficient when considering a client-server implementation of the model. Since the model is also a growth model, this yields a client-server *relational* growth model. In a peer to peer network only the asset configuration data would be store and not transferred by the nodes. Just the key or ID of the solution view or coupler in question would be needed to be sent.

The seed solution is a representative of its solution views. With an added coupler, we obtain the future views to be evaluated. If we perform this step several times we get a sequence of growth. This growth is kept in the last system state. Let us re-frame again the problem on a client-server basis as a model for low coupling, communication architecture. Let us put the model on the server.

Assuming no ad-hoc configuration will be needed later, all asset data could be on the server. We only need the graphs on the client side as the minimal presentation. To change the state (growth) we need to transmit one coupler or, since we have all information on the server, just any identifier (ID) will do. We can just attach the coupler (may also be visually) and keep the client prime function: presentation (interfacing).

Behaviour explained by its intrinsic state machine (from a model-wide point of view). We only need to transmit "the event" and get back some results from the "evolved state". We transmit back from the server just the relevant information from the evolved state to make any view from the client meaningful and useful. This arrangement is optimal from the point of view of communication channel efficiency . The model shares the low coupling and also the communication efficiency properties of the client-server model because it models a communication protocol.

7.4 A (Scale-Free) network of assets assembler

Some aspects related to how relationships grow the network using the model are precondition for a scale-free network behaviour generation but not proof. Like in ReST, the system is featured by a set of operations with an uniform and easy access to resources. In ReST this is achieved through the use of URIs. This gives

it easy access to resources on the network, just with a few links. Furthermore, the number of how many of such links (relationships) can be created is not limited.

7.4.1 Unlimited relationships

Every step in the evolution acts a modulator on the degrees of freedom available to perform the next evolution step. The coupler graph chosen to traverse unlocks new possibilities for expansion. A principle of ReST architecture is "Hypermedia as the engine of application state" or HATEOAS. A strong parallel can be found. In our model, the state is the last expanded graph and isonodes are acting as hyperlinks (relations) to new equivalence classes (families). By simply adding nodes (and these relationships) to an equivalence class we are establishing the connection (link), and thus, also enabling new exploration pathways. Also, these new nodes set the stage for new evolution paths to take place executed by the *evolve* Δ operation. It is the shifting or branching, by the practitioner (explicit) or automated (implicit), which will steer the state and the growth to whatever new path.

The unrestrained power of relating or linking is precondition of a scale-free network as well as preferential attachment based on node properties [94]. These preconditions are met. The model **insofar** complies with the Barabasi model preconditions should grow a scale-free network. A Node fitness criterion [160] is guaranteed, by isomorphism in our case, so the strength of the claim is reaffirmed.

7.4.2 Existence of Simpler Solution families

This scale-free network guarantees high connectedness, the small world effect. The connectedness allows for solutions families to have a shorter (smaller) discoverable instances. This feature enhances a replacement mechanism using the family instance detection previously discussed. This mechanism allows for simpler software with no known property loss.

7.4.3 The need for complex families

Reaching a solution may require long chains of artefact evolutions. The reason for this is to achieve the degree of property aggregation needed for a solution to be acceptable. Furthermore, complex solution richness, gained by its node count,

may exhibit special future freedom to expand. The main point to take is that not always the simplest solution will be the best choice. A deliberately long solution may be more *evolvable* for a variety of reasons. The Cloud evolution is a good example of this phenomenon.

7.5 Model Implications and Re-Interpretations

7.5.1 The coupler as a solution subspace

We could model both the solution view and the coupler creating two sets with two governing \simeq . It became clear after implementing the prototype that, recycling the concept of artefact to include the coupler, made sense from both a development and theoretical perspective. A new scenario emerges if both solution view and coupler view belong to their respective $[view\ graph]_{\simeq}$. We talked about the FSM created by the model events affecting the solution subspace. The G_i (states) not only is a hierarchy but also the crossing point if we generate **another FSM from the coupler view perspective**, accepting only compatible solution views. Either shift has to happen first, narrowing subsequent choices. The tree is created by the composition ($\Delta \circ \Delta$) of many binary functions as depicted in Figure 7.1. Instead of adding a coupler graph to expand the existing original graph we consider both graph to be parameters (solution subspaces instance or views). One subspace has to act as the coupler subspace (by searching and selecting an instance or view). Hence, a new **binary** operator Δ allows constructing binary trees of artefact assemblages (Figure 7.1. With this perspective we are just acknowledging the solution space as explicitly searchable or not fixed. In terms of implementation, for instance, this could turn a subspace into a internal or external variable (regarding the modular arrangement).

$$\Delta(\Delta(A, C), \Delta(B, D)) = \Delta(X, Y) = Z$$

The way it is currently implemented, the solution evolves (itself) with a coupler. The solution and the coupler are artefacts. Choosing the coupler first would limit the solutions available, but that, could be of interest in some engineering scenarios. If the solution space is equivalent to a web browser with a set properties, a compatible extension performing the desired task would be the coupler. If we swap the roles of solution and coupler many solutions can be added as couplers. This could be useful for the need to find a custom and compatible complex

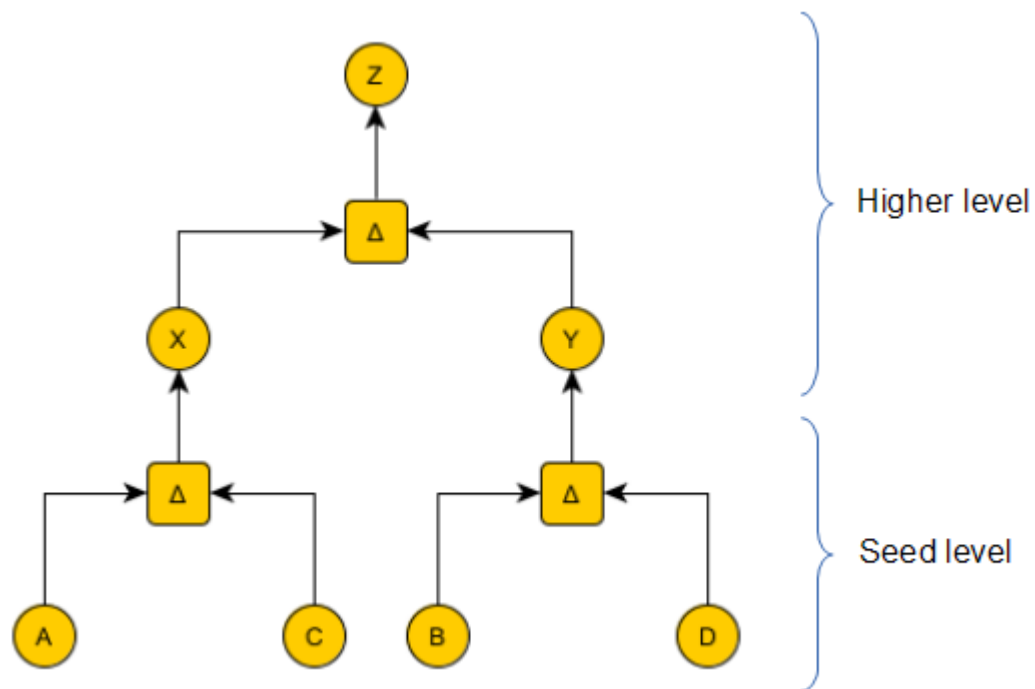


Figure 7.1: 1 family. 2 levels. 4 ordered sequences of Equivalent solutions operations. 4 paths.

component using solution as couplers. Using the previous situation, the extension is chosen (fixed) and any compatible browser or other piece of compatible software is the variable option. The implementation forces a choice order, *solution_view.evolve(coupler_view)*, where a method would use *evolve(solution_view, coupler_view)*. Fortunately, upgrading the original implementation to illustrate this is straightforward. In this perspective the new $[view\ graph]_{\simeq}$ albeit an artefact is governed by a different \simeq , that is, a new different subspace. This will model all the potential solution space as a binary tree. That is, the model can be viewed as modelling solution sub-spaces (families) as a binary tree of hypergraphs as artefacts (hierarchies).

7.5.2 Solution sub-space iterator

The operations can be implemented as functors by adding an evaluation function as parameter. The artefact as hiding an abstract container of compatible by isomorphism equivalence classes. The model as an iterator implementation of

solutions sub spaces advance is performance by iterating over known solution views.

7.5.3 Emerging fractal spaces

Since the operations, including *Delta*, are also an iterative function they are the basis for fractal [163] space modelling. The model operations work as a plane-filling function (a space filling curve), albeit for solution spaces [164]. Using the binary tree modelling view its easier to deduce the model to be a type L-system for solution sub-spaces [165]. Every Δ is a refinement toward a solution.

7.5.4 $[\]_{\simeq}$ as a Closure

Here *closure* is used from the Set Theory perspective. $[\]_{\simeq}$ identifies members of the same family. The outcome is part of the same set so its closed for both shifts. Since this is implemented as a class *Artefact* where a concrete solution view growth state is the artefact instance. This makes the *Artefact Set* close with respect to Δ evolve method as it produces a new artefact. This is only true from the hypergraph point of view as the same hypergraph encoding remains. The set definition $[\]_{\simeq}$ will change to incorporate a new isomorphism, making Δ not close from that perspective.

7.6 Critical appraisal

7.6.1 Terminology and concepts

First and foremost, the terminology of abstract concepts is complicated to balance. On one hand there is the need to be clear and understood. On the other hand the abstraction potential should not be lost. There is also a clash with commonly used words which make them hard to use. New terminology was not invented nor created. This has the cost of making the reader get lost in abstract or general concepts. There are some freedoms taken that should not prevent understanding. For instance, "software solution, logical or any other view" gets shorten to solution view or just "solution" since we are talking about compatible solution sets. The term *Artefact* is also troublesome. It clashes with the common denomination of general engineering documentation outputs, as well as with any software related device or widget. We could term it configured solution sub-space but that is too

long albeit accurate. Artefact, subspace, solution, view or $[view\ graph]_{\simeq}$ are used depending on the emphasis required by subtle differences regarding the topic at hand.

7.6.2 Models can be dangerous

Models are simplifications of reality. Sometimes oversimplifications. They are not exempt of being contaminated by all the cognitive biases plaguing any research. The model definition is complex but it is simple enough as it operates. It is based on formal mathematical definition so there is an element of safety in that sense. It may not model all unforeseen engineering aspects or it may not be the best way of modelling software relationships for any particular purpose. Therefore, it is best to evaluate this and other models on a cases by case applicability basis supported by empirical results. There exists an epistemic unsuitability under the heavy uncertainty created by change events caused by the passage of time. This is of lesser concern as it is covered because the *episteme* the model is based on is Set Theory and Graphs, i.e., well known Mathematics. The risk should fall on the applicability to future software engineering concerns.

7.6.3 The need for empirical evidence

The prototype supports the theory but it is quite incomplete as a practical demonstration of some aspects. There is a need for **empirical proof**. The formal model only validates the relational model but not the consequences of operating it. This is a good line of new research. The space partitioning and traversal in any form is also formally validated. What is also not known is its uses of a particular engineering need beyond **asset relationships**.

7.6.4 Prototype implementation

Albeit for didactic purposes the choices for data formats are not ideal for every purpose. For a Python only implementation YAML would have been a better choice for straightforward operation. For graph incidence tables as text (CSV) could have been used. However, it is useful to show how to convert to useful formats like JSON. That way also need to transmit data as JSON was anticipated. XML implementation could be implemented in a more general way as not all scenarios are covered, underutilising its potential. Some aspects could be element

attributes and some other may need levels of nesting. These text files are appropriate to configure the system but a database will be best to persist the state of larger networks. Since the model is relational, Redis (a key-value database), could be a good choice. A graph database is also a sound choice for the same reason.

Software engineering concerns

The prototype has been tested but not exhaustively. A battery of unit tests is need to assure all known conditions are covered. Since the development was so experimental, such level of testing were obviated. Here it became evident how hard is to keep concepts and prototype in synch. Refactoring solves most of the problems in this case but not every one. Furthermore, the propagation of changes when some basic concept is reassessed is noticeable. This may point to a design flaw(s) or the change is so fundamental is causes an overhaul of the implementation. Given these, solid testing is left for a future production stage. However, these are interesting engineering concerns to take into account.

7.6.5 Automation of craftsmanship

The model allows exploring complex relationships of assets. The emphasis was on architectural and higher level (or abstract) components modelling applications. The prototype is evidence we can automate the analysis of those intricate relationships, even if applied to other designs elements like modules or classes.

The tool can be used and expanded as a companion aid to document development. Links to Jupyter notebooks can be added to git as extra documentation. The notebooks can also be used to document development stories. A visual way of modelling can be added for extra ease of use.

The prototype shows how easy is to configure a system and these configurations can be easily traced with no overhead as they are text files. Their changes can be tracked with Git (or other VCS) as just another part of any engineering project.

The microservice facility shows how fit it is for a web ReSTful implementation and it also has great extension possibilities. This could be the foundation of an independent case tool. With any deployment proper companion tools to create

configurations, including internal *wikis* or a repository system to classify the assets.

The model have been demonstrated to be feasible and viable using current software. However, it could have been more complete with extra evaluation functions or more detailed metadata. Any of these may require enough good engineering data for any analysis to make sense.

Human friendly automation

The history of software development is littered with examples of shifts to higher abstractions levels. Opportunistic reuse is an old strategy and several component models were successful in the past. It needs to be enhanced by planned reuse like in SPLs.

Parallel to this, computer languages are increasingly higher level and started supporting multi-paradigm features. This feature acquisition is noticeable with regard to software patterns. Software patterns are a way of reusing known solutions to known problems. In some instances, they reveal a gap in the language a missing feature. Now features of the languages are tuned to match software patterns (like with iterators). This can be modelled as an asset or a property o the asset.

This fits well with our initial goal of automation of craftsmanship. Moreover, it empowers the practitioners to evaluate the options at any desired level without limiting their creativity.

7.7 Recommendations for Future Research

7.7.1 Visual Modelling

There is a lack of support for visual modelling. It was more interesting to implement the model and obviate the need for a visual interfacing. Such interfacing falls in other implementation domains. There are many ways of display graph in Python but is nascent scene for interactive graphs, particularly on the web. The

web is the de facto presentation target. There is D3.js using SVG as the drawing system. An alternative, Bokeh uses the HTML5 canvas element and is supposed to be faster for larger graphs. Both use JavaScript. To study specific JavaScript implementations was explored but ultimately considered to be out of scope. There are other systems integrating D3 into Python (to also use it in Jupyter). Since most are quite incomplete (as of 2016) except for D3 and they are under heavy development right now. These and others may be a good choice to upgrade any prototype future version.

7.7.2 The research of other implementations

It would be of great engineering interest to see original implementation of the models. Perhaps using other language features too. As said, the engineering needs may vary and not all the implementation approaches are the same. Even written in Python there may exist a better implementation. This can be studied in depth.

7.7.3 Automatic stop using conditional traversal

Automatic evaluation of metadata for automated exploring has not been implemented. This has the utility to advance on the hierarchy branches until a node configuration metadata condition is met. Only large data configurations would justify this but it is still a really useful feature.

7.7.4 Complex Data Configurations

Advanced online web harvesting

Harvesting/scraping online or offline resource has great potential as demonstrated. This could serve to create or enhance configuration files or other novel ways to alter metadata, perhaps while on execution. Other relations can be created automatically to try to push the model to its limit. Using the technique shown, we can build an isomorphism database that can be enhanced with the aggregate knowledge of practitioners. There is more work on the data gathering aspect. This is good research line. This includes mining real SPL repositories. All this could be integrated as a whole CASE tool with an enhanced ReST API access to allow this new aspects.

Intelligent Scanning

We use these branches to traverse the whole solution space as $[seed_artefact\ graph]_{\approx}$ expands and evolves multidimensionally. This is poised to be exponentially easier closer to the seed when the underlying hypergraph is smaller. Intelligent operation based on machine learning or soft computing techniques could help to analyse all the artefact information and branching up to a level. They can also help to grow autonomously until conditions are met.

These are operations that go further in complexity and implement heuristics, AI or any other soft-computing techniques. This complexity is to add get to a level of interaction or configuration beyond what can be achieved by other complex operation implementations. There are many possibilities in this domain. We would briefly discuss a sample application for illustrative purposes.

Configuring schema parameters with Genetic algorithms

To deal with large solution spaces or combination of variable we can make use of Genetic Algorithms (GAs). It is possible to devise operations beyond plain operations on the graphs. In order to reduce the size of the solution space we may want to apply some meta-operation with specific goals.

We could use genetic algorithms to discard solutions that not meet some criteria and therefore highlight some pockets of valid solutions. The advantage of genetic algorithms is that they excel at searching large solutions spaces finding *satisficing* solutions that are exponentially better as an aggregate of the improvement of averages after each iteration [166].

We could construct such operations with a clear defined function concise enough to minimise redundancy of operation. This multidimensional search is defined by the encoding of the genes. Binary encoding is the worst case scenario (further from the description of the problem) and some higher abstracted level encoding the better one (closer to the description of the problem). A genetic algorithm is a process that discards a lot of unsuitable specimens in an implicitly parallel fashion. It could be used to find pockets or regions of acceptable solutions in the solution space. Best way to picture this is a multidimensional slicer (or discarder, depending on the point of view). A family of operations can be created with various levels of complexity.

For instance, it can be expected that a *cohesion_maximiser* or a *modules_reducer* (as a knapsack problem, which is NP-hard) operations could be developed. Julstrom [167] describes the Knapsack problem and suggests a greedy heuristic to enhance the *naive* genetic algorithm approach. Such optimisations could be implemented as an operator. A collection of n objects with two positive numerical values as attributes need to fit a conceptual knapsack with weight capacity capped at C . Being these two attributes size and weight, we need to maximise the size of the objects while minimising their total weight up to the constraint C . Therefore, being v and w the value vectors:

$$\begin{aligned} & \forall v, g \in V, G \\ & x_1, x_2, \dots, x_n \text{ and } w_1, w_2, \dots, w_n \in v_i \text{ modules_reducer} : v \rightarrow g \\ & W = \sum_{i=1}^n x_i w_i \leq C \end{aligned}$$

We can interpret the values in any way suitable to the optimisation to be performed. We may want to maximise some property while minimising others. This adaptation could be an interesting line of research.

Bibliography

- [1] S. R. Schach and A. Tomer, "A maintenance-oriented approach to software construction," *Journal of Software Maintenance: Research and Practice*, vol. 12, no. 1, pp. 25–45, 2000. xi, 5, 36
- [2] S. Schach and A. Tomer, "Development/maintenance/reuse: Software evolution in product lines," in *Proceedings of the First Conference on Software Product Lines : Experience and Research Directions: Experience and Research Directions*. Norwell, MA, USA: Kluwer Academic Publishers, 2000, pp. 437–450. xi, 5, 36
- [3] A. Khan and S. Balbo, "A tale of two methodologies: Heavyweight versus agile." [Online]. Available: <http://ausweb.scu.edu.au/aw04/papers/edited/balbo/> [Accessed: 2014-02-11] xi, 20, 21
- [4] "CodeCity." [Online]. Available: <http://www.inf.usi.ch/phd/wettel/codecity.html> [Accessed: 2013-06-07] xi, 24, 25
- [5] "The moose book: Table of contents." [Online]. Available: http://www.themoosebook.org/book/table-of-contents?_s=OafoYNdOVLUBkxVl&_k=E6gvB8TO&_n&8 [Accessed: 2013-06-19] xi, 24, 26, 32
- [6] "ERCIM news 88." [Online]. Available: <http://ercim-news.ercim.eu/en88> [Accessed: 2013-06-09] xi, xix, 24, 26, 27, 28, 29, 30, 31, 33, 34, 35, 37, 38, 45, 91
- [7] "Softwareonaut: The Book." [Online]. Available: <http://scg.unibe.ch/softwareonaut/book> [Accessed: 2013-07-01] xi, 25, 27
- [8] "FRASR - FFramework for analyzing software repositories." [Online]. Available: <http://www.frasr.org/description/> [Accessed: 2013-07-01] xi, 28, 32
- [9] "Starlink." [Online]. Available: <http://starlink.sourceforge.net/Starlink.html> [Accessed: 2013-07-01] xi, 29

- [10] S. Abrahão, J. Gonzalez-Huerta, E. Insfrán, and I. Ramos, "Software evolution in model-driven product line engineering," *ERCIM News*, vol. 2012, no. 88, 2012. [Online]. Available: <http://ercim-news.ercim.eu/en88/special/software-evolution-in-model-driven-product-line-engineering> [Accessed: 2017-01-11] xi, 37, 38
- [11] "graph_tool.topology - assessing graph topology graph-tool 2.2.28 documentation." [Online]. Available: <http://graph-tool.skewed.de/static/doc/topology.html> [Accessed: 2014-01-27] xi, 41
- [12] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976. xi, 41
- [13] "Mathematician claims breakthrough in complexity theory | Science | AAAS." [Online]. Available: <http://www.sciencemag.org/news/2015/11/mathematician-claims-breakthrough-complexity-theory> [Accessed: 2017-03-11] xi, 43
- [14] J. G. Stell, "Relations on hypergraphs," in *Proceedings of the 13th International Conference on Relational and Algebraic Methods in Computer Science*, ser. RAMiCS'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 326–341. xi, 50, 51
- [15] G. Ausiello, P. G. Franciosa, and D. Frigioni, "Directed Hypergraphs: Problems, Algorithmic Results, and a Novel Incremental Approach," in *Theoretical Computer Science*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Oct. 2001, pp. 312–328. xi, 51
- [16] B. Harvey, "Computer Science Logo Style vol 3 ch 1: Automata Theory." [Online]. Available: <https://people.eecs.berkeley.edu/~bh/v3ch1/fsm.html> [Accessed: 2016-09-11] xii, 33, 86
- [17] "Quickstart: Launch the Deployer | Google Cloud Datalab | Google Cloud Platform." [Online]. Available: <https://cloud.google.com/datalab/docs/deployer/quickstart> [Accessed: 2016-09-27] xii, xiv, xviii, 96, 100, 221
- [18] "Force-directed graph." [Online]. Available: <http://blocks.org/mbostock/4062045> [Accessed: 2014-02-13] xii, 120
- [19] M. Bostock, "Mobile Patent Suits." [Online]. Available: <https://gist.github.com/mbostock/1153292> [Accessed: 2016-09-19] xii, 118, 120, 208

- [20] "File:Logistic cdf.svg - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/File:Logistic_cdf.svg [Accessed: 2017-03-10] xiii, 143
- [21] "Microsoft .net framework." [Online]. Available: <https://www.microsoft.com/net/> [Accessed: 2017-07-20] xvii
- [22] A. v. Hoorn, *Model-Driven Online Capacity Management for Component-Based Software Systems*. Norderstedt: Books On Demand, October 2014, pp. 305–310. xvii, xviii, xix, xx, xxi, 30, 33, 34, 68, 73
- [23] S. Schulze, M. Schulze, U. Ryssel, and C. Seidl, "Aligning coevolving artifacts between software product lines and products," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, ser. VaMoS '16. New York, NY, USA: ACM, 2016, pp. 9–16. xvii, xviii, 35, 37
- [24] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, 2005th ed. Berlin u.a.: Springer, Sep 2005, p. 21. xvii, xviii, xix, 33, 37
- [25] "Getting Started - Ajax | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/AJAX/Getting_Started [Accessed: 2017-01-06] xvii
- [26] "Anaconda package list | Continuum Analytics: Documentation." [Online]. Available: <https://docs.continuum.io/anaconda/pkg-docs> [Accessed: 2016-08-24] xvii, 92
- [27] "Webassembly concepts." [Online]. Available: <https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts> [Accessed: 2017-03-06] xvii, xx, xxi, 64, 65
- [28] K. C. Loudon and Lambert, *Programming Languages: Principles and Practices*. Cengage Learning, Jan. 2011, p. 216. xvii
- [29] "Microsoft azure." [Online]. Available: <https://azure.microsoft.com/en-us/> [Accessed: 2017-07-19] xvii, 96
- [30] "bokeh." [Online]. Available: <http://bokeh.pydata.org/en/latest/> [Accessed: 2017-07-19] xvii

- [31] "Beatifulsoup." [Online]. Available: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/> [Accessed: 2017-07-19] xvii
- [32] "C# guide." [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/index> [Accessed: 2017-07-19] xvii, 92
- [33] "Using a Content Management System (CMS)." [Online]. Available: <https://www.ucl.ac.uk/building-great-websites/managing-your-website/using-a-cms> [Accessed: 2017-03-11] xvii
- [34] J. Martin, *Managing the Data Base Environment*. Pearson Education, Limited, 1983, p. 381. xvii, 73, 75
- [35] "All Standards and Drafts - W3C." [Online]. Available: <https://www.w3.org/TR/> [Accessed: 2017-01-06] xvii, xviii, xx, xxi, 30, 32, 33, 34, 36, 41
- [36] "curl command line tool and library." [Online]. Available: <https://curl.haxx.se/> [Accessed: 2017-07-19] xvii, 136
- [37] "cygwin." [Online]. Available: <https://www.cygwin.com/> [Accessed: 2017-07-19] xviii
- [38] "D3.js - data-driven documents." [Online]. Available: <http://d3js.org/> [Accessed: 2014-02-13] xviii, 118
- [39] "Linux debian." [Online]. Available: <https://debian.org> [Accessed: 2017-07-18] xviii, 68
- [40] "Emscripten." [Online]. Available: <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Emscripten> [Accessed: 2017-07-18] xviii, 64
- [41] "Flask." [Online]. Available: <http://flask.pocoo.org/> [Accessed: 2017-07-19] xviii, 97, 124
- [42] B. Khossainov and A. Nerode, *Automata Theory and its Applications*. Springer Science & Business Media, Jun. 2001, pp. 1–18,19–23,44–103. xviii, 154
- [43] "Cloud Launcher." [Online]. Available: <https://console.cloud.google.com/launcher> [Accessed: 2016-08-23] xviii

- [44] "Git - About Version Control." [Online]. Available: <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control> [Accessed: 2017-03-08] xviii, xx, xxi, 20, 24, 141
- [45] "The GraphML File Format." [Online]. Available: <http://graphml.graphdrawing.org/> [Accessed: 2017-01-06] xviii
- [46] "Licenses by Name | Open Source Initiative." [Online]. Available: <https://opensource.org/licenses/alphabetical> [Accessed: 2017-03-07] xviii
- [47] M. Pilgrim, *Greasemonkey Hacks: Tips & Tools for Remixing the Web with Firefox*. "O'Reilly Media, Inc.", 2006. xviii, 60, 61
- [48] R. T. Fielding, "REST: architectural styles and the design of network-based software architectures," Doctoral dissertation, University of California, Irvine, 2000. [Online]. Available: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> [Accessed: 2017-03-14] xviii, xx, 73, 75, 128
- [49] F. Pérez and B. E. Granger, "Ipython: A system for interactive scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 21–29, 2007. xix, 93, 120
- [50] "Jinja." [Online]. Available: <http://jinja.pocoo.org/> [Accessed: 2017-07-19] xix, 97, 118, 139
- [51] "jquery." [Online]. Available: <https://jquery.com/> [Accessed: 2017-07-19] xix, 118
- [52] "JSON." [Online]. Available: <http://www.json.org/> [Accessed: 2017-01-06] xix
- [53] "Powered By Jupyter: A Survey of the Project Ecosystem," Mar. 2016. [Online]. Available: <http://blog.ibmjstart.net/2016/03/21/powered-by-jupyter/> [Accessed: 2016-07-28] xix, 93, 97
- [54] "IPython/Jupyter project receives 6 million US\$ funding - Computational Modelling Group." [Online]. Available: <http://cmg.soton.ac.uk/news/2015/07/ipythonjupyter-project-receives-6-million-us/> [Accessed: 2016-09-27] xix, 93, 97, 120
- [55] "Linux." [Online]. Available: <https://linux.org> [Accessed: 2017-07-18] xix, 68

- [56] "The LLVM Compiler Infrastructure Project." [Online]. Available: <http://llvm.org/> [Accessed: 2017-01-06] xix, 64
- [57] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. xix, 97, 117
- [58] A. R. da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems & Structures*, vol. 43, pp. 139–155, 2015. xix, 30, 33
- [59] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Information and Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014. xix, 30, 34
- [60] "ASP.NET MVC Overview." [Online]. Available: [https://msdn.microsoft.com/en-us/library/dd381412\(v=vs.108\).aspx](https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx) [Accessed: 2017-01-06] xix, 41
- [61] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA USA, Aug. 2008, pp. 11–15. xix, 92, 97
- [62] "What is POSIX?" [Online]. Available: <https://kb.iu.edu/d/agjv> [Accessed: 2017-01-06] xx, 34
- [63] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python." [Online]. Available: <http://www.scipy.org/> [Accessed: 2016-06-22] xx, 92
- [64] M. Moreira, *Software Configuration Management Implementation Roadmap*, ser. Software Configuration Management Implementation Roadmap. Wiley, 2004, no. v. 1. xx, 32
- [65] "SHA1 version 1.0." [Online]. Available: https://www.w3.org/PICS/DSig/SHA1_1_0.html [Accessed: 2017-01-11] xx
- [66] "Mozilla technologies." [Online]. Available: <https://developer.mozilla.org/en-US/docs/Mozilla/Tech> [Accessed: 2016-09-03] xx, xxi, 60, 64, 65

- [67] “Software product lines | case studies | catalog of software product lines.” [Online]. Available: <http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm> [Accessed: 2014-02-03] xx, 4, 33, 37
- [68] C. Hursch and J. Hursch, *SQL: The Structured Query Language*, ser. TAB books. TAB Professional and Reference Books, 1988. xx, 73
- [69] “Subversion.” [Online]. Available: <https://subversion.apache.org/> [Accessed: 2017-07-18] xx, 24
- [70] “Tampermonkey.” [Online]. Available: <https://tampermonkey.net/> [Accessed: 2017-07-19] xx
- [71] “V8. google’s open source javascript engine.” [Online]. Available: <https://developers.google.com/v8/> [Accessed: 2017-07-20] xx
- [72] “VF2 algorithm. NetworkX 1.7 documentation.” [Online]. Available: <http://networkx.lanl.gov/reference/algorithms.isomorphism.vf2.html> [Accessed: 2014-02-09] xx, 44, 127
- [73] “Gnu wget.” [Online]. Available: <https://www.gnu.org/software/wget/> [Accessed: 2017-07-19] xxi, 136
- [74] “Cross-site scripting - Glossary | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Cross-site_scripting [Accessed: 2017-01-06] xxi
- [75] “The Official YAML Web Site.” [Online]. Available: <http://yaml.org/> [Accessed: 2017-01-06] xxi
- [76] R. Frigg and S. Hartmann, “Models in Science,” in *The Stanford Encyclopedia of Philosophy*, fall 2012 ed., E. N. Zalta, Ed., 2012. 2
- [77] “Evolutionary Software Development,” 2008, NATO IST-026/RTG-008. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.215.6318> [Accessed: 2017-03-14] 2, 9, 19, 21, 22
- [78] “Software factories: Assembling applications with patterns, models, frameworks, and tools,” Apr. 2013. [Online]. Available: <http://msdn.microsoft.com/en-us/library/ms954811.aspx> [Accessed: 2013-04-23] 3, 4, 5, 9, 14

- [79] J. Greenfield and K. Short, "Software factories: assembling applications with patterns, models, frameworks and tools," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 16–27. 3, 4, 5, 9, 14, 39, 40
- [80] J. White, J. A. Galindo, T. Saxena, B. Dougherty, D. Benavides, and D. C. Schmidt, "Evolving feature model configurations in software product lines," *Journal of Systems and Software*, vol. 87, pp. 119–136, 2014. 3, 35
- [81] I. Herraiz, D. Rodriguez, G. Robles, and J. M. Gonzalez-Barahona, "The evolution of the laws of software evolution: A discussion based on a systematic literature review," *ACM Comput. Surv.*, vol. 46, no. 2, pp. 1–28, Dec. 2013. 3, 64
- [82] M. Godfrey and Q. Tu, "Growth, evolution, and structural change in open source software," in *Proceedings of the 4th International Workshop on Principles of Software Evolution*, ser. IWPSE '01. New York, NY, USA: ACM, 2001, pp. 103–106. 3, 70, 71
- [83] M. W. Godfrey and D. M. German, "On the evolution of lehman's laws," *J. Softw. Evol. Process*, vol. 26, no. 7, pp. 613–619, Jul. 2014. 3, 70
- [84] P. Bhattacharya, M. Iliofotou, I. Neamtii, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 419–429. 4, 39, 143
- [85] M. Harn, V. Berzins, Luqi, and B. Shultes, "A formal model for software evolution," in *Third International Conference on Computational Intelligence and Multimedia Applications, 1999. ICCIMA '99. Proceedings, 1999*, pp. 143–147. 4, 36, 39, 47, 90, 91
- [86] M. Harn, V. Berzins, Luqi, and A. Mori, "Software evolution process via a relational hypergraph model," in *1999 IEEE/IEE/JSAI International Conference on Intelligent Transportation Systems, 1999. Proceedings, 1999*, pp. 599–604. 4, 36, 39, 47, 90, 91
- [87] B. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," Aug. 2009, pp. 44–51. 4, 41, 42

- [88] D. Simon and T. Eisenbarth, "Evolutionary Introduction of Software Product Lines," in *Proceedings of the Second International Conference on Software Product Lines*, ser. SPLC 2. London, UK, UK: Springer-Verlag, 2002, pp. 272–282. 6
- [89] G. D. Crnkovic, "Constructive Research and Info-computational Knowledge Generation," in *Model-Based Reasoning in Science and Technology*, ser. Studies in Computational Intelligence, L. Magnani, W. Carnielli, and C. Pizzi, Eds. Springer Berlin Heidelberg, 2010, no. 314, pp. 359–380, doi: 10.1007/978-3-642-15223-8_20. 7
- [90] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980. 8, 19
- [91] M. Lehman, J. Ramil, P. D. Wernick, D. Perry, and W. Turski, "Metrics and laws of software evolution-the nineties view," in *Software Metrics Symposium, 1997. Proceedings., Fourth International*, Nov. 1997, pp. 20–32. 8, 19, 142
- [92] M. M. Lehman, "Laws of software evolution revisited," in *Proceedings of the 5th European Workshop on Software Process Technology*, ser. EWSPT '96. London, UK, UK: Springer-Verlag, 1996, pp. 108–124. 8, 19
- [93] L. Northrop, "SEI's software product line tenets," *IEEE Software*, vol. 19, no. 4, pp. 32–40, Jul. 2002. 9
- [94] A.-L. Barabási, *Network Science*, 1st ed. Cambridge University Press, Aug. 2016, ch. 5. 13, 81, 144, 156
- [95] L. Osterweil, "Software processes are software too," in *Proceedings of the 9th International Conference on Software Engineering*, ser. ICSE '87. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 2–13. 13
- [96] "Anaconda package list \textbar Continuum Analytics: Documentation." [Online]. Available: <https://docs.continuum.io/anaconda/pkg-docs> [Accessed: 2016-08-24] 14
- [97] N. Vizcaino and M. Manjunathaiah, "Software evolution: a graph based model," *Lecture Notes on Software Engineering*, vol. 3, no. 3, p. 164, 2015. 16
- [98] R. S. Pressman and D. Ince, *Software engineering: a practitioner's approach*. London: McGraw-Hill, 2000. 18

- [99] B. Heller, E. Marschner, E. Rosenfeld, and J. Heer, "Visualizing collaboration and influence in the open-source software community," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 223–226. 18
- [100] N. Curtis, *Modular Web Design creating reusable components for user experience design* /Nathan Curtis. Berkeley, Ca: New Riders, 2010. 18
- [101] T. O'Reilly, "Open source paradigm shift." [Online]. Available: http://oreilly.com/tim/articles/paradigmshift_0504.html [Accessed: 2014-02-11] 18
- [102] 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 *IEEE International Conference on Software Maintenance, 2001. Proceedings, 2001*, pp. 292–300. 18, 38
- [103] J. Baragry and K. Reed, "Why is it so hard to define software architecture?" Dec. 1998, pp. 28 –36. 19
- [104] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Physical Review E*, vol. 68, no. 4, Oct 2003. [Online]. Available: <https://arxiv.org/abs/cond-mat/0305575> [Accessed: 2017-01-11] 19
- [105] M. Jazayeri, "Species evolve, individuals age," in *Eighth International Workshop on Principles of Software Evolution*, Sep. 2005, pp. 3 – 9. 19
- [106] A. Marco, F. Gallo, P. Inverardi, and R. Ippoliti, "Towards a stem architecture description language for self-adaptive systems," in *2010 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, Oct. 2010, pp. 269–270. 19
- [107] G. Xie, J. Chen, and I. Neamtii, "Towards a better understanding of software evolution: An empirical study on open source software," in *IEEE International Conference on Software Maintenance, 2009. ICSM 2009*, Sep. 2009, pp. 51–60. 20
- [108] M. N. K. Boulos, I. Maramba, and S. Wheeler, "Wikis, blogs and podcasts: a new generation of web-based tools for virtual collaborative clinical practice and education," *BMC medical education*, vol. 6, no. 1, p. 41, 2006. 20

- [109] F. Keenan, "Agile process tailoring and problem analysis (APPLY)," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 45–47. 20
- [110] V. Rajlich, "Changing the paradigm of software engineering," *Commun. ACM*, vol. 49, no. 8, pp. 67–70, Aug. 2006. 20
- [111] O. S. P. Abrahamsson, "Agile software development methods," *Relatório Técnico, Finlandia*, 2002. 21
- [112] "Microformats part 2: The fundamental types. alex faaborg," <http://blog.mozilla.org/faaborg/2006/12/13/microformats-part-2-the-fundamental-types/>. [Online]. Available: <http://blog.mozilla.org/faaborg/2006/12/13/microformats-part-2-the-fundamental-types/> [Accessed: 2012-04-12] 21
- [113] "Introduction to software Engineering/Architecture/Anti-Patterns - wikibooks, open books for an open world." [Online]. Available: http://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Architecture/Anti-Patterns [Accessed: 2014-02-11] 21
- [114] N. N. Taleb, *Antifragile: Things that Gain from Disorder*. Penguin, Nov. 2012. 22
- [115] J. Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Addison-Wesley, Jul. 2013, pp. 273–274. 22
- [116] S. Cook, H. Ji, and R. Harrison, "Software evolution and software evolvability," pp. 1–12, 2000, DOI 10.1.1.21.9272. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?sessionid=A9BBD33EBA9045E996A226F794D2C0E3?doi=10.1.1.21.9272> [Accessed: 2017-03-14] 22, 35
- [117] "Reveal group." [Online]. Available: <http://www.inf.usi.ch/lanza/reveal.html> [Accessed: 2017-07-18] 24
- [118] M. Meyer, T. Gîrba, and M. Lungu, "Mondrian: An agile information visualization framework," in *Proceedings of the 2006 ACM Symposium on Software Visualization*, ser. SoftVis '06. New York, NY, USA: ACM, 2006, pp. 135–144. 24

- [119] W. Poncin, A. Serebrenik, and M. v. d. Brand, "Process mining software repositories," in *2011 15th European Conference on Software Maintenance and Reengineering*, March 2011, pp. 5–14. 28, 32
- [120] "INRIA lille nord europe - ADAM team web site: Topics/SPL." [Online]. Available: <http://adam.lille.inria.fr/pmwiki.php/Topics/SPL> [Accessed: 2013-07-09] 29
- [121] "Rascal - WebHome." [Online]. Available: <http://www.rascal-mpl.org/> [Accessed: 2013-07-09] 31
- [122] "Smalltalk." [Online]. Available: <http://web.cecs.pdx.edu/~harry/musings/SmalltalkOverview.html> [Accessed: 2017-07-18] 32
- [123] "Argouml." [Online]. Available: <http://argouml.tigris.org/> [Accessed: 2017-07-18] 32
- [124] "The java tutorials." [Online]. Available: <http://docs.oracle.com/javase/tutorial/> [Accessed: 2017-07-18] 33, 34
- [125] "Eclipse." [Online]. Available: <https://eclipse.org/home/> [Accessed: 2017-07-18] 34
- [126] "Iso c++." [Online]. Available: <https://isocpp.org/> [Accessed: 2017-07-18] 34, 64, 66
- [127] "Python." [Online]. Available: <https://python.org/> [Accessed: 2017-07-18] 34, 64, 91
- [128] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 419–429. 35
- [129] 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 *IEEE International Conference on Software Maintenance, 2001. Proceedings, 2001*, pp. 292–300. 35
- [130] J. M. Barnes, D. Garlan, and B. Schmerl, "Evolution styles: foundations and models for software architecture evolution," *Software & Systems Modeling*, vol. 13, no. 2, pp. 649–678, 2014. 35

- [131] J. I. Maletic, M. L. Collard, and B. Simoes, "An xml based approach to support the evolution of model-to-model traceability links," in *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, ser. TEFSE '05. New York, NY, USA: ACM, 2005, pp. 67–72. 36, 76, 80
- [132] T. Mens and J. Klein, "Evolving software - introduction to the special theme," *ERCIM News*, vol. 2012, no. 88, 2012. [Online]. Available: <http://ercim-news.ercim.eu/en88/special/evolving-software-introduction-to-the-special-theme> [Accessed: 2017-01-11] 38, 39
- [133] "Approaches to solving the graph isomorphism problem," Nov. 2012. [Online]. Available: <http://smartech.gatech.edu/handle/1853/6494> [Accessed: 2012-11-29] 45
- [134] R. Kazman, "Assessing architectural complexity," in *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, 1998, Mar 1998, pp. 104–112. 48
- [135] E. Weisstein, "Partial Order." [Online]. Available: <http://mathworld.wolfram.com/PartialOrder.html> [Accessed: 2014-12-01] 48
- [136] E. W. Weisstein, "Equivalence Class." [Online]. Available: <http://mathworld.wolfram.com/EquivalenceClass.html> [Accessed: 2014-12-01] 49
- [137] A. Bretto, *Hypergraph Theory: An Introduction*. Springer Science & Business Media, Apr. 2013. 50
- [138] E. Pafilis, S. I. O'Donoghue, L. J. Jensen, H. Horn, M. Kuhn, N. P. Brown, and R. Schneider, "Reflect: augmented browsing for the life scientist," *Nature Biotechnology*, vol. 27, no. 6, pp. 508–510, Jun. 2009. 59, 122
- [139] "Zotero." [Online]. Available: <https://www.zotero.org/> [Accessed: 2016-08-24] 59
- [140] "The Mozilla platform." [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/The_Mozilla_platform [Accessed: 2016-09-03] 60, 65

- [141] R. Filman, "Taking back the Web," *IEEE Internet Computing*, vol. 10, no. 1, pp. 3–5, Jan. 2006. 61
- [142] "The c programming language." [Online]. Available: <http://groups.engin.umd.umich.edu/CIS/course.des/cis400/c/c.html> [Accessed: 2017-07-18] 66
- [143] P. T. Endo, G. E. Gonçalves, J. Kelner, and D. Sadok, "A survey on open-source cloud computing solutions," in *Brazilian Symposium on Computer Networks and Distributed Systems*, 2010. 67
- [144] "Google compute engine." [Online]. Available: <https://cloud.google.com/compute/> [Accessed: 2017-07-18] 68
- [145] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. Pearson Education, 2000, pp. 165–166,174. 73
- [146] G. Gigerenzer, "Why Heuristics Work," *Perspectives on Psychological Science*, vol. 3, no. 1, pp. 20–29, Jan. 2008. 73
- [147] J. Bentley, D. Knuth, and D. McIlroy, "Programming pearls: A literate program," *Commun. ACM*, vol. 29, no. 6, pp. 471–483, Jun. 1986. 74
- [148] "Free pascal reference guide." [Online]. Available: <https://www.freepascal.org/docs-html/current/ref/ref.html#QQ2-7-6> [Accessed: 2017-07-18] 74
- [149] "Relational Calculus." [Online]. Available: <http://jcsites.juniata.edu/faculty/rhodes/dbms/reldcalc.htm> [Accessed: 2016-09-11] 75
- [150] N. H. Madhavji, J. Fernandez-Ramil, and D. Perry, *Software Evolution and Feedback: Theory and Practice*. John Wiley & Sons, Aug. 2006, pp. 72–75. 76
- [151] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. Pearson Education, 2000, p. 173. 76
- [152] "Docker - Build, Ship, and Run Any App, Anywhere." [Online]. Available: <https://www.docker.com/> [Accessed: 2016-09-27] 96
- [153] D. Cunliffe, C. Taylor, and D. Tudhope, "Query-based navigation in semantically indexed hypermedia," in *Proceedings of the Eighth ACM Conference on Hypertext*, ser. HYPERTEXT '97. New York, NY, USA: ACM, 1997, pp. 87–95. 122

- [154] O. Díaz and C. Arellano, “The augmented web: Rationales, opportunities, and challenges on browser-side transcoding,” *ACM Trans. Web*, vol. 9, no. 2, pp. 8:1–8:30, May 2015. 122
- [155] “Redis.” [Online]. Available: <https://redis.io/> [Accessed: 2017-07-19] 124
- [156] “Logistic Functions.” [Online]. Available: http://wmueller.com/precalculus/families/1_80.html [Accessed: 2016-08-31] 142
- [157] W. Scacchi, “Understanding Open Source Software Evolution 181,” in *Software Evolution and Feedback*, N. H. d. Chair, J. C. F.-R. B. Lecturer, MEM, and D. E. P. C. D. Co-Editor, Eds. John Wiley & Sons, Ltd, 2006, ch. 9, pp. 181–205. 142
- [158] S. Valverde and R. V. Sole, “Hierarchical Small Worlds in Software Architecture,” *arXiv:cond-mat/0307278*, Jul. 2003, arXiv: cond-mat/0307278. [Online]. Available: <http://arxiv.org/abs/cond-mat/0307278> [Accessed: 2016-08-20] 144
- [159] A.-L. Barabási, *Network Science*, 1st ed. Cambridge University Press, Aug. 2016, ch. 10. 144
- [160] K. Nguyen and D. A. Tran, “Fitness-Based Generative Models for Power-Law Networks,” in *Handbook of Optimization in Complex Networks*, ser. Springer Optimization and Its Applications, M. T. Thai and P. M. Pardalos, Eds. Springer US, 2012, no. 57, pp. 39–53, doi: 10.1007/978-1-4614-0754-6_2. 144, 156
- [161] T. Chaikalas and A. Chatzigeorgiou, “Forecasting java software evolution trends employing network models,” *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 582–602, 2015. 144
- [162] V. Musco, M. Monperrus, and P. Preux, “A Generative Model of Software Dependency Graphs to Better Understand Software Evolution,” *arXiv:1410.7921 [cs]*, Oct. 2014, arXiv: 1410.7921. [Online]. Available: <http://arxiv.org/abs/1410.7921> [Accessed: 2016-09-26] 149
- [163] D. Shiffman, “The Nature of Code.” [Online]. Available: <http://natureofcode.com/book/chapter-8-fractals/> [Accessed: 2016-09-12] 159

-
- [164] E. W. Weisstein, "Plane-Filling Function." [Online]. Available: <http://mathworld.wolfram.com/Plane-FillingFunction.html> [Accessed: 2016-09-12] 159
- [165] J. Song, F. Kui, and C. Yan, "Model Based on the L-System's Binary Tree Structure and Its Application," *Procedia Engineering*, vol. 15, pp. 4446–4450, Jan. 2011. 159
- [166] M. Mitchell, *An Introduction to Genetic Algorithms*, new edition ed. MIT Press, Apr. 1998. 164
- [167] B. A. Julstrom, "Greedy, genetic, and greedy genetic algorithms for the quadratic knapsack problem," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 607–614. 165

Appendices

Appendices

NOTE: Sources edited manually for line-break adjustment. This usually follow some logic. In the case of python scripts, in a pythonic way (any { [()] }).

A.1 Prototype notebook examples

A.2 Python files

A.2.1 csv2assets.py

```
1  #!/usr/bin/env python3
2  """
3  Utility module to build a XML configuration fragment from custom a CSV file of asset data.
4  These configuration can be layered and only overwrite node metadata on tag name clash.
5  The subset categories used is naive but serves as illustration. Further and more meaningful
6  subsets can be created. Also serves as example how configurations can be customised.
7  """
8  __author__ = 'Noel Vizcaino'
9
10 import csv
11 import xml.etree.ElementTree as customTree
12 import xml.dom.minidom as dom
13 from model import Artefact as subspace
14
15 GENERATOR_MESSAGE = 'Generated by csv2assets.py. \n' \
16                     'Google Cloud Launcher features popular \n' \
17                     'open source packages that have been configured by \n' \
18                     'Bitnami or Google Click for easy deployment.'
19
20 infile = r'.\GoogleLauncher\googlelauncher.csv'
21 outfile = r'.\GoogleLauncher\googlelauncher_layer.xml'
22 XML_RELATION_TAG = subspace.expansion_relation_tag
23 XFILTER = 'Standard'
24
25 def googlecsv2assets():
```

```

26     """
27     Writes the csv encoded assets into a XML assets file.
28     It features many options from the model like creating a graph from
29     the set of the cartesian product of a reduced assets set.
30     """
31     root = customTree.Element('assets')
32     root.set('name', 'Google Cloud Launcher Assets')
33     root.append(customTree.Comment(GENERATOR_MESSAGE))
34     with open(infile, encoding='UTF-8' ) as f:
35         reader = csv.reader(f,delimiter='|')
36         for row in reader:
37             link, name, icon, description = row
38             if '?cat=INFRASTRUCTURE' in link:
39                 category = 'Infrastructure'
40             elif '?cat=OS' in link:
41                 category = 'OS'
42             elif '?cat=DATABASE' in link:
43                 category = 'Database'
44             elif '?cat=BLOG' in link:
45                 category = 'Bloggging'
46             elif '?cat=CMS' in link:
47                 category = 'CMS'
48             elif '?cat=CRM' in link:
49                 category = 'CRM'
50             elif '?cat=DEVELOPER_TOOLS' in link:
51                 category = 'Developer tools'
52             elif '?cat=OTHERS' in link:
53                 category = 'Other'
54             else:
55                 category='Unknown origin'
56             customTree.SubElement(root, 'asset',{ 'icon':icon,
57                                                    'link':link,
58                                                    'description':description,
59                                                    'category':category,
60                                                    'name':name.strip(),
61                                                    })
62         for valid_category in ['OS','Database','Bloggging','CMS','CRM','Developer tools']:
63             root = relation2tag(root,valid_category, XFILTER)
64         #NOTE:minidom changes the order or the attributes, only annoying for humans
65         xml = dom.parseString(customTree.tostring(root)).toprettyxml(indent="  ")
66         #print(ET.tostring(root))
67         #print(xml)
68     with open(outfile, "w", encoding='UTF-8') as fout:

```

```

69         fout.write(xml)
70         print('Done!')
71
72 def relation2tag(root,category, xfilter):
73     """
74     Cartesian product of the asset set with itself using valid categories.
75     Other more meaningful relational subsets could be created.
76     :param root: root of the xml tree
77     :param category: a valid category
78     :param xfilter: a undesired string
79     :return: amended xml tree
80     """
81     for assetA in root.iter('asset'):
82         nameA = assetA.get('name')
83         catA = assetA.get('category')
84         for assetB in root.iter('asset'):
85             nameB = assetB.get('name')
86             catB = assetB.get('category')
87             if catA in category and catB in category:
88                 if nameA!=nameB and xfilter not in nameA and xfilter not in nameB:
89                     isonode = customTree.SubElement(assetA, XML_RELATION_TAG)
90                     isonode.text= nameB
91     return root
92
93 if __name__ == '__main__':
94     googlecsv2assets()

```

A.2.2 launcherharvest.py

```

1  #!/usr/bin/env python3
2  """
3  Utility module to parse Google Cloud Launcher item cell subtree structure as
4  of sept 2016 (fetch online or offline)
5  to build a custom CSV file of asset data.
6  """
7  __author__ = 'Noel Vizcaino'
8
9  import urllib.request, urllib.error
10 from bs4 import BeautifulSoup
11 import ssl
12
13
14 XML_PARSER = "lxml"
15 offlinepath = r'.\GoogleLauncher\Cloud Launcher Marketplace Solutions.html'

```

```

16 onlineURL = 'https://console.cloud.google.com/launcher?q=*'
17 google_launcher_cell_tag = 'a'
18 google_launcher_cell_class = 'p6n-mp-solution-card-link'
19 csv_filename = r'.\GoogleLauncher\googlelauncher.csv'
20 csv_separator = '|'
21
22 def harvest_online(url):
23     """
24     Get text/HTML resource as utf-8 from site url
25     :param url: url of the site to harvest or scrub
26     :return: text/html
27     """
28     try:
29         ctx = ssl.create_default_context()
30         ctx.check_hostname = False
31         ctx.verify_mode = ssl.CERT_NONE
32         # user_agent = 'Mozilla/5.0 (iPad; CPU OS 6_0 like Mac OS X) AppleWebKit/536.26' \
33             #'(KHTML, like Gecko) Version/6.0 Mobile/10A5355d Safari/8536.25'
34         user_agent = 'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:40.0)' \
35             'Gecko/20100101 Firefox/48.0'
36         response = urllib.request.urlopen(urllib.request.Request(
37             url, headers={'User-Agent': user_agent}), ctx)
38         html = response.read().decode('utf-8')
39         return html
40     except (urllib.error.HTTPError, urllib.error.URLError, ssl.SSLError):
41         print('=====[ Online access error.
42             Try offline by fetching the file ]====')
43         return None
44     finally:
45         print('=====[ Online access errors.
46             Try offline by fetching the file ]====')
47         return None
48
49
50 def html_parse(html, tag, class_):
51     """
52     Parser of the current Google Cloud Launcher item cell subtree structure as of
53     sept 2016 (saved offline for convenience)
54     name class-->
55         ['p6n-space-below-xs p6n-color-primary ' \
56         'p6n-mp-solution-card-text-name'].string
57         -->h3
58     icon class-->

```



```

59         ['p6n-mp-solution-card-icon-image']['src']
60         -->img
61         description
62         class-->['p6n-space-last p6n-color-secondary' |
63         ' p6n-mp-solution-card-description'].string]
64         -->div
65
66     :param html:
67     :param tag: tag of the HTML element of interest
68     :param class_: class of the HTML element of interest
69     :return: list of rows with fields separated by csv_separator
70     """
71     lines=[]
72     soup = BeautifulSoup(html, XML_PARSER)
73     cells = soup.find_all(tag, attrs={"class": class_})
74     if cells:
75         print('#:',len(cells))
76         for link in cells:
77             #tmp=(repr(item).replace("'", ''))
78             print(repr(link))
79             fields =[link['href'],#a
80                     link.h3.string,#h3
81                     link.img['src'],#img
82                     link.find('div',class_='p6n-space-last p6n-color-secondary p6n-mp-
83                             solution-card-description').string]#div
84             line = csv_separator.join(fields)
85             lines.append(line)
86         return lines
87
88 def googlelauncher2csv():
89     """
90     Fetches and/or scrubs html data from Google Cloud Launcher as example of the
91     possibilities to generate
92     asset sets.
93     """
94     with open(csv_filename, 'w', encoding='utf-8') as fout:
95         with open(offlinepath, encoding='utf-8') as fin:
96             text = harvest_online(onlineURL)
97             if not text:
98                 text = fin.read()
99             fout.write('link|name|icon|description' + '\n')
100            for row in html_parse(text, google_launcher_cell_tag, google_launcher_cell_class):
101                if row:

```

```
102         fout.write(row + '\n')
103
104
105 if __name__ == '__main__':
106     googlelauncher2csv()
```

artefact_views_test_cleaned.py

A.2.3 microservice.py

```
1  #!/usr/bin/env python3
2  """
3  Attempt to implement a example cloud microservice to illustrate how to visualise
4  (configured) artefact
5  instances saved as JSON files on the server.
6
7  NOTICE:artefact names and IDs are equivalence via URL for convenience.
8  The logical thing in practice
9  would be to saved them as hash-id.json files and ignore the names.
10
11 *Views include its originator, because it is a view.*
12
13 When running open http://127.0.0.1:5000/artefact/seed/viewlist for available
14 view list.
15 to initialise and see a saved (naive) sample seed 'artefact.json' and others.
16
17 There is normal operation or operation based on a plot_relation pre-made
18 scenario. Switch with:
19     * http://127.0.0.1:5000/normal_scenario (default on init)
20     * http://127.0.0.1:5000/test_scenario (where some normal operations
21       were tested online)
22
23 Test sample operation evolve using POST (in practice JSON with any AJAX exchange
24 would do):
25     * http://127.0.0.1:5000/operationtest
26
27 IT should be noted that only operations and ID would be transmitted except on
28 creation where the new
29 artefact would have to be downloaded to the client.
30
31 Operations related to HTTP verbs POST PUT DELETE would go here:
32     * http://127.0.0.1:5000/api
33
34 Resources access:
```

```

35
36 =====
37                                URL
38 =====
39 http://127.0.0.1:5000/<serverartefact>/id
40 http://127.0.0.1:5000/<serverartefact>/<viewname>
41 http://127.0.0.1:5000/<serverartefact>/<viewname>/JSON
42 http://127.0.0.1:5000/<serverartefact>/<viewname>/info
43 http://127.0.0.1:5000/<serverartefact>/viewlist
44 http://127.0.0.1:5000/<serverartefact>/tileviews
45 http://127.0.0.1:5000/<serverartefact>/info
46 http://127.0.0.1:5000/<serverartefact>/metadata/<tag>
47 =====
48
49 =====
50                                Resources
51 =====
52 SHA-1 hash (40 char), just as ID
53 Renders a graph of any available view
54 The graph with metadata as JSON
55 Basic textual information for any views
56 Equivalence class known views
57 Graphs take a time to load if there are too many. Experimental.
58 Basic textual information from
59 Tags used in XML configurations
60 =====
61
62 This is also an example of how to remote assets as ReST resources. They can be
63 fetched with wget or curl too.
64 It tries to honour the ReSTful API style AFAIK.
65
66 Browsers:
67 Chromium v44 seems to be more responsive than Firefox v48.
68
69 Further testing and experimenting:
70     * curl is a nice tool similar to wget, we can do some testing from the
71       command line.
72     * $curl --header "Content-type: application/json"
73         POST http://127.0.0.1:5000/api --data "@operation.json"
74
75 Truth is only the IDs and operators need to be transmitted for maximum
76 communication efficiency.
77 The artefacts should be downloaded just once by the client. From there,

```

```
78     update progressively on creation.
79     This microservice can be operated by Jupyter to create nice and shareable
80     development narratives.
81     """
82     __author__ = 'Noel Vizcaino'
83
84     DEFAULT_HOST = '127.0.0.1'
85     #Make sure the port is open!!
86     MICROSERVICE_PORT = 5000
87
88
89     from flask import Flask, render_template, jsonify, abort, request
90     from model import Artefact as subspace
91     import ioutils as io
92     import os
93
94     app = Flask(__name__)
95     ARTEFACT_PATH = r'.\artefacts\'
96     DEFAULT_SEED = 'evolved'
97
98     class Scenario:
99         """
100         Simple strategy pattern to account for the tests scenarios vs normal operation
101         """
102         def __init__(self, function=None):
103             """
104             Initialises the default scenario
105
106             :param function: the function to use as replacement
107             """
108             if function:
109                 self.use = function
110
111         def use(self, artefact_json_name):
112             return normal_scenario(artefact_json_name)
113
114
115     def test_scenario(artefact_json_name):
116         """
117         Simple Model and API online testing.
118
119         :param artefact_json_name: the name on disk of the seed artefact
120         :return: sample artefact
```

```

121     """
122     seed = load_JSON_artefact(DEFAULT_SEED)
123     seed.evolve(subspace.from_relationships([('CustomBinary1', 'Redis')],
124         'rediscoupler'), 'redisexpansion')
125     solution = seed.evolve_delta(subspace.from_relationships([('CustomBinary1',
126         'bs4')], 'bs4coupler'), 'bs4expansion')
127     solution.bind_context_fragment('views.xml') # or any not included before
128         #saving in any JSON artefact
129     #isonodes or properties must be updated together via XML, otherwise the
130         #other data persists as is not overridden
131     solution.bind_context_fragment(r'.\GoogleLauncher\googlelauncher_layer.xml')
132     solution.update_views()
133     return solution
134
135 def normal_scenario(artefact_json_name):
136     """
137     Loads a store JSON artefact and updates the expanded
138         :math:[artefact \setminus graph] \setminus simeq \text{views}
139
140     :param artefact_json_name: the name of the stored JSON artefact
141     :return: the artefact with its views updated
142     """
143     art = load_JSON_artefact(artefact_json_name)
144     art.update_views()
145     return art
146
147 #default scenario
148 active_equivalence_class = Scenario(normal_scenario)
149
150 @app.route('/scenario/<function_name>')
151 def setscenario(function_name):
152     """
153     Sets the scenario to be ANY predefined one without touching the source code.
154     NOTICE: Not ReSTful!, just for convenience not purity. This should be added
155         as part of any API using HTTP verbs.
156
157     :param function_name: the name of the function of the scenario to use
158     :return: Confirmation
159     """
160     global active_equivalence_class
161     active_equivalence_class = Scenario(globals()[function_name])
162     return 'Scenario {} is set.'.format(function_name)
163

```

```

164 def load_JSON_artefact(name):
165     """
166     Loads JSON file produced with the same API format. JSON carries metadata
167     along but not evolved step marks
168     or red arrows.
169
170     :param name: Name used in the JSON filename
171     :return: The fully configured seed artefact
172     """
173     with io.load_graph_context(io.load_json(ARTEFACT_PATH+name+'.json')) as g:
174         seed =subspace.from_graph(g, name)
175         return seed
176
177 @app.route("/operationtest")
178 def operationstest():
179     """
180     Naive sample operation how-to plot_relation
181
182     :return: form to evolve any two JSON stored artefacts
183     """
184     return ('<html><head><style></style></head>'
185            '<body><div>This only works with artefact names on disk as JSON<br/>'
186            '<form action="/api" method="POST">'
187            '<label for="seed">Seed : </label><input name="seed"> '
188            '<label for="coupler">Coupler : </label><input name="coupler">'
189            '<input type="submit" value="evolve"></form></div></body></html>')
190
191 @app.route('/api', methods=['POST'])
192 def operations():
193     """
194     Evolve two artefacts by (JSON) name (It should use ID at later stage)
195
196     :return: resulting dynamic graph view
197     """
198     seed = active_equivalence_class.use(request.form['seed'])
199     coupler = active_equivalence_class.use(request.form['coupler'])
200     view = seed.evolve(coupler,seed.name+'_delta_'+coupler.name)
201     #return request.form['seed']+ '_delta_' +request.form['coupler']+ ' = '+ view.id
202     io.dump_json(ARTEFACT_PATH+view.id+'.json',view.graph)
203     # another fancy option
204     return render_template('viewtemplate.html', d3data = io.d3_data(view))
205
206 @app.route('/<server_artefact>/id')

```

```
207 def getID(server_artefact):
208     """
209
210     :param server_artefact: file basename of stored artefact
211     :return: the SHA-1 hash of the name as stored
212     """
213     return active_equivalence_class.use(server_artefact).id
214
215 @app.route('/<server_artefact>/<viewname>')
216 def artefact_view(server_artefact,viewname):
217     """
218     Shows the corresponding dynamic view.
219
220     :param server_artefact: file basename of stored artefact
221     :param viewname: instance name or ID
222     :return: A jinja2 template implementing a D3 Force graph with view data
223     """
224     #parsedname=urlllib.parse.unquote(viewname)
225     for view in active_equivalence_class.use(server_artefact).views:
226         if view.name==viewname or view.id==viewname:
227             return render_template('viewtemplate.html', d3data = io.d3_data(view))
228
229 @app.route('/<server_artefact>/<viewname>/JSON')
230 def artefact_view_JSON(server_artefact,viewname):
231     """
232     JSON corresponding to a view
233
234     :param server_artefact: file basename of stored artefact
235     :param viewname: instance name or ID
236     :return: text/JSON
237     """
238     #parsedname=urlllib.parse.unquote(viewname)
239     for view in active_equivalence_class.use(server_artefact).views:
240         if view.name==viewname or view.id==viewname:
241             return str(io.to_json(view.graph))
242
243 @app.route('/<server_artefact>/viewlist')
244 def artefact_viewlist(server_artefact):
245     """
246     Browser quirks mode will help display the list of artefact views
247
248     :param server_artefact: file basename of stored artefact
249     :return: List of seed views currently configured and their IDs
```

```

250     """
251     #parsedname=urlib.parse.unquote(viewname)
252     return ('<br/>\n'.join(['<a href="http://{}/{}/{}>ID:{}</a><br/>\n'
253                             'Name:{}'.format(DEFAULT_HOST,
254                                             MICROSERVICE_PORT,
255                                             server_artefact,
256                                             view.id,
257                                             view.id,
258                                             view.name)
259                                     for view in active_equivalence_class.
260                                     use(server_artefact).views]))
261
262 @app.route('/<server_artefact>/tileviews')
263 def tiled_views(server_artefact):
264     """
265     Attempt to create a tile view of current available views using iframes.
266     IDs work when dragged from source
267     but not when clicked. Bugged. I suspect it does not scale well.
268
269     :param server_artefact: file basename of stored artefact
270     :return: A jinja2 template with iframes. Each loads another template
271             implementing a D3 Force graph with view data
272     """
273     return render_template('tiletemplate.html',
274                             solutionviews =
275                             active_equivalence_class.use(server_artefact).views,
276                             artefact=server_artefact)
277
278 @app.route('/<server_artefact>/info')
279 def seed_info(server_artefact):
280     """
281     Basic artefact text information to self pretty-print on the web
282     as a naive stdweb ;)
283
284     :param server_artefact: file basename of stored artefact
285     :return: artefact web friendly textual representation
286     """
287     return str(active_equivalence_class.use(server_artefact)).replace('\n', '<br/>\n')
288
289 @app.route('/<server_artefact>/<viewname>/info')
290 def artefact_info(server_artefact,viewname):
291     """
292     Basic artefact view text information to self pretty-print on the web

```



```

293
294     :param server_artefact: file basename of stored artefact
295     :param viewname: artefact name or hash ID
296     :return: artefact web friendly textual representation
297     """
298     for view in active_equivalence_class.use(server_artefact).views:
299         if view.name==viewname or view.id==viewname:
300             return str(view).replace('\n','<br/>\n')
301
302 @app.route('/<server_artefact>/metadata/<tag>')
303 def artefact_tag(server_artefact,tag):
304     """
305     Artefact stored metadata by tag
306
307     :param server_artefact: file basename of stored artefact
308     :param tag: tag used by any configuration fragment. Remember past layering as
309                 could have been overwritten.
310     :return: artefact web friendly textual representation of the metadata
311     """
312     return str(active_equivalence_class.use(
313                                     server_artefact).mapinfo(tag)).replace('\n','<

```

A.2.4 ioutils.py

```

1  #!/usr/bin/env python3
2  """
3  Utility module dedicated to input/output helper functions.
4  These comprise stdout (console output) printing functions
5  and disk IO methods for various data formats.
6  """
7  __author__ = 'Noel Vizcaino'
8
9  import json
10 from networkx.readwrite import json_graph
11 import networkx as nx
12 import matplotlib.pyplot as plt
13 import xml.etree.ElementTree as element_tree
14 #from xml.dom import minidom
15 from contextlib import contextmanager
16 #import os
17
18 DEFAULT_CONTEXT_FILE = 'assets.xml'
19 DEFAULT_NODE_SIZE = 300
20

```

```
21 def dump_json( filename, g):
22     """
23     Writes graph g in JSON format to a new file with a name filename in default
24     working directory.
25     This saves the state of the graph, including metadata loaded.
26
27     :param filename: the name of the file
28     :param g: NetworkX graph
29     """
30     data = to_json(g)
31     with open(filename, 'w', encoding='UTF-8') as fout:
32         fout.write(json.dumps(data))
33
34
35 def to_json(graph):
36     """
37     Returns JSON data encoding of the graph
38
39     :param graph: NetworkX graph
40     :return: JSON data
41     """
42     data = json_graph.node_link_data(graph)
43     return data
44
45
46 def load_json( filename):
47     """
48     Loads a graph g from a JSON file
49
50     :param filename: name of the JSON file
51     :return: a new NetworkX graph built from the data
52     """
53     with open(filename, 'r', encoding='UTF-8') as fin:
54         g = json_graph.node_link_graph(json.load(fin))
55     return g
56
57 def save_png(solution, filename):
58     """
59     Saves a PNG snapshot of the drawn solution from the plot window
60
61     :param solution: artefact instance
62     :param filename: filename of the PNG image
63     """
```

```
64     nx.draw(solution.graph, pos=nx.spring_layout(solution.graph),
65             node_size=plot_sizes(solution), with_labels=True )
66     plt.savefig(filename)
67     plt.clf()
68
69 def print_adjacency_matrix(graph):
70     """
71     Prints to stdout the adjacency matrix corresponding to the graph
72
73     :param graph: a NetworkX graph
74     :return: matrix text stdout print
75     """
76     print(nx.adjacency_matrix(graph))
77
78 def show(solution, tag):
79     """
80     Draws the solution on a Matplotlib plot window
81
82     :param solution: artefact instance
83     :param tag: tag from a context to base the size of the nodes on
84     """
85     pos=nx.spring_layout(solution.graph)
86     nx.draw(solution.graph, pos, node_size=plot_sizes(solution, tag), with_labels=True )
87     nx.draw_networkx_edges(solution.graph, pos,
88                           edgelist=[],
89                           width=4, alpha=0.5, edge_color='r')
90     plt.show()
91     plt.clf()
92
93 def show_graph(graph):
94     """
95     Draws any NetworkX graph on a Matplotlib plot window
96
97     :param graph:
98     :return:
99     """
100    nx.draw_spring(graph, with_labels=True)
101    plt.show()
102    plt.clf()
103
104 def plot_sizes(solution, tag):
105     """
106     Calculates node sizes for plotting on a Matplotlib plot window
```

```

107
108     :param solution: artefact instance
109     :param tag: tag from a context to calculate the actual rendering size of the nodes
110     """
111     sizes = [DEFAULT_NODE_SIZE*(1+size) for size in solution.count(tag)]
112     return sizes
113
114 def show_multiple_artefacts(solutions, tag):
115     """
116     Draws a solution list on several simultaneously spawning Matplotlib plot windows
117
118     :param solutions: artefact instance list
119     :param tag: tag from a context to base the size of the nodes on
120     """
121     for i,solution in enumerate(solutions):
122         plt.figure(i)
123         pos=nx.spring_layout(solution.graph)
124         nx.draw(solution.graph,pos,node_size=plot_sizes(solution, tag),
125                with_labels=True )
126     plt.show()
127     plt.clf()
128
129 @contextmanager
130 def load_graph_context(graph, filename=DEFAULT_CONTEXT_FILE):
131     """
132     Populates a graph with the node metadata specified by the context in
133     (fullpath)file filename
134
135     :param graph: NetworkX graph to populate
136     :param filename: XML file with context data
137     :returns: a generator yielding to the populated graph
138     """
139     tree = element_tree.parse(filename)
140     root = tree.getroot()
141     for asset in root.findall('asset'):
142         assetname = asset.get('name')
143         if assetname in graph:
144             tags = list(set([ child.tag for child in asset]))
145             if tags:
146                 for tag in tags:
147                     nx.set_node_attributes(graph, tag, {assetname:
148                    [item.text.strip() for item in asset.findall(tag)]})
149     yield graph

```

```
150
151 def print_d3_data(solution):
152     """
153     Prints custom formatted data to be loaded in JavaScript
154
155     :param solution: artefact instance
156     """
157     data = d3_data(solution)
158     print(data)
159
160 def write_d3js(solution, filename='customdata.js'):
161     """
162     Writes a JavaScript file containing custom formatted data
163
164     :param solution: artefact instance
165     :param filename: name of the JavaScript file, default is customdata.js
166     """
167     with open(filename, 'w', encoding='UTF-8') as fout:
168         fout.writelines(d3_data(solution))
169
170 def d3_data(solution):
171     """
172     Generates customised text data (a JavaScript source code snippet)
173     based on the artefact graph
174
175     :param solution: artefact instance
176     :return: JavaScript source code snippet
177     """
178     tmp=[]
179     for i,e in enumerate(solution.graph.edges()):
180         if e in solution.evolved_edges:
181             line = 'source: "{source}" , target:"{target}", type: "{type}"'.format(
182                 source=e[0],target = e[1],type='coupler_added')
183         else:
184             line = 'source: "{source}" , target:"{target}", type: "{type}"'.format(
185                 source=e[0],target = e[1],type='dependency')
186         tmp.append('{'+line+'}')
187     out= ',\n'.join(tmp)
188     return 'var links = ['+out+'];'
189
190 def print_pairs(graph):
191     """
192     Prints the edges of a graph as key and value pairs
```

```

193
194     :param graph: A NetworkX graph
195     """
196     for k,v in graph.items():
197         print(k,v)
198
199 def print_graph(graph):
200     """
201     Prints the nodes in the graph one by one
202
203     :param graph: A NetworkX graph
204     """
205     for node in graph:
206         print('Node: ',repr(node))
207
208 def write_evolved_tile_views(seed, coupler, filename, div=True):
209     """
210     Generates HTML file with iframes based for assist in the viewing of graphs
211     as a tile
212
213     :param seed: artefact instance
214     :param coupler: artefact instance with coupler graph
215     :param filename: HTML file
216     :param div: If div tag is needed, default is True
217     """
218     # write_d3js(seed, filename)
219     with open('tileview.html','w',encoding="UTF-8") as tv:
220         tv.write('<html>')
221         #tv.write('<div style="padding:4px;">{</div><br><iframe scrolling="no" '
222             #width="270px" #height="270px" src="{}.html"></iframe></div>'.
223             #format(seed.name,filename))
224     for index,sol in enumerate(seed.explore_isonodes()):
225         level2 = sol.evolve(coupler, sol.name+'<-'+coupler.name)
226         name = 'view'+str(index)+'.js'
227         write_d3js(level2, name)
228         with open('viewgraph.html',encoding="UTF-8") as f:
229             text = f.read()
230             newtext = text.replace(r'server/customdata.js',name)
231             with open(name+'.html','w',encoding="UTF-8") as fout:
232                 fout.write(newtext)
233         if div:
234             tv.write('<div style="padding:4px;">{</div><br><iframe scrolling="no" '
235                 'width="270px" height="270px" src="{}.html">')

```

```

236         '</iframe></div>'.format(name,name))
237     else:
238         tv.write('<iframe scrolling="no" width="270px" height="270px"'
239                '&src="{}.html"></iframe>'.format(name,name))
240     tv.write('</html>')

```

A.2.5 model.py

```

1  #!/usr/bin/env python3
2  """
3  Module dedicated to the designed Elements of Evolution.
4  This features operators (methods) and operands related to artefacts
5  This includes the Class Artefact as a solution view belonging to its equivalence
6  class under an isomorphism :math: '[instance graph]_||simeq'
7  The view is represented by its graph we can name 'itself' or *base graph*.
8
9  """
10 --author-- = 'Noel Vizcaino'
11
12 import networkx as nx
13 from networkx.algorithms import isomorphism
14 from ioutils import load_graph_context, DEFAULT_CONTEXT_FILE
15 import hashlib
16
17
18 class Artefact:
19     """
20     Class Artefact is a solution subspace instancing known views belonging to its
21     equivalence class under an isomorphism represented by
22     :math: '[instance graph]_||simeq'
23     This is the key or ID to a subspace of the whole potential solution space.
24     The subspace is further configured with configuration fragments that can be
25     layered into a whole within the instance.
26     The fragments contain useful metadata to be loaded into the nodes.
27     This is kept glued together by a chain of isomorphisms representing
28     the relational associations of the nodes(or vertices) understood as assets
29     relationships.
30     :math: '\Delta's create a new isomorphic subspaces by graph expansion
31     abandoning previous chain to create a new one. This means old instances from
32     the same origin are sub-graph isomorphic.
33
34     """
35     node_property_tag = 'property'
36     expansion_relation_tag = 'isonode'

```

```
37
38 def __init__(self, name):
39     """
40     Creates basic artefact with empty digraph and blanked fields.
41
42     :param name: Name of the artefact
43     """
44     self.name = name
45     self.id = self.generateID(self.name)
46     self.graph = nx.DiGraph()
47     self.couplerlist = []
48     self.evolved_edges = []
49     self.contexts = []
50     self.views = []
51
52 def generateID(self, data):
53     """
54     Generates a new one way cryptographic hash using sha1 algorithm.
55     The hexadecimal digest is returned as ID based on the utf8 self.name
56     encoding, just for convenience.
57
58     :param data: string to use as unicode bytes to be hashed
59     :return: the hex digest
60     """
61     return hashlib.sha1(data.encode()).hexdigest()
62
63 @classmethod
64 def from_relationships(cls, data, name):
65     """
66     To create an artefact using an assets pairs relationship list
67
68     :param data: list of asset pairs
69     :param name: name for this artefact
70     :return: new artefact with a default attached context (nodes metadata)
71     """
72     artefact = cls(name)
73     artefact.graph.add_edges_from(data)
74     artefact.bind_context_fragment()
75     return artefact
76
77 @classmethod
78 def from_graph(cls, g, name):
79     """
```



```
80         To create an artefact using a graph containing asset relationships
81
82         :param g: a NetworkX graph
83         :param name: name for this coupler artefact
84         :return: new artefact with a default attached context (nodes metadata)
85         """
86         artefact = cls(name)
87         artefact.graph = g
88         artefact.bind_context_fragment()
89         return artefact
90
91     def update_views(self):
92         """
93         Creates or updates views with known solutions views.
94         This is the equivalence :math:'[instance graph]_||simeq' class partial
95         expansion
96         """
97         self.views = self.explore_isonodes()
98
99     def bind_context_fragment(self, config=DEFAULT_CONTEXT_FILE):
100         """
101         Binds a metadata node information layer (context fragment) to its (base)
102         graph node metadata
103
104         :param config: path to node metadata file
105         """
106         with load_graph_context(self.graph, config) as g:
107             self.graph = g
108             self.contexts.append(config)
109
110     @property
111     def properties(self):
112         """
113         Access to tag property node metadata as dictionary
114
115         :return: properties dictionary, use properties[node]
116         """
117         ps = self.tagmap(Artefact.node_property_tag)
118         return ps
119
120     def count(self, tag):
121         """
122         Counts and returns node metadata instances as id by an existing tag context
```

```
123
124         :param tag: an existing(initialised) tag context
125         :return: metadata instances or entries count
126         """
127         return [len(self.tagmap(tag)[n]) for n in self.graph]
128
129     @property
130     def isonodes(self):
131         """
132         Access to tag isonode node metadata as dictionary
133
134         :return: isonodes dictionary, use isonodes[node]
135         """
136         ps = self.tagmap(Artifact.expansion_relation_tag)
137         return ps
138
139     def _fixmiss(self, assetmap):
140         """
141         Check fixes a dictionary for unknown node metadata hit misses by blanking them
142
143         :param assetmap:
144         :return: assetmap with blanked array misses
145         """
146         for n in self.graph:
147             try:
148                 assetmap[n]
149             except KeyError:
150                 assetmap[n] = []
151         return assetmap
152
153     def tagmap(self, tag):
154         """
155         Returns nodes stored tagged metadata
156
157         :param tag: name of the metadata dictionary
158         :return: a tag dictionary where nodes are the keys
159         """
160         assetmap = nx.get_node_attributes(self.graph, tag)
161         self._fixmiss(assetmap)
162         return assetmap
163
164     def evolve(self, coupler, name):
165         """
```

```

166         Expand, with coupler graph, current base graph carrying existing
167         configurations. This is operator (thin) :math:'\Delta' over
168         the representative view of
169         :math:'[instance graph]_||simeq' where
170         the equivalence class evolves but implicitly.
171
172         :param coupler: the artefact with the graph to attach
173         :param name: a name for the resulting artefact
174         :return: expanded solution view
175         """
176         self.name = name
177         self.id = self.generateID(self.name)
178         common = set(coupler.graph) & set(self.graph)
179         # print('common:', common)
180         self.graph = nx.compose(self.graph, coupler.graph)
181         for node in common:
182             for edge in coupler.graph.edges():
183                 if node in edge:
184                     self.evolved_edges.append(edge)
185         self.couplerlist.append(coupler)
186         return self
187
188     def evolve_delta(self, coupler, name):
189         """
190         Explicitly evolve the known views belonging to
191         :math:'[instance graph]_||simeq' currently known.
192         A trunk :math:'\Delta' operation.
193
194         :param coupler: the artefact with the graph to attach to all views
195         :param name: a name for the resulting artefact
196         :return: expanded equivalence class
197         """
198         self.update_views()
199         for index, view in enumerate(self.views):
200             self.name=name
201             self.views[index].id = self.generateID(self.name)
202             self.views[index]=view.evolve(coupler, view.name+' <- '+coupler.name)
203             self.views[index].couplerlist.append(coupler)
204             self.views[index].evolved_edges=list(set(self.evolved_edges))
205         return self
206
207     def explore_isonodes(self):
208         """

```

```

209         Gathers the solutions views that can be currently generated based on stored
210         isonode metadata.
211
212         :return: A list with (valid) solution views belonging to
213         :math:'[instance graph]_\\|simeq'
214         """
215         artefact_list = []
216         mapping = {}
217         for n in self.graph:
218             for alternative in self.isonodes[n]:
219                 mapping.update({n: alternative})
220                 # print(repr(tmp)) # add original node!
221                 graph = nx.relabel_nodes(self.graph, mapping)
222                 name = 'View '+str(repr(mapping).replace(':', ' by '))
223                 artefact = self.from_graph(graph, name)
224                 artefact.evolved_edges = self.evolved_edges #plot_relation
225                 artefact.couplerlist = list(set(self.couplerlist))
226                 artefact_list.append(artefact)
227         artefact_list.append(self)
228         return artefact_list
229
230     def isonodes_graph(self):
231         """
232         Creates graph with the isonodes name enumerations as nodes.
233         Bijection is broken
234         as on node point to many
235         making it a hypergraph view.
236
237         :return: a new graph
238         """
239         mapping = {}
240         for n in self.graph:
241             assets = self.isonodes[n]
242             assets.append(n)
243             newnode = ', '.join(assets)
244             mapping.update({n: newnode})
245         graph = nx.relabel_nodes(self.graph, mapping)
246         # NOTE: ignoring other data as not needed
247         return graph
248
249     @property
250     def assortativity(self):
251         """

```

```
252         The degree of assortative or preferential attachment calculated for the
253         artefact graph. This is calculated using a fast internal
254         provided by NetworkX implementation of the Pearson algorithm.
255         This would serve to track its evolution through branches and to compare
256         the assortativity with that of other networks.
257
258         :return: floating point coefficient
259         """
260         return nx.degree_pearson_correlation_coefficient(self.graph)
261
262     def is_subgraph_isomorphic(self, subartefact):
263         """
264         Finds if current artefact itself contains ANY other isomorphic artefact
265
266         :param subartefact: sub-artefact with a relevant sub-graph to check with
267         :return: True if found.
268         """
269         iso = isomorphism.DiGraphMatcher(self.graph, subartefact.graph)
270         return iso.subgraph_is_isomorphic()
271
272     def find_isomorphic_pairs(self, subartefact, tags, values):
273         """
274         Check if current artefact contains other isomorphic sub-artefact considering
275         metadata information as a filter.
276         And provides the isomorphic result of the filter.
277
278         :param subartefact: sub-artefact with a relevant sub-graph to check with
279         :param tags: affected node metadata context tags list
280         :param values: corresponding node metadata context actual datum list
281         :return: Returns generator of dictionaries with found isomorphic pairs
282         """
283
284         iso = isomorphism.DiGraphMatcher(self.graph, subartefact.graph,
285                                         node_match=
286                                         isomorphism.categorical_node_match(tags,
287                                         values))
288
289         return iso.isomorphisms_iter()
290
291     def __str__(self):
292         """
293         Basic artefact text information to self pretty-print to stdout with standard
294         print function
```

```

295
296         :return: textual representation of the artefact
297         """
298         out = '-----[ Artefact: {} ]' \
299             '-----\n'.format(self.name)
300         out += self.mapinfo(Artefact.node_property_tag)
301         out += self.mapinfo(Artefact.expansion_relation_tag)
302         out += str('\nContext sequence: {}'.format(self.contexts))
303         out += str('\nCouplers so far: {}'.format([coupler.name
304                                                     for coupler in self.couplerlist]))
305         out += str('\nEvolution edges: {}'.format(repr(self.evolved_edges)))
306         # out += str('\nProperty Count: {}'.format(self.count('property')))
307         return out
308
309     def mapinfo(self, tag):
310         """
311         Gather nodes tagged text metadata to self pretty-print to stdout with
312         standard print function
313
314         :param tag: an existing context tag
315         :return: Tagged node metadata as text
316         """
317         out = '\n'.join('Node {} {} entries: '.format(n,tag)+repr(self.tagmap(tag)[n])
318                         for n in self.graph)
319         return(str(out))

```

A.2.6 artefact_views_test.py

A.3 Produced Support data

GPLv3 licensed component by Michael Bostock customised to display D3.js force graph display in all D3 based views using Jinja2 template system. Source: <http://bl.ocks.org/mbostock/1153292> [19]

A.3.1 The tile view Jinja2 template

```

1  |<!--!DOCTYPE html>
2  |<html>
3  |<head>
4  |<meta charset="UTF-8">
5  |</head>

```

```
6 <body>
7
8 {%- for view in solutionviews %}
9 <iframe sandbox="allow-scripts"
10 scrolling="no" width="500px"
11 height="500px"
12 src="http://127.0.0.1:5000/{{artefact|safe}}/{{view.id|safe}}">
13 </iframe>
14 {%- endfor %}
15
16 </body>
17 </html>
```

A.3.2 The single view Jinja2 template

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="UTF-8">
5 <!--Based on http://bl.ocks.org/mbostock/1153292
6 need to customize, this is just for the presentation layer
7 -->
8
9 <style>
10
11 .link {
12     fill: none;
13     stroke: #666;
14     stroke-width: 1.5px;
15 }
16
17 #coupler_added {
18     fill: red;
19 }
20
21 .link.coupler_added {
22     stroke: red;
23 }
24
25 .link.otherel {
26     stroke-dasharray: 0,2 1;
27 }
28
29 circle {
```

```
30   fill: #ccc;
31   stroke: #333;
32   stroke-width: 1.5px;
33 }
34
35 text {
36   font: 10px sans-serif;
37   pointer-events: none;
38   text-shadow: 0 1px 0 #fff, 1px 0 0 #fff, 0 -1px 0 #fff, -1px 0 0 #fff;
39 }
40 </style>
41 </head>
42 <body>
43 <script src="http://d3js.org/d3.v3.min.js"></script>
44 <script>
45
46   {{d3data|safe}}
47
48
49   var nodes = {};
50
51   // Compute the distinct nodes from the links.
52   links.forEach(function(link) {
53     link.source = nodes[link.source] || (nodes[link.source] = {name: link.source});
54     link.target = nodes[link.target] || (nodes[link.target] = {name: link.target});
55   });
56
57
58   var width = 480,
59       height = 480;
60
61   var force = d3.layout.force()
62     .nodes(d3.values(nodes))
63     .links(links)
64     .size([width, height])
65     .linkDistance(50)
66     .charge(-300)
67     .on("tick", tick)
68     .start();
69
70   var svg = d3.select("body").append("svg")
71     .attr("width", width)
72     .attr("height", height);
```



```
73
74 // Per-type markers, as they don't inherit styles.
75 svg.append("defs").selectAll("marker")
76   .data(["dependency", "coupler_added", "otherel"])
77   .enter().append("marker")
78     .attr("id", function(d) { return d; })
79     .attr("viewBox", "0 -5 10 10")
80     .attr("refX", 15)
81     .attr("refY", -1.5)
82     .attr("markerWidth", 6)
83     .attr("markerHeight", 6)
84     .attr("orient", "auto")
85   .append("path")
86     .attr("d", "M0,-5L10,0L0,5");
87
88 var path = svg.append("g").selectAll("path")
89   .data(force.links())
90   .enter().append("path")
91     .attr("class", function(d) { return "link " + d.type; })
92     .attr("marker-end", function(d) { return "url(#" + d.type + ")"; });
93
94 var circle = svg.append("g").selectAll("circle")
95   .data(force.nodes())
96   .enter().append("circle")
97     .attr("r", 6)
98     .call(force.drag);
99
100 var text = svg.append("g").selectAll("text")
101   .data(force.nodes())
102   .enter().append("text")
103     .attr("x", 8)
104     .attr("y", ".31em")
105     .text(function(d) { return d.name; });
106
107 // Use elliptical arc path segments to doubly-encode directionality.
108 function tick() {
109   path.attr("d", linkArc);
110   circle.attr("transform", transform);
111   text.attr("transform", transform);
112 }
113
114 function linkArc(d) {
115   var dx = d.target.x - d.source.x,
```

```

116     dy = d.target.y - d.source.y,
117     dr = Math.sqrt(dx * dx + dy * dy);
118     return "M" + d.source.x + "," + d.source.y + "A" + dr + "," +
119           dr + " 0 0,1 " + d.target.x + "," + d.target.y;
120 }
121
122 function transform(d) {
123     return "translate(" + d.x + "," + d.y + ")";
124 }
125
126
127
128 </script>
129 </body>
130 </html>

```

A.3.3 Single view javascript data sample

```

1  var links = [{source: "SpiderMonkey" , target:"DCOM", type: "dependency"},
2  {source: "JS Libraries" , target:"SpiderMonkey", type: "dependency"},
3  {source: "JS Libraries" , target:"DCOM", type: "dependency"},
4  {source: "JS Libraries" , target:"Javascript", type: "dependency"},
5  {source: "XUL" , target:"SpiderMonkey", type: "dependency"},
6  {source: "XUL" , target:"Javascript", type: "dependency"},
7  {source: "CustomBinary1" , target:"WebGL wasm binary", type: "dependency"},
8  {source: "CustomBinary1" , target:"wasm capability", type: "coupler_added"},
9  {source: "wasm capability" , target:"SpiderMonkey", type: "dependency"}];

```

A.3.4 Sample JSON Artefact graph format

Graph data

```

1  {"graph": {"name": "compose( , )"}, "nodes": [{"licenceview": ["None"],
2  "id": "C++ wasm target", "property": ["C++ libraries", "NO filesystem IO",
3  "fastest", "load-time-efficient"], "softwareview": ["Language"]},
4  {"id": "WebGL wasm binary"}, {"id": "CustomBinary1"}, {"softwareview":
5  ["Presentation"], "id": "SpiderMonkey", "property":
6  ["JavaScript engine", "asmj.js", "wasm", "language:JavaScript",
7  "Mozilla application framework"], "isonode": ["V8", "Chakra"],
8  "licenceview": ["MPL"]}, {"softwareview": ["UX/Presentation"],
9  "id": "XUL", "property": ["GUI", "web", "XML", "cross platform",
10 "Mozilla application framework"], "isonode": ["WinForms", "Swing",
11 "wxWidgets", "Kivy", "QT"], "licenceview": ["MPL"]}, {"id": "CustomBinary2"},

```

```

12 {"softwareview": ["Components"], "id": "XPCOM", "property": ["business layer",
13 "component model", "enable:plugins", "text processing", "cross platform",
14 "Mozilla application framework"], "isonode": ["DCOM", ".NET", "BONOBO"],
15 "licenceview": ["MPL"]}, {"softwareview": ["Language"], "id": "Javascript",
16 "property": ["language", "imperative", "Object Oriented",
17 "text processing", "web"], "isonode": ["Python", "TypeScript"],
18 "licenceview": ["None"]}, {"id": "wasm capability"}, {"id": "JS Libraries"}],
19 "multigraph": false, "directed": true,
20 "links": [{"target": 3, "source": 4}, {"target": 7, "source": 4},
21 {"target": 8, "source": 2}, {"target": 1, "source": 2},
22 {"target": 8, "source": 5}, {"target": 0, "source": 5},
23 {"target": 6, "source": 3}, {"target": 6, "source": 9},
24 {"target": 3, "source": 9}, {"target": 7, "source": 9},
25 {"target": 3, "source": 8}]]

```

A.4 Python stdout(console) prints

Artefact basic information print output:

```

1 -----[ Artefact: wasm capability enabled Firefox ]-----
2 Node XPCOM property entries: ['business layer', 'component model', 'enable:plugins', 'text processing']
3 Node SpiderMonkey property entries: ['JavaScript engine', 'asmj.js', 'wasm', 'language:JavaScript', 'M']
4 Node Javascript property entries: ['language', 'imperative', 'Object Oriented', 'text processing', 'wel']
5 Node XUL property entries: ['GUI', 'web', 'XML', 'cross platform', 'Mozilla application framework']
6 Node JS Libraries property entries: []
7 Node wasm capability property entries: []Node XPCOM isonode entries: ['DCOM', '.NET', 'BONOBO']
8 Node SpiderMonkey isonode entries: ['V8', 'Chakra']
9 Node Javascript isonode entries: ['Python', 'TypeScript']
10 Node XUL isonode entries: ['WinForms', 'Swing', 'wxWidgets', 'Kivy', 'QT']
11 Node JS Libraries isonode entries: []
12 Node wasm capability isonode entries: []
13 Context sequence: ['assets.xml']
14 Couplers so far: []
15 Evolution edges: []
16 Seed Assort: -0.148522131447
17 -----[ Artefact: CustomScript jQuery and D3.js ]-----
18 Node XPCOM property entries: ['business layer', 'component model', 'enable:plugins', 'text processing']
19 Node CustomBinary1 property entries: []
20 Node XUL property entries: ['GUI', 'web', 'XML', 'cross platform', 'Mozilla application framework']
21 Node WebGL wasm binary property entries: []
22 Node CustomBinary2 property entries: []
23 Node Javascript property entries: ['language', 'imperative', 'Object Oriented', 'text processing', 'wel']
24 Node JS Libraries property entries: []

```

```
25 Node C++ wasm target property entries: ['C++ libraries', 'NO filesystem IO', 'fastest', 'load-time-eff
26 Node SpiderMonkey property entries: ['JavaScript engine', 'asmj.js', 'wasm', 'language:JavaScript', 'M
27 Node wasm capability property entries: []Node XPCOM isonode entries: ['DCOM', '.NET', 'BONOBO']
28 Node CustomBinary1 isonode entries: []
29 Node XUL isonode entries: ['WinForms', 'Swing', 'wxWidgets', 'Kivy', 'QT']
30 Node WebGL wasm binary isonode entries: []
31 Node CustomBinary2 isonode entries: []
32 Node Javascript isonode entries: ['Python', 'TypeScript']
33 Node JS Libraries isonode entries: []
34 Node C++ wasm target isonode entries: []
35 Node SpiderMonkey isonode entries: ['V8', 'Chakra']
36 Node wasm capability isonode entries: []
37 Context sequence: ['assets.xml', 'views.xml']
38 Couplers so far: ['wasml', 'wasm2']
39 Evolution edges: [('CustomBinary1', 'wasm capability'), ('CustomBinary2', 'wasm capability')]
40 Sol Assort: -0.0185185185185
41 Node XPCOM softwareview entries: ['Components']
42 Node CustomBinary1 softwareview entries: []
43 Node XUL softwareview entries: ['UX/Presentation']
44 Node WebGL wasm binary softwareview entries: []
45 Node CustomBinary2 softwareview entries: []
46 Node Javascript softwareview entries: ['Language']
47 Node JS Libraries softwareview entries: []
48 Node C++ wasm target softwareview entries: ['Language']
49 Node SpiderMonkey softwareview entries: ['Presentation']
50 Node wasm capability softwareview entries: []
51 Node XPCOM licenceview entries: ['MPL']
52 Node CustomBinary1 licenceview entries: []
53 Node XUL licenceview entries: ['MPL']
54 Node WebGL wasm binary licenceview entries: []
55 Node CustomBinary2 licenceview entries: []
56 Node Javascript licenceview entries: ['None']
57 Node JS Libraries licenceview entries: []
58 Node C++ wasm target licenceview entries: ['None']
59 Node SpiderMonkey licenceview entries: ['MPL']
60 Node wasm capability licenceview entries: []
61 True
62
63 Process finished with exit code 0
```

A.5 XML configuration files

Configuration files with metadata to be loaded on artefact instances.

A.5.1 Artefact XML sets master configuration schema to validate master

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema attributeFormDefault="unqualified"
3    elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
4    <xs:element name="assets" type="assetsType">
5      <xs:annotation>
6        <xs:documentation>if I define it
7        &lt;assets name="asset repository"
8        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
9        xsi:noNamespaceSchemaLocation="./repository.xsd"></xs:documentation>
10     </xs:annotation>
11   </xs:element>
12   <xs:complexType name="assetsType">
13     <xs:sequence>
14       <xs:element type="assetType" name="asset" maxOccurs="unbounded" minOccurs="0">
15         <xs:annotation>
16           <xs:documentation>inside the artefact they are just listed</xs:documentation>
17         </xs:annotation>
18       </xs:element>
19     </xs:sequence>
20     <xs:attribute type="xs:string" name="name"/>
21     <xs:attribute type="xs:string" name="version"/>
22   </xs:complexType>
23   <xs:complexType name="assetType">
24     <xs:choice maxOccurs="unbounded" minOccurs="0">
25       <xs:element type="xs:string" name="property"/>
26       <xs:element type="xs:string" name="isonode"/>
27     </xs:choice>
28     <xs:attribute type="xs:string" name="name" use="optional"/>
29   </xs:complexType>
30 </xs:schema>

```

A.5.2 Artefact XML assets master configuration example

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!--
3  if I define it

```

```
4 <assets name="asset repository"
5 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6 xsi:noNamespaceSchemaLocation="./repository.xsd">
7 -->
8 <assets name="master" version="1.0">
9 <!-- inside the artefact they are just listed -->
10 <asset name="DCOM">
11 <property>Windows Only</property>
12 </asset>
13 <asset name="WebGL wasm binary">
14 <property>NO filesystem IO</property>
15 <property>fastest</property>
16 <property>web</property>
17 <property>Render:Canvas</property>
18 <property>language:JavaScript</property>
19 <property>Smallest size</property>
20 <property>Sandboxed</property>
21 </asset>
22 <asset name="C++ wasm target">
23 <property>C++ libraries</property>
24 <property>NO filesystem IO</property>
25 <property>fastest</property>
26 <property>load-time-efficient</property>
27 <property>Access to legacy code</property>
28 <property>Low level</property>
29 <property>Smallest size</property>
30 <property>Sandboxed</property>
31 </asset>
32 <asset name="bs4">
33 <property>HTML processing</property>
34 <property>HTML error tolerance</property>
35 <property>language:python</property>
36 <property>text processing</property>
37 <property>web</property>
38 </asset>
39 <asset name="Kivy">
40 <property>GUI</property>
41 <property>cross platform</property>
42 <isonode>WinForms</isonode>
43 <isonode>Swing</isonode>
44 <isonode>wxWidgets</isonode>
45 <isonode>XUL</isonode>
46 <isonode>QT</isonode>
```

```
47 </asset>
48 <asset name="D3">
49   <property>web</property>
50   <property>language: JavaScript</property>
51   <property>Render: SVG</property>
52   <isonode>Bokeh</isonode>
53 </asset>
54 <asset name="Bokeh">
55   <property>web</property>
56   <property>Render: Canvas</property>
57   <property>language: JavaScript</property>
58   <isonode>D3</isonode>
59 </asset>
60 <asset name="jQuery">
61   <property>language: JavaScript</property>
62   <property>web</property>
63   <property>declarative</property>
64   <property>cross platform</property>
65   <isonode>Zepto</isonode>
66   <isonode>AngularJS</isonode>
67   <isonode>umbrella.js</isonode>
68 </asset>
69 <asset name="Firefox">
70   <property>web</property>
71   <property>browser</property>
72   <property>cross platform</property>
73   <property>enable: plugins</property>
74   <property>Mozilla application framework</property>
75   <isonode>Chrome</isonode>
76   <isonode>IE</isonode>
77   <isonode>Opera</isonode>
78   <isonode>Edge</isonode>
79   <isonode>Lynx</isonode>
80 </asset>
81 <asset name="GreaseMonkey">
82   <property>enabler: XSS</property>
83   <property>web</property>
84   <property>language: JavaScript</property>
85   <property>Firefox: extension</property>
86   <isonode>Scriptish</isonode>
87   <isonode>Tampermonkey</isonode>
88   <isonode>Violent monkey</isonode>
89 </asset>
```

```
90 <asset name="XPCOM">
91   <property>business layer</property>
92   <property>component model</property>
93   <property>enable:plugins</property>
94   <property>text processing</property>
95   <property>cross platform</property>
96   <property>Mozilla application framework</property>
97   <isonode>DCOM</isonode>
98   <isonode>.NET</isonode>
99   <isonode>BONOBO</isonode>
100 </asset>
101 <asset name="SpiderMonkey">
102   <property>JavaScript engine</property>
103   <property>asmj.js</property>
104   <property>wasm</property>
105   <property>language:JavaScript</property>
106   <property>Mozilla application framework</property>
107   <isonode>V8</isonode>
108   <isonode>Chakra</isonode>
109 </asset>
110 <asset name="Javascript">
111   <property>language</property>
112   <property>imperative</property>
113   <property>Object Oriented</property>
114   <property>text processing</property>
115   <property>web</property>
116   <isonode>Python</isonode>
117   <isonode>TypeScript</isonode>
118 </asset>
119 <asset name="XUL">
120   <property>GUI</property>
121   <property>web</property>
122   <property>XML</property>
123   <property>cross platform</property>
124   <property>Mozilla application framework</property>
125   <isonode>WinForms</isonode>
126   <isonode>Swing</isonode>
127   <isonode>wxWidgets</isonode>
128   <isonode>Kivy</isonode>
129   <isonode>QT</isonode>
130 </asset>
131 <asset name="Ubuntu Trusty">
132   <property>OS</property>
```



```

133     <property>Linux</property>
134     <isonode>Windows Server 2012 R2</isonode>
135     <isonode>Red Hat Enterprise Linux 7</isonode>
136     <property>Google Cloud Platform</property>
137 </asset>
138 <asset name="Redis">
139     <property>Database</property>
140     <property>key-value</property>
141     <property>no-sql</property>
142     <isonode>PostgreSQL-test</isonode>
143     <isonode>Cassandra-test</isonode>
144     <isonode>MongoDB-test</isonode>
145 </asset>
146 <asset name="nginx">
147     <property>server</property>
148     <property>high-performance server</property>
149     <isonode>node.js</isonode>
150     <isonode>Apache</isonode>
151     <isonode>LAMP</isonode>
152     <isonode>LAPP</isonode>
153 </asset>
154 <asset name="node.js">
155     <property>server</property>
156     <property>language:JavaScript</property>
157     <property>asynchronous</property>
158     <property>evented</property>
159     <isonode>vertx.io</isonode>
160     <isonode>Tornado</isonode>
161 </asset>
162 <asset name="LAMP">
163     <property>Server Stack</property>
164     <property>language:JavaScript</property>
165     <property>language:sql</property>
166     <isonode>LAPP</isonode>
167     <isonode>Apache</isonode>
168     <isonode>language:PHP</isonode>
169     <isonode>MySQL</isonode>
170 </asset>
171 </assets>

```

A.5.3 Artefact XML asset extra configuration fragment example

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- if I define it

```

```
3 <assets name="asset repository"
4 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5 xsi:noNamespaceSchemaLocation="./repository.xsd">
6 -->
7 <assets name="views" version="1.0">
8 <!-- inside the artefact they are just listed-->
9 <asset name="C++ wasm target">
10 <licence>Not applicable</licence>
11 <softwareview> Language</softwareview>
12 </asset>
13 <asset name="bs4">
14 <licence>MIT</licence>
15 <softwareview> business logic</softwareview>
16 </asset>
17 <asset name="Kivy">
18 <licence>Dual GPL</licence>
19 <softwareview> UX/Presentation</softwareview>
20 </asset>
21 <asset name="jsforcegraph">
22 <licence>GPLv3</licence>
23 <softwareview> Presentation</softwareview>
24 </asset>
25 <asset name="jQuery">
26 <licence>MIT</licence>
27 <softwareview> Presentation</softwareview>
28 </asset>
29 <!-- define firefox as an artefact -->
30 <asset name="Firefox">
31 <licence>MPL</licence>
32 <softwareview> framework </softwareview>
33 </asset>
34 <asset name="GreaseMonkey">
35 <licence>MPL</licence>
36 <softwareview>plugin</softwareview>
37 </asset>
38 <asset name="XPCOM">
39 <licence>MPL</licence>
40 <softwareview> Components</softwareview>
41 </asset>
42 <asset name="SpiderMonkey">
43 <licence>MPL</licence>
44 <softwareview> Presentation</softwareview>
45 </asset>
```

```

46 <asset name="Javascript">
47   <licence> Standard</licence>
48   <softwarereview> Language </softwarereview>
49 </asset>
50 <asset name="XUL">
51   <licence>MPL</licence>
52   <softwarereview> UX/Presentation</softwarereview>
53 </asset>
54 </assets>

```

A.5.4 Google Cloud Launcher configuration fragment(layer)

Just selected assets entries out of 1755 source lines [17].

```

1 <?xml version="1.0" ?>
2 <assets name="Google Cloud Launcher Assets">
3   <!--Generated by csv2assets.py.
4   Google Cloud Launcher features popular
5   open source packages that have been configured by
6   Bitnami or Google Click for easy deployment.-->
7   <asset category="Unknown origin" description="description"
8   icon="icon"
9   link="link" name="name"/>
10   <asset category="Unknown origin" description=" Universal artifact repository "
11   icon="Cloud%20Launcher%20Marketplace%20Solutions_files/XvX05GaH5MSyRwmozajoTjIziG
12   pBl1ps5Kg8nQ0G88aRqvofM.png"
13   link="https://console.cloud.google.com/launcher/details/jfrog-app/artifactory"
14   name="JFrog Artifactory"/>
15   <asset category="OS" description=" Ubuntu Trusty Linux (14.04 LTS) "
16   icon="Cloud%20Launcher%20Marketplace%20Solutions_files/BuywmfS24sUuqEjgyg_VRZrGN2m
17   -CPqMg0MAa1MaLlT-LtCXJk.png"
18   link="https://console.cloud.google.com/launcher/details/ubuntu-os-cloud/
19   ubuntu-trusty?cat=OS"
20   name="Ubuntu Trusty">
21     <isonode>Windows Server 2012 R2</isonode>
22     <isonode>Red Hat Enterprise Linux 7</isonode>
23     <isonode>Ubuntu Xenial</isonode>
24     <isonode>Windows Server 2008 R2</isonode>
25     <isonode>Ubuntu Precise</isonode>
26     <isonode>Red Hat Enterprise Linux 6</isonode>
27     <isonode>SUSE Linux Enterprise Server 11</isonode>
28     <isonode>SUSE Linux Enterprise Server 12</isonode>
29     <isonode>Debian 8</isonode>
30     <isonode>CentOS 7</isonode>

```

```

31     <isonode>CentOS 6</isonode>
32 </asset>
33 <asset category="Database" description=" Advanced key-value cache and store "
34 icon="Cloud%20Launcher%20Marketplace%20Solutions_files/V2nVMPQcimm14jDWD-QKf_
35 VToeFZOIt7-5GOF7Saqt_mMJKR0I.png"
36 link="https://console.cloud.google.com/launcher/details/click-to-deploy-images/
37 redis?cat=DATABASE"
38 name="Redis">
39     <isonode>Cassandra</isonode>
40     <isonode>EDB Postgres Enterprise</isonode>
41     <isonode>Aerospike</isonode>
42     <isonode>MySQL</isonode>
43     <isonode>MongoDB</isonode>
44     <isonode>PostgreSQL</isonode>
45     <isonode>Cassandra</isonode>
46     <isonode>Percona</isonode>
47     <isonode>MongoDB Multi-VM</isonode>
48     <isonode>CouchDB</isonode>
49     <isonode>DataStax Enterprise</isonode>
50     <isonode>ClearDB</isonode>
51 </asset>
52 <asset category="Blogging" description=" The most popular and ready-to-go CMS "
53 icon="Cloud%20Launcher%20Marketplace%20Solutions_files/DvnB0pJ0iEM2LW4mhC8fZBEMKxB
54 7DAARTrOTUp3mKrAkwyM5n8.png"
55 link="https://console.cloud.google.com/launcher/details/bitnami-launchpad/
56 wordpress?cat=LOG"
57 name="WordPress">
58     <isonode>WordPress Multisite</isonode>
59     <isonode>Ghost</isonode>
60     <isonode>Publify</isonode>
61     <isonode>Chyrp</isonode>
62     <isonode>Roller</isonode>
63 </asset>
64 <asset category="CMS" description=" The most popular and ready-to-go CMS "
65 icon="Cloud%20Launcher%20Marketplace%20Solutions_files/DvnB0pJ0iEM2LW4mhC8fZBEMKxB
66 B7DAARTrOTUp3mKrAkwyM5n8.png"
67 link="https://console.cloud.google.com/launcher/details/bitnami-launchpad/
68 wordpress?cat=CMS"
69 name="WordPress">
70     <isonode>WordPress Multisite</isonode>
71     <isonode>Joomla!</isonode>
72     <isonode>Drupal</isonode>
73     <isonode>Alfresco Community</isonode>

```

```
74     <isonode>concrete5</isonode>
75     <isonode>MODX</isonode>
76     <isonode>Tiki Wiki CMS Groupware</isonode>
77     <isonode>CMS Made Simple</isonode>
78     <isonode>Pimcore</isonode>
79     <isonode>ProcessWire</isonode>
80     <isonode>Open Atrium</isonode>
81     <isonode>SilverStripe</isonode>
82     <isonode>eZ Publish</isonode>
83     <isonode>Sitecake</isonode>
84     <isonode>TYPO3</isonode>
85     <isonode>Plone</isonode>
86     <isonode>Ametys</isonode>
87     <isonode>Neos</isonode>
88     <isonode>XOOPS</isonode>
89     <isonode>Refinery CMS</isonode>
90     <isonode>ocPortal</isonode>
91     <isonode>EnanoCMS</isonode>
92 </asset>
93 <asset category="CRM" description=" Flexible customer relationship management
94 solution "
95 icon="Cloud%20Launcher%20Marketplace%20Solutions_files/ftOAI13fKyeVPMFaGGkYFvOSj
96 ToDYswN77zLT0QT0oML3IQPb0.png" link="https://console.cloud.google.com/launcher/
97 details/bitnami-launchpad/sugarcrm?cat=CRM"
98 name="SugarCRM">
99     <isonode>Odoocrm</isonode>
100    <isonode>SuiteCRM</isonode>
101    <isonode>Mautic</isonode>
102    <isonode>ERPNext</isonode>
103    <isonode>Dolibarr</isonode>
104    <isonode>CiviCRM</isonode>
105    <isonode>EspoCRM</isonode>
106    <isonode>OpenERP</isonode>
107    <isonode>OroCRM</isonode>
108    <isonode>Zurmo</isonode>
109    <isonode>X2Engine Sales CRM</isonode>
110    <isonode>Fat Free CRM</isonode>
111 </asset>
112 <asset category="Developer tools" description=" Binary Repository Manager for
113 Maven, Ivy,
114 Gradle modules "
115 icon="Cloud%20Launcher%20Marketplace%20Solutions_files/-oLJxURmB1qwQYk1Zbe412B7u
116 6cdToF5rv5mW
```

```
117     JeHhEczeaaHcs.png"
118     link="https://console.cloud.google.com/launcher/details/bitnami-launchpad/
119     artifactory?cat=DEVELOPER_TOOLS" name="Artifactory">
120         <isonode>PHP 5.6 - Zend Server Developer Edition</isonode>
121         <isonode>Jenkins</isonode>
122         <isonode>Redmine</isonode>
123         <isonode>GitLab</isonode>
124         <isonode>Kafka</isonode>
125         <isonode>DreamFactory</isonode>
126         <isonode>RabbitMQ</isonode>
127         <isonode>Subversion</isonode>
128         <isonode>Phabricator</isonode>
129         <isonode>TestLink</isonode>
130         <isonode>Eclipse Che</isonode>
131         <isonode>Parse Server</isonode>
132         <isonode>ActiveMQ</isonode>
133         <isonode>Mantis</isonode>
134         <isonode>PHP 7.0 - Zend Server Developer Edition</isonode>
135         <isonode>Trac</isonode>
136         <isonode>Review Board</isonode>
137         <isonode>Codiad</isonode>
138         <isonode>Squash</isonode>
139         <isonode>JFrog Artifactory</isonode>
140         <isonode>Kong</isonode>
141     </asset>
142     <asset category="Other" description=" Cogito understands the meaning of written
143     language "
144     icon="Cloud%20Launcher%20Marketplace%20Solutions_files/c3xcso5w75hN6sjRNWoR4BYne9R
145     aFcCezmM275pc9cu5Zh00T1.png"
146     link="https://console.cloud.google.com/launcher/details/cogito-api/cogito?cat=OTHERS"
147     name="Cogito API Core"/>
148     <asset category="Other" description=" Popular eCommerce software and platform "
149     icon="Cloud%20Launcher%20Marketplace%20Solutions_files/xQhecg2kZcu0SYaomVko4qpWGyk
150     _3WS1_7KkNEqtlubr0oo07I.png"
151     link="https://console.cloud.google.com/launcher/details/bitnami-launchpad/
152     magento?cat=OTHERS"
153     name="Magento"/>
154     <asset category="Other" description=" Universal artifact repository "
155     icon="Cloud%20Launcher%20Marketplace%20Solutions_files/XvX05GaH5MSyRwmozajoTjIziGp
156     llps5Kg8nQ
157     OG88aRqvonfM.png"
158     link="https://console.cloud.google.com/launcher/details/jfrog-app/artifactory?cat=OTHERS"
159     name="JFrog Artifactory"/>
```

```
160     <asset category="Other" description=" Extremely powerful, scalable wiki
161         implementation "
162     icon="Cloud%20Launcher%20Marketplace%20Solutions_files/OPnVKdrbvyCzz1uA8qhWrUGMOLj
163     nX-kb9f17EsTb8Cphntk6Ws.png"
164     link="https://console.cloud.google.com/launcher/details/bitnami-launchpad/
165     mediawiki?cat=OTHERS"
166     name="MediaWiki"/>
167     <asset category="Other" description=" Free e-commerce platform for online
168         merchants "
169     icon="Cloud%20Launcher%20Marketplace%20Solutions_files/t_mZQ0un4z_MSkRpmF_Fl
170     VdxBnNh2zluwu_TR8e
171     8qln7W5Lug.png"
172     link="
173         <asset category="Other" description=" Intuitive to-do list app for easy
174             collaboration "
175     icon="Cloud%20Launcher%20Marketplace%20Solutions_files/I7nZXExPyqXIbwfWb7ul
176     GnOuGg1HwAbjcqMdHg6
177     -lrKGDZj7gd.png"
178     link="https://console.cloud.google.com/launcher/details/bitnami-launchpad/
179     tracks?cat=OTHERS"
180     name="Tracks"/>
181     <asset category="Other" description=" Popular Open Source ePortfolio and
182         social networking
183         web app "
184     icon="Cloud%20Launcher%20Marketplace%20Solutions_files/WtnYURyuHYbw0tTai7qEj
185     yzhEFX1MiJIVtTaXu5w
186     5QTbNOA9_2.png"
187     link="
188         <asset category="Other" description=" Popular personal web server "
189     icon="Cloud%20Launcher%20Marketplace%20Solutions_files/sbXZDQSLT-KPnRIOVaxnd
190     E9uLwgfNLgenpeRjXi
191     6e43YORpRLg.png"
192     link="https://console.cloud.google.com/launcher/details/bitnami-launchpad/
193     diaspora?cat=OTHERS"
194     name="Diaspora"/>
195     <asset category="Other" description=" Multi-purpose, fully featured web
196         gallery "
197     icon="Cloud%20Launcher%20Marketplace%20Solutions_files/cf1dfpaQRU_GKGIfFAj5CH
198     1ZUqC4z4X7groQv01ZmiaJq683zY.png"
199     link="https://console.cloud.google.com/launcher/details/bitnami-launchpad/
200     coppermine?cat=OTHERS"
201     name="Coppermine"/>
202 </assets>
```

A.6 Large images/captures

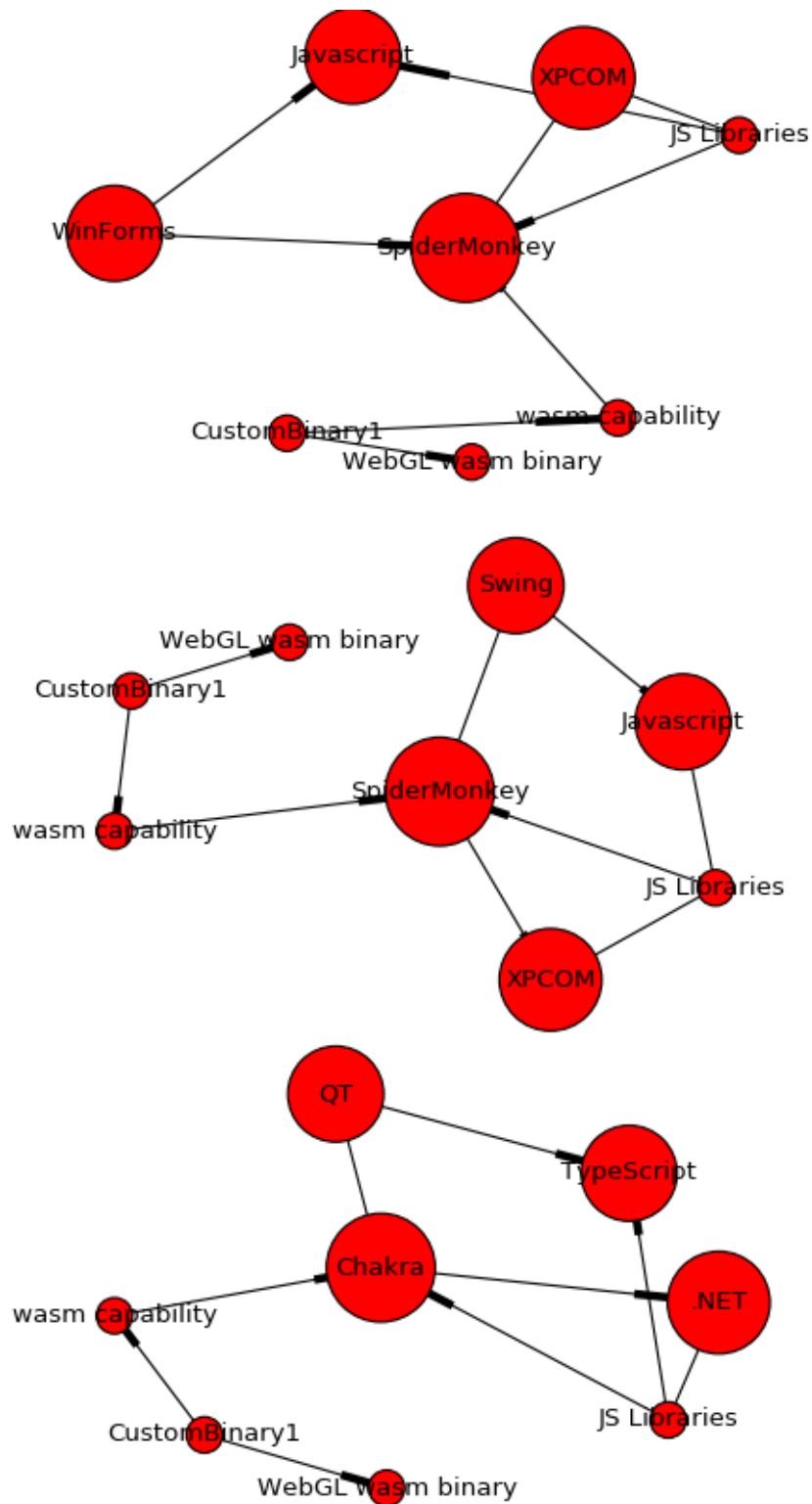


Figure A.1: Equivalence class generation of sample known static views.

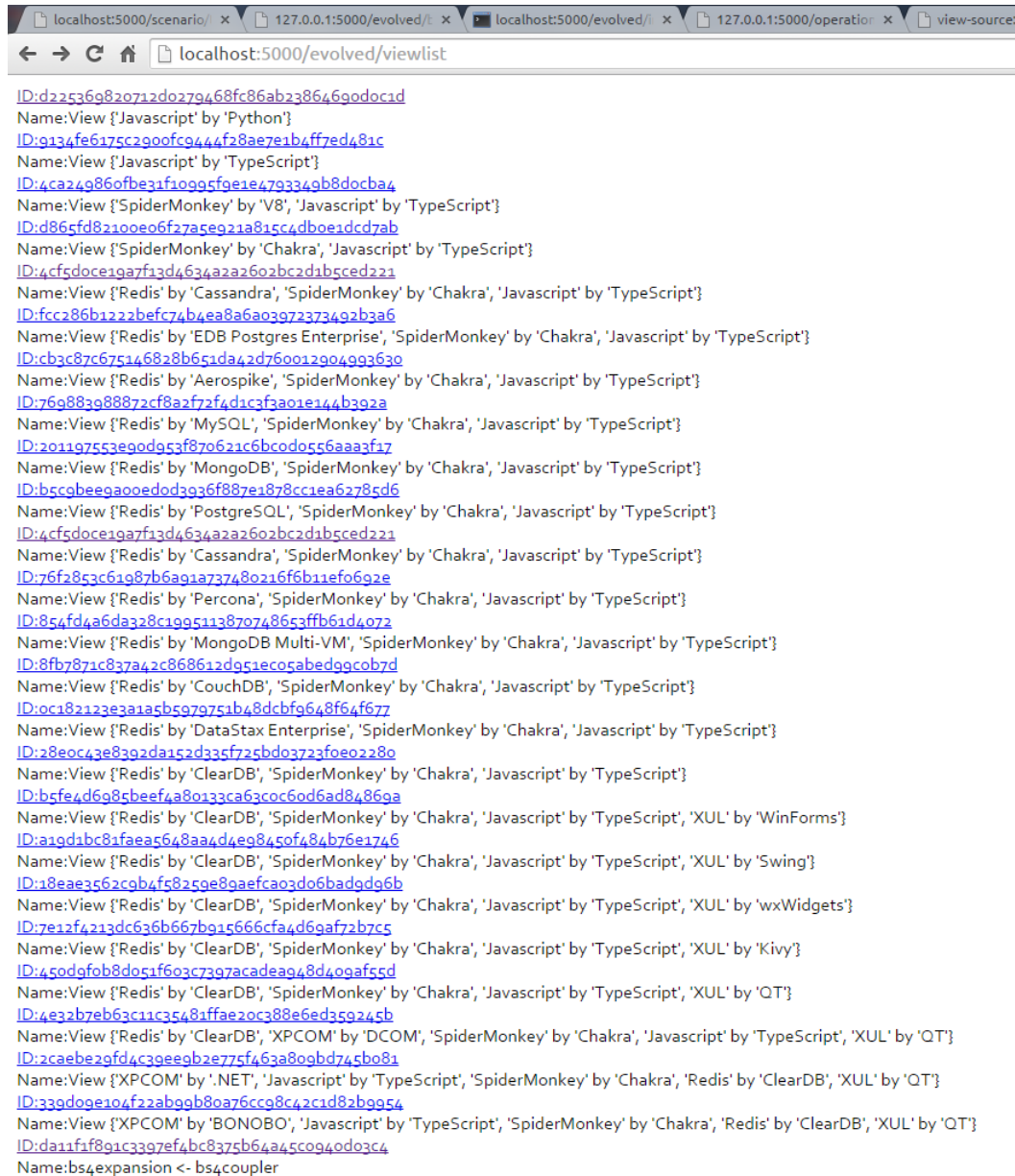


Figure A.2: All views evolved with the same artefact using the *evolve_delta()* thick Δ method.

```

-----[ Artefact: wasm capability enabled Firefox ]-----
Node JS Libraries property entries: []
Node XPCOM property entries: ['business layer', 'component model', 'enable:plugins', 'text processing', 'cross platform', 'Mozilla application framework']
Node Javascript property entries: ['language', 'imperative', 'Object Oriented', 'text processing', 'web']
Node SpiderMonkey property entries: ['JavaScript engine', 'asm.js', 'wasm', 'language:JavaScript', 'Mozilla application framework']
Node wasm capability property entries: []
Node XUL property entries: ['GUI', 'web', 'XML', 'cross platform', 'Mozilla application framework']Node JS Libraries isonode entries: []
Node XPCOM isonode entries: ['DCOM', '.NET', 'BONOBO']
Node Javascript isonode entries: ['Python', 'TypeScript']
Node SpiderMonkey isonode entries: ['V8', 'Chakra']
Node wasm capability isonode entries: []
Node XUL isonode entries: ['WinForms', 'Swing', 'wxwidgets', 'Kivy', 'QT']
Context sequence: ['assets.xml']
Couplers so far: []
Evolution edges: []
Seed Assort: -0.148522131447
-----[ Artefact: CustomScript jQuery and D3.js ]-----
Node CustomBinary2 property entries: []
Node JS Libraries property entries: []
Node SpiderMonkey property entries: ['JavaScript engine', 'asm.js', 'wasm', 'language:JavaScript', 'Mozilla application framework']
Node CustomBinary1 property entries: []
Node XPCOM property entries: ['business layer', 'component model', 'enable:plugins', 'text processing', 'cross platform', 'Mozilla application framework']
Node C++ wasm target property entries: ['C++ libraries', 'NO filesystem IO', 'fastest', 'load-time-efficient']
Node Javascript property entries: ['language', 'imperative', 'Object Oriented', 'text processing', 'web']
Node WebGL wasm binary property entries: []
Node wasm capability property entries: []
Node XUL property entries: ['GUI', 'web', 'XML', 'cross platform', 'Mozilla application framework']Node CustomBinary2 isonode entries: []
Node JS Libraries isonode entries: []
Node SpiderMonkey isonode entries: ['V8', 'Chakra']
Node CustomBinary1 isonode entries: []
Node XPCOM isonode entries: ['DCOM', '.NET', 'BONOBO']
Node C++ wasm target isonode entries: []
Node Javascript isonode entries: ['Python', 'TypeScript']
Node WebGL wasm binary isonode entries: []
Node wasm capability isonode entries: []
Node XUL isonode entries: ['WinForms', 'Swing', 'wxwidgets', 'Kivy', 'QT']
Context sequence: ['assets.xml', 'views.xml']
Couplers so far: ['wasm1', 'wasm2']
Evolution edges: [('CustomBinary1', 'wasm capability'), ('CustomBinary2', 'wasm capability')]
Sol Assort: -0.0185185185185
Node CustomBinary2 softwareview entries: []
Node JS Libraries softwareview entries: []
Node SpiderMonkey softwareview entries: ['Presentation']
Node CustomBinary1 softwareview entries: []
Node XPCOM softwareview entries: ['components']
Node C++ wasm target softwareview entries: ['Language']
Node Javascript softwareview entries: ['Language']
Node WebGL wasm binary softwareview entries: []
Node wasm capability softwareview entries: []
Node XUL softwareview entries: ['UX/Presentation']
Node CustomBinary2 licence entries: []
Node JS Libraries licence entries: []
Node SpiderMonkey licence entries: ['MPL']
Node CustomBinary1 licence entries: []
Node XPCOM licence entries: ['MPL']
Node C++ wasm target licence entries: ['Not applicable']
Node Javascript licence entries: ['Standard']
Node WebGL wasm binary licence entries: []
Node wasm capability licence entries: []
Node XUL licence entries: ['MPL']
True

```

Figure A.3: Notebook text output. Includes data before and after evolution and configuration layering.

```

%matplotlib inline
__author__ = 'Noel Vizcaino'

from model import Artefact as subspace
import ioutils as io

def main():
    r1 = [('XUL', 'SpiderMonkey'), ('XUL', 'Javascript'),
          ('JS Libraries', 'Javascrpt'), ('JS Libraries', 'XPCOM'),
          ('SpiderMonkey', 'XPCOM'), ('JS Libraries', 'SpiderMonkey'), ('wasm capability', 'SpiderMonkey')]
    r2 = [('CustomBinary1', 'wasm capability'), ('CustomBinary1', 'WebGL wasm binary')]
    r3 = [('CustomBinary2', 'C++ wasm target'), ('CustomBinary2', 'wasm capability')]
    seed = subspace.from_relationships(r1, 'wasm capability enabled firefox')
    coupler1 = subspace.from_relationships(r2, 'wasm1')
    coupler2 = subspace.from_relationships(r3, 'wasm2')
    solution = seed.evolve(coupler1, 'CustomScript jQuery')
    solution = solution.evolve(coupler2, 'CustomScript jQuery and D3.js')
    solution.bind_context_fragment('views.xml')
    io.print_d3_data(solution)
    solution.update_views()
    io.show_multiple_artefacts(solution.views, 'property')

if __name__ == '__main__':
    main()

var links = [{source: "JS Libraries", target:"javascript", type: "dependency"},
             {source: "JS Libraries", target:"SpiderMonkey", type: "dependency"},
             {source: "JS Libraries", target:"XPCOM", type: "dependency"},
             {source: "CustomBinary1", target:"WebGL wasm binary", type: "dependency"},
             {source: "CustomBinary1", target:"wasm capability", type: "coupler_added"},
             {source: "XUL", target:"Javascrpt", type: "dependency"},
             {source: "XUL", target:"SpiderMonkey", type: "dependency"},
             {source: "CustomBinary2", target:"C++ wasm target", type: "dependency"},
             {source: "CustomBinary2", target:"wasm capability", type: "coupler_added"},
             {source: "SpiderMonkey", target:"XPCOM", type: "dependency"},
             {source: "wasm capability", target:"SpiderMonkey", type: "dependency"}];

```

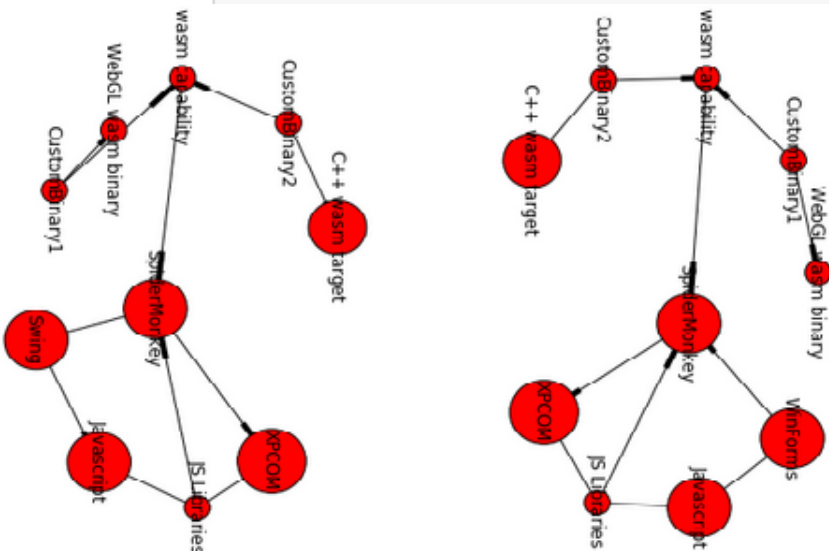


Figure A.4: Sample test interactive output as a Jupyter notebook: "multiple simultaneous static graphs.ipynb"

