Halo: A Framework for End-User Architecting

Vishal Dwivedi

CMU-S3D-22-110
December 2022

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

David Garlan (Chair)
James D. Herbsleb
Christian Kästner
Ian Gorton (Northeastern University)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

This thesis was sponsored by the Department of Defense (DOD) under Contract No. FA8721-05-C-0003, Office of Naval Research (ONR) under grant ONR-N000140811223, Army Research Office (ARO) under Award No. W911NF1310155, and the National Security Agency (NSA) under grant H9823018D0008. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DOD, ONR, ARO, NSA or the U.S. Government.

Keywords: Programming environments, software frameworks, software environments for composition, end-user software engineering, end-user architecture, HALO

Dedicated to my adviser, David
For all his support and guidance through the years!

Abstract

A large number of domains today require end users to compose various heterogeneous computational entities to perform their professional activities. However, writing such end-user compositions is hard and error-prone. Compared to the capabilities of modern programming environments, end users have relatively few tools for things like composition analysis, compilation into efficient deployments, interactive testing and debugging (e.g., setting breakpoints, monitoring intermediate results, etc.), history tracking, and graceful handling of run-time errors.

To overcome these limitations, we pose this thesis: "It is possible to build an end-user composition framework that can be instantiated to provide high-quality composition environments at relatively low cost compared to existing hand-crafted environments for a broad class of composition domains."

As a solution to this problem, we have designed a new technique called "end-user architecting" that associates end-user specifications in a particular domain as instances of architectural styles. This allows cross-domain analyses, systematic support for reuse and adaptation, powerful auxiliary services (e.g., mismatch repair), and support for execution, testing, and debugging.

To allow a wider adoption of this technique, we have designed the "Halo framework" that can be instantiated across a large number of domains, with composition models varying from data flows, publish-subscribe, and workflows. The Halo framework supports most of the common compositions tasks such as Search, Reuse, Construction, Analysis, Execution, and Debugging support and provides general and reusable infrastructures with well-defined customization points to build composition environments with these common features. Halo also provides adapters that can be customized for different user interfaces, runtime environments, and various analyses based on domain-specific constraints. This allows developers to systematically customize Halo and develop composition environments by using Halo's building blocks.

This approach can reduce the cost of development of end-user composition platforms (compared to developing them from scratch) and improve the quality of end-user compositions.

Acknowledgments

I deeply appreciate my advisor and mentor, Prof. David Garlan, for his everlasting patience, great insights, kind encouragement and all the effort he put in to guide me through this thesis. Also, special thanks to Dr. Bradley Schmerl for all his advice, collaborations on various systems and research, and guidance over the years, which helped me to formulate this thesis research. I also thank my committee members Prof. Christian Kastner, Prof. James Herbsleb and Dr. Ian Gorton for reviewing my work to strengthen and elevate its quality. I also thank Mary Shaw for her lessons on writing, research and abstraction.

I am also thankful to my various colleagues and collaborators: Dr. Ivan Ruchkin, Dr. Perla Velasco Elizando, Dr. Jose Maria Fernandez and Prof. Kathleen M. Carley whose collaborations helped me to define this research. I am also indebted to my various friends and colleagues in the CMU Software Engineering program who helped me with various ideas, critiques and support over the years. My special thanks to my ISR colleagues Ashutosh, Daniel, Ben, and Michael for their support and feedback during my PHD.

Most importantly, my special gratitude to my wife Dr. Srividya Ranganathan for her motivation to finish my PhD. Seeing her go through a rigorous medical residency program where she would spend 27 hour shifts in the E.R. and would still be willing to work through the next day was a big motivation.

I am also very thankful to my parents for all their motivation, support and love during my academic years. Many special thanks to my father in law, Mr. Rangathan Seshadri, who always motivated me to finish my thesis defense. After I started my career in Canada, besides David, I don't think anyone was more curious to understand my research. And like my conversations with David, I had to be always prepared - because there was always the next question coming, which required a precise answer and forced me to think.

Additionally, I owe special thanks to Nishad Prakash - my manager at Instacart. A significant chunk of this thesis was done during my ABD status while I was working at Instacart. During the peak of Covid growth, work at a startup was no different than my wife's long ER shifts (or at least it seemed that way). Nishad's support, and nudges to finish my PhD, were extremely helpful through this period.

Finally, there are several other friends and family members who made this thesis possible through their encouragement and support. My apologies for not being able to mention their names.

Contents

List of Figures

List of Tables

We call them novices for being ignorant about software engineering. They call us novices for being ignorant about their domain. Perhaps, we both are novices...

—Reflections on a complex relationship

CHAPTER 1

Introduction

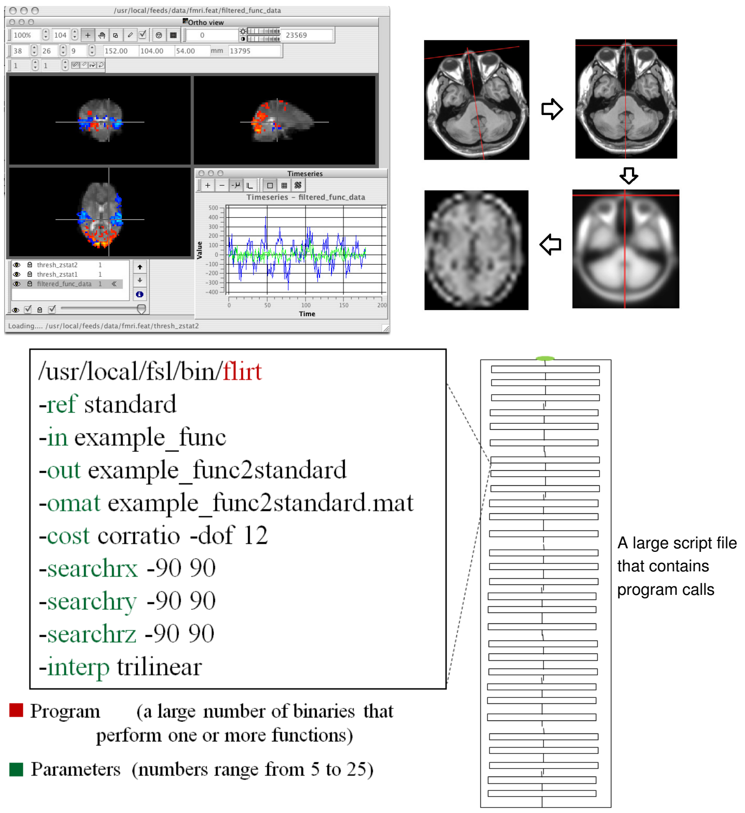
Increasingly, end users rely on computations to support their professional activities. In some cases, turnkey applications and services are sufficient to carry out computational tasks. However, in many situations users must adapt computing to their specific needs. These adaptations can take many forms: from setting preferences in applications, to programming spreadsheets, to creating orchestrations of services in support of some business process. This situation has given rise to an interest in end-user programming [Nar93], and, more generally, end-user software engineering [KAB$^+$11] or end-user computing [Goo97]. This emerging field attempts to find ways to better support users who, unlike professional programmers, write programs not as their primary job function, but to support the goals of their domain. Such end users may be experts in their domains, but must find ways to harness the power of computation to support their tasks.

In these domains, professionals typically have access to a large number of existing applications and data sets, which must be composed in novel ways to gain insight, carry out "what if" experiments, generate reports and research findings. Unfortunately, assembling such elements into coherent compositions is not trivial. In many cases, users must have detailed low-level knowledge of aspects such as application parameter settings, application invocation idiosyncrasies, file locations and naming conventions, data formats and encodings, ordering restrictions, and scripting languages.

For example, in the field of brain imaging, scientists study samples of neural activity to diagnose disease patterns such as tumours and injuries. Research in this domain requires that scientists compose a large number of tools and apply them to brain-imaging data sets to diagnose problems, such as malformations and structural or functional deformities. There also exist a large number of brain image processing tools for image recognition, image alignment, filtering, volumetric analytics, mapping, etc. Figure 1.1 illustrates a popular neuroscience tool suite, called FSL, that is used to create scripts for analyzing FMRI [Pek06] data. The figure illustrates part of a complex script composed of a collection of program calls to brain-imaging libraries and with a number of input parameters. In such a scenario, end users (who are neuroscience professionals) must create and execute detailed scripts, which is often hard and error-prone.

End users such as this have to address various concerns in such a scenario [GDRS12]. For example:

- **Compatibility concerns**: (1) Can the data produced by a component be consumed as an input of another? (2) Are any input-output conversion tools required to assemble a set of mismatched components?
- **Reuse concerns**: (1) How can the composition be packaged and shared with other users? (2) Where can one find a component to reuse?
- **Execution concerns**: (1) Is it possible to execute a composition on a particular server? (2) How



```
/usr/local/fsl/bin/flirt
-ref standard
-in example_func
-out example_func2standard
-omat example_func2standard.mat
-cost corratio -dof 12
-searchrx -90 90
-searchry -90 90
-searchrz -90 90
-interp trilinear
```

🟥 Program (a large number of binaries that perform one or more functions)
🟩 Parameters (numbers range from 5 to 25)
A large script file that contains program calls
Ortho view
Timeseries
Timeseries – filtered_func_data
Value
Time

Figure 1.1: End users in the neuroscience domain invoke a variety of programs and services.

much time will the execution take?

- **Quality concerns**: (1) Is the composition secure? (2) Is it possible to execute a composition faster with a lower fidelity result?
- **Reproducibility and data-provenance concerns**: (1) Can an in-silico[*] experiment to be repeated later? (2) What are the composition steps required to repeat a particular result?

Further, it may be difficult for such end users to determine whether a set of components can be composed at all, and, if not, what to do about it. For example, differences in data encodings may make direct component composition infeasible without the inclusion of one or more format converters. Even when a valid composition can be achieved, it may not have the performance (or other quality attributes) critical to the needs of the end users. And, even when a suitably performing composition can be created, it may be difficult to share it with peers or reuse it in different settings.

Today there exist a large number of domains that depend on composing existing components but have to rely on impoverished composition environments that make the task of end-user composition error-prone and complex. Table 1.1 lists examples of some of these domains.

In most cases, end users are expected to either become programmers or live with impoverished support (and sometimes both). Studies have shown that across many domains, end users are novices who are forced to spend about 40% of their time doing programming activities [HH11], often resort to copy-paste, and in the process make frequent mistakes [BGL+09].

Such end users today form large communities where their needs are not served by conventional software composition tools. These communities could benefit from powerful domain-specific composition

[*]A scientific experiment conducted or produced by means of computer modeling or computer simulation.

Domain	**Examples of compositions**
Astronomy	electromagnetic image processing tasks [DSS+05]
Bioinformatics	biological data-analysis services [Let05]
Digital music production	audio sequencing and editing [MH11]
Environmental Science	spatio-temporal experiments [VAR09]
Film production	scripting Maya and Adobe after-effects and Poser and Blender components [PKR06]
Gaming	scriptable and composable 3D engines [KBK08]
Geospatial Analysis	interactive visualization of geographical data [MCC11]
Home Automation	control schedule for home devices [LNH03]
Neuroscience	brain-image processing libraries [DEF+11]
Scientific computing	transformational workflows [Seg07]
Socio-technical Analysis	dynamic network creation, analysis, reporting and simulation [SGD+11]
Virtual Instrumentation	experiment pipelines in LabView [Joh97]

Table 1.1: Domains involving end-user compositions.

environments where compositions are designed using high-level abstractions (or components) that can be packaged, documented and reused through easily-accessible repositories. Additionally, such end users would like to have intuitive interfaces that match the computational model of their domain and are natural to their tasks [MPK04]. And furthermore, end users could benefit from analyses that can flag problems and guide them to avoid or repair mistakes. These tools could provide feedback about software qualities such as security, performance and fidelity, about which end users cannot easily reason on their own, enabling them to meet requirements in a cost effective way [AGI98, SG98].

Indeed, a number of such platforms have been developed in recent years that have tried to achieve the above goals. Examples include Taverna for life sciences [OGA+06], the Ozone Widget Framework (OWF) [MCC11] for geospatial analysis, Loni Pipeline for brain imaging [RMT03], VisTrails for data exploration and visualization [BCS+05], Steinberg's Virtual Studio Technology (VST) for composing music effects [MH11], etc.

While many of these platforms have been successful and several are in widespread use, they are typically handcrafted specifically for each new end-user domain, often at a high cost. The designers of such platforms are forced to engineer common capabilities from scratch such as component search, access-control, type-checking and execution support. This leads to high cost in development time and effort. As a consequence of this high development cost, we see many impoverished composition tools that lack capabilities to support end-user composition.

In this thesis, we advocate an approach to these problems that exploits the similarity between such compositions and software architecture, and attempts to leverage the considerable advances made within that field over the past two decades. The key idea is to view end-user composition activities as analogous to engaging in architectural design within a domain-specific style, and to represent those architectures explicitly. We argue that such explicit representation can offer many benefits — it can (i) allow one to compose components, rather than write code, (ii) provide criteria for evaluating the soundness and quality of a composition, (iii) support reuse and parametrization, and (iv) establish a platform for a host of task-enhancing services such as program synthesis, analysis, compilation, execution, and debugging.

By decomposing the problem in this way, we identify a new field of concern, which we term *end-user architecting*. Similar to end-user programming [Nar93], we recognize up front that the key issue is bridging the gap between available computational resources and the skill set of the users who must harness them — users who typically have weak or non-existent programming skills. But unlike end-user programming, end-user architecting seeks to find higher-level abstractions that leverage the considerable advances in software architecture languages, methods, and tools to support component composition, analysis and execution.

In this thesis, we argue that the key to doing this is to use an approach in which there is an explicit architectural representation of the compositions created by end users. For a given domain the architectures that could be created would be associated with a domain-specific architectural style corresponding to natural computational models for the domain (such as some variant on workflow, publish-subscribe, or data-centric styles) [SG96]. Furthermore, associated with the style and corresponding infrastructure, there would be a set of architecture services that could support analysis, execution, etc. Finally, all of these features would be made available to users through a graphical front end that supports access to component repositories, architecture construction, system execution, and various additional support services.

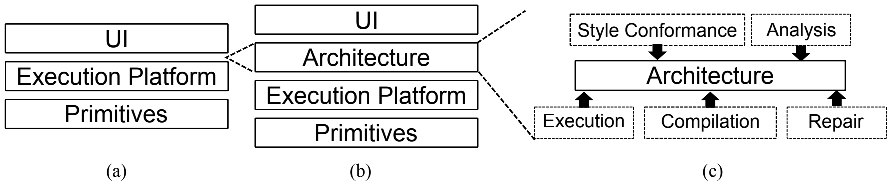


Figure 1.2: End-user Architecting Approach

This leads to a general framework of system organization in support of end-user architecting, as illustrated in Figure 1.2. Part (a) of the figure shows the current state of affairs: users must translate their tasks into the computational model of the execution platform, and become familiar with the low-level details of that platform and the primitive computational elements (applications, services, files, etc.) leading to the problems described in more detail in Chapter 3. Part (b) illustrates the new approach. Here, end-user architectures are explicitly represented as architectural models defined in a domain- specific architectural style. These models and the supporting infrastructure can then support a host of auxiliary services, including checking for style conformance, quality attribute analysis, compilation into efficient deployments, execution and debugging mechanisms, and automated repair — as shown in part (c).

In this thesis, we show that such an end-user architecting capabilities can be made available through a customizable framework that can be instantiated across a large number of domains. Concretely, we demonstrate a framework named Halo that implements this approach.

1.1 Thesis Statement

Specifically we investigate the following thesis:

> **It is possible to build an end-user composition framework that can be instantiated to provide high-quality composition environments at relatively low cost compared to existing hand-crafted environments for a broad class of composition domains.**

Next, we break down the thesis statement and elaborate the elements that compose it.

It is possible to build an end-user composition framework that can be instantiated...

Software frameworks are powerful tools that provide significant reuse benefits to the software engineering community [GS03]. By providing a general scaffolding infrastructure of libraries, their APIs, and plugins, they reduce the development effort of the developers who can now develop richer tools by plugging in into the framework functionality. Halo is one such framework that provides generic and reusable building blocks that can be reused by platform developers to create quality end-user composition environments without developing everything from scratch.

In particular, Halo provides the following capabilities:

- Reuse of existing components and compositions.
- Creation of a domain-specific vocabulary that conforms to a particular composition style (for e.g., dataflows, or publish-subscribe)
- Analyses that provide feedback for compatibility and correctness concerns (for e.g., QoS tradeoffs, performance, etc.)
- Code generation plugins that map compositions to standard execution technologies (for e.g., generation of BPEL and other scripts)
- Execution support by compiling compositions into code, execution, and feedback about execution status and results.

Platform developers can customize and reuse the building blocks provided by the Halo framework to design their own custom composition environments without developing everything from scratch.

...to provide high quality composition environments...

In this thesis, we demonstrate that the Halo framework improves the quality of end-user composition by allowing the end users to focus on assembly and configuration of computations rather than the low-level application invocation idiosyncrasies that require programming expertise. Some of the quality-enhancing features, which many composition environments lack today but that are supported by Halo include:

- **Domain specificity**: The ability to design a composition in its domain-specific vocabulary for components and computations.
- **Flexible modeling**: The ability to design a composition at an abstract level with appropriate defaults for various component properties.
- **Interactivity and feedback**: The ability to offer interactive feedback during the composition design and reuse process, and help with debugging.
- **Mismatch resolution**: The ability to resolve mismatches based on domain-specific rules. Examples include auto-correction for data mismatches, incorrect ordering of components, etc.
- **Reuse and share**: The ability to support reuse of pre-existing components and compositions and share new compositions with other developers.

...at relatively low cost compared to existing hand-crafted environments...

Ideally, end users would like a composition environment that is tailored to their own composition domain and has the qualities that we described above. However, supporting these quality requirements comes at a cost, and more so when such capabilities need to be hand-crafted from scratch.

We demonstrate that the Halo framework lowers this cost by providing building blocks that can be customized for various composition environments. Composition environments can reuse a large part of the framework, in many cases with only a few lines of code and implement capabilities such as conformance checking, reuse support, execution support and various analyses. Furthermore, as opposed to handcrafting composition environments from scratch, using Halo lowers the cost for many other reasons. First, the design decisions and components necessary when starting development from scratch are already built into the Halo framework, and can be reused. Second, working with standardized and reusable styles facilitates automated error-checking as styles encode the rules for component compositions. Last but not least, the framework addresses the needs of the larger ecosystem of stakeholders consisting of domain-experts, component developers, platform developers, etc., who currently have to handcraft everything from scratch. Having Halo's reusable building blocks reduces the design costs for such stakeholders.

...for a broad class of composition domains.

Within an increasing number of domains, end-user composition environments are an emerging trend. Even though they look very different in terms of their composition vocabulary, they often conform to common computation models — such as (i) dataflow, (ii) publish-subscribe, or (iii) call-return. For example, workflows composed in Taverna [HWS+06] in the e-science domain conform to dataflow style, widgets interactions in Ozone [MCC11] for the geospatial analysis domain conform to a variant of a publish-subscribe style, while scientific workflows in Kepler [LAB+06] correspond to a mix of data-flow and call-return style. We demonstrate that the Halo framework is general enough to support such common computation models for end-user compositions.

We demonstrate that we can associate architectural representations with end-user specifications [DEF+11] and use them for analysis. Specifically, software architecture allows one to define *architectural styles*, where each style denotes a family of systems that shares a common vocabulary and semantics of composition, conforms to rules for combining components, and identifies analyses that can be applied to systems in that family [SG96]. Such styles can be defined using a declarative language like Acme, which (in many cases) can be directly compiled for constructing systems in that style and for checking conformance with the constraints of the style. The Halo framework associates end-user compositions as instances of architecture styles and uses them as a basis for analysis, code generation and guidance to end users. This allows us to build a generic framework that (a) is reusable across different domains, (b) can support different types of computation models, and (c) support various potential analyses and quality requirements.

1.2 Thesis Evaluation

To evaluate this thesis, we applied our approach as exemplified by the Halo framework to a number of systems, each belonging to one of three architectural styles, across very different domains. Together, these example applications serve to demonstrate applicability across a breadth of styles. We then assessed the cost-effectiveness of the framework using a task-based estimation of effort and informal user feedback. Specifically, as we we will show in Chapter 6.3, Chapter 6.1, and Chapter 6.2 our approach satisfies the three claims as follows.

To evaluate **generality**, we demonstrated that the Halo framework is general enough to be applied across different domains withing common computation models such as dataflow, pub-sub and call-return. We do this validation through the following steps:

- End-User Architecting in 2 Computation Styles (Dataflow and Publish-Subscribe)
- End-User Architecting in 3 Domains (Arithmetic Expressions, Brain Imaging, and Widget framework)

Besides these domain-specific instantiations, we also demonstrate generality by different instantiations of the framework building blocks (which we will explain in detail in Chapter 3. Specifically, we have the following demonstration steps:

- Demonstrate End-User Architecting with 3 User Interface Adapters (Arithmetic Expressions, Brain Imaging, and Widget framework)
- Demonstrate End-User Architecting with 2 Execution Adapters (BPEL, SCA)
- Demonstrate End-User Architecting with 3 Analysis Adapters (Arithmetic Expressions, Brain Imaging, and Widget framework)
- Demonstrate End-User Architecting with 2 Repository adapters (Arithmetic Expressions in SwiFT and in Dyanamo)

To evaluate **cost-effectiveness**, we demonstrate that the Halo framework reduces the cost of development of end-user composition platforms. We show that the Halo framework provides common and reusable infrastructures, which are flexible to customize to develop end-user composition environments. In effect, Halo saves engineers time and development effort to build end-user architecting environments. To show effort savings, we characterize the end-user composition tasks and provide coarse-grained, task-based estimation of effort, then qualitatively assess and evaluate savings with the Halo framework relative to current practice. In particular, we perform this task analysis for three different environments.

To evaluate **support for high-quality**, we carried out a qualitative study [DHG17] where we identified the set of quality dimensions needed for end-user compositions across a number of domains, including: (a) Search and Explore, (b) Reuse support, (c) Composition Construction, (d) Analysis Support, (e) Execution support, and (f) Debugging. We demonstrated that the Halo framework improves the quality of end-user composition by supporting all of these quality dimensions.

1.3 Thesis Contributions

This thesis advances the state-of-art in the field of *end-user software engineering* by providing an approach that allows easier construction and analysis of end-user compositions. Specifically, this thesis makes the following contributions:

- **A novel technique** for end-user architecting that dramatically reduces the time, cost and difficulty of building a significant class of end-user composition environments. This technique benefits composition environment developers as they can rapidly and incrementally customize composition environments at significantly lower cost than the existing hand-crafted environments.
- **A reusable framework for end-user architecting** that provides interfaces, libraries, controls structures and the necessary plug-in points for developing high quality end-user composition environments.
- **A set of analyses** that improve the end-user composition experience. Examples include: ordering analysis, security and privacy analysis, performance analysis, and analyzing composition deploy-

ment while considering trade-offs such as performance vs. fidelity.

- **A collection of styles** that can be refined and specialized to model end-user compositions.

Additionally, the thesis contributes to the field of *software architecture* through extensions to architecture description languages to support:

- **Generic and reusable analytic capabilities** that are designed for end-user compositions, but are also relevant for architecture modeling. Examples include automatic mismatch detection and resolution, fidelity trade-offs, performance and security analysis based on styles.
- Support for **additional user interface and execution mappings** associated with the architecture description. Today, most architecture tools allow representation of architecture vocabulary consisting of properties and constraints and some minimal user-interface specifications describing how the tools can visually display the vocabulary. Halo extends this representation with additional mappings to user interface and run-time representations that could potentially be used for a number of other use cases besides end-user architecting.

Some of the publications that have arisen from this work include:

- Vishal Dwivedi, James Herbsleb, and David Garlan. What ails end-user composition: A cross-domain qualitative study. In *End-User Development. IS-EUD 2017*, volume 10303 of *Lecture Notes in Computer Science*. Springer, 2017
- Vishal Dwivedi, David Garlan, Jürgen Pfeffer, and Bradley R. Schmerl. Model-based assistance for making time/fidelity trade-offs in component compositions. In *11th International Conference on Information Technology: New Generations, ITNG 2014, Las Vegas, NV, USA, April 7-9, 2014*, pages 235–240, 2014
- Perla Velasco Elizondo, Vishal Dwivedi, David Garlan, Bradley R. Schmerl, and José Maria Fernandes. Resolving data mismatches in end-user compositions. In *IS-EUD*, pages 120–136, 2013
- David Garlan, Vishal Dwivedi, Ivan Ruchkin, and Bradley R. Schmerl. Foundations and tools for end-user architecting. In *Monterey Workshop*, pages 157–182, 2012
- Vishal Dwivedi, Perla Velasco Elizondo, José Maria Fernandes, David Garlan, and Bradley R. Schmerl. An architectural approach to end user orchestrations. In *The European Conference on Software Architecture (ECSA)*, pages 370–378, 2011
- Bradley R. Schmerl, David Garlan, Vishal Dwivedi, Michael W. Bigrigg, and Kathleen M. Carley. Sorascs: a case study in soa-based platform design for socio-cultural analysis. In *ICSE*, pages 643–652, 2011

1.4 Document Roadmap

In the remainder of this thesis, Chapter 2 describes the background of this work and discusses related solutions and their limitations. Chapter 3 describes the overall end-user architecting approach, and sets the context for introducing the Halo framework. In Chapter 4, I describe the technical architecture of Halo with key customization points and how they facilitate reuse and low-effort customization towards creating a target end-user composition environment. In Chapter 6, I discuss the evaluation of the thesis and how the Halo framework provides a general-purpose framework (Chapter 6.3) that allows building high-quality (Chapter 6.1) composition environments at a low Cost (Chapter 6.2). Chapter 7 describes the design trade-offs along with issues and limitations associated with end-user architecting and the Halo framework. Chapter 8 concludes this thesis, highlighting future work that can build on the foundations and tools developed in this dissertation.

If I have seen further, it is by standing on the shoulders of giants.

—Sir Isaac Newton

CHAPTER 2

Related Work

Software architecture emerged as a subfield of software engineering in the 1990s as a way to tackle the increasing complexity of software systems design. As computing became more pervasive, the field of end-user software engineering emerged in the early 2000's. As use of computers and programming grew across a number of domains, the field of end-user computing to study how we can realize the potential for high-end computing to perform problem-solving in a trustworthy manner across these domains. While the principle idea behind software architecture has been to allow software engineers to treat system design at a high-level of abstraction, representing a system as a composition of interacting components. In this thesis, we argue that these architectural abstractions can also be used to support end-user compositions.

The following areas have influenced the formulation and direction of this work:

2.1 End-user software engineering

End-user software engineering is a research area at the intersection of computer science and human-computer interaction [KAB+11]. It aims to empower users who write programs, but not as their primary job function. Such users may not have the skills of professional software developers and they often they face many of the same software engineering challenges: understanding requirements, carrying out design activities, supporting reuse, quality assurance, etc. As noted earlier, studies have shown that across many domains, such end-users spend about 40% of their time doing programming-related activities, but employ few of the tools and techniques used by modern software engineering [HH11]. As as result, creating computations often leads to systems that are brittle, contain numerous bugs, have poor performance, cannot be easily reused or shared, and lead to a proliferation of idiosyncratic solutions to similar problems within a domain [BGL+09].

To date, most of the research in end-user software engineering has focused on end-user *programming*, where novel forms of programming languages have been developed for enhanced usability within a domain. These include visual programming languages [Mye90], programming-by-demonstration [Cyp93], direct manipulation programming languages [HHN85a], and domain-specific languages [Fow11]. The most popular end-user development tool is the spreadsheet. Due to their unrestricted nature, spreadsheets allow relatively unsophisticated computer users to write programs that represent complex data models, while shielding them from the need to learn lower-level programming languages.

Furthermore, many end-user development activities are collaborative in nature, including collaboration between professional developers and end-user developers and collaboration among end-user developers. In addition, such end-user developers benefit from online and offline communities, where end-user developers can collaboratively solve end-user development problems of shared interest or for mutual benefit.

In contrast, this work focuses on domains in which component composition is the primary form of end-user system construction, an activity that we have termed end-user architecting. For such domains, we have argued, it makes sense to explore ways to adapt the tools and techniques of software *architecture*, rather than software *programming*. However, similar to end-user development activities, this work looks into the collaborative nature of composition environment development, where professional developers collaborate with domain experts to build the architecture styles and other parts of the end-user architecting environments.

2.2 Software architecture

There exists a large body of foundational work on software architecture that has paved the way for architecture to be used as a model to reason about a software system. The landmark paper by Perry and Wolf defined software architecture and established it as a discipline, drawing analogies from building architectural styles and forming a basis for using architectures as system models [PW92]. Shaw and Garlan characterized and codified many common styles of system architecture [SG96]. Bass, Clements, and Kazman investigated the practical issues of applying software architecture through many case studies, providing techniques for designing and analyzing architecture [BCK07].

In this work we build directly on that heritage. Key influences have been architecture description languages [MT97], the use of architectural styles [SG96, MKMG97], and architecture-based analyses [GS06]. An architectural style [AAG93] defines the vocabulary of element types, properties common to the element types in theses systems, a set of constraints on the permitted composition, and the associated analyses for reasoning about this class of systems. In essence, architectural style is useful for capturing commonalities between systems of a particular class and, consequently, variability across different classes of systems. In the past decade, there has been significant work on how styles can be formalized and applied to design systems and provide analysis capabilities [AAG93, MKMG97]. The decoupling of style from system during design makes it feasible to build generic infrastructure that can be tailored to specific domains. Consequently, Architecture description languages (ADLs) have been developed for various domains, for modeling systems, and even interchange, including Acme, C-2, Meta-H, Rapide, SADL, UniCon, and Wright, to name a few [MT97].

The premise of the end-user architecting approach is to recognize that the computational design activities performed by many communities are fundamentally architectural in nature, and therefore architectural techniques can be applied here. However, there also remain a number of gaps and challenges that require additional research and adaptation of those techniques to the needs of end users. While the end-user architecting approach builds on the rich literature of software architecture, it adds many contributions to this field.

Today, there is limited support for export and effective reuse of architectural specifications. The end-user architecting approach adds this capability through APIs that allow packaging, search and reuse of compositions. Supporting generic and reusable analytic capabilities is an even harder problem in the field of software architecture where the end-user architecting approach provides useful contributions. Furthermore, there exists limited research on providing bridging mechanisms between architectural specifications and user-interface or execution-infrastructure layers that makes end-user architecting a challenging problem. Even though, architecture definition languages like Acme provide a rich set of base styles (client-server, publish-subscribe, etc.), it is often non-trivial to design end-user composition styles from these — mainly because it requires both technical and domain-expertise to determine the rules of the domain. The end-user architecting approach demonstrates how a collection of styles can be defined across various domains through refinement and specialization of base styles [DEF$^+$11].

2.3 Software product lines and module systems

Software product lines is another related research approach where a program can be tailored to a specific application scenario based on different user-selected features [CN02]. In the last decade several methods have been established to create product line architectures. Examples include, COPA, FAST, FORM, KobrA and QADA [Mat04]. Recently, Norbert Siegmund et al. in their work [SRK$^+$11, SKK$^+$12] demonstrated how to derive non-functional properties based on user-selected features. While end-user compositions have similarities with product lines in terms of assembly of components, they differ in terms of the computation model that defines the component assembly. For example, end-user compositions could be in the form of dataflow or publish-subscribe where a configuration is not merely an assembly of components, but also the connectors that facilitate the communication between them. Similarly, there has been significant work towards designing module systems [KOE12] where developers can decompose a large system into subsystems, or modules, which can be combined into meaningful configurations.

Modularization has been a well-known mechanism for improving the flexibility and comprehensibility of a software system since the early 70s [Par72]. Today most languages use module systems in some form. For instance, modules have been used as a collection of definitions of possibly heterogeneous components (such as, functions, variables, exceptions, objects and classes) that are made available through an interface. Like product lines, module systems offer an efficient technique for packaging systems and hiding information about implementation. End-user compositions on the other hand, assemble domain-specific functions (or components) that need to be mapped to code but are not necessarily code blocks. The end-user architecting technique facilitates this mapping between end-user concepts (defined in a visual specification) to code, using an intermediate architectural specification for analysis and code generation.

2.4 Domain specific languages (DSL)

A domain-specific language (DSL) is a type of programming language or specification language that offers expressive power focused on a particular problem domain. DSLs raise the level of abstraction in software development by providing constructs to express high-level concepts from which lower-level implementations can be generated. This abstraction has allowed DSLs to be used for software construction [VDKV00], data-processing (SQL), generating documents (tex), pictures (PIC), scripting (Unix shell scripts) and other scenarios.

Architecture specifications can be considered as a form of domain-specific language. Therefore many of the benefits of DSLs also apply to architecture specifications. The similarities between architecture specifications and DSLs could be best understood from Figure 2.1.

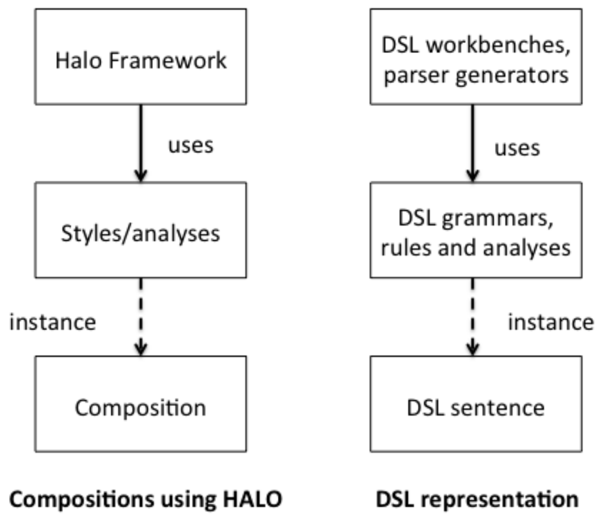


Figure 2.1: DSL based frameworks and Halo.

While both DSLs and architecture specifications allow abstraction over the concrete realization of concepts, the differences are in what they encode and how. In a broad range of application domains, DSLs are used as simple programming languages (or extensions to programming languages) via built-in abstractions and notations that simplify writing code. Software architectures, on the other hand, represent a system as a composition of interacting components independent of the programming language that implements them. Properties and constraints on use of these components (and their compositions) can then be specified in a style that allows designers to analyze systemic quality attributes and tradeoffs, such as performance, reliability, security, availability, maintainability, and so on. DSLs, on the other hand, provide similar analyses by type-checking and analyzing language grammars and rules.

Traditionally, DSLs have been used as special-purpose languages and therefore re-purposing (or re-targeting) DSLs to different domains has been a difficult problem [HV10]. Although, there has been some recent work towards DSL extension, restriction, unification, self-extension and composition [EGR12], but it still remains a hard problem. Similarly, there has been some work towards style-based composition and generalization [NA00, DEF^{+}11, AAG93, MG96] and there exist many tools and approaches to address this problem.

Furthermore, style-based composition has similarities with frameworks for creating DSLs —called DSL workbenches — examples of which include Intentional programing [Sim95], Eclipse-based Xtext [EV06], JetBrains's Meta Programming System [VS10], Microsoft's Software Factories [GS03] and SugarJ [ERKO11]. These workbenches allow language designers to define a DSL in three main parts: schema, editor(s), and generator(s). These are used for type-checking and other analyses. Similar to these DSL workbenches, the Halo framework provides the basic building blocks for code generation, analysis and feedback that can be used for end-user composition.

2.5 Tools and frameworks for end-user composition

A large number of domains require technically-naive users to compose computational elements into novel configurations, such as workflows and scripts for experiments and analyses. Such users often form large communities that share a common set of tasks, vocabulary, and computational needs. As I described in Chapter 1, examples of these communities include astronomy [DSS+05], bioinformatics [Let05], environmental sciences [VAR09], intelligence analysis [SGD+11], neuroscience [Neu], and scientific computing [Seg07]. In such communities simple turnkey or parameterized implementations are inadequate, since it is impossible to anticipate all possible configurations — hence the need for tools that can help users in creating, executing, and sharing compositions.

As a consequence, a number of powerful composition environments have been created for particular problem domains. Examples include: Loni-pipeline [RMT03] for brain-imaging compositions; Galaxy [GRH+05] for genomics; and Vistrails [BCS+05] for data-exploration and visualization for scientific applications. Other more generic composition environments, such as Taverna [OGA+06], Kepler [LAB+06], WINGS [GRD+07], and Ozone [MCC11], can be used across several domains, but typically only support a specific computation model — such as workflow or publish-subscribe.

In contrast to these efforts, the end-user architecting approach represents compositions as instances of domain-specific architectural styles, which can be analyzed for correctness and quality. This thesis attempts to lay the foundation for viewing this class of tools and frameworks as supporting a form of architecture design, and argues that there are considerable benefits in taking this point of view. Among those benefits are the ability to reuse compositions, create cross-domain analyses, provide systematic support for reuse and adaptation, support powerful auxiliary services (e.g., mismatch repair), and support execution, testing, and debugging.

2.6 Architecture-based code generation

There has been a lot of work towards code generation through the translation of architecture descriptions specified in an ADL to a programming language, and association of architecture and code. Shaw and Deline did some early work in modelling of architectures and associating them with their implementation artifacts [SDK+95]. Similarly, Aldrich et. al. created ArchJava [ACN02], which combines software architecture specifications to Java implementation in order to ensure conformance between architecture and code, and thus, support the co-evolution of both architecture and implementation. Medvidovic et al built an environment called Dradel that extended the C2 language to support architecture based evaluation of code [MRT99]. Similarly, the Model driven architecture (MDA) framework by the OMG group [BCD+03] used UML abstractions to generate code artifacts, which has been used across various domains. There have many other similar systems that used the underlying technique of mapping architecture model to code artifacts and generation of code. Examples include, generating code from design patterns [BFVY96], generating code from event models [MS11], and various other model based techniqes [SGT20]. However, the focus of many of these techniques has been to provide a generic code-generation approach rather than targeting towards supporting a framework for building a composition or a development environment based on architecture or other component models.

More recently, there has been a lot of practical work at NASA where architectural descriptions

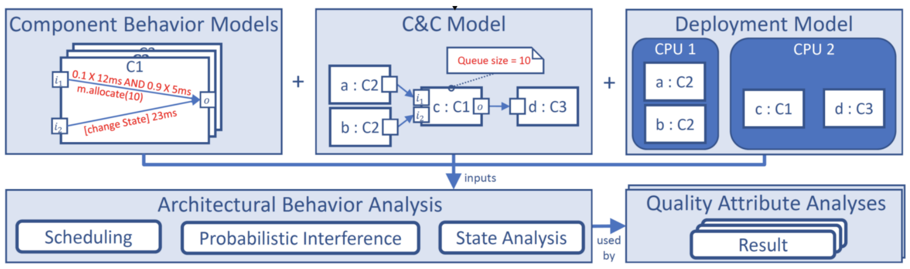


Figure 2.2: F’ architecture based analysis and code generation

have been used to generate code and test artifacts, which are used for various small satellite missions. FPrime [BCW+18] is one such open-source flight software framework developed at JPL and tailored to small-scale systems such as CubeSats, SmallSats, and instruments. As these NASA missions have modest budgets, and tight schedules, having an architectural framework that allows building blocks such as architecture, infrastructure, tools, and reusable software components reduces the overall development time and improves the quality of software. FPrime, similar to our approach, uses (1) an architecture that decomposes flight software into discrete components with well-defined interfaces; (2) a C++ framework that provides core capabilities such as message queues and threads; (3) tools for specifying components and connections and automatically generating code; (4) a collection of ready-to-use components; and (5) tools for testing flight software at the unit and integration levels.

2.7 Limitations to State-of-the-Art Addressed

In this chapter, we presented an overview of the related work to show how the state-of-the-art partially serves our thesis objective. Advances in software research provide the language, model, and analysis to represent and reason about a system’s software architecture, giving us the powerful notion of architectural style. Research in end-user software engineering domain, helps with techniques related to collaborative aspects of software development in domains where end users and software professionals work together for end-user development. Research in product lines and DSLs helps us with a better understanding of supporting variability of the frameworks. Furthermore, the various end-user composition tools and frameworks provide us a better understanding of the types of analyses supported (and often missing) in such environments.

However, current approaches present a number of limitations and unresolved issues, which we address in this thesis. In particular, traditional end-user computing approaches have focused on relatively unsophisticated computer users (rather than programmers with advanced skills). Segal refers to such advanced programmers as “professional end-user developers" — people such as research scientists who work in highly technical, knowledge-rich domains and who develop software in order to further their professional goals [Seg07]. End-user architecting is a common activity for such end users. Naturally, while the traditional end-user software engineering research has focused on spreadsheets, visual programming, and programming by example, this thesis primarily focuses on the needs of ‘professional end-user developers’ such as neuroscientists, intelligence analysts, financial mathematicians, planetary scientists, etc., that used specialized end-user composition environments with complex analytic needs. Architecture-based analyses are not only useful to such end users and problem domains, but it also allows development of rich composition environments with advanced analytic features [DEF+11]. Our approach allows such capabilities to be made available to a wide number of composition domains.

These areas of research has helped to clarify the key requirements and features of end-user composition tools that we detail in Chapter 3.

An approach to support design and analyses of compositions using high-level abstractions designed using architectural styles...

—End-User Architecting

CHAPTER 3

End-User Architecting

The objective of this thesis is to empower platform developers with an approach and the tools to add analytic capabilities to end-user composition environments. In this chapter, we outline the key features of our approach, which consists of a generic framework called Halo that can be instantiated across various domains, and discuss the technical challenges that must be addressed to make this approach work.

3.1 End-user architecting overview

Across many domains, the key design activity of end users comprises of composing various computational elements while satisfying different domain-specific constraints, and run various functional and non-functional analyses. We argue that the computational design activities performed by end users in these communities are fundamentally architectural in nature. Recognizing that, one can then explore how modern techniques and tools in support of *software architecture* can be applied to this new area of *end-user architecting*, which we define as:

> "*End-user architecting is an approach that provides ways to incorporate software architecture based techniques to design quality end-user composition platforms, which can be used by end users to compose, analyze, and execute high-quality compositions in an efficient manner.*" [GDRS12]

Software Architecture emerged in the 1990s as an important subfield in software engineering. End-user architecting aims to build on the large body of work in Software Architecture. Since its emergence, there has been a significant development of foundations, tools, and techniques to aid software architects. These include formal and semi-formal architecture description languages (ADLs) [MT97], architecture-based analyses [GS06], architecture reconstruction tools [SAG+06], architecture evaluation methods [CKK01], architecture handbooks [BMR+96], architecture style definition and enforcement [GMW00], and many others.

The principle idea behind software architecture is to allow software engineers to treat system design at a high-level of abstraction, representing a system as a composition of interacting components. Properties of those components and their compositions can then be specified in a way that allows designers to analyze systemic quality attributes and tradeoffs, such as performance, reliability, security, availability, and maintainability [SG96].

We extend this to *end-user compositions* through an explicit architectural representation of the compositions. The approach relies on the fact that for a given domain, the end-user specifications could be associated with a domain-specific architectural style corresponding to natural computational models for the domain (such as some variant on workflow, publish-subscribe, or data-centric styles). Further, associated with the style and corresponding infrastructure, there could be a set of architecture services that could support analysis, execution, etc. Finally, all of these features could be made available to users through a graphical front end that supports access to component repositories, architecture construction, system execution, and various additional support services. As we discussed earlier in Chapter 1, Figure 3.1 describes the high-level idea behind this approach.

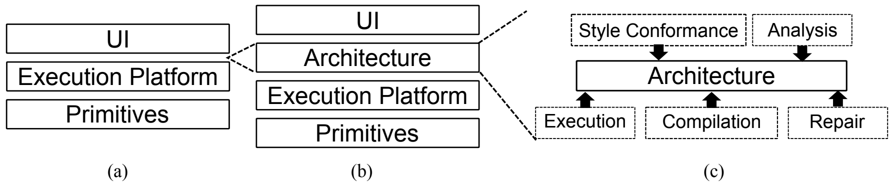


Figure 3.1: End-user Architecting Approach

Part (a) of the figure shows the current state of affairs: users must translate their tasks into the computational model of the execution platform, and become familiar with the low-level details of that platform and the primitive computational elements (applications, services, files, etc.). Part (b) illustrates the new approach. Here, end-user architectures are explicitly represented as architectural models defined in a domain-specific architectural style. These models and the supporting infrastructure can then support a host of auxiliary services, including checking for style conformance, quality attribute analysis, compilation into efficient deployments, execution and debugging mechanisms, and automated repair — as shown in Part (c).

The following aspects of software architecture provide us the mechanisms to handle some of the requirements for end-user composition:

- **Component composition:** Software architecture represents a system as a composition of components, supporting a high-level view of the system and bringing to the forefront issues of assignment of function to components, component compatibility, protocols of interaction between components, and ways to package component compositions for reuse.

- **Domain-specific computation models:** Software architecture allows developers to represent a system using compositional models that are not restricted by the implementation platform or programming language, but can be chosen to match the intuition of designers. Specifically, software architecture allows one to define *architectural styles*, where each style denotes a family of systems that shares a common vocabulary of composition, conforms to rules for combining components, and identifies analyses that can be applied to systems in that family [SG96]. Styles may represent generic computational models such as publish-subscribe, pipe-filter, and client-server. Or, they may be specialized for particular domains [Mon99a, MG96].

- **Analysis:** Software architecture allows developers to perform analysis of quality attributes at a systems level. This is typically done by exposing key properties of the components and their interactions, and then using those properties in support of calculations to determine expected component compatibility, performance, reliability, security, and so on [GS06]. This in turn allows developers to make engineering tradeoffs, for example balancing attributes like fidelity, performance, and cost of deployment to match the particular business context. Additionally, in some cases it is possible to build analytic tools that not only detect problems, but also suggest possible solutions [SG03].

- **Reuse:** Software architecture supports several kinds of reuse. First, architectural styles provide a basis for sharing components that fit within that style [Mon99a, MG96]. Modern examples of this include platforms like J2EE and frameworks like Eclipse. Second, software architectures permit the definition of reusable patterns that can be used to solve specific problems [BCK07, BMR+96]. Third, most architectural models support hierarchical description, whereby a component can be treated as a primitive building block at one level of composition, but refined to reveal its own sub-architecture.
- **Execution support:** For some architectural styles tools can generate implementations. Typically this is done by using a repository of components that conform to the style, and then compiling the system description into executable code [GRS+05]. Additionally, software architectures can be used for run-time monitoring and debugging [YGS+04].
- **Reducing implementation cost through architecting:** Implementations supported by software architecture may significantly reduce the time-consuming tasks for implementation. Robert Monroe demonstrated in his thesis [Mon99b] that design of programming environments can be done in a cost-effective manner by using architectural specification via styles, which can define both the computation specific elements as well as domain-specific constraints. Furthermore, such an architecture layer can be used to generate code, which can lower the overall cost of software development. Examples include, systems such as F' **??** that allow generation of code for flight satellite systems at NASA. Finally, styles can be designed by domain experts allowing the platform developers to reuse the pre-designed domain-expertise and analytical capabilities.

3.2 Architecture-based component composition

In the subsequent part of this chapter we discuss an in-depth example of how the end-user architecting approach is used to build an end-user composition environment in the Neuroscience domain. But before we do that, we must discuss 3 key techniques for end-user architecting.

One of the first techniques is the explicit representation of a composition vocabulary in an architectural style. Examples of such styles include: variations of dataflow, publish-subscribe, etc. This allows us to define the various constraints and rules for compositions, and the analyses that can be run on them. End-user architects who need to design such styles may need not start from scratch; they are provided various base styles that can be extended and refined to define rules for composition in a particular domain. In Section 3.3 we describe in detail what such a style looks like in the Neuroscience domain, and a general-purpose approach to build such styles.

The second technique, and the centerpiece of the end-user architecting approach, which is key to building tooling for end-user architecting, is the explicit use of end-user architecture as a layer (as shown in Figure 3.1) to drive end-user composition. While there may be low-level tools, programs or scripts, they can be composed together as services, and an explicit architecture-based middleware can guide creating the compositions, debugging and executing them. This end-user architecture layer can provide support for guidance, analysis, reuse support, and execution without end users having to worry about implementation-specific technical details. In Section 3.4, we describe an example of an end-user architecture layer for the

Neuroscience domain.

This brings us to our third technique, where the instantiation of the end-user architecture layer is supported by the Halo framework. While the end-user architecture layer can be purpose-built for each environment, it is costly to do so and Halo provides a set of building blocks to reduce this cost. In Section 3.5, we introduce the Halo framework for end-user achitecting and what the application of Halo means in the domain of end-user architecting.

3.3 Using architecture style to define Neuroscience composition vocabulary

The first major technique employed by the end-user architecting approach is the use of a stylized software architecture model to represent software compositions. In one of our previous works, we demonstrated how different architecture styles can be used for assembling computations in various domains [DEF+11]. Not only do such styles allow the definition of a domain-relevant vocabulary, but through refinements and specializations these styles can be used to represent dataflows, publish-subscribe communications and a mix of other computation models. The use of such styles gives us leverage to use existing architectural analysis techniques to provide advice and guarantees to users about their compositions via various formal analyses. Furthermore, architecture styles allow building domain-vocabularies that can support composition and debugging, which is critical to support various common end-user composition tasks.

Software architecture provides the high-level structure of a system, consisting of components, connectors, and properties [SG96]. While it is possible to model the architecture of a system using such generic high-level structures, it is crucial to use a more specialized architectural modeling vocabulary that targets a family of architectures for a particular domain. This specialized modeling vocabulary is known as an architectural style [SG96] and it defines the following elements:

- **Component types:** represent the primary computational elements and data stores.
- **Connectors types:** represent interactions among components.
- **Properties:** represent semantic information about the components and connectors.
- **Constraints:** represent restrictions on the usage of components or connectors, e.g. allowable values of properties, topological restrictions.

Acme [GMW97b] is an architectural definition language (ADL) that provides support to define such styles. Acme's predicate-based type system allows styles to inherit characteristics from other styles. When a style element (or the style itself) inherits other elements, not only does it inherit the properties, but also the constraints defined on its usage. Acme allows definition of components and connect and their composition constrains through styles, which allow compositions in a high-level visual language. Such visual compositions can be compiled into code and executed. The constraints of the visual composition are defined in styles, which can be inherited and refined based on various properties.

An example style for dataflow compositions,created as part of my research, is SCORE [DEF+11], which provides a vocabulary for description of workflows architectures. It abstracts the specification of workflows to only the core properties and constraints of concern that are relevant to the specific domain. The SCORE style specifies rules that are evaluated at design time, enforcing restrictions on the kinds of components users can compose. For example, a given domain can impose restriction on the types of data a component can consume and this can specified in a style that is used by the composition. Writing these rules involves some degree of technical expertise, but these are associated with the architectural style, which is written once by an end-user architect, and then used by end users for modeling workflows based on the style.

Table 4.4 shows SCORE architectural types, functions and constraints that are used to specify workflows using SCORE. These constrants are based on Acme's architecture constraint language, where constraints are expressed as first-order predicates over architecture structure and properties of the workflow elements. The basic elements of the constraint language include constructs such as conjunction, disjunction, implication and quantification. Various domain-specific constraints can be defined based on Acme's constraint language. Such constrains not only prohibits end users from creating inappropriate service compositions, but also promotes soundness by ensuring feedback mechanism via marking errors when a component fails to satisfy these constraints. More details of how SCORE is refined and specialized with multiple domain-specific constructs are explained later in this chapter.

Properties and constraints on architectural elements can be used to analyze systems defined using SCORE. Table 3.2, for example, displays some analyses that are built using SCORE properties, such as analyzing a workflow for correctness, and various domain-specific analyses based on workflow properties. Some of the examples of such analyses written in Acme ADL are presented in [GS06]. The rules for these analyses are written as predicates, which allow compositions to be analyzed for correctness while end users design them.

3.3.1 Neuroscience example

Functional magnetic resonance imaging (fMRI) is a common form of analysis performed by neuroscientists in the brain-imaging domain to understand the behavior of the human brain [Pek06]. A typical fMRI analysis consists of sequences of computations over brain image data to support hypotheses or interpretations, such as assessing the evolution of cognitive deficits in neurodegenerative diseases [Eid09]. Figure 3.2 illustrates a typical image translation process.

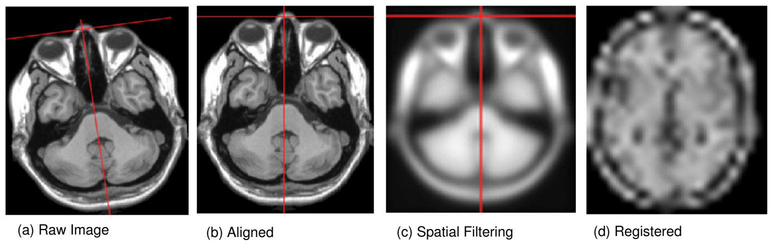


Figure 3.2: Brain image data viewed after individual pre-processing steps.

Neuroscientists today have at their disposal large repositories of brain imaging data, such as the BIRN Data Repository [Bio] and the Portuguese Brain Imaging Network Project [BIN]. Neuroscientists also have access to a large variety of processing tools, which perform functions such as those listed in Table 3.3.

Today, while professional neuroscientists can easily identify the steps required for processing brain imaging data, because of a proliferation of possible tool implementations for each step and their idiosyn-

Components	Description
`DataStore`	Components for data-access (such as file/SQL data-access)
`LogicComponent`	Components for conditional logic (such as join/split etc)
`Service`	Components that are executed as a service call
`Tool`	Components who's functionality is implemented by tools
`UIElement`	Special-purpose UI activity for human interaction
Connectors	Description
`DataFlowConnector`	Supports dataflow communication between the components.
`DataReadConnector`	Read data from a DataStore Component
`DataWriteConnector`	Write data to a DataStore Component
`UIDataFlowConnector`	Provides capabilities to interact with UIElements
Ports	Description
`configPort`	Provides an interface to add configuration details to components
`consumePort`	Represents data-input interface for a component.
`providePort`	Represents data-output interface for a component.
`readPort`	Provides data-read interface for DataStore component
`writePort`	Provides data-write interface for DataStore component
Roles	Description
`consumerRole`	Defines input interface to DataFlow/UIDataflow connectors
`providerRole`	Defines output interface to DataFlow/UIDataflow connectors
`dataReaderRole`	Defines input interfaces for the DataRead/DataWrite connectors
`dataWriterRole`	Defines output interfaces for the DataRead/DataWrite connectors
Acme Functions	Description
`Workflow.Connectors`	The set of connectors in a workflow
`ConnectorName.Roles`	The set of the roles in a connector
`self.PROPERTIES`	All the properties of a particular element
`size( )`	Size of a set of workflow elements
`Invariant`	A constraint that can never be violated
`Heuristic`	A constraint that should be observed but can be selectively violated
Constraint types	Example

`Structural`	Checking that connectors have only two roles attached
	`rule onlyTwoRoles = heuristic size(self.ROLES) = 2;`
	`rule allValues = invariant forall p in self.PROPERTIES | hasValue(p);`

`Membership`	Ensuring that a workflow contains only 2 types of components `rule membership-rule = invariant forall e: Component in self.MEMBERS |declaresType(e,ComponentTypeA) OR declaresType(e,ComponentTypeB);`

Table 3.1: Illustrative example of composition elements in SCORE style.

cratic parameterization requirements, they find it difficult to choose and assemble tools to implement these steps. Furthermore, while these experts can debug a processing script by examining the outputs, novices are typically unable to do this. As an example of the complexity introduced by tool parameterization, Figure 1.1 illustrates a part of a typical script in which a single logical processing step requires the specification of 9 parameters*.

Additional complexity arises because of implicit sequencing constraints. For example, a mandatory step in fMRI analysis is to perform pre-processing operations on brain image data to remove or control

*In practice, the number of parameters ranges from 5 to 25.

	STRUCTURAL ANALYSIS	TYPE
`Data Integrity`	Data-format of the output port of the previous connector matches the format of the input port	Predicate based
`Semantic correctness`	Membership constraints for having only limited component types are met	Predicate based
`Structural soundness`	All Structural constraints are met, and there are: - no dangling ports - no disconnected data elements	Predicate based
	DOMAIN-SPECIFIC ANALYSES	**TYPE**
`Security/Privacy Analysis`	Identify potential security/privacy issues based on rules	Program based
`Order Analysis`	Evaluate if ordering of two services makes sense	Program based

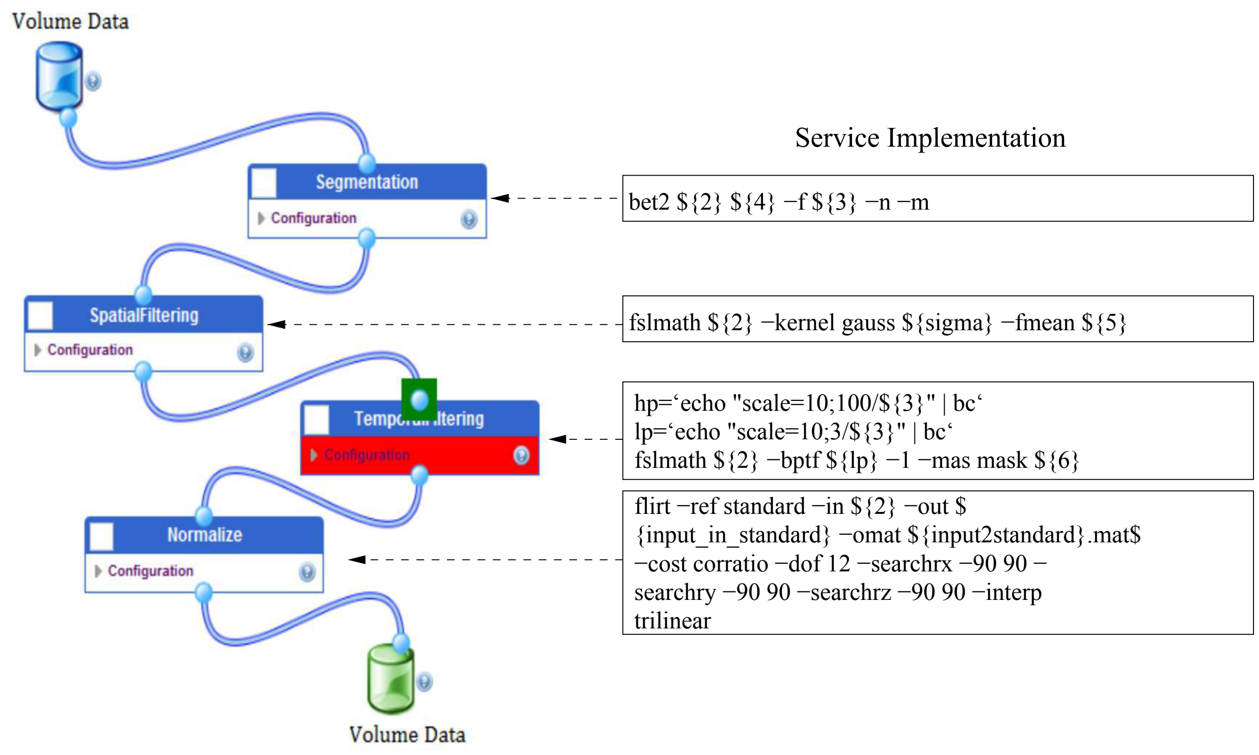
Table 3.2: Illustrative examples of some architecture-based analyses

Operation	Description	Tool name
Align	Alignment of an fMRI sequence based on a reference volume (i.e. motion correction, direction correctness)	*fslmaths, fslroi, mcflirt*
Segmentation	Segmentation of a brain mask from the fMRI sequence	*bet2, fslmaths, fslstats*
Spatial Filtering	Compute spatial density estimates for Neuroscience images, and filter the volumes accordingly	*fslmaths, susan*
Temporal Filtering	Blur the moving parts of images, while leaving the static parts.	*fslmaths*
Normalize	Translating, rotating, scaling, and may be wrapping the image to match a standard image template	*flirt*
Register	Align one brain volume to another using linear transformation operations (such as rotation, translations, etc.) or non-linear transformations (such as warping, local distortions, etc.)	*flirt, fnirt*

Table 3.3: Some tools for brain-imaging processing.

some aspects that can affect the overall analysis [Str06] (such as aligning one brain volume to another using linear transformations operations like rotation, translation, etc.). While experts may learn these constraints through trial and error, there are no tools to guide less-expert end users.

There are many possible ways to encode image data and analysis results, and neuroscientists must ensure that encodings match between steps. This further complicates composition because neuroscientists must be aware of these formats and carefully select compatible steps or manually locate transducers that can bridge mismatches.



```
bet2 ${2} ${4} −f ${3} −n −m
```

```
fslmath ${2} −kernel gauss ${sigma} −fmean ${5}
```

```
hp=‘echo "scale=10;100/${3}" | bc‘
lp=‘echo "scale=10;3/${3}" | bc‘
fslmath ${2} −bptf ${lp} −1 −mas mask ${6}
```

```
flirt −ref standard −in ${2} −out $
{input_in_standard} −omat ${input2standard}.mat$
−cost corratio −dof 12 −searchrx −90 90 −
searchry −90 90 −searchrz −90 90 −interp
trilinear
```

Figure 3.3: A problematic Neuroscience workflow that misses ‘alignment’ of data before ‘temporal filtering’.

Key features of our end-user architecting approach to this domain are:

1. **Architecture representation:** Architectures are explicitly represented in a system layer that stores compositions as workflows and provides a repository of processing steps and transducers. The main components made available in this prototype were derived from the FSL tool suite (e.g., *bet2, fslmath, flirt*) [FMR].

2. **Architecture style:** Compositions are defined using a formal workflow architectural style, which defines computational elements specific to the Neuroscience domain, and (b) it provides additional properties and domain-specific constraints (such as checking ports for different data encodings and other content of brain-image data) that allow the correct construction of workflows within the Neuroscience domain.

3. **Analysis:** The properties of the style elements are used for designing various domain-specific analyses for the brain imaging domain. An example is data mismatch analysis to support the detection of data mismatches in the Neuroscience compositions and to suggest repairs that can resolve these mismatches based on an end user’s quality of service requirements [VEDG+12].

4. **Execution support:** Workflows are compiled into BPEL scripts, which are executed on a service-oriented platform, providing feedback and debugging capabilities to the end users.

5. **Services:** The brain imaging platform provides services to end users tracking the history of operations performed and access to brain imaging data sets.

6. **Reuse:** Workflows are encapsulated as parameterized components for later reuse and adaptation.
7. **User Interface:** A web-based graphical interface is provided for workflow construction, analysis, and execution.

3.3.2 Architecture vocabulary for Neuroscience compositions

As we briefly mentioned in the Section 3.3, domain-specific architecture styles for the composition vocabulary can utilize incremental refinement and specialization of common architecture styles based on data flow, publish-subscribe, etc. and this allows easier representation of various types of compositions. Figure 3.4 shows how a SCORE data flow style (defined in Acme) is refined to the domain of Neuroscience.

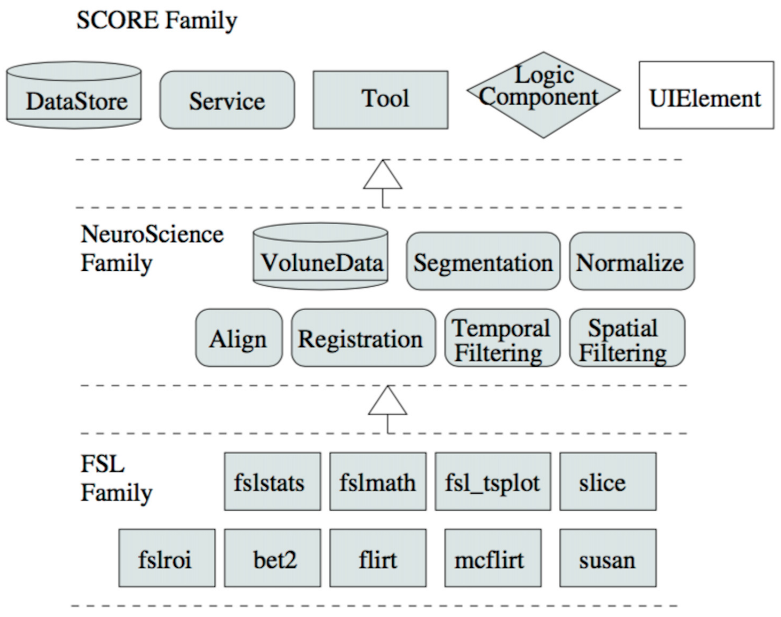


Figure 3.4: Style derivation by inheritance.

We introduced the SCORE data flow style in Section 3.3. For Neuroscience compositions, such a base style can be extended and components can be further refined with additional properties. A snippet of such style refinement in Acme is shown below. For instance, this snippet of Acme style definition defines a brain-imaging component type VolumesData that extends the generic component DataStore from the Score style with additional properties from the Neuroscience domain. Similarly, ports and roles can be further specialized with additional properties and constraints relevant to the Neuroscience domain.

Such an incremental construction of domain-specific vocabulary allows a greater degree of reuse as styles from one problem domain can be built based on the base styles from other domains. Acme currently provides tooling to define such style in AcmeStudio where end-user architects can use drag-and-drop interfaces on AcmeStudio to compose these and the tooling assists in debugging and fixes.

Program 1 Example snippet of Score Acme style.

```
import families/SCORE-FAM.acme;
Family BING extends SCORE-Fam with {
    Connector Type writeData extends DataWriteConnector with {
        Role provider = {
        }
        Role dataWriter = {
        }

   Component Type VolumesData extends DataStore with {
        Port readData = {
        }
        Port writeData = {
        }
        Property NumVolumes : int;
        Property numberOfInputs : int;
        Property highPassFilterCutoff : float;
        Property TimeOfAquiringVolume : float;
        Property outputDirectory : string;
   }
   Port Type VolumeReadPort extends readT with {
        Property volumeListName : string;
        Property data : BrainImaging.dataType;
   }
   ...
}
```

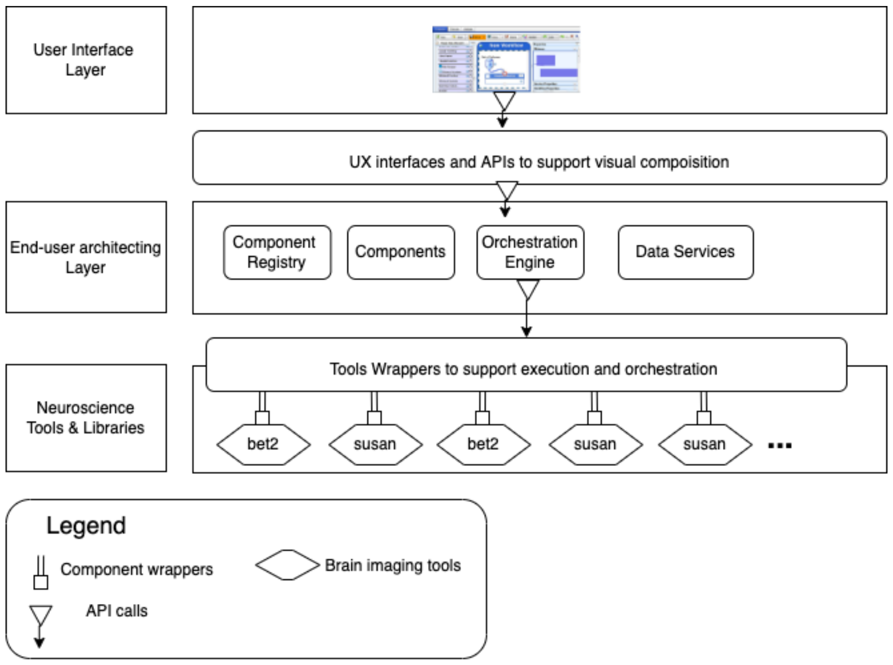
3.4 Using end-user architectures to drive Neuroscience compositions

In section 3.3 we described how end-user composition vocabulary for Neuroscience compositions can be defined with architectural styles in Acme. In this section we describe how we can make an explicit use of an end-user architecture layer to drive end-user composition, support reuse and execution.

As an illustrative example of this, Figure 3.5 shows a typical implementation architecture of a Neuroscience workflow execution environment. At the lowest level of granularity are various Neuroscience tools and scripts that end users must combine together, the user interface for this can be supported by an intermediate architecture layer that not only makes this orchestration possible, but also supports various domain-specific analyses that would otherwise not be possible if such a layer did not exist.

As a first step to build such an environment, individual programs and libraries need to be componentized so they can be reused within the framework. While the execution model for such components may vary, for our implementation we assumed a Service Oriented Architecture (SOA) and migrated the brain imaging scripts and libraries to services. Each component in this model performs a key business function, and we defined services with their input and output signatures. The service definition can use existing frameworks like Apache SCA. For instance, the table below defines some key brain-imaging operations and the fsl programs that implement them. Each of these operations is exposed as a service by wrapping a script-execution as a web-service.

Once the key functions are identified, the individual scripts can be wrapped as components by using



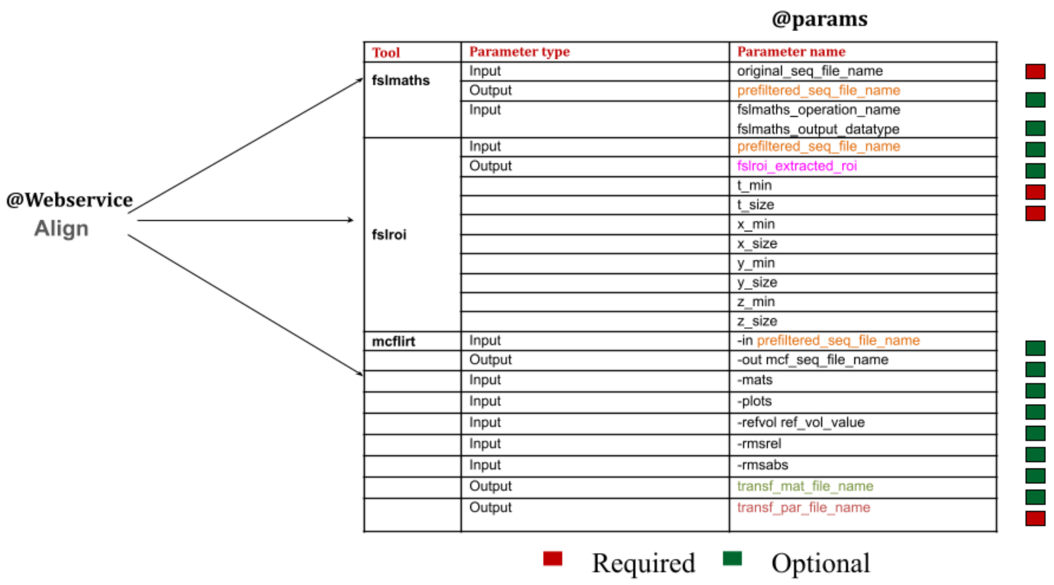
User Interface Layer
UX interfaces and APIs to support visual compoisition
End-user architecting Layer
Component Registry
Components
Orchestration Engine
Data Services
Neuroscience Tools & Libraries
Tools Wrappers to support execution and orchestration
bet2
susan
bet2
susan
susan
...
Legend
Component wrappers
Brain imaging tools
API calls

Figure 3.5: Using end-user architectures to drive Neuroscience compositions

Operation	**FSL Programs that implement this operation**
Align	fslmaths, fslroi, mcflirt
Segmentation	bet2, fslmaths, fslstats
Spatial filtering	fslmaths, susan
Temporal filtering	fslmaths
Normalize	flirt

Table 3.4: Mapping brain-imaging programs to functions

the framework interfaces and are exposed as services. For example, Figure 3.6 shows how we define web-service components and their signatures by wrapping up low-level scripts. End users in this world can build compositions using front-ends that use these individual services. The architecture layer then is responsible for orchestration of these components and help with the execution, analysis and debugging. While the individual visual composition, runtime and assembly of components can vary across environments, having a general-purpose framework that can assist in this construction helps with the overall development and keeps the costs low. We will discuss on how this can be done in further detail in the next few Sections.



Tool	Parameter type	Parameter name	
fslmaths	Input	original_seq_file_name	🟥
	Output	prefiltered_seq_file_name	🟩
	Input	fslmaths_operation_name	🟩
		fslmaths_output_datatype	🟩
fslroi	Input	prefiltered_seq_file_name	🟩
	Output	fslroi_extracted_roi	🟩
		t_min	🟥
		t_size	🟥
		x_min	
		x_size	
		y_min	
		y_size	
		z_min	
		z_size	
mcflirt	Input	-in prefiltered_seq_file_name	🟩
	Output	-out mcf_seq_file_name	🟩
	Input	-mats	🟩
	Input	-plots	🟩
	Input	-refvol ref_vol_value	🟩
	Input	-rmsrel	🟩
	Input	-rmsabs	🟩
	Output	transf_mat_file_name	🟩
	Output	transf_par_file_name	🟥

Figure 3.6: Componentization of scripts

3.5 Using a framework to build a composition environment in the Neuroscience domain

While end-user architecting platforms described in Section 3.4 can be custom built for each domain, doing so is costly. Such costs can be amortized across such developments if we had a framework that provided much of these features as building blocks. Platform developers can then use these building blocks to build composition environments for their domains. Not only can this lower the overall cost of platform development, this approach could lead to better reuse and more analytic support that can potentially be shared across different use-cases.

To fulfil this goal, we created a framework, called Halo, that provides generic and reusable infrastructure than can be tailored to particular system styles and further customized to specific composition scenarios. This customizable end-user architecting framework has many advantages. Providing a substantial base of reusable infrastructure greatly reduces the cost of development. Providing separate customization mechanisms allow developers to tailor the framework to different composition environments with relatively small increments in effort.

This is enabled by a two-step approach: (i) a generic reusable infrastructure of libraries to represent end-user architectures, and (ii) a collection of customizable parts to integrate the end-user architecture layer with their user interfaces and runtime environment with the help of various adapters.

In particular, Halo provides the following mechanisms to support end-user composition:

1. **Architecture representation:** By having compositions explicitly represented in an architectural layer, compositions can be formally represented and analyzed. Compositions can be defined using a formal architectural style that defines the composition constraints and the overall vocabulary (as described in Section 3.3).

2. **Analysis:** The properties of the style elements can be used for designing various domain-specific analyses for given domain. Examples include: data mismatch analysis [VEDG+12], privacy analysis, etc.
3. **Execution support:** Compositions can be compiled into executable scripts that run on a execution platform.
4. **Reuse:** Compositions can be encapsulated as parameterized components for later reuse and adaptation.
5. **Adapters:** Extensible bridging components allow integration of the architecture layer with the UI, runtime, and analysis plugins.

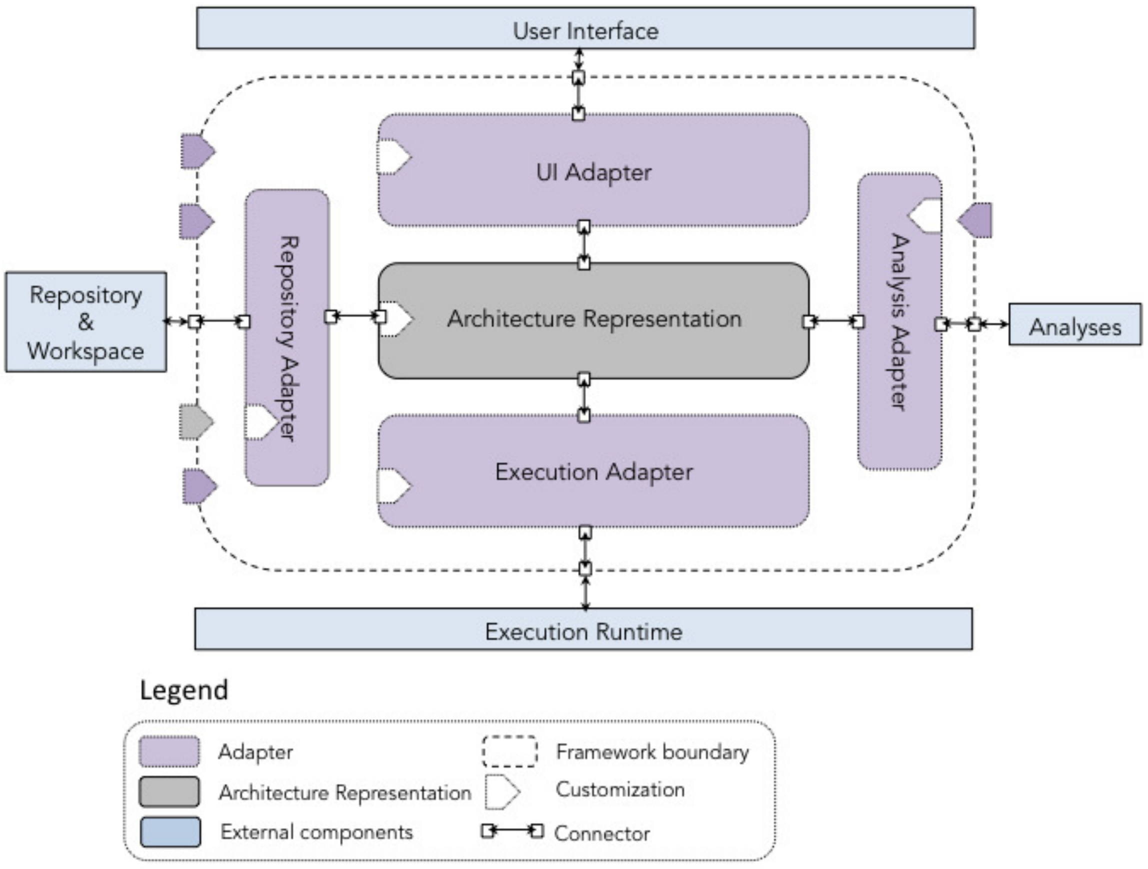


Figure 3.7: Halo framework with notional customization points

We will discuss the design of Halo in detail in Chapter 4. However, Figure 4.1 shows a high-level view of the different layers and the customization points. At a high-level Halo as an end-user architecting framework provides the following capabilities: (a) the ability to represent different types of compositions and constraints in Acme, (b) an end-user architecture layer that can map visual compositions to the corresponding architecture, and (c) a framework that provides a library of components, including various adapters that allow integration of individual layers that can be independently customized to build the composition environments.

The next section provides an example application of the Halo framework in the Neuroscience domain, and explains the tasks involved in such an instantiation.

3.5.1 The building blocks of Halo

The building blocks of the framework provide support for construction, reuse, advanced analysis, execution and sharing and are shown in Figure 4.1.

The centrepiece of Halo is the *Architecture layer*, which is customizable with style definitions to support different domain-specific composition vocabularies and constraints. This layer provides support for architectural type-checking and architectural analysis. Furthermore, it mediates the translation between visual specifications and their executable representations allowing end users to execute their compositions.

The *User interface* allows end users to compose visual components and the *Execution platform* supports their execution. While these are important elements of any end-user composition environment, the Halo framework provides hooks to integrate UI and run times. A large number of UI technologies exist today to build interfaces. Similarly, a large number of execution environments exist today that allow execution of compositions. The Halo framework provides generic adapters that can be customized to integrate the UI and execution platforms.

Halo provides four types of Adapters: a *UI Adapter*, an *Execution Adapter*, an *Analysis Adapter*, and a *Repository Adapter*. These adapters allow API calls to support integration of different building blocks and bridging of vocabularies. The UI adapter for example, provides task support for common end-user tasks such as drawing compositions, search, analysis, execution and reuse. The execution adapter provides libraries for invoking execution runtimes. The Analysis adapter allows integration and invocation of external analyses. Furthermore, the Repository Adapter allows integration of external compositions into Halo by providing plugins for language translation and import functions.

Next, we show a simple example of how different stakeholders can instantiate different building blocks of the framework to create a brain imaging composition environment.

3.5.2 Example: Building a Neuroscience environment using Halo

As an example of Halo instantiation, consider the domain of Neuroscience, for the problem scenario that we defined in this previous section. Halo provides various building blocks that can be customized and reused, instead of developing them from scratch.

Next, I describe how Halo supports the different stakeholders to build a Neuroscience composition environment described in the above example.

Legend

▨	Outside framework
🟦	An instance provided (support may exist for other types)
☐	Within framework

Role	*Task*	*Brain imaging (BI) instance*	*Framework Support*
Component Developer	1. Implement tools to perform specific computations. (Referred to as components)	1. Fslmaths, fslroi, bet2, and fslstats (ref. Table 4.1)	1.
	2. Create/Collect data required for composition. (Also treated as components)	2. BI datasets* in file formats such as NIFTI, FSL (vtk), ASCII etc.	2.
Component Integrator	1. (a) Wrap components to provide Halo interfaces. (b) Wrap data as data-service by implementing the Halo data-service interface.	1. (a) BI tools as a web service. (b) BI data as a web service.	1. Wrapper API that define access protocols, execution method, and asynchronous or synchronous behavior.
	2. Provide the required component meta-data in the registry	2. Brain-imaging component functions, ownership details, and other meta-data in the registry.	2. A registry and schema to store and search component data.
	3. Deploy components	3. BI web services deployed on a web server.	3. A sample Tomcat based execution and deployment infrastructure.
Domain† Expert	1. Select the visual presentation for compositions	1. Workflow based composition for brain-imaging computations	1. Several representative examples of composition layouts, such as data-flows, workflows, pub-sub, etc.
	2. Identify the composition vocabulary‡ in a particular domain.	2. Informal vocabulary for a brain-imaging composition style (ref Sec 5.1.2)	2. A tutorial on composition styles; with example styles including data-flows, workflows and pub-sub.
	3. Provide component classification schema (for component browsing and search).	3. Information about component meta-data such as tool functionality, type hierarchies, and access rules that aids in better classification.	3. A tutorial on component classification along with an ontology editor with sample classification data.
	4. Define a set of analyses (or select existing analyses) that should be performed on the composition style.	4. Analyses relevant for brain-imaging compositions such as mismatch-resolution, ordering analysis etc.	4. Collection of sample analysis plugins, such as for performance, security, ordering and mismatch analysis, etc.
	1. Develop a user interface for the	1. A workflow-based UI for BI compositions.	1. An example web-based workflow composition environment

* From brain imaging data repositories such as BIRN (www.birncommunity.org) and INCF (www.incf.org)

† The domain expert has expertise in the composition domain, and defines the vocabulary, consisting of 'informal' descriptions about computations, their properties and constraints. A framework instantiator encodes this in a formal architectural style.

	composition tool. Choices (a) Reuse the framework-provided UI (b) Create a new UI.		(SWIFT) developed using Halo framework.
	2. Develop the execution platform for compositions. Choices (a) Reuse the framework-provided SOA execution platform. (b) Create a new execution platform.	2. SOA-based execution infrastructure running BPEL based orchestrations	2. An example SOA execution platform support that executes compositions compiled into BPEL orchestrations. Support to plug in other kinds of execution platforms, such as for shell scripts, tool pipeline etc.
Framework Instantiator	3. Customize the Halo framework through plugins (a) Create a domain-specific architecture style to instantiate the architectural representation plugin. (b) Implement the analyses identified by the domain expert. (c) Enter the component classification information in a domain-ontology.	3. Halo framework customized with plugins to support brain-imaging workflows. (a) Brain imaging style (b) Analyses relevant for brain-imaging compositions such as mismatch-resolution, ordering analysis etc. (c) Brain imaging ontology.	3. Plugins for (i) architectural representation, (ii) reuse support, (iii) component and data reference (iii) analyses, iv) component registry, and (v) execution-support (ref. Section 4.3) (a) An architectural representation plugin that associates compositions with their architectural representations. (b) An API that provides the skeleton for developing analysis plugins, and Sample analysis plugins, such as for performance, security, ordering and mismatch analysis, etc. (c) Component and data reference plugin that uses the ontology to support classification and search.
	4. Define mappings between the 3 layers represented by {end-user specification, architecture, and executable code}. Specifically, (a) Translation from end-user specification to architecture representation (b) Translation from Acme specification into executable code Choices • Reuse the framework-provided plugins. • Create new plugins	4. (a) Mapping from BI compositions (visual language) to architectural representation (b) Mapping from Acme to BPEL.	4. Compiler plugins that translate block structured patterns in one specification to block structured patterns in another --- based on the technique adapted from [ODHv06] Examples include: • Brain imaging scripts to Acme • Compositions in JSON to Acme • Acme to BPEL • BPMN to BPEL (b) An API that provides the skeleton for developing compiler plugins.
	5. Synchronize the models in the 3 layers on	5. Halo event mechanism to for communications	5. An event mechanism implemented by the Halo

3.6 Summary

In this chapter, we argued that the computational activities of end users in many domains are analogous to that of software architects, and that rather than forcing end users to become programmers, we could instead provide architecture-based tools and techniques to support their tasks. Across various domains, end users need to compose computational entities either via writing scripts or some visual tools. End-user architecting can help such end users by providing frameworks and tools that can help with building powerful analytical tools, at a lower cost than developing them from scratch. There are three core techniques that can drive this: (1) the use of domain-specific architecture styles that define the properties and constraints for the compositions, (2) supporting the visual construction using end-user architectures that allow component assembly, reuse and analysis, and finally (3) providing a framework that provides the building blocks so that construction of such composition environments can be lowered as opposed to building everything from scratch. We illustrated how this approach can be applied to a Neuroscience domain where neuroscientists need to use various tools for imaging analysis. Finally, we described what building a composition environment for such a domain entails and how an end-user architecting framework like Halo can provide support for such development. Next, we describe the Halo framework and its building blocks.

A framework that implements the end-user architecting approach

—Halo

CHAPTER 4

The Halo Framework

In the previous Chapter, we argued for a generic approach that we termed “end-user architecting", which provides ways to incorporate software architecture analytic techniques to design better quality end-user environments. To fulfil the end-user architecting requirements outlined in the previous Chapter, we built a framework, called Halo, that provides generic and reusable infrastructure that can be tailored to various composition styles and domains.

Halo framework design is primarily based on the fact that, for most composition environments used today, one of the common activities [HHN85b] is to combine computational elements and data sources by drag-and-drop, analyze the composition for errors, and execute the composition. These composition steps usually entail some common direct manipulation activities and a number of user commands, such as adding composition elements, specifying their properties, and relationships between composition elements. While the composition styles, UI technologies and domains for such a composition may vary, most of the time the architectural style governing this composition is quite similar; usually a data-flow or publish-subscribe.

In the previous section we identified the key features that end-user architecting must support: visual composition, reuse and import of existing components, analysis of the composition for errors, and execution to produce an output. The Halo framework adopts a layered architecture, where modules or components with similar functionalities are organized in horizontal layers, where each layer can be individually customized by adapters that provide customization and integration points. For example, while the user-interface may allow drag and drop composition support, a UI adapter layer would define general-purpose libraries for common UX operations, which can be extended by an integrator. Similarly, an analysis adapter would define a generic set of functions to invoke an analysis, and a runtime adapter would define the library of execution commands that would allow integration with various execution environments. These adapters provide a customizable mechanism to build composition environments based on the requirements of the individual domains.

This customizable end-user architecting framework has many advantages. Providing a substantial base of reusable infrastructure and plugins greatly reduces the cost of development. Providing a separate customization mechanism allows developers to tailor the framework to different composition environments with relatively small increments in effort.

In this chapter, we describe the design and engineering of the Halo framework, focusing particularly on the customization points and how the individual components and layers can be combined together to facilitate the construction of composition environments. By providing a layered architecture that allows various customization points via the adapters, we enable a general-purpose framework that can be instantiated for multiple domains. Instead of building everything for scratch, which is very expensive, composition environment can be developed at a much cheaper cost.

Next, we present the architecture of the Halo framework and provide more details on how it can be customized through the various adapters.

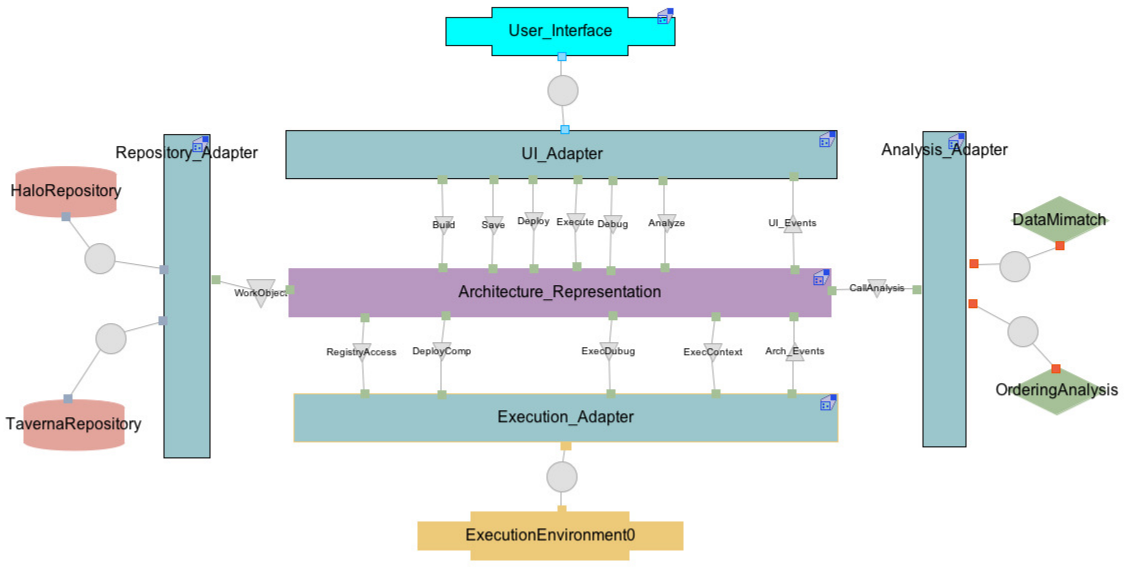


Figure 4.1: Halo architecture diagram

Halo provides *UI adapters* that allow integration of user interface and the business logic layer and interpretation of user commands involved with construction of compositions. A similar scenario holds true for execution, where a number of execution platforms are used by composition environments. Halo provides *Execution adapters* to integrate these execution environments.

4.1 Architecture and Design of Halo

As we discussed earlier, Halo provides a general-purpose, reusable framework that can be tailored to different composition styles. The architecture of the Halo framework, described in Acme and modeled in a graphical architecture design environment called AcmeStudio [SG04], is diagrammed in Figure 4.1. Table 4.1 lists the key architectural types with a brief description of the Halo architecture family. The complete architectural description is available in Appendix A.

This architecture provides a natural decomposition that (a) logically separates the customization points on individual components, (b) allows runtime separation of concerns across different components, and (c) localizes the integration and vocabulary mappings in the adapters. The ports on the Adapters provide points of customization for the framework, the framework defines the high-level interface, and the individual customization points are implemented by an instantiator.

While Table 4.1 describes the key Halo components and connectors, Figure 4.2 describes the overall component interactions and the various customization points.

Table 4.1: Halo Architectural style description — main family

Components	**Functional Description**
UserInterfaceT	Build compositions using an interface
UIAdapterT	Mediates between UI and architecture. Provides general-purpose libraries to integrate UI environments.
ArchitectureRepT	Provides architectural representation of compositions.
ExecutionAdapterT	Mediates between execution platform and architecture. Provides general-purpose libraries to integrate and invoke commands on execution environments.
RepositoryAdapterT	Provides general-purpose libraries for import and export of compositions.
AnalysisAdapterT	Provides general-purpose libraries to invoke external analyses.
RepositoryComponentT	A store for packaged compositions
AnalysisComponentT	A self contained program that runs on a composition and performs various analyses
ExecutionEnvironmentT	A platform that can execute a composition
Connectors	**Functional Description**
RepositoryAccessConT	An HTTP call to read external repository
AnalysisInvocationConT	A call to invoke an analysis plugin
ExecutionCallConnT	A call to the runtime
EventsConnT	An event on a message bus

Table 4.2: Halo Artifacts: Who does what?

Artifact	**Component/Layer**	**Who**	**How?**
Customized Style	Architecture	Domain Expert	Define Type, rules, properties (in Acme Studio)
Services/Executable components	Execution Layer	Component Integrator	Implementation (Wrapper API)
UI-Command modules	UI Adapter	System Instantiator	Implementation
Execution modules	Execution Adapter	System Instantiator	Implementation
Analyses	Analysis adapter and the Analysis layer	System Instantiator	Implementation
Imported Composition	Repository adapter	System Instantiator	Implementation

Furthermore, a summary of who does what is shown in Table 4.2. The composition environment development is performed by designer and developers who take on various roles. Starting with a domain expert, who defines the architecture vocabulary consisting of composition rules, constraints and analysis; to component integrator who uses framework provided wrappers to adapt existing components and make them executable in Halo; to a system integrator who extends the various adapters to build the composition environment.

Next, we describe how these individual components are integrated together to a build a feature rich composition environment and the building blocks Halo assists in this development. We first define a runtime view showing the various component interactions in Section 4.1.1. Next, in Section 4.1.2 we walk through a module view focusing on extension mechanisms and code that developers have to write to customize the Halo building blocks without writing everything from scratch.

4.1.1 Halo Runtime Architecture

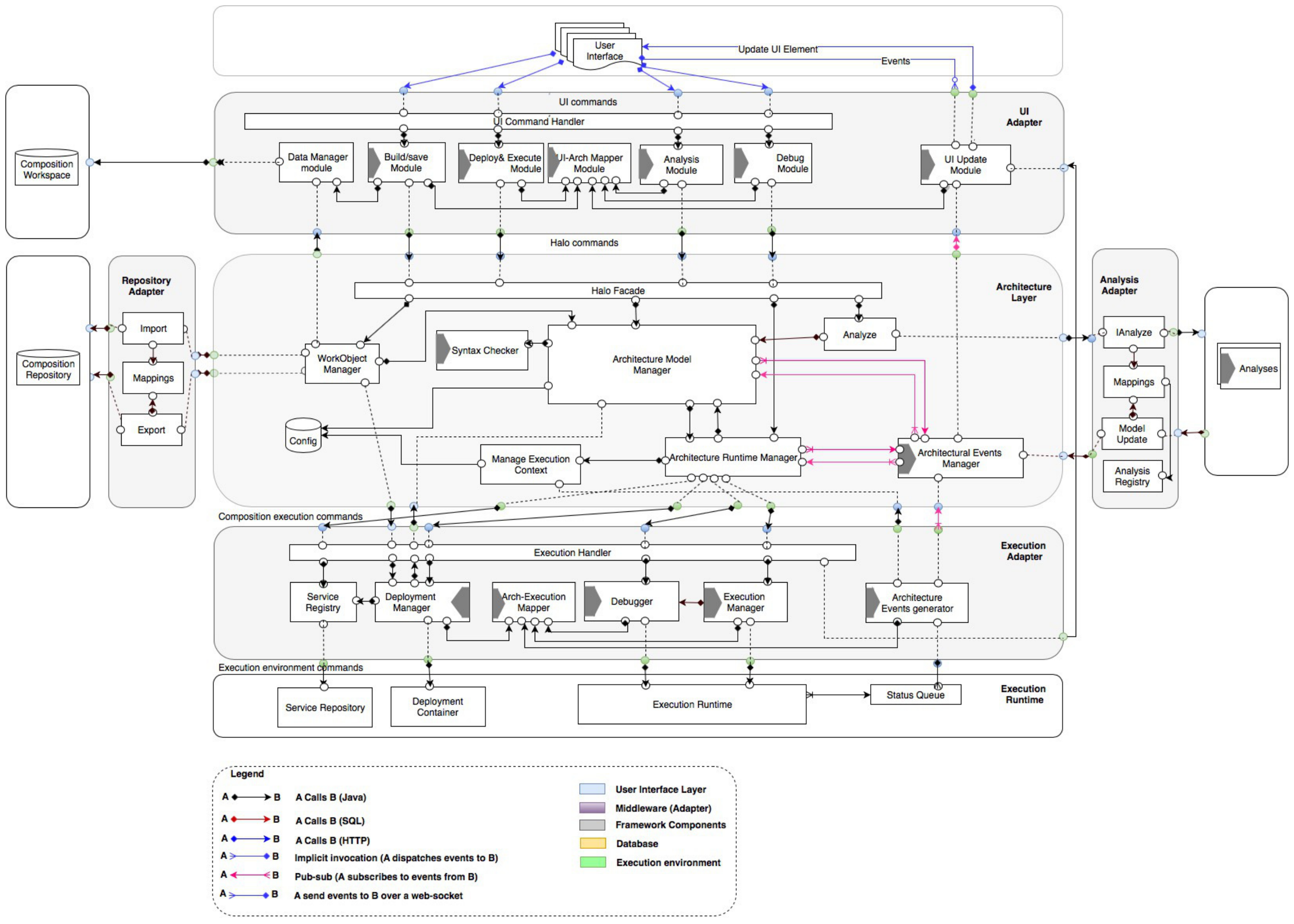


Figure 4.2: Detailed Halo runtime architecture.

As shown in Figure 4.2, an instantiation of Halo consists of an integration of multiple layers. The layers in grey compose the core building blocks of Halo that provide components for end-user architecting. The layers in white consist of components that are integrated to create a feature rich composition environment. As we have mentioned before, while the framework gives example instances of UI, execution runtime, repositories and analyses (marked as components in the white layer), it doesn't prescribe any fixed set of technologies. Instead, it allows integration with a number of such UI and execution environments through generic and customizable adapters that allow cost-effective integration of end-user architecting features. The key functionality of the framework is localized to individual layers so as to maintain separation of concerns. Next, we describe some of the key components in this layout for an illustrative composition environment.

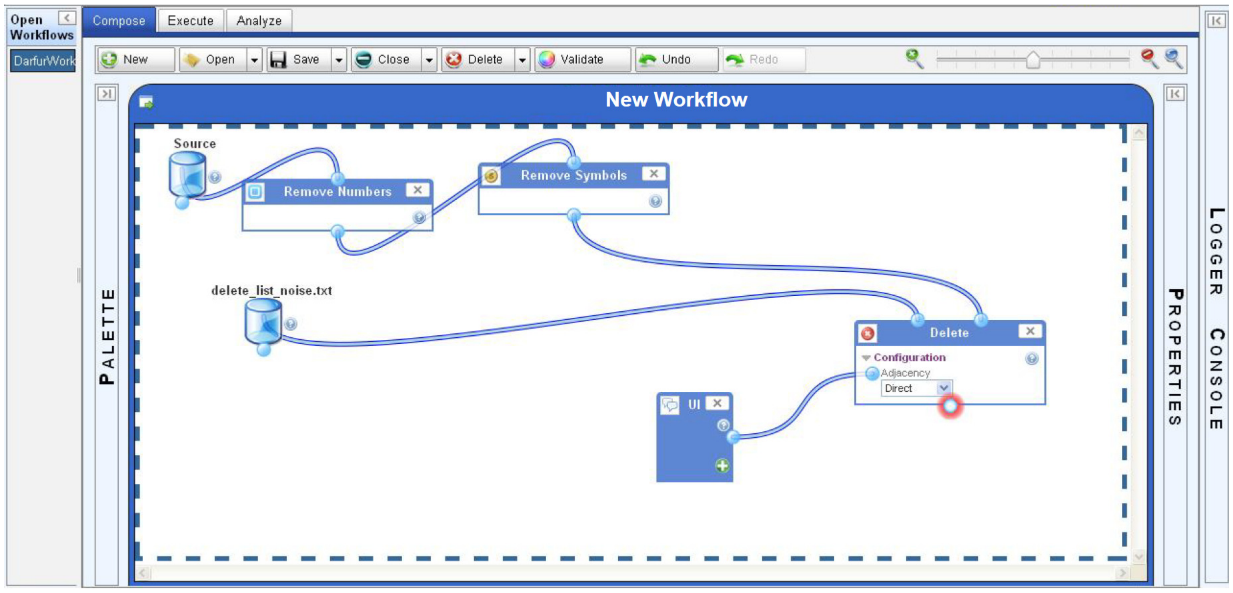


Figure 4.3: An illustrative composition environment.

Figure 4.3 shows an instance of a composition interface that supports end-user composition tasks that are supported through various API interactions. While the nature of compositions and the UI libraries used varies (and implementing these libraries is beyond the scope of the framework), Halo allows integration of such interfaces through generic adapters that can be customized for different user interfaces. An example of the key runtime components are listed below:

- **User Interface:** The front-end of most composition environments consists of a drag and drop (or other kinds of drawing) interface to end-users to drag and drop components and connect them in meaningful combinations. The style and vocabulary of such end-user compositions may vary from dataflows, to publish-subscribe widgets, to mix of composition styles. Halo provides customizable APIs to translate these end-user compositions to architecture representation and execute them. An example of such an end-user composition interface is shown in Figure 4.3 where three services, two pieces of data and a UI visualization service are combined in the SWiFT composition environment [GSD$^+$11]. End-Users can use the instantiated composition environment to compose new workflows, edit existing workflows, and save their workflows. They have access to not only their own work, but also any shared workflows. Besides basic construction, most composition environments also facilitate reuse and packaging and search for compositions in the repositories.

- **UI Adapter:** UI Adapter is the layer that provides translation from user vocabulary to architecture vocabulary. While the specific implementations of UI adapters vary across composition environments, Halo provides a generic set of customizable APIs to build these UI adapters. An example of such an adapter is shown in Figure 4.7, showing the high-level components that are responsible for receiving requests from the User Interface and mediating between the UI and the architecture layer.

  In this example, the adapter provides a generic Command handler with JavaScript functions that translate the user-interface commands to architecture commands. Furthermore, each command is processed by a task module that provides functionality to handle composition, build and save, deployment, analysis and UI-Update functionality. Halo provides the high-level interfaces for each of these modules that can be customized by a framework instantiator for a specific User interface.

- **Architecture Representation:** Architecture Layer provides the key functionality of architectural representation, analysis and execution management. This layer provides the APIs to define compositions using domain-specific architecture styles, which specify the vocabulary of element types and constraints on compositions [12]. Some example of constraints include prohibition of cycles, dangling connectors, unattached interfaces, and mismatched communication channels (where the data produced by one component is incompatible with the data consumed by a successor component). Besides such end-user feedback, this layer also provides execution capabilities such as compilation into executable code.

- **Execution Adapter** Execution Adapter is the layer that provides translation from architecture vocabulary to execution semantics. Like the UI adapter, the implementations of execution adapter vary across different runtime environments. Halo provides a generic set of runtime APIs that can be further customized. An example of such execution adapter is discussed in Section 4.2.3, where we describe an integration of SWiFT workflow environment with a BPEL runtime engine.

- **Execution Layer:** This Layer provides execution support for end-user compositions and consists of basic execution infrastructure to execute a composition. Besides the execution infrastructure, this layer provides the APIs for common execution and monitoring commands. Most common capabilities of such runtime environments include execution, debugging, repositories, status queues, and other auxiliary runtime capabilities.

- **Analysis Adapter:** Analysis Adapter is the layer that allows integration of external analysis into Halo. Individual analysis can be based on different models or written with different assumptions, but the analysis adapter provides a generic mechanism not only to invoke the analyses but it also provides the necessary protocols to interface with the architecture layer allowing easier integration with different environments.

- **Repository Adapter:** Repository Adapter is the layer that allows integration of external repositories into Halo. As Halo could be instantiated for multiple domains, the vocabulary of compositions would vary not only in terms of language constructs but also computation style. The repository adapter provides mechanisms to import a composition written into an external vocabulary to by transforming it into an architecture based vocabulary used by Halo. This import and export mechanism allows easier integration of a number of composition vocabularies through Halo.

We will revisit the runtime architecture and the key decisions in Chapter **??**. However, having given an example of the runtime architecture we next describe the module view to illustrate how developers build these layers in a modular fashion and the support the framework provides in creating a cost effective and customizable manner instead of writing everything from scratch.

4.1.2 Halo Module View

The modular design and construction of software, as designed for Halo, is not new. In 1979, David Parnas wrote about the advantages of extensible software and constraints around building extensible software [Par79] such as:

- Adding simple features without significant code changes
- Building product variants by adding or removing functionality
- Delivering an early release with a subset of features

David Parnas further gave some examples of how reusable software artifacts could be developed by engineers in a fast and reliable way. Since then, a significant number of extension and reuse mechanisms have been proposed and are used in practice today. The Halo framework enables such modular decomposition and reuse by breaking down the general architecture of composition environments as multiple layers, and providing modules and packages to build these individual layers.

Figure 4.4 shows a simplified module view of the Halo framework. While the UI and the runtime are key to any composition environment, the focus of the Halo framework is not to provide specific UI or execution libraries, but ways to integrate such UI and execution runtimes through adapters. The Adapters handle this integration by providing generic Command Handler packages that can be extended to integrate different types of commands. The adapters also provide generic modules that implement the abstract classes for individual composition method.

4.2 Customizing the Halo Framework

To fulfill the requirements for a general-purpose architecture as outlined in Chapter 3, and to support a general-purpose architecture, Halo supports various customization points for the framework. The key design approach is to divide the framework functionality in layers that enables separation of concerns. We list some such customizations in Table 4.3.

This level of customization through different layers has a number of advantages. First, the adapters and the other reusable infrastructure greatly reduce the cost of development. The adapters and the other customization mechanisms allows engineers to tailor the framework to different systems with relatively smaller development effort. This allows to satisfy the generality, cost effectiveness and quality requirements for the thesis.

Next, we describe some of the customization techniques as they are implemented by different layers of the framework.

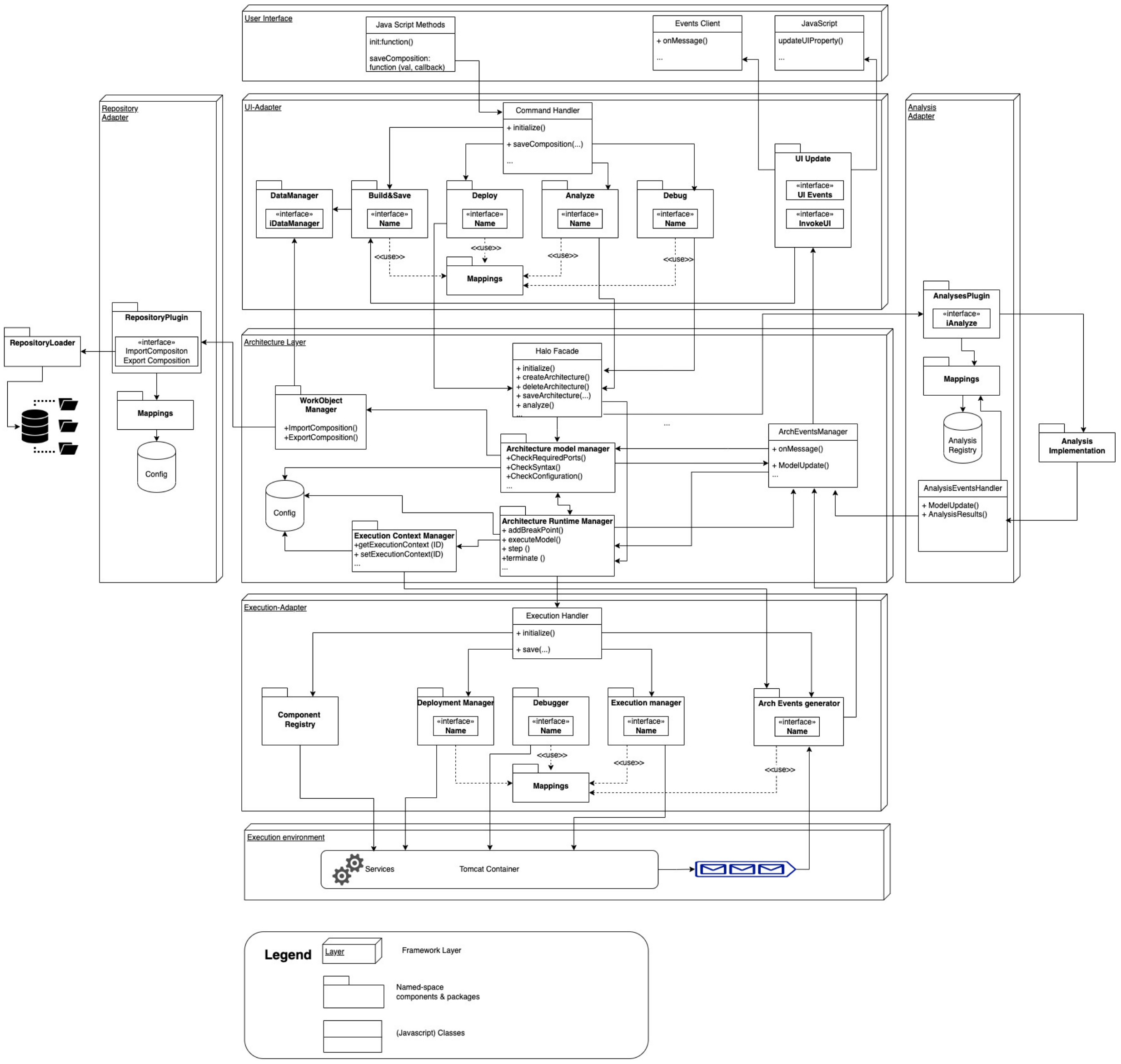


Figure 4.4: A cross-section of Halo modules in an example instantiation.

4.2.1 User interface customization

Figure 4.3 shows an example end-user composition environment that is instantiated using the Halo framework. A high-level control flow for such an environment instantiated through Halo is shown in Figure 4.5. Common composition commands in such an environment consist of initializing a new composition, creating compositions, deleting, analyzing, executing, and saving a composition. Halo provides extensible command modules that allows creation and customization of these commands so that they could be provided into the UI layer through the adapter. In this model, the user interface layer is responsible for representation of end-user interactions, while the UI commands are interpreted into an architecture vocabulary by the UI Adapter. While the UI layer and its interaction mechanisms with the architecture

Table 4.3: Halo framework extensibility and reuse

Component/Layer	Techniques	Description
User Interface	Modular/Extensible command library	Command UI composition commands wrapped as Javascript calls
	Loosely coupled updates	UI updates pushed in as messages
UI Adapter	Loose-coupling through Abstract classes and APIs	Customizable commands by extending the common interface
	Command Factories	Extensible UI Commands
Architecture Layer	Vocabulary Refinements and Extension	Architecture vocabulary represented in Acme can be refined by styles
Execution Adapter	Loose-coupling through Abstract classes and APIs	Customizable commands by extending the common interface
	Command Factories	Extensible Runtime Commands
Execution Runtime	Loosely coupled Message-notification	Runtime messages pushed to a queue
Analysis Adapter	Plugin extension	Individual analysis plugins implement a common interface
Repository Adapter	Plugin extension	Extensible common interface for composition import and export
System Wide	Separation of concerns	UI, Architecture, Adapters have constrained communication
	Configuration	Individual layer allows configuration for independent customization

layer could vary significantly based on technologies used, the Halo framework supports general-purpose adapters to plug in arbitrary user interfaces, provided that they implement the end-user commands defined by the Halo framework.

An example instantiation of the UI adapter for this environment involves supporting various commands that can be invoked from the user interface. The Command-Handler in the UI Adapter implements a Java servlet that intercepts the UI commands and provides extensible interfaces for interpreting the individual composition commands and compiling them into architecture vocabulary. The individual modules provide interfaces that are extended to implement architecture commands.

The UI and Execution adapters implement a command factory pattern to encapsulate information needed to trigger common actions. They provide a Handler that encapsulates the various commands. The user interface adapter for example, provides a command handler that is commonly implemented as a servlet than receives commands and makes callbacks to the user interface. The Analysis adapter provides an events handler that works as a publisher for analysis events that are pushed to an events queue.

The UML model for this layer is shown in Figure 4.7. The UI Adapter provides the command handler that intercepts the commands from the user interface and delegates it to the architecture layer, which provides a generic facade for executing architecture commands. The Halo framework provides various

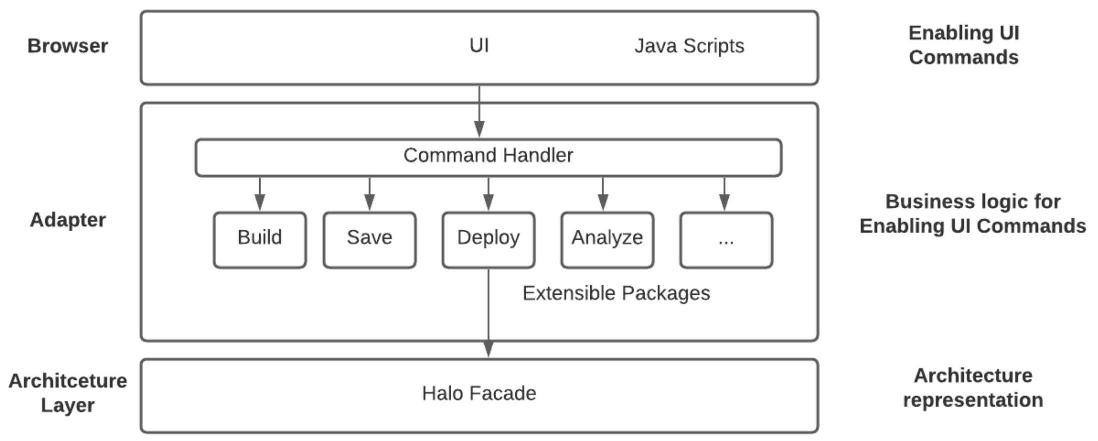
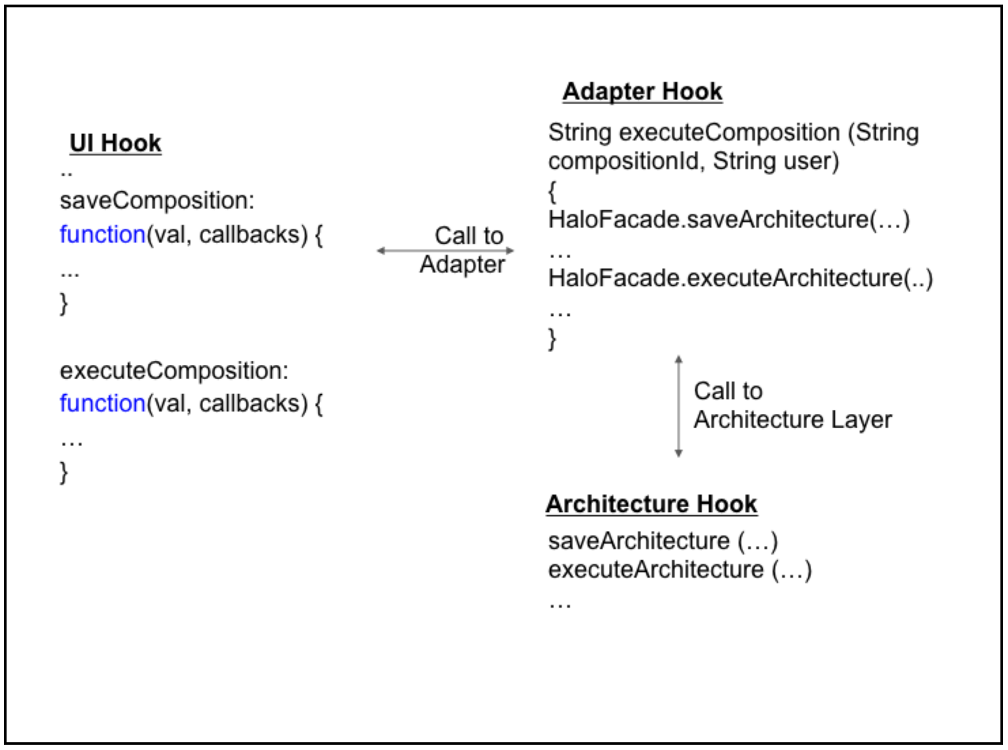


Figure 4.5: High-level UI Adapter control flow



```
UI Hook
..
saveComposition:
function(val, callbacks) {
...
}

executeComposition:
function(val, callbacks) {
…
}
```

```
Adapter Hook
String executeComposition (String
compositionId, String user)
{
HaloFacade.saveArchitecture(…)
…
HaloFacade.executeArchitecture(..)
…
}
```

```
Architecture Hook
saveArchitecture (…)
executeArchitecture (…)
…
```

Figure 4.6: UI command delegation through an adapter.

hooks to intercept these commands. In the UI layer, an example of such a hook is various java-script calls that invoke the commands. These javascript calls are intercepted by the adapter that implements a servlet that receives the command and delegates it to the architecture layer. The communication between the user interface and the adapter servlet is implemented through a Reverse Ajax design pattern that allows sending data from client to server and pushing server data back to the browser.

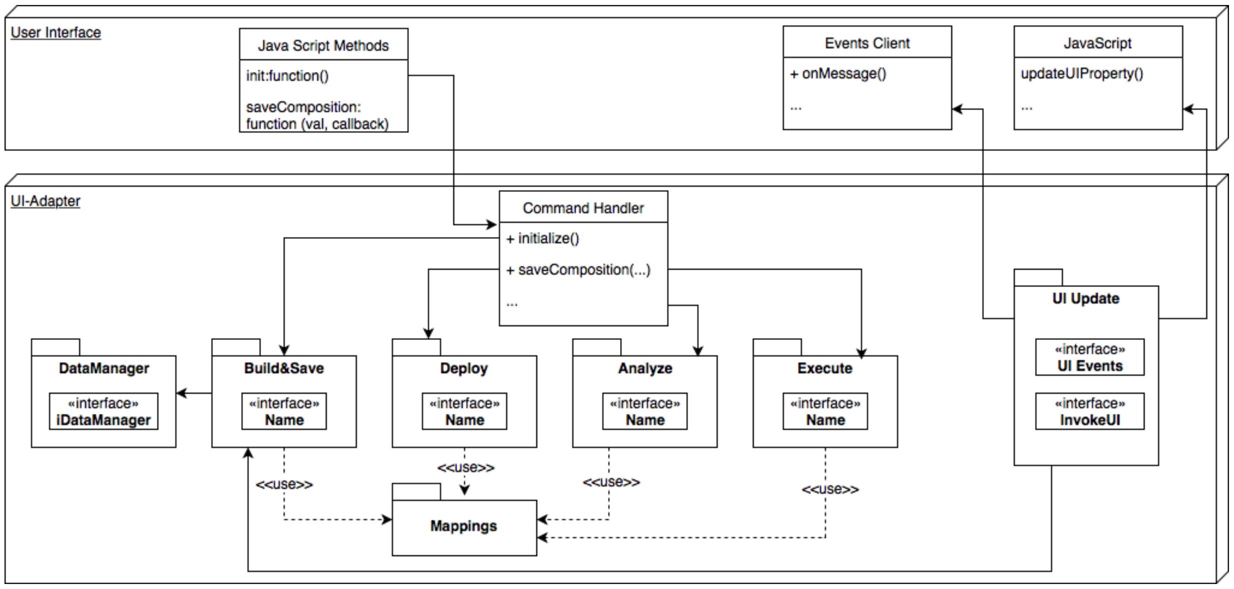


Figure 4.7: UI Adapter customization.

In this model, the individual layers promote separation of concerns, while allowing significant customization and reuse of modules and interfaces across use cases. While the user interfaces and technologies may vary significantly, this technique allows some level of decoupling and reuse of the design patterns, if not the code fragments directly where individual layer could be independently customized for new technologies.

4.2.2 Architecture Layer Customization

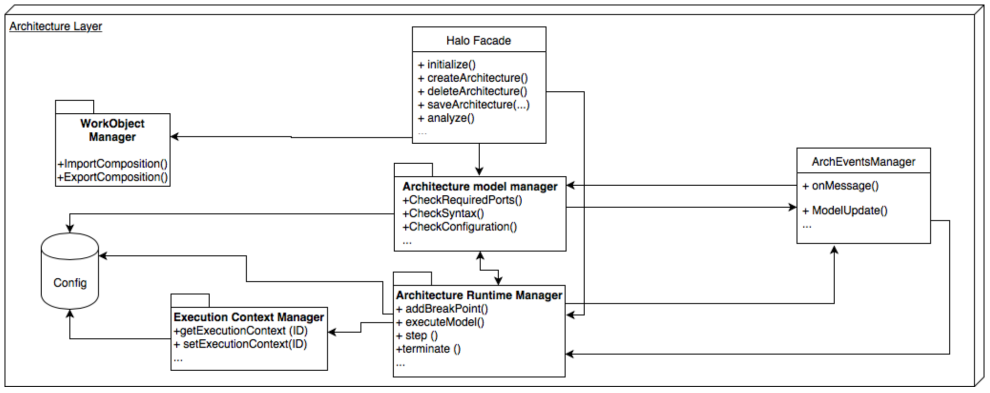


Figure 4.8: Architecture layer - key components

While user interfaces support the specification of the composition, the architecture layer implements the End-user architecting approach. As we have emphasized before, the ability to associate representations for end-user compositions and using the architectural representations to enforce analysis, reuse and execution, while ensuring this could be implemented in a generic, cost effective way is core to the framework.

These are some of the key components for the Halo Architecture layer:

- **Halo Facade**: is an entry point to the Halo architecture layer and implements a handler that is responsible for delegation of architecture commands to the individual components. While the framework implements basic initialization, creation, update, delete and other basic functions on the architecture layer, it allows extension of these commands. However, while architecture commands can be independently customized, they do have some dependency on the user interface commands. To reduce this dependency, the UI Adapter implements the mapping logic that bridges the UI and architecture vocabulary and must be extended or modified to extend/modify the facade.
- **Architecture Model Manager**: manages access to the architecture model and evaluates the conformance of the architecture model to a predefined set of design rules, expressed in an ADL called Acme [GMW97b]. This component leverages various tools and techniques built around Acme for type checking and architectural analysis. Architecture styles are key to customization of the models used for Halo compositions as they capture the dimensions of variability and representation in architectural design. Style has been formalized and applied in many system designs [AAG93, MKMG97]. This style-based refinement makes it feasible to build generic infrastructure that can be tailored to various domains.

  The model manager registers compositions in Acme ADL and uses that for architecture analysis. Acme, a generic ADL, supports the explicit notion of styles and provides a constraint language similar to UML's OCL to capture system design constraints [DEF+11] and domain-specific analysis such as data-mismatches [EDG+13].

  Table 4.4 shows a high-level example of architectural types, functions and constraints that are used to specify compositions for Halo. The style represents the key components and some constraints that are based on Acme's first order predicate logic, where they are expressed as predicates over properties of the workflow elements. The basic elements of the constraint language include constructs such as conjunction, disjunction, implication and quantification [GMW97b].

  Not only do architecture styles allow representation of domain specific compositions, but they can be specialized through refinement and inheritance as shown in Figure 4.9. This requires construction of sub-styles that extend the base styles by adding additional properties, domain-specific constraints, and rules that allow the correct construction of workflows within that domain.

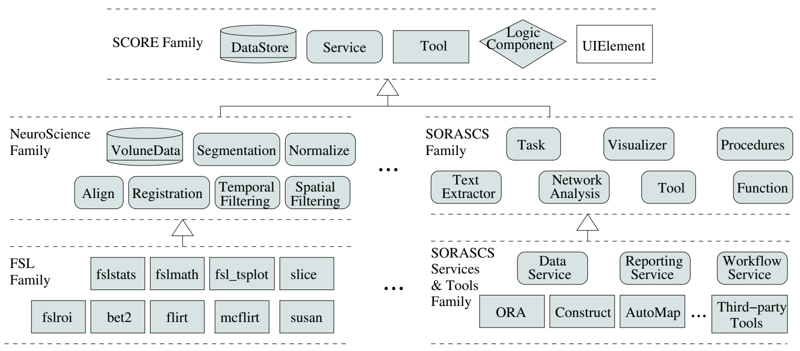


Figure 4.9: Style derivation by inheritance.

Components	Description
`DataStore`	Components for data-access (such as file/SQL data-access)
`LogicComponent`	Components for conditional logic (such as join/split etc)
`Service`	Components that are executed as a service call
`Tool`	Components who's functionality is implemented by tools
`UIElement`	Special-purpose UI activity for human interaction
Connectors	Description
`DataFlowConnector`	Supports dataflow communication between the components.
`DataReadConnector`	Read data from a DataStore Component
`DataWriteConnector`	Write data to a DataStore Component
`UIDataFlowConnector`	Provides capabilities to interact with UIElements
Ports	Description
`configPort`	Provides an interface to add configuration details to components
`consumePort`	Represents data-input interface for a component.
`providePort`	Represents data-output interface for a component.
`readPort`	Provides data-read interface for DataStore component
`writePort`	Provides data-write interface for DataStore component
Roles	Description
`consumerRole`	Defines input interface to DataFlow/UIDataflow connectors
`providerRole`	Defines output interface to DataFlow/UIDataflow connectors
`dataReaderRole`	Defines input interfaces for the DataRead/DataWrite connectors
`dataWriterRole`	Defines output interfaces for the DataRead/DataWrite connectors
Acme Functions	Description
`Workflow.Connectors`	The set of connectors in a workflow
`ConnectorName.Roles`	The set of the roles in a connector
`self.PROPERTIES`	All the properties of a particular element
`size( )`	Size of a set of workflow elements
`Invariant`	A constraint that can never be violated
`Heuristic`	A constraint that should be observed but can be selectively violated
Constraint types	Example
`Structural`	Checking that connectors have only two roles attached `rule onlyTwoRoles = heuristic size(self.ROLES) = 2;`
`Structural`	Checking if a specific method of the service called exists `rule MatchingCalls = invariant forall request: !ServiceCallT in self.PORTS |exists response: !ServiceResponseTin self.PORTS| request.methodName == response.methodName;`
`Property`	Checking if all property values are filled in `rule allValues = invariant forall p in self.PROPERTIES | hasValue(p);`
`Membership`	Ensuring that a workflow contains only 2 types of components `rule membership-rule = invariant forall e: Component in self.MEMBERS |declaresType(e,ComponentTypeA) OR declaresType(e,ComponentTypeB);`

Table 4.4: An example of composition representation in Acme

The Model Manager not only supports common architectural checks, such as Check for Syntax, rules and configuration, but it also allows checking for domain-specific vocabulary that is represented in Acme. Properties and constraints on architectural elements as defined in Acme, allow the Model Manager to enforce the architectural checks.

- **Architecture Runtime Manager**: manages the mappings between architecture and execution, compilation into runtime architecture and invocation of various execution commands. Having an architectural representation of the composition, the runtime manager enforces the execution of these models when requested by the users. However, in order to do so, it needs to compile the architecture representation into a runtime model, and also maintain the runtime context to allow capabilities such as debugging, status of currently executing components and mappings between architecture components and executable instances.

  The methods implemented by the Halo runtime manager are primitive execution, setting breakpoints, stepping and termination. While these are methods over the architecture model, their runtime implementation is left to the Execution adapter that implements each method based on the runtime selected. Halo supports extension to the Model manager, which could be implemented by changing the model manipulation logic or adding new methods.

  The Architecture Context Manager manages the various mappings related to runtime context, debugging and execution state. Currently, this is implemented by a generic set of APIs and the information is stored in a SQL database. While the schema and the type of context could change, at the very least the framework supports storing mappings with architectural components and its runtime equivalent, and the execution ids for each composition to allow repeat executions and display of intermediate execution status whenever possible through the user interface. The methods available through the Context Manager are extendable to add more detailed information depending upon the instantiation use-case.

- **Architecture Events Manager**: implements a light-weight publish-subscribe mechanism for architecture and runtime events. It uses multiple channels for UI events, analysis events, runtime events and architecture events that are consumed by different components. It provides a wrapper for publish and subscribe of the events queue.

  The Halo framework uses Active MQ as a messaging bus for the events manager. The framework implements a generic client over the pub-sub message queue that provides a wrapper over ActiveMQ messaging APIs. As an extension mechanism, the client provides interfaces for the client library that can be extended depending on instantiation use case. A framework instantiator, for instance, may chose to write its own custom handlers for onMessage() method that can support different integration use cases.

- **WorkObject Manager**: manages import and export and transformation of compositions into a format that can be read by Halo. While the nature of the compositions and their domain specific representation may change across different environments, the work object manager enables import from a custom DSL to Acme and vice verse. This import and export is enabled by a plugin framework that allows framework instantiators to add new types of imports and exports. While Halo supports only basic imports and exports, the design of the Work Object Manager can also support more complex use cases, such as packaging of architecture specifications, user interface specs to allow packaging of display elements, etc should the composition environment designers need to implement this.

4.2.3 Execution Layer Customization

The execution layer follows a similar design pattern as the user interface and its adapter. Since a key goal for Halo is to provide a general-purpose framework that could support a variety of composition styles and technologies, Halo supports this via a general-purpose adapter design. Similar to the UI Adapter design, the Execution Adapter too provides a generic Handler to intercept execution commands, which it delegates

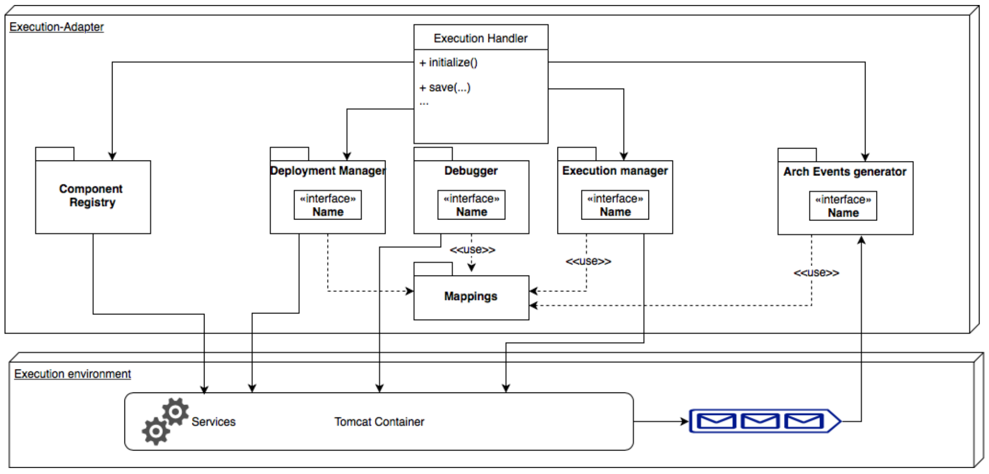


Figure 4.10: Execution layer - key components

to the runtime environment.

Halo provides extensible command modules that allows creation and customization of these commands so that they could be executed. The commands that are delegated from the UI to the architecture components, are interpreted into runtime commands by the Execution Adapter and implemented as callbacks. As shown in Figure 4.10, Halo supports this through various modules. While the execution runtimes can change from web-servers, to SCA to language CLIs, the Adapter allows mapping execution calls into calls to the runtime APIs. Halo provides the interfaces that can be implemented by the Execution Adapter to implement those calls.

4.2.4 Analysis and Repository Customization

Similar to the user interface and execution runtime, the types of composition languages and reusable analyses can vary significantly. However, such third-party analyses and compositions can be quite useful to the composition environment and Halo supports their integration through a generic adapter. As shown in Figure 4.4, both these types of adapters provide interfaces that can be extended by framework instantiators to integrate the third-party plugins and compositions. Halo provides a generic Facade for integration of analysis plugins and compositions, providing methods for their import and export.

Besides implementing handlers and command factories, the framework allows a plugin mechanism for adding external repositories and analyses. Figure 4.11, for example, shows an interface for the Analysis plugins that each analysis must implement. This allows integration of third party libraries as they can be invoked through a common invocation protocol. At this point the user interface and execution adapters are not available as plugins because there is more effort required to map UI/execution commands into architecture and therefore this needs a developer to write those mappings. But this is something that could be extended further as pluggable modules through further automation.

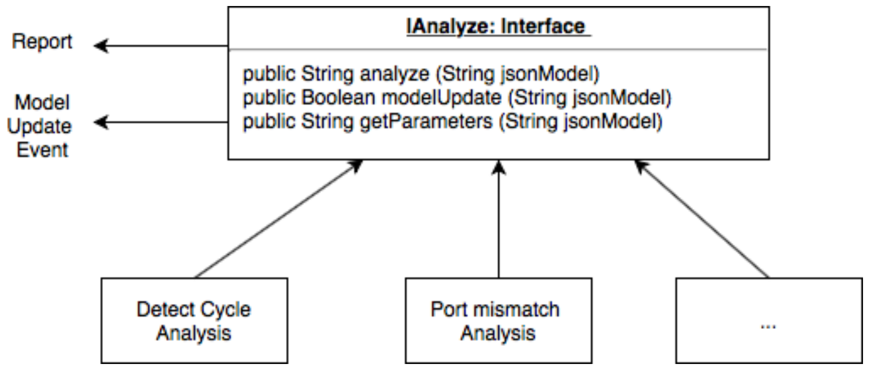

IAnalyze: Interface
public String analyze (String jsonModel)
public Boolean modelUpdate (String jsonModel)
public String getParameters (String jsonModel)
Report
Model Update Event
Detect Cycle Analysis
Port mismatch Analysis
...

Figure 4.11: Analysis Handler

4.3 Summary

In this chapter, we described the key Halo components. We reiterated how the modular and layered breakdown of Halo supports the requirements of generality and cost-effectiveness to engineer end-user architecting environments. We also described how individual Halo layers can be customized through techniques like modularization, vocabulary refinement, plugin extensions and command factories and handlers. While the architecture representation is key to the end-user architecting approach, we demonstrated how the use of adapters enables integration of the architecture layer and can be enforced through customization and extensions rather than a complete rewrite.

CHAPTER 5

Examples and Supporting Evidence

In Section 1.2, we described the evaluation plan for three thesis claims: **generality** to a broad spectrum of styles and composition environments that can be built using the framework; **cost-effectiveness** to engineer and develop the composition environments using the Halo Framework; and **improved quality** of the end-user composition in supporting major composition and analytic tasks. In this chapter we demonstrate how the Halo framework supports those claims through various validation steps. Specifically, we present the following evidence:

1. Demonstrate End-User Architecting in 2 Computation Styles (DF + Pub Sub)
2. Demonstrate End-User Architecting in 4 Domains (Arithmetic expressions, Dynamic Network Analysis, Neurosciences, and Widget compositions)
3. Demonstrate End-User Architecting with 2 UI Adapters
4. Demonstrate End-User Architecting with 2 Execution Adapters
5. Demonstrate End-User Architecting with 2 Analysis Adapters
6. Demonstrate End-User Architecting with 2 Repositories

In this chapter, we describe four instantiations of Halo, within the two computation styles of dataflow and publish-subscribe along with adapters to integrate them. We also give examples of two analyses based on end-user architectures. Together, these examples provide validation evidence for Halo. In Chapter 6, we provide a recap of how the end-user architecting approach supports the thesis claims.

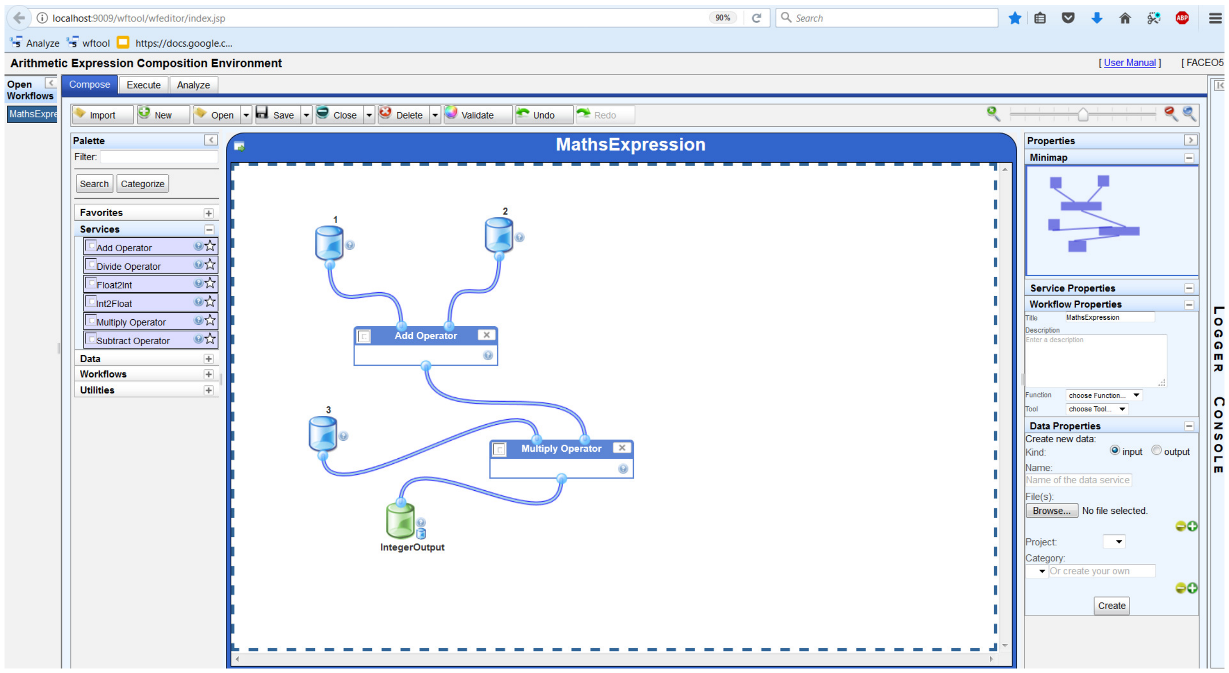
5.1 Arithmetic expression composition

As a simple hello-world example, we instantiated Halo to create an Arithmetic expression evaluator. The key components for this application are operators and operands, which can be combined together to build arithmetic expressions.

The key end-user architecture building blocks for this domain are as follows:

1. **Architecture representation:** The end-user architecture for this is a dataflow where operators and operands are combined together to produce compositions, that produce an output. The arithmetic operators include basic maths functions like add, subtract, multiply and divide that are wrapped together and exposed as Halo services. The operands are float and integer inputs, which can be combined with the operators.

2. **Architecture style:** Compositions are defined using an Arithmetic Expression style, which is based on the SCORE dataflow style (as defined in Chapter 3.3.2) and defines the rules for composing operators and operands.
3. **Adapters:** The four adapters provide support to wrap the operators and operands, aid their composition, and execution on the Apache Tuscany runtime environment.
4. **Analysis:** Examples of analyses include checking for Divide By Zero, and Checking for Cycles.
5. **Execution support:** The compositions are compiled and executed on the Apache Tuscany runtime environment.
6. **Reuse:** Arithmetic expressions can be saved as workflows in a repository and reused.



5.2 Dynamic Network Analysis

Dynamic Network Analysis (DNA) is a domain of computation that focuses on the analysis of network models, which represent entities, relations, and their properties. DNA is increasingly being used in a variety of fields, including anthropology, sociology, business planning, law enforcement, and national security, where networks capture the relationships between people, knowledge, tasks, locations, etc. [Car06].

End users in these fields are typically analysts who extract entities and relations from unstructured text (such as web sites, blogs, twitter feeds, email, etc.) to create network models, and who then use those models to gain insight into social, organizational, and cultural phenomena through analysis and simulation.

For example, an analyst interested in understanding disaster relief after the Haiti earthquake in 2010 [ZGM11] might build a network from open source news data provided through a source such as LexisNexis [Lex].

This unstructured textual data needs to be processed into a usable form, or "cleaned," to filter out headers, remove noise, and normalize concepts. From this processed data a dynamic network can be generated representing associations between people, places, resources, knowledge, tasks, and events. Using network analysis algorithms, insights can then be gained. For example, analysis can determine things like the primary organizations and people involved in the relief effort, how information about food and medical supplies propagated through the network, and how these evolved over time.

Similar kinds of analyses are routinely carried out in law enforcement (where analysts use crime reports and statistics to determine drug-related gang activities), healthcare and disease control (where analysts use medical reports from hospitals and pharmacies to understand disease vectors), and anthropology (where social scientists can understand belief systems and how they relate to demographics).

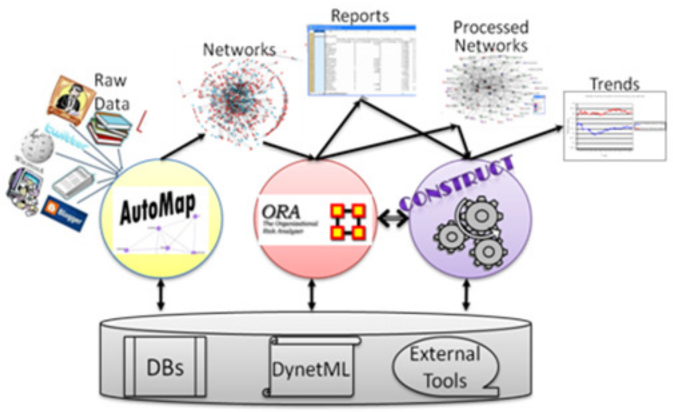


Figure 5.1: Typical tools for socio-cultural analysis.

Within this broad domain of dynamic network analysis, analysts typically engage in a process of composing a variety of existing tools to extract networks, analyze them, and display results. Figure 5.1 illustrates a typical toolset used for such analyses consisting of the following: AutoMap for extracting networks from natural language texts, ORA for analyzing and visualizing networks, and Construct for "what-if" reasoning about the networks using simulation [SGD+11].

Conceptually the computations that analysts create can be viewed as workflows, where each step in the workflow requires the invocation of some data transformation step that consumes the data from previous steps and produces results for the next step. However, traditionally, to achieve this kind of composition analysts would need to understand the idiosyncracies of each of tool, manually invoke them on data stored in various file locations using a variety of file naming schemes and data formats, and preserve the results of the analysis in some location that they would have to keep track of, before invoking another tool to carry out the next step.

To apply the end-user architecting approach to this domain, we adapted the end-user architecting framework of Figure 3.1 by creating an environment, called SORASCS (Service ORiented Architecture for Socio-Cultural Systems), for dynamic network analysis [GCS+09, SGD+11], and illustrated in Figure 5.2. Key features of this environment are as follows:

1. **Architecture representation:** Architectures are explicitly represented in an architecture layer, called the socio-cultural analysis layer. This layer stores compositions as workflows. It also pro-

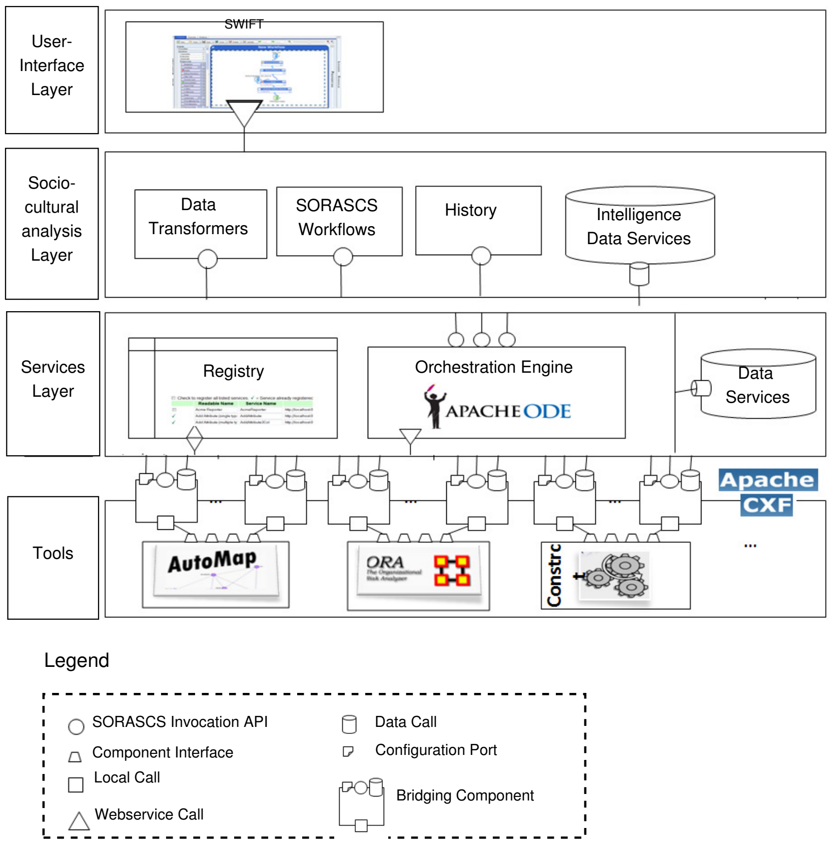


Figure 5.2: SORASCS Organization.

vides a repository of data transformers, which act as component building blocks for creation of new workflows.

2. **Architecture style:** Compositions are defined using a formal workflow architectural style based on SCORE (see Section 3.3.2), which specifies the vocabulary of element types and constraints on compositions [DEF+11]. Element types include data transformers, data sources, and data sinks. Constraints of the workflow style prohibit the introduction of cycles, dangling connectors, unattached interfaces, and mismatched communication channels (where the data produced by one component is incompatible with the data consumed by a successor component).

3. **Adapters:** The four Adapters allow integration of the user interface with a BPEL-based runtime and integration of analysis plugins and a storage repository.

4. **Analysis:** The SORASCS workflow style supports a number of analyses including (a) data privacy analysis, which identifies potential privacy issues in the information flows, (b) a security analysis, which identifies potential security issues based on workflow properties, (c) an ordering analysis, which uses machine-learning to evaluate whether the ordering of transformation steps is consistent with previously constructed workflows, and (d) performance analysis, which estimates the amount of time that will be taken to complete an analysis of a specified data set.

5. **Execution support:** Workflows are compiled into BPEL scripts, which are run within the Services Layer using standard SOA infrastructure. The compilation process attempts to optimize performance by parallelizing workflow execution. Additionally, there is execution support for long-

duration transformations and graceful error handling — typically not provided by baseline SOA infrastructure. Further, it is possible for a user to set breakpoints, execute the workflow one transformation at a time, and preserve intermediate data for later inspection.

6. **Reuse:** Workflows can be encapsulated as parameterized components for later reuse and adaptation. These are stored in a repository of available data transformers, which may be used as primitives, or "opened" to reveal their substructure and possibly edited for new usage contexts.

To illustrate how SORASCS works, Figure 5.5 shows a workflow that analyzes a user's emails to generate a social network of his/her contacts. Table 5.1 lists the computational elements that are used for this workflow. The `Mail Extractor` workflow step acquires security credentials to connect to a remote mail server in order to gain access to the user's emails. The composition then transmits the user's email data to `Filter Text`, followed by `Delete`, which in combination remove irrelevant words and symbols. This data is then passed to `Generate Meta-Network`, which generates a social-network of the people and concepts referred to in the email text. `HotTopics` then creates a report listing important keywords in this social network. The workflow also uses two data sources that provide the inputs to the text processing steps.

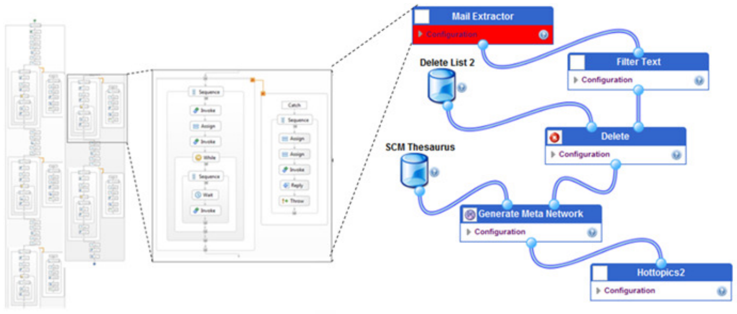


Figure 5.3: A dynamic network analysis workflow with a security flaw.

When a security analysis is run on this workflow, SORASCS detects a security problem. In this case, data security requirements mandate the use of 'token-based authentication' by all services. However the above workflow includes the `Mail Extractor` service, which uses 'password-based authentication' — indicating a security violation. The analysis flags this as a problematic workflow by highlighting the inappropriate service in red.

Once analysis is complete and the errors have been corrected, the user can compile the workflow into the BPEL script illustrated in Figure 5.5, which can then be executed. Although not illustrated here, as execution proceeds, the user is given feedback through the SORASCS user interface to show which workflow step is currently being executed.

5.3 Widget composition environment

The Ozone Widget Framework (OWF) [Pot12] – or just Ozone – is a web platform for integrating web-based widgets, which run on a distributed set of processors hosted by multiple organizations. Such web applications widgets are lightweight visual applications, and OWF allows end users to open and com-

Operation	**Description**
`Mail Extractor`	Extracts email from a server to a text file
`Filter Text`	Removes undesirable information from text files
`Delete`	Removes a set of common keywords using a standard dictionary (such as: a, an, the, etc) from a text file
`Generate Meta-Network`	Creates a dynamic network based on the information in the text file
`Hot Topics`	Creates a report about important keywords in a social network

Table 5.1: DNA operations used in the workflow of Figure 5.5.

pose a set of widgets through a web "dashboard" in their browser. Users interact with widgets, which communicate among each other using the OWF framework.

An example of an Ozone dashboard is shown in Figure 5.4. The right-most window is the launch menu from which end users can add widgets to their dashboard. There are four widgets displayed on the dashboard, displaying information of different types, some in chart form, others (in the background) on maps. These widgets may pass information between each other to ensure that they are focused on the same map region, for example, or to display updated information as it becomes available from a database or data stream. This dashboard and the arrangement of widgets can be shared between developers by exchanging textual configuration files.

Ozone widgets interact in a publish-subscribe style [CBB+10]: widgets can publish events to channels and subscribe to channels to receive events.[*] All widgets that have subscribed to a channel receive data published to that channel by any other widget. Widget developers who wish to integrate with other developers must agree on the names of channels to publish to, and the format of the data that is published. To offer additional control over communication, Ozone also allows end users to restrict potential communication between widgets by indicating pairs that are allowed to communicate, thereby implicitly restricting other widgets from participating in those communications.

While end users are free to choose which widgets appear in their dashboard, considerable care must be taken to ensure sensible configurations. In particular, it is important to make sure that widgets both publish and subscribe to the appropriate channels, and that the type of data published is consistent with that expected by subscribers.

The existence of complex interconnection rules and behavior lead naturally to the use of architectural modeling of widget compositions, which could support the end-user architecting process through automated constraint checking. For example, a widget topology can be checked to conform to a privacy constraint that widgets containing private data do not communicate it to third-party untrusted widgets. Another application is widget topology generation: a user would specify what pairs of widgets should and should not interact, and a set of topologies would be generated. While developing a full-fledged widget

[*]Events in Ozone are plain-text strings or JSON objects.

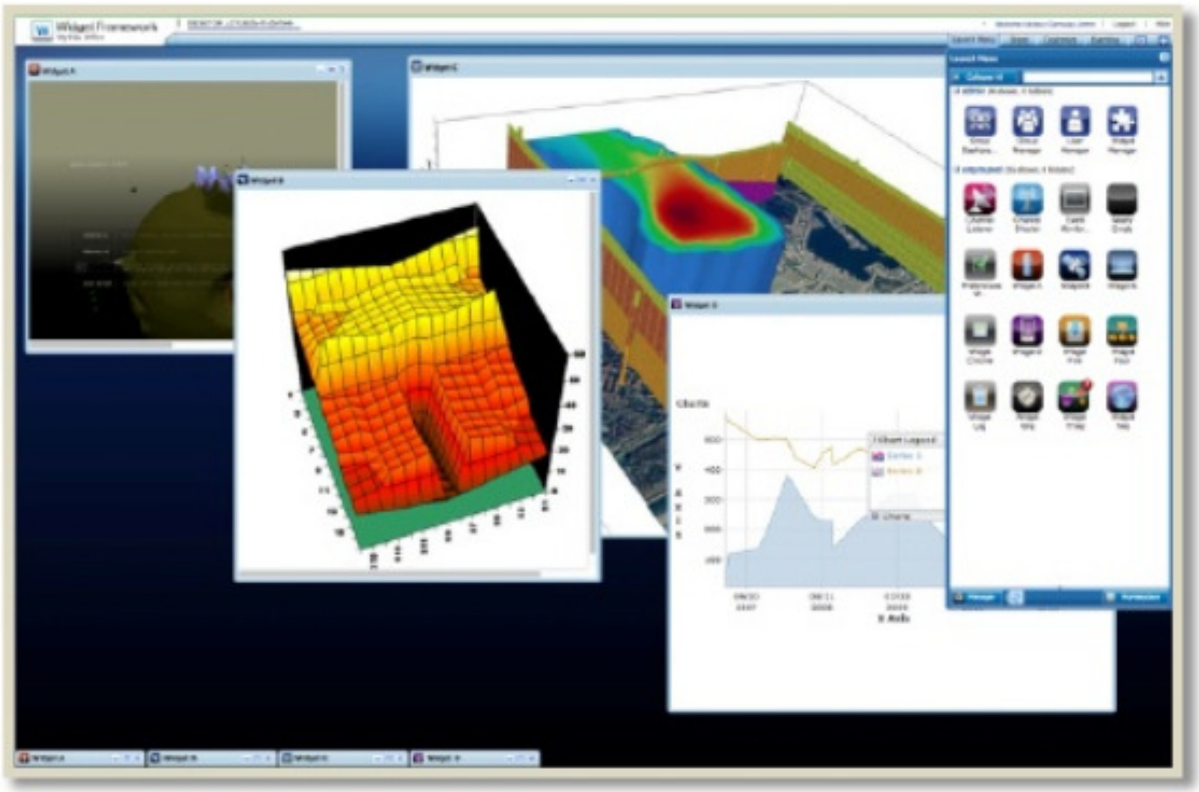


Figure 5.4: An Ozone dashboard example from [HV12].

environment was out of scope for this thesis, in our work we modeled Ozone widget compositions in Acme [GMW97a] and used that for analysis.

Key features of our end-user architecting approach to this domain are:

1. **Architecture Representation:** Ozone widget configurations are represented as explicit architectural models, that indicate which widgets are involved in a composition and the communication topology.
2. **Architectural Style:** Compositions are defined using a variant of a publish-subscribe style that takes into account restrictions on widget communications. Element types include Widgets, which have publish and subscribe interfaces, and two types of connectors representing public channels and private (restricted) channels.
3. **Adapters:** We built a simple UI-adapter that can accept a widget assembly, an execution adapter than can evaluate the execution and analysis adapter than can execute an analysis displaying the results in another widget
4. **Analysis:** The framework allows integration of various analysis plugins that could take a composition and provide analytic results that are displayed in another widget. Examples of analytic results include displaying whether there are data mismatches over publish-subscribe channels, whether information is lost (e.g., because there is no widget subscribed to information on a particular channel), etc.

5. **Reuse:** Dashboard setups (i.e., configurations) that are shared between analysts as textual configuration files. Embellishing this with architectural representations allows end users to check whether adaptations to existing compositions retain prior communication channels, and whether it is feasible to substitute one widget for another.

5.4 Neuroscience workflow composition

We introduced the neuroscience workflow composition in Section 3. As described before, the end-user compositions in this domain comprise of compositing various tools and libraries and construct workflows consisting of a series of operations to process brain imaging data.

The Halo framework was applied to this domain to provide the following components:

1. **Architecture representation:** Similar to arithemtic expression environment, architectures are explicitly represented in a system layer that stores compositions as workflows and provides a repository of processing steps and transducers. The main components made available in this prototype were derived from the FSL tool suite (e.g., *bet2, fslmath, flirt*) [FMR].

2. **Architecture style:** Compositions are defined using a formal workflow architectural style, which is similar to the one used for arithmetic expression environment.[†] The neuroscience style differs in two respects: (a) it defines computational elements specific to the neuroscience domain, and (b) it provides additional properties and domain-specific constraints (such as checking ports for different data encodings and other content of brain-image data) that allow the correct construction of workflows within the neuroscience domain.

3. **Adapters:** The four adapters allow the integration of the user interface, execution runtime, analysis plugins and the neuroscience component repository

4. **Analyses:** Similar to arithemtic expression environment, the properties of the style elements are used for designing various domain-specific analyses for the brain imaging domain. An example is data mismatch analysis (described in detail in Section 5.5.2) to support the detection of data mismatches in the neuroscience compositions and to suggest repairs that can resolve these mismatches based on an end user's quality of service requirements [VEDG$^+$12].

5. **Execution support:** Workflows are compiled into BPEL scripts, which are executed on a service-oriented platform, identical to SORASCS, providing similar feedback and debugging facilities.

6. **Reuse:** The workflows in this domain can be encapsulated as parameterized components for later reuse and adaptation.

7. **User Interface:** A web-based graphical interface is provided for workflow construction, analysis, and execution.

Not only did Halo provide the building blocks, which lowered the cost for constructing this environment (as opposed to building everything from scratch), it also allowed capabilities to plug various analyses. In the next section, we describe one such analysis that allows automated mismatch repair based on utility theory.

[†]In fact, using the formal architectural description language of Acme[MKMG97], we have defined a common root style for both the dynamic network analysis domain and the neuroscience domain [DEF$^+$11].

5.5 Examples of analyses based on end-user architectures

In the previous Section we described how compositions can be represented as architectures. Next, we describe some example scenarios where such an end-user architecture can assist end users in their composition tasks using various architecture-based analyses.

5.5.1 Constraint analysis

An architecture model for compositions allows us expose system integrity constraints explicitly, that supports checking their validity during compositions. Consider, for example, a dataflow composition that is implemented via refinement of the SCORE style. The style defines specific constraints on composition of the components, which constrain how these can be combined together. Such constraint specification is based on Acme, which facilitates the definition of logical expressions that capture such relationships as connectedness, type conformance, and hierarchy. Halo supports building analyses that provide guidance to the end users when such constraints are violated.

As an example of analyses based on constraints, see the figure 5.1, where the architecture vocabulary provides constraints that define the permitted composition of components. For example, the vocabulary may define constraints on component interfaces (or ports) based on some expected property value.

Listing 5.1 Illustration of a constraint on a property

```
Port Type In = {
    ...
    Property structure: legalInternalStructure;
}
Port Type Out = {
    ...
    Property structure: legalInternalStructure;
}
Component Type flirt extends Registration = {
    Port In: in;
    Port Out: out;
 }
...
for all c1, c2 : Service | connected (c1, c2) ->
     (c1.out.structure == c2.in.structure)
```

A large number of analyses can be expressed via such architectural constraints. Not only can such constraints define legal ways of combining various components, more complex rules can be defined on top of these constraints — for example, checking for cycles, checking for various structural properties, etc. When such constraints are violated, these can be easily exposed via an interface that allows feedback and guidance to an end user so that the user may correct the issues with the composition. Such nudges improve the quality of end-user composition and make end users more efficient where they can focus on the task at hand, instead of delving into code and scripts to figure out what was the problem.

Besides analysis based on architectural constraints (violations), Halo supports more advanced forms of analyses where analyses are treated as plugins that act on the architectural vocabulary and can per-

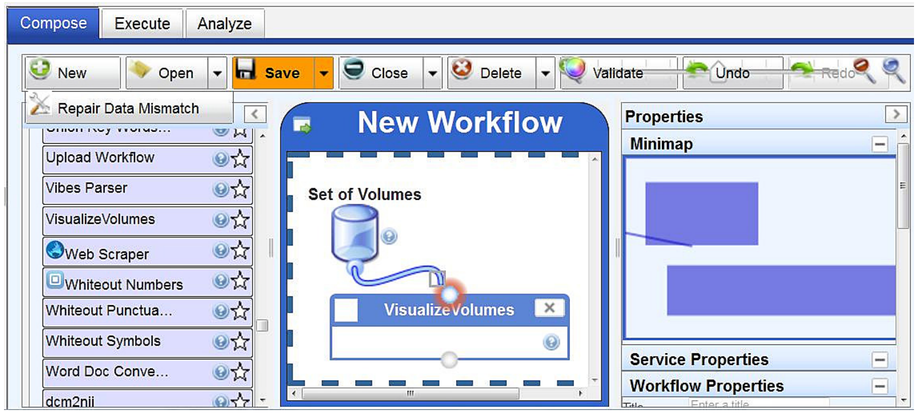


Figure 5.5: An example analysis based on constraint violation.

form more complex tasks – for example, checking for performance (based on related properties and their analysis), automated mismatch repair, and so on.

In the next section, we describe one such analysis in the brain imaging domain that allows automated repair of a compositions.

5.5.2 Data mismatch analysis

Many domains such as scientific computing and neuroscience require end users to compose heterogeneous computational entities to automate their professional tasks. However, an issue that frequently hampers such composition is data-mismatches between computational entities. Although, many composition frameworks today provide support for data mismatch resolution through special-purpose data converters, end users still have to put significant effort to deal with data mismatches, e.g., identifying the available converters and determining them, or combination of them, meet their QoS expectations. Often end users have to compose computational entities that have conflicting assumptions about the data interchanged among them (as shown in Table 5.2).

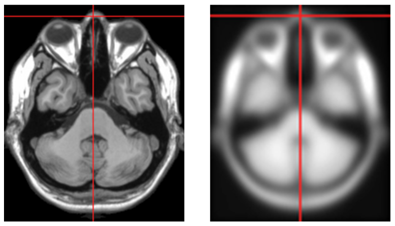
As introduced earlier, consider the typical scenario in the neuroscience domain where scientists study samples of human brain images and neural activity to diagnose disease patterns. This often entails analyzing large brain-imaging datasets by processing and visualizing them. Such datasets typically contain 3D volumes of binary data divided to voxels.[‡] Across many such datasets, besides the geometrical representation, brain volumes also differ in their orientation. Therefore, when visualizing different brain volumes a scientist must "align" them by performing registration. When two brain volumes A and B are *registered*, the same anatomical features have the same position in a common brain reference system, i.e., the nose position in A is in the same position in B (as shown in Figure 5.6 (a)). Thus, registration of brain volumes allows integrated brain-imaging views.

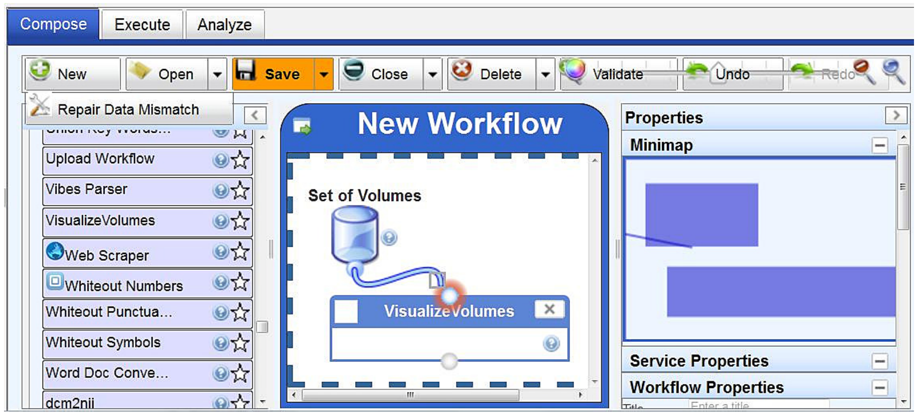
Processing and visualizing data sets require scientists in this domain composing a number brain-imaging tools and services provided by different vendors. The selection of tools and services is carried out manually and often driven by analysis-dependent values of domain-specific QoS constraints, e.g., accuracy, data loss, distortion. In this context, the heterogeneous nature of services and tools often leads

[‡]A voxel is a unit volume with specific coordinates and dimensions, e.g. width, length and height.

Type	Description
DataType	Results from conflicting assumptions on the signature of the data and the components that consume it, e.g., a computation requires different data type.
Format	Results from conflicting assumptions on the format of the data being interchanged among the composed parts, e.g., xml *vs.* csv (comma separated values).
Content	Results from conflicting assumptions on the data scope of the data being interchanged among components, e.g., the format of the output carries less data content than is required by the format of the subsequent input.
Structural	Results from conflicting assumptions on the internal organization of the data being interchanged among the composed parts, e.g., different coordinates system such as Polar *vs*. Cartesian data or different dimensions such as 3D *vs*. 4D.
Conceptual	Results from conflicting assumptions on the semantics of the data being interchanged among the composed parts, e.g., brain structure *vs.* brain activity or distance *vs.* temperature.

Table 5.2: Common types of data mismatches.


(a)



(b)

Figure 5.6: (a) Registered volumes with same brain reference system and (b) Data mismatch detection during composition

to have data mismatches; thus, scientists also need to select conversion tools and services to resolve them.

Consider that during workflow composition a scientist needs to visualize a set of brain-image volumes. These volumes store brain images of the same person as 3D DICOM volumes. The volumes are not registered, i.e., they are not aligned to the same brain reference system. To visualize this data, the scientist tries to compose the Set of Volumes data service –which can read the actual store where the volumes are, and the Visualize Volumes service –which enables their visualization. Table 5.3 shows an excerpt of the specifications of the operations' parameters of these two services. As can be seen, the Visualize Volumes service requires data that is already registered and in 'NIfTI' format (see its registered='Yes' and format='NIfTI' input parameters). Thus, these two services cannot be composed as they have both a *format* and a *structural mismatch*, i.e. the interchanged data has both a different format and internal organization.

Most impoverished environments would find it hard to even detect such mismatches, not to mention

Table 5.3: An excerpt of the parameter specifications of the services in the example.

Service	Operation	Input parameters	Output parameters
Set of Volumes	read Volumes		name='out' type='files' format='DICOM' registered='No' sameSubject='Yes'
Visualize Volumes	view	name='in' type='files' format='NIfTI' registered='Yes' sameSubject='Yes|No'	
dinifti	DICOM toNIfTI	name='in' type='files' format='DICOM' registered='No|Yes'	name='out' type='files' format='NIfTI' registered='Yes|No'
dcm2nii	dc2nii	name='in' type='files' format='DICOM' registered='No|Yes' sameSubject='Yes|No'	name='out' type='files' format='NIfTI' registered='Yes|No' sameSubject='Yes|No'
flirt	register	name='in' type='files' format='NIfTI' registered='No' sameSubject='Yes|No'	name='out' type='files' format='NIfTI' registered='Yes' sameSubject='Yes|No'
fnirt	register	name='in' type='files' format='NIfTI' registered='No' sameSubject='Yes|No'	name='out' type='files' format='NIfTI' registered='Yes' sameSubject='Yes|No'

fixing them. It is usually left to the scientists to discover why such a mismatch occurred and then manually fix it. The ideal workflow, the scientists would have wanted that would fix this composition scenario is shown in Figure 5.10. It is possible to build analyses that could automatically fix this composition, and even suggest multiple alternatives when more than one option is possible.

Architecture-based data mismatch repair

In one of our prior works [EDG+13], we developed an approach to fix such mismatch analysis using a utility-based analysis of architecture representations. The general idea behind the approach was to build a mismatch repair evaluation engine, which when fed with architectural descriptions describing a composition would generate alternative choices that could be ranked by a utility function and presented to the end user. Figure 5.7 shows the high-level steps to such an analysis.

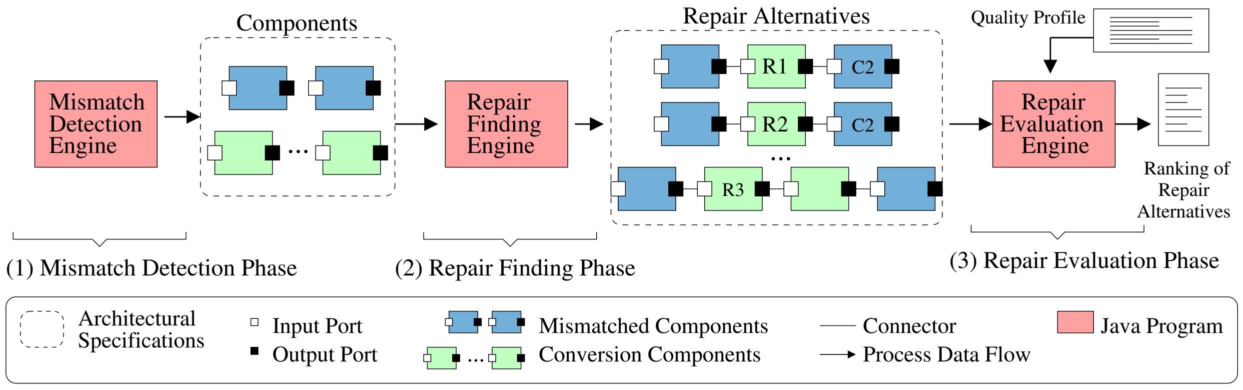


Figure 5.7: The three main phases of the approach to data mismatch detection and resolution.

Program 2 shows a snippet of an Acme specification that illustrates specialization of the neuroscience style where data *format* and data *structure* are represented as properties of the ports of the `flirt` service component. Such an architectural specification can be used to automatically check constraints to detect various types of violations in compositions.

Program 2 Example of data ports with format and structural information.

```
Property Type legalFormats = Enum {NIfTI, DICOM};
Property Type legalInternalStructure = Enum {Aligned, NotAligned};
Port Type In = {
   Property format: set of legalFormats;
   Property structure: legalInternalStructure;
}
Port Type Out = {
   Property format: set of legalFormats;
   Property structure: legalInternalStructure;
}
Component Type flirt extends Registration = {
   Port In: in;
   Port Out: out;
 }
```

For instance, this predicate can be used to detect a data mismatch involving both format and structural aspects:

```
for all c1, c2 : Service | connected (c1, c2) ->
    size(intersection(c1.out.format, c2.in.format)) > 0
    AND (c1.out.structure == c2.in.structure)
```

The predicate states that it is not enough for a pair of connected Services $c1$ and $c2$ to deal with data with the same format (e.g., DICOM or NIfTI[§]), but also the data must have same structural properties (e.g., Aligned or NotAligned). Once a mismatch is detected by the type checker, the Mismatch Detection Engine can retrieve the architectural specification of the pair of mismatched components and outputs this to the repair finding phase, which is shown in Figure 5.8.

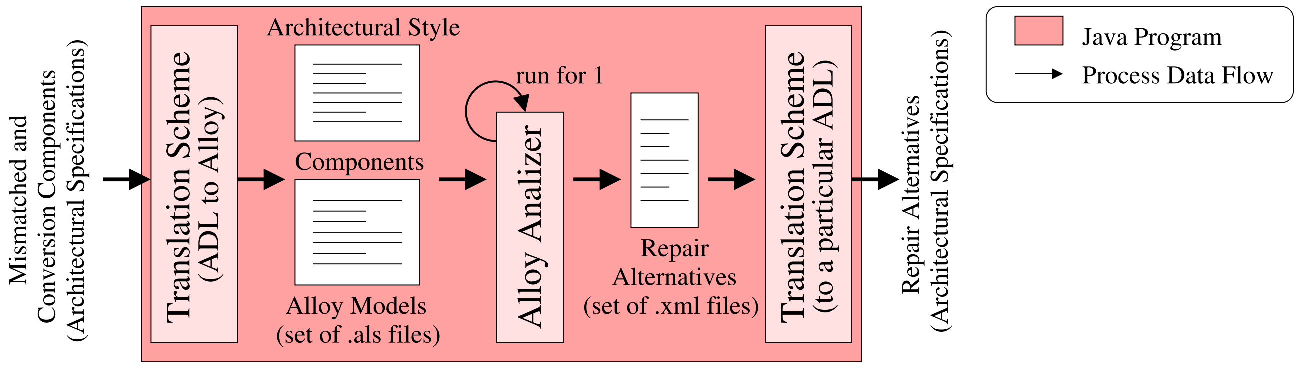

Mismatched and Conversion Components (Architectural Specifications)
Translation Scheme (ADL to Alloy)
Architectural Style
Components
Alloy Models (set of .als files)
run for 1
Alloy Analizer
Repair Alternatives (set of .xml files)
Translation Scheme (to a particular ADL)
Repair Alternatives (Architectural Specifications)
Java Program
Process Data Flow

Figure 5.8: The Repair Finding Engine.

In this phase, the declarative specifications of the pair of mismatched composition elements are converted into Alloy specifications, which generates a set of possible alternatives for the model. The Repair

[§]DICOM and NIfTI are data formats used to store volumetric brain-imaging data.

Finding Engine thus finds all the valid instances of a repair alternative by having multiple runs of this command. The Alloy Analyzer stores these instances as XML files. These files are then transformed to architectural specifications to be processed in the next phase of the approach as shown in Figure 5.8.

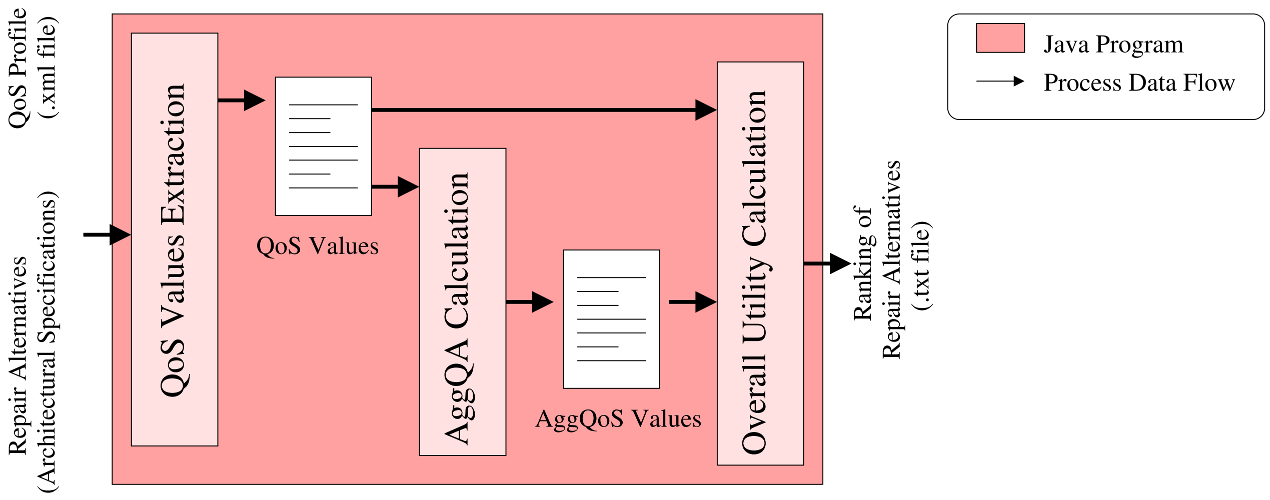

QoS Profile (.xml file)
Repair Alternatives (Architectural Specifications)
QoS Values Extraction
QoS Values
AggQA Calculation
AggQoS Values
Overall Utility Calculation
Ranking of Repair Alternatives (.txt file)
Java Program
Process Data Flow

Figure 5.9: The Repair Evaluation Engine.

While there are number of ways one could prioritize the generated compositions (if there are more than one), one way to handle this is through a utility based approach, where a QoS profile could be associated with each possible component to decide the overall QoS values of repair alternatives.

Let's assume that the scientist has specific QoS requirements for a repair. He would like to have no distortion in the brain-image; he would like to have an optimal speed and accuracy, but would be OK with their average values. However, low value of speed or accuracy, or distortion is not acceptable for this composition. This information, specified in the QoS Profile, can be summarized as follows:

Accuracy: $\langle$(Optimal, 1.0), (Average, 0.5), (Low, 0.0)$\rangle$,
Speed: $\langle$(Optimal, 1.0), (Average, 0.5), (Low, 0.0)$\rangle$ and
Distortion: $\langle$(Y, 0.0), (N, 1.0)$\rangle$, with the 0.5, 0.1 and 0.4 weight values respectively.

Based on the QoS information, and using a set of built-in domain-specific functions, the Repair Evaluation Engine calculates the following aggregated quality attribute values:[¶]

RA_1: $aggQA_{Dist}$ = N, $aggQA_{Sp}$ = Ave, $aggQA_{Acc}$ = Opt.
RA_2: $aggQA_{Dist}$ = Y, $aggQA_{Sp}$ = Ave, $aggQA_{Acc}$ = Opt.
RA_3: $aggQA_{Dist}$ = N, $aggQA_{Sp}$ = Opt, $aggQA_{Acc}$ = Opt.
RA_4: $aggQA_{Dist}$ = Y, $aggQA_{Sp}$ = Ave, $aggQA_{Acc}$ = Opt.

With all this available information, the Repair Evaluation Engine can compute the overall utility of each repair alternative via the utility function. The obtained results are ranked and presented to the scientist.

[¶]Dist = Distortion, Sp = Speed, Acc = Accuracy, Opt=Optimal, Ave=Average.

Program 3 Illustrative QoS specification for FSL services.

```
-- dinifti
<QoSSpecification>
 <att><name>Distortion</name><val>N</val></att>
 <att><name>Speed</name><val>Average</val></att>
 <att><name>Accuracy</name><val>Optimal</val></att>
</QoSSpecification>
-- dcm2nii
<QoSSpecification>
 <att><name>Distortion</name><val>N</val></att>
 <att><name>Speed</name><val>Optimal</val></att>
 <att><name>Accuracy</name><val>Optimal</val></att>
</QoSSpecification>
-- flirt
<QoSSpecification>
 <att><name>Distortion</name><val>N</val></att>
 <att><name>Speed</name><val>Optimal</val></att>
 <att><name>Accuracy</name><val>Optimal</val></att>
</QoSSpecification>
-- fnirt
<QoSSpecification>
 <att><name>Distortion</name><val>Y</val></att>
 <att><name>Speed</name><val>Average</val></att>
 <att><name>Accuracy</name><val>Optimal</val></att>
</QoSSpecification>
```

Figure 5.10 shows part of the workflow after resolving the mismatch. As can be observed, the alternative that has the highest utility is selected.

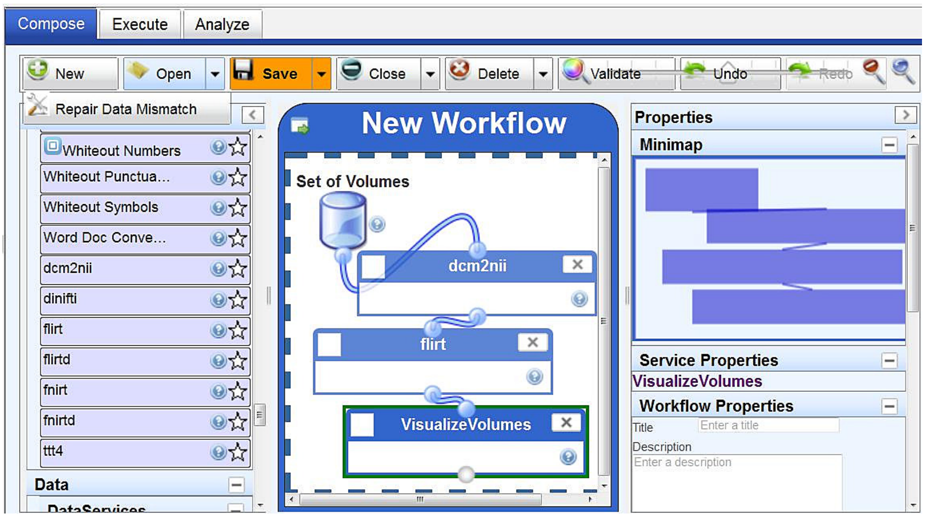


Figure 5.10: The workflow after resolving the mismatch.

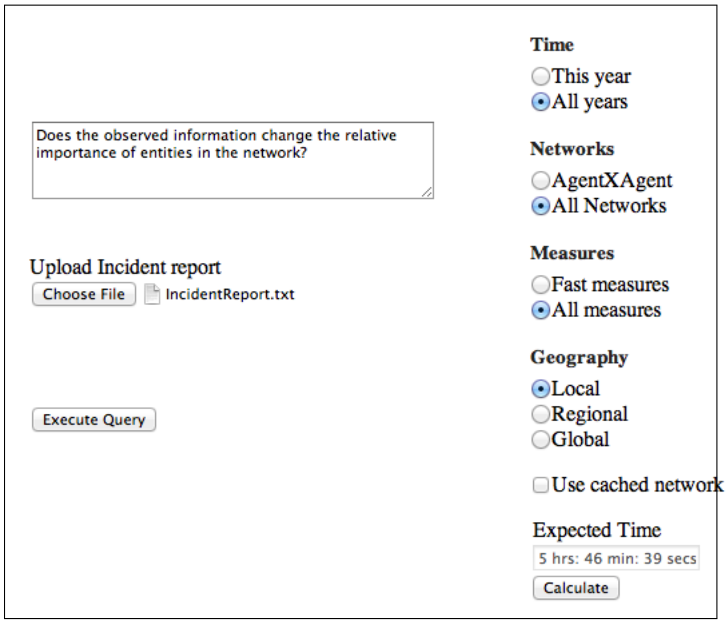
5.5.3 Fidelity vs. timeliness analysis

Another end-user architecting analysis is a common scenario in many scientific fields, where simulations and analyses require computations with varying fidelity expectations. For example, scientists may perform a quick approximation using lesser data, or perform computations with various fidelity trade-offs. In many such domains, composing heterogeneous computational entities, usually in the form of workflows or component assemblies, allows scientists to execute their analyses. In these scenarios, often the fidelity selection of datasets, components, and their configurations determines the timeliness of queries (and vice-versa).

As an illustration, consider an example in the field of military intelligence where soldiers rely on analysis and simulations to guide their operations. Today, such analytic capabilities are provided by tools and mechanisms that can transform information sources captured as unstructured input (e.g., incident reports, news sources, miscellaneous geo-spatial data) into complex network models that aid sophisticated analysis such as situational awareness, key entities, fact identification, and what-if exploration [CP12].

A common querying scenario is when a soldier observes suspicious activity and sends an incident report (see example below) to an operating base, where analysts and other experts can analyze the incident and respond back with a report.

```
Incident Report:  Lt.  Col.  Liz Abreams (Date:  2/16/2011) Set up sensor
alert at checkpoint zulu-1, border crossing between Talodi and Malakal.
Position sensor picked up 15 vehicles.  Darfur escapees.  Overcast.  Positive
ID on LP 6VES512.  Orange.  Passengers were Hasim Makul, Hassan Sayid
Deng, Jon Deng, and Mary Okulo.  Visible knifes.  Possible narcotics."
Query:  Should we detain?  Will maintain position till 1800.
```


Does the observed information change the relative importance of entities in the network?

Upload Incident report

Choose File IncidentReport.txt

Execute Query

Time
This year [✓ All years]

Networks
AgentXAgent [✓ All Networks]

Measures
Fast measures [✓ All measures]

Geography
[✓ Local] Regional Global

Use cached network

Expected Time
5 hrs: 46 min: 39 secs

Calculate

Figure 5.11: A simple UI to perform fidelity time trade-offs.

In order to assess the situation and to answer the soldier's question, an analyst in a forward operating base must decide whether the new information leads to any significant changes in the existing network structures in that geographical area. Such an analyst may have user interfaces that may allow him/her to perform various tradeoffs on the various input parameters to the query (as shown in Figure 5.11). However, for each of these selected variations in input parameters the underlying computation may vary significantly. An illustrative computational workflow (shown in Figure 5.12) for this query involves: (i) processing this incident report along with the network data, (ii) converting it into a graph-based model and (iii) using network algorithms to create and visualize the impact of the new event.

In this domain, a typical network contains millions of nodes with information about people, events, and locations. Therefore, it may save significant time to pre-process and cache this data at the expense of using potentially stale information. Other fidelity variations include: (a) reducing the quantity of data based on dimensions such as time (e.g., only consider this year, vs. consider all years), (b) space (e.g., only consider sources associated with Darfur and Sudan), (c) source (e.g., only consider sources from local reports), and (d) using faster approximations vs. slower but accurate algorithms. These, and other fidelity choices, may lead to different component assemblies with different execution times.

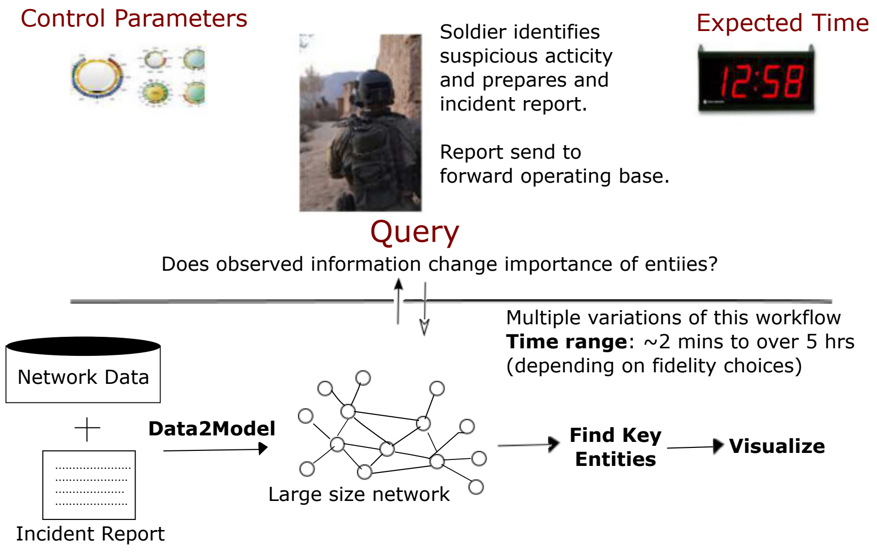

Control Parameters
Soldier identifies suspicious acticity and prepares and incident report.
Report send to forward operating base.
Expected Time
12:58
Query
Does observed information change importance of entiies?
Multiple variations of this workflow
Time range: ~2 mins to over 5 hrs (depending on fidelity choices)
Network Data
+
Data2Model
Incident Report
Large size network
Find Key Entities
Visualize

Figure 5.12: Querying based on fidelity vs. timeliness.

A variation of the workflow from Figure 5.12 is illustrated in Figure 5.13 that has fidelity reductions in terms of using cached data with a faster approximation algorithm for computing key-entities. The end user (here, an analyst) provides the control parameters or fidelity expectations that can inform him about the expected execution time and help in the generation of the right computation assembly that serves his operational needs. While the workflow in Figure 5.12 takes more than 5 hours to execute, the one in Figure 5.13 takes about 2 minutes. This dramatic time saving is achieved by approximating the results by using a slightly older, cached network and a faster algorithm that uses a subset of the networks that deals with relationships between people from the Sudan network data (instead of using a collection of other relationships such as knowledge, resources, geospatial or temporal information, etc. that can provide a detailed, but slower, analysis).

To perform an analysis like this, we could uses an architecture based approach where architectural models can represent the composition vocabulary and the constraints (as shown in Figure 5.14) and can be used to not only create the executable components, but also provide estimations of the execution time for the overall computation. The execution time for such computation could vary extensively from a few minutes to few hours, and the analyst could benefit from an estimation of the expected execution time. If the analyst knew about the time estimations, he/she could vary the fidelity of the computation before

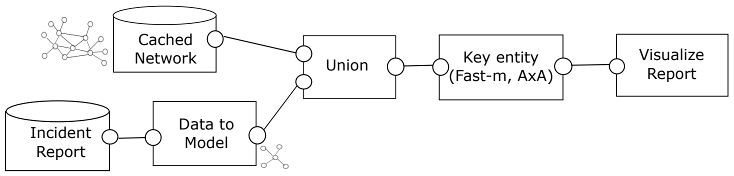


Figure 5.13: A variation of the workflow in Figure 5.12.

actually submitting the request.

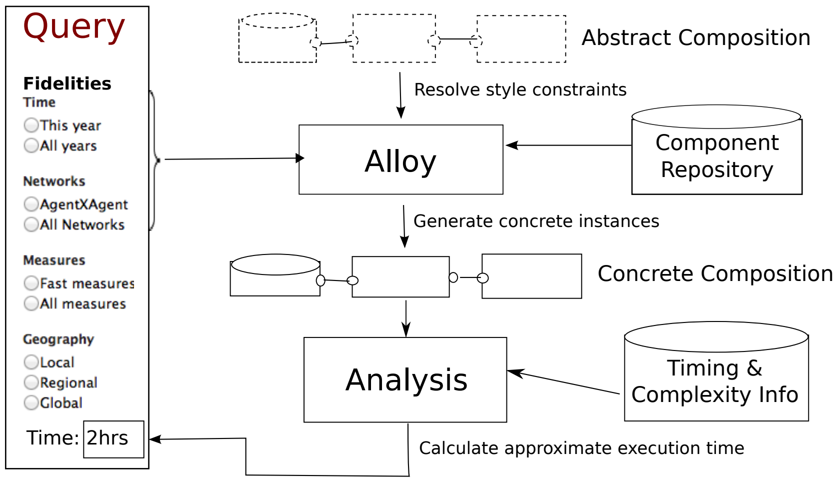


Figure 5.14: Generating concrete workflow and timing.

We implemented a simple architecture based analysis, where the UI specification is represented in an Acme, including the properties, constraints and functions. This Acme specification is converted into an Alloy model that generates a number of concrete specifications that match the constraints for the model. Of these concrete specifications, the specification that meets the time-fidelity criteria is further compiled into an executable BPEL script and executed (if the analysts decides the timeliness and fidelity is a good fit).

More details about the overall approach can be found in this paper [DGPS14]. While the architectural representation and the types of analysis could vary significantly across domains and environments, we wanted to emphasize that advanced analyses can be developed based on architectural specifications, which could enrich end-user composition and aid the development of quality end-user architecting environments.

5.6 Summary

In this chapter, we demonstrated the application of end-user architecting across two styles (dataflow and publish-subscribe), across four domains: Arithmetic expressions, Dynamic Network Analysis, Neurosciences, and Widget compositions. In the next chapter, we evaluate the extent to which the end-user architecting approach, supported by evidence from these instantiations, fulfills the thesis claims.

CHAPTER 6

Thesis Evaluation

In the thesis statement, I claimed that it is possible to build an end-user composition framework that can be instantiated to provide **high-quality** composition environments at relatively **low cost** compared to existing hand-crafted environments for a broad class of composition domains. The discussion of the end-user architecting approach and the Halo framework in Chapter 3, and Chapter 4 (respectively) demonstrates the feasibility of this claim — that it is possible to do at all. In Chapter 4 we described the Halo building blocks and how to customize them and in the previous chapter we demonstrated instantiations of Halo across various dimensions. In this chapter, I further validate the claims around improved quality, low cost and generality of the framework.

6.1 Claim: Quality assurance through Halo

For validating **quality improvements** with Halo, and getting a general understanding of what are the quality dimensions associated with composition environments, we performed a **qualitative study** to understand how end-users use their composition environments across different domains and what are the key aspects and features that indicate high quality for their composition environments. Our validation approach relies on Halo supporting all the key quality features identified in our qualitative study.

Next, we describe the study and findings.

6.1.1 Research Question

In order to validate the claim that Halo framework allows development of high quality composition environments, we address the following research question(s):

What are the common quality features that end users care about?

To what extent, and in what contexts, does Halo improve the quality of composition environments?

While quality is a subjective aspect, we hypothesized that there are a common set of quality features for composition environments across domains. Feature-rich, quality environments support these quality features to a greater extent, and the more impoverished ones don't. We evaluated this via our qualitative study, which we describe in the following sections.

6.1.2 Research method: Qualitative Study

We chose an exploratory, qualitative research method that aims to understand how end-users used their composition environments across different domains and problems faced by them. Our method consists of three main phases:

- The case selection and protocol design phase, in which we developed the research protocol and identified a diverse set of composition environments with different composition styles and application domains.
- The interview phase, wherein we elicited responses from the selected end-users
- The qualitative data analysis phase, in which we coded the interview transcripts and systematically drew inferences from the data.

Next, we describe the 3 phases of our study.

Case Selection

As shown in Table 1.1, composition environments today use a wide variety of composition models, varying from dataflows (e.g., Loni Pipeline and Taverna) to publish-subscribe (e.g., Ozone Widgets) to state-based transitions (e.g., SimMan3G simulation) to mix of composition styles (e.g., Kepler). An important consideration for our study was to explore the differences across these domains and composition models. For instance, did end-users face the same problems while designing workflows as they did while composing states? We selected 4 candidate environments that were quite different in their domain of application and composition models. Besides this, we conducted a pilot study using an industrial composition environment called “Appian modeler", which is a dataflow based composition environment.

We provide a brief description of these composition environments below:

1. **Loni Pipeline:** is a dataflow-based composition environment for neuroscience workflows. The compositions in the Loni Pipeline environment reference data, services and tools as components that can be assembled together through a drag and drop interface. As per a software usage survey [*] conducted by NeuroDebian in 2011, Loni Pipeline was one of the top 20 environments in the neuroscience domain.
2. **Taverna:** is a dataflow-based composition environment for designing and executing web-services compositions. Initially designed for bioinformatics, Taverna is currently being used by users in many domains, such as bioinformatics, cheminformatics, medicine, astronomy, social science, music, and digital preservation.
3. **SimMan3G:** is a state-based patient simulation system that facilitates health-care training by simulating real-life medical scenarios such as a cardiac arrest, breathing complications and change of vital signs on the high-fidelity manikins. Medical training professionals can combine a sequence of such activities to create a medical scenario (such as an asthma attack) and the complications that go along with it. These activities can be currently programmed in a composition and automatically executed on a manikin or a simulator.

[*]http://neuro.debian.net/survey/2011/results.html

Table 6.1: Study Participants.

Tool	Participant	Expertise Level
Appian modeler	P0 (Pilot)	Beginner
Taverna	P1	Beginner
Taverna	P3	Expert
Taverna	P4	Expert
SimMan3G	P5	Beginner
SimMan3G	P6	Expert
SimMan3G	P7	Beginner
Kepler	P8	Beginner
Kepler	P9	Expert
Loni Pipeline	P10	Beginner
Loni Pipeline	P11	Expert

4. **Kepler:** Kepler is a composition environment for designing and executing scientific workflows that uses a mix of dataflow and control flow semantics. Using Kepler's graphical user interface, users can compose various analytic components and data sources to create a scientific workflow. The Kepler software helps users share and reuse data, workflows, and components developed by the scientific community to address common needs.

For the composition environments described above, we recruited 10 participants (plus one additional for the pilot) who had a different degree of expertise in using the composition environment. The average total interview time per participant for each interview was about 35 minutes. Our participants consist of a mix of beginners (with less than a year experience) and experts (who had been using their composition environment for many years). Table 6.1 shows the list of participants for the study. It is to be noted that our "expertise level" criteria was fairly subjective and was reinforced during the interview through direct questions about the participants background and the level of their experience and expertise using their composition environments.

Semi-structured Interviews

For our qualitative study, we followed a semi-structured interviewing discipline [EH13], which means that although the interviews were guided by an explicit interview protocol that defined the general topics that the interviews would examine, we were free to devise new questions to further probe interviewees on specific subjects.

All subjects were asked to either draw a composition (as a homework task), or reproduce an existing composition they had previously drawn. During the interview, all participants were asked to open up their composition and they were interviewed about their experience writing that composition. The general technique used was to start with open-ended questions such as "What problems did you face in creating this composition?", and then ask detailed questions about specific types of problems.

Our interviews consisted of an introductory script to secure informed consent followed by a series of topics to be covered including the following:

- Questions about a participant's role and background and expertise
- Questions about a recently drawn composition (before the interview) that participants needed to open up and use as a recall mechanism

Table 6.2: Sample (first-level) codes for the study.

1st Cycle Codes.	
1. Composition motives (mot) • Simulation (mot:simulation) • Experimentation (mot:experiment) • Teaching (mot:teaching) • Automation (mot:automation) • Other (mot:other)	4. Resolution of problems (res) • Analysis tools (res: tools) • Intuition (res: intuition) • Execution (res: execution) • Reference Documentation (res: docs) • Other (res: other)
2. Nature of Composition (nat) • Computation model (nat: compModel) • Abstraction level (nat: abstractionLevel) • Other (nat: other)	5. Desired Feature (des) • General Purpose (des: general) • Tool-specific feature (des: specific)
3. Quality issues with composition environments (issue) • Technical detail (issue: techDetail) • Reuse support (issue: reuseSup) • Execution support (issue: execSup) • Analysis support (issue: analysis) • Computation model mismatch (issue: compMismatch) • Other (issue: oth)	6. Skill level of end user (skill) • Beginner (skill: beginner) • Expert (skill: expert) • Unknown (skill: Unknown) 7. Rating (rating) • Highly important (rating: highImp) • Low importance (rating: lowImp) • Unknown (rating: Unknown)
Other codes...	

- Questions about features used to create that composition
- Questions about problems faced and quality issues of the environment
- Ratings of quality issues
- Suggestions: how can limitations be addressed?

We instructed participants to speak out loud and explain their actions while working with the composition environments. The recorded audio statements of participants were further transcribed and analyzed.

Data Analysis and Interpretation

Given the exploratory nature of our research questions, “Content Analysis" [MHS13] is the main analytic method used in our study. The content analysis technique allows building an understanding of underlying reasons and motivations of participants while using unstructured or semi-structured data (such as interviews).

We recorded all participant interviews and used Amazon Turks to transcribe the audio into text, which needed some post processing. We used coding theory [Sal15] to link the findings about end-user preferences to the interview dataset and validate whether our observations were consistent. In particular, we employed a two-cycle coding method: in the first cycle, we applied the “hypothesis coding” method to our dataset using the predefined code list. In the second cycle, we applied axial/pattern coding to discover patterns from the dataset [Sal15].

A selection of sample 1st cycle codes is listed in Table 6.2. As a second-cycle coding activity, we identified patterns and selective heuristics that led to some of the key findings for the study that we discuss in the Results section.

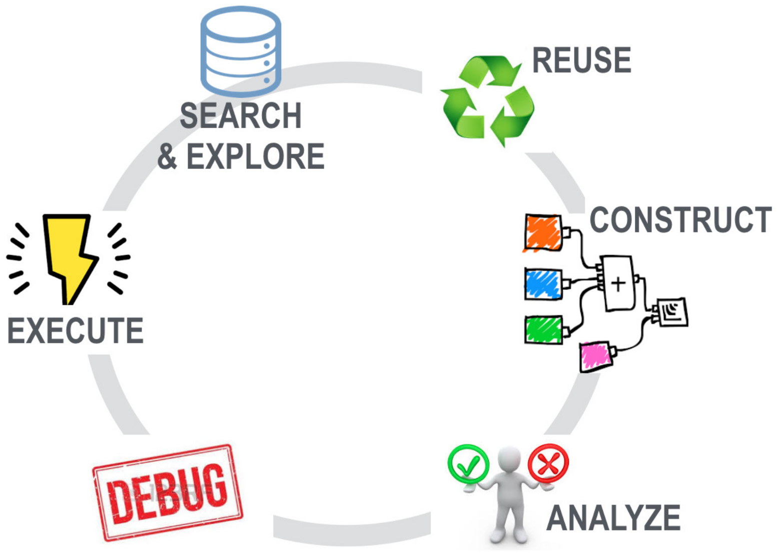


Figure 6.1: Types of End-User tasks.

6.1.3 Results

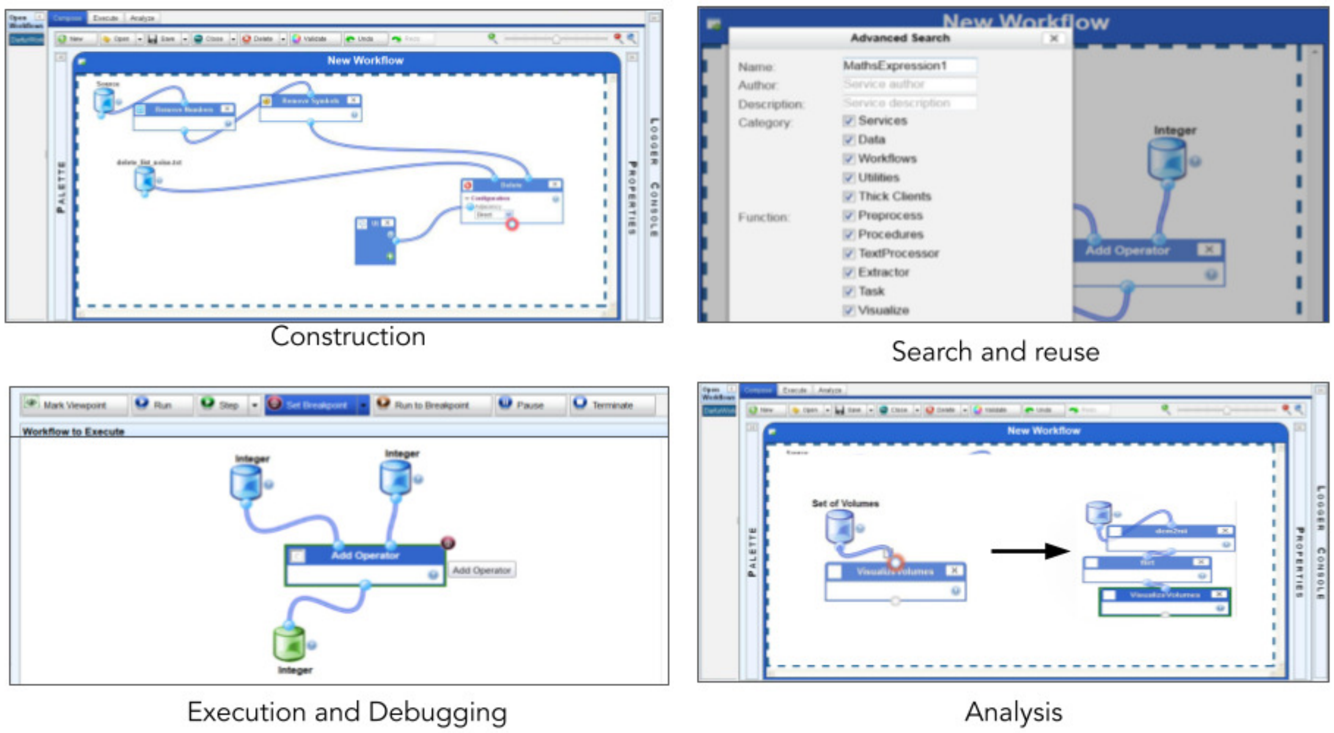
To address *ResearchQuestion* (“What key quality features do end-users want in their composition environments"), we evaluated the first-cycle attribute codes for the code (quality) issue. As a conclusion from the study, we identified that the end-users primarily used their composition environments for following “6 types" of tasks:

1. **Search and explore**: Across many domains and environments, the first step for composition usually starts with search of existing compositions on online forums, desktops or component repositories, followed by some level of reuse, experimentation and debugging. In domains like bioinformatics, brain imaging and e-sciences, there is an increasing trend to provide curated registries where users can look up specific components and compositions and download them for their use. However, searching through such repositories still remains a challenge.
2. **Reuse**: Self-reuse (using one’s own composition in a different context) and External reuse (using someone else’s composition) is often a common problem scenario for many end-user composition environments. Often professionals need to share their components and compositions with others. For instance, brain researchers may want to replicate the analyses of others, or to adapt an existing analysis to a different setting (e.g., executed on different data sets). Packaging such compositions in a reusable and adaptable form is difficult, given the low-level nature of their encodings, and the brittleness of the specifications.
3. **Construction**: Combining visual (and computational) elements by drag-and-drop is a common activity across many composition environments. These composition steps usually entail some common direct manipulation activities and a number of user commands such as adding composition elements, specifying their properties, and relationships between composition elements. While semantically many of these composition commands look similar, the exact nature varies across composition styles, UI technologies and domains.
4. **Analysis**: End-user compositions often enforce restrictions on legal ways to combine elements, dic-

tated by things like format compatibility, domain-specific processing requirements, ordering constraints, and access rights to data and applications. Discovering whether a composition satisfies these restrictions is usually a matter of trial and error, since there are few tools to automate such checks. Moreover, even when a composition does satisfy the composition constraints, its extra-functional properties — or quality attributes — may be uncertain. For example, determining how long a given computation will take to produce results on a given data set can often be determined only by time-consuming experimentation.

5. **Execution**: End-users often need to interactively execute their compositions to learn about the domain or perform computation tasks. However, compared to the capabilities of modern programming environments, end-users have relatively few tools for things like compilation into efficient deployments, interactive execution and monitoring intermediate results. This follows in part from the fact that in many cases compositions are executed in a distributed environment using middleware that is not geared towards interactive use and exploration by technically naive users.

6. **Debugging**: Debugging support is often the least mature capability across most composition environments. Capabilities such as interactive testing and debugging by setting breakpoints, monitoring intermediate results, history tracking, and graceful handling of run-time errors are challenging for composition environments. Furthermore, the vocabulary of execution is often very different to the vocabulary of construction and many naive end-users are not proficient in low-level computational details. It is not surprising that support for debugging is missing in most environments.



Name: MathsExpression1
Author: Service author
Description: Service description
Category: ✓ Services ✓ Data ✓ Workflows ✓ Utilities ✓ Thick Clients
Function: ✓ Preprocess ✓ Procedures ✓ TextProcessor ✓ Extractor ✓ Task ✓ Visualize

Figure 6.2: Halo support for the end-user tasks.

In our findings, analysis and execution support were the most important features for end-users as they not only helped in debugging but also interactive learning of compositions. Support for reuse was also an important requirement to address composition problems. However, the form of reuse varied across environments. “Self-reuse" was the primary form of reuse. “External reuse" via repositories was mainly used for experimentation and learning. This was not a surprising result as prior studies have reported similar observations [GCG+14].

Furthermore, by examining some of the successful composition platforms such as LoniPipeline [RMT03], Taverna [HWS+06], Galaxy [GRH+05] and Wings [GRD+07], it was possible for us to identify the larger ecosystem consisting of various domain-experts and developers. We identified various developer roles in such ecosystems. Examples of such roles include [†] component developers, platform developers, domain experts, etc. Even though impoverished composition environments are common today (where end users perform many of these roles by themselves), rich compositions environments manage this synchronization quite well[‡]. This allows them to develop higher quality of composition environments.

A quality framework not only supports most of the end user tasks, but it also needs to be able to support all these developer roles. Here are some of the ways Halo supports these developer roles:

1. *Component developer*: implements components such as applications, services, libraries, scripts and data elements, and makes them available in a repository. Halo defines the interfaces that these components need to implement so that they could participate in end-user compositions.

2. *Component integrator*: wraps these components to provide Halo interfaces, which ensures that the components provide the access protocols and the methods required by the framework.

3. *Domain expert*: identifies the composition vocabulary that consists of computational elements in the domain, their properties, the composition constraints, and the high level classification schema that will aid in browsing and search. A domain-expert uses his expertise to define the ground rules for compositions in that domain. Furthermore, based on the quality objectives of a particular domain, a domain expert also defines the types of analyses that may be performed over the compositions. Domain experts communicate this information to a *framework instantiator* who formally encodes the composition vocabulary in styles, and develops the analyses.

4. *Framework instantiator*: instantiates the Halo framework to create an end-user composition platform. It is his responsibility to customize the Halo framework by using appropriate plugins for (i) architectural representation, (ii) reuse support, (iii) component and data reference (iii) analyses, (iv) component registry, and (v) execution-support. He creates the user interface, execution platform, and the intermediate architecture layer and integrates them together to create a working environment for composition construction, execution, and analysis.

5. *End-user architects*: use an end-user composition environment to create compositions and templates for end-users. They design, analyze, and execute high quality compositions and package and register their compositions as reusable templates for end-users to use for their specific composition tasks.

Not only does Halo provide support for each of these roles, it improves the quality of the developed environments by supporting all the key end-user tasks. While, how these end-user tasks are presented

[†]Note that the roles that we identify for Halo closely correspond to these roles.

[‡]For example, composition environments such as WINGS and Taverna are built on an ecosystem of developers and domain-experts who design components, ontologies and rules to promote easier composition

to the end user would vary across environments, supporting them through the framework allows easier development of these qualitative features.

6.1.4 Discussion and limitations

One of the research questions we asked earlier was "To what extent, and in what contexts, does Halo improve the quality of the composition environments". Our qualitative study delved deeper into the types of composition features that are "typically needed" across composition environments, the various roles that are engaged in the development of composition environments, and how Halo can support both of these. While these are common features that indicate quality, these are in no way exhaustive. Additionally, even with these features, a buggy implementation or other discrepancies can make the composition platform (and the associated user experience) undesirable. On the extreme end, there may exist domains where the nature of composition construction may be completely ad-hoc, the nature of analysis may not be architectural; or even if associated architectural composition and analysis constructs can be defined, their utility may not justify the effort.

If we set aside these extreme cases, one way to think about the applicability criteria of Halo would be to think of the common computational tasks associated with composition. As illustrated in Table 6.3, even for some very diverse domains, if the composition modelling, execution and analysis activities rely on fixed component types, with well-defined domain constraints, it is feasible to define an end-user architecture associated with such composition and drive composition and analysis through Halo. A framework integrator would still need to customize the Halo building blocks; and the related costs would vary. However, by customizing and integrating Halo building blocks composition platform developers can get many of these quality features in an off-the-shelf manner.

Table 6.3: Activities involved in end-user composition.

Activities / **Tools**	**Composition**				**Execution**		**Analysis**	
	Conception	**Reuse Support**	**Modeling**	**Model Analysis**	**Distribution**	**Monitoring**	**Visualization**	**Query**
Taverna (Bioinformatics)	Computational pipelines	Search web-service registry tags	Web-service compositions	Correctness/ Availability	Remote server	Client-side display	Model composition and execution	Execution results
Ozone (Geospatial analysis)	Widget based interactive visualization	Widget lookup	Publish subscribe messages	Message flow control	Remote server	Widget display	Widgets composition and subscriptions	Execution results
VST (Digital Music Production)	Audio sequencing affects	Audio filters	Pipeline of audio filters	Correctness / Aesthetics	Local	Music production	Music filter assembly	Execution results
SWiFT (Dynamic Network Analysis)	Pipeline of DNA services	Component Registry	DNA workflows	Correctness, Security, Performance etc	Workflow server	Workflow execution, analysis and intermediate results	Views for composition, analysis and execution	Specific analyses/results
WINGS (Sci. computing)	Computational pipelines	Semantic templates	Semantic compositions	AI-planning/ Semantic reasoning	Workflow server	Client-side display	Model, Results	Workflow execution results

6.2 Claim: Cost Improvements through Halo

For validating **cost-effectiveness** of Halo, we use a formal **task analysis approach** that compares the cost of building key composition environment features with and without the framework. We assessed the overall engineering effort to build composition environments by reusing and customizing Halo components instead of building everything from scratch.

Next, we describe our cost evaluation approach.

6.2.1 Research Question

In order to validate the claim that Halo framework reduces the costs of developing high-quality composition environments, we address the following research question:

To what extent, and in what contexts, does Halo reduce the cost of composition environments?

6.2.2 Research method: Task comparison framework

This section describes an analysis of the tasks required to design and construct a customized composition environment comparing the cost of building an environment ground-up, using the end-user architecting approach in which the Halo building blocks of the composition environment are incrementally customized and integrated via adapters. This Task framework was proposed by Monroe et al. [Mon99b] and used by Cheng et al. [Che08] for evaluation of framework instantiations.

Each of the tasks described in this analysis is given a time estimate that includes best-case, average case, and worst case times for each stage along with the criteria that determines what makes these projects fall into best case, worst case, or average case categories. The effort numbers assigned in these task categories are rough estimates based on data from the various case study implementations conducted as a part of this research, informal estimates from teams who have built similar tools, and estimates based on lines of code and effort estimates from code repositories and published data. These estimates can, of course, widely vary depending on the scope and complexity of composition environments and the various conditions and the rigour of project execution. However, these should provide a ballpark estimate of the likely amount of time needed for constructing these environments. In Section 6.2.3 I further explain how each of these estimates were calculated.

Task evaluation for building composition environments

The Halo framework allows the developers to customize various APIs and use architectural styles to build and analyze compositions. The previous chapters demonstrated how Halo allows various customization mechanisms and supports creation of quality end-user composition environments. In this chapter, I further discuss how Halo reduces the cost of this development.

As a first step to evaluate the cost effectiveness of using the Halo framework it is important to assess the engineering effort required for overall design and implementation needed to instantiate Halo in a particular domain. The Halo framework provides the building blocks to support end-user architecting, including the adapters needed to assemble the architecture layer and other components (i.e., the user interface, a repository, a set of analyses, and an execution platform). However, the complexity of this assembly varies significantly based on these factors:

	Arithmetic Expression Environment (Best Case)	**Brain-imaging composition Environment (Average Case)**	**Widget Composition Environment (Worst Case)**
Complexity of Architecture Vocabulary	Small, homogeneous set of design elements	Larger, heterogeneous set of design elements	Complex, poorly-understood, heterogenous design elements
Ease of Customization	Limited customization required Easy reuse of adapters, and UI	Some effort needed to customize the adapters Needed new components for compilation, visualization and display	Significant effort to integrate all components Lack of existing components and adapters

Table 6.4: Criteria that impact cost of Halo instantiation.

1. **Complexity of Architecture Vocabulary:** The level of complexity of end-user architecting environments can vary significantly based on level of documentation, familiarity of the designer with the domain, homogeneity of design rules, and the number of design elements involved. For instance, a simple arithmetic composition environment may have smaller number of design constructs, while a full-fledged brain-imaging environment composition environment may have pose higher complexity for a designer; or worse, a composition environment can have complex/poorly-understood, heterogeneous design constructs that can make architecture modeling complex.
2. **Ease of Customization:** Another factor that greatly impacts the cost of Halo instantiation is the level of effort required to customize existing components to put together an end-user architecting environment. If the instantiation requires straight forward assembly without significant modifications, it lowers overall cost. On the other end, if the customization requires major changes, the costs are much higher.

Table 6.4 summarizes the main criteria that determine the overall instantiation cost of Halo. In our case studies demonstrating Halo instantiation, the arithmetic expression environment was the best case scenario as it involved least development costs. While the Arithmetic expression environment was an illustrative example, the Brain imaging environment is an average case scenario that involves significantly more complex domain vocabulary, where although some components are provided for the framework instantiators, they still need to customize the adapters and write some code to integrate all the pieces. The Widget composition environment, on the other hand, is the worst case scenario where the framework instantiator needs to spend some effort not only in modelling the complex vocabulary, but also they have to spend some effort writing custom adapters and other integration pieces to instantiate Halo.

Cost estimation technique used and data gathering

For the time estimates presented in this thesis, mainly three data points were used:

1. **Time estimates based on framework implementations by the author:** One of the primary sources of data for the time it took to perform various task categories was based on actual instantiations of the framework by the author. For both best-case and average-case instantiations mentioned above, we had full instantiations of Halo. This gave us a ballpark estimate of approximate time for such

framework instantiations.

2. **Time estimates based on similar environment implementation by external members:** A second source of data was how much time it took to build similar composition environments by external teams. One of the candidates for this was a workflow environment developed as a part of SORASCS project [SGD+11] by a Carnegie Mellon masters team. As a part of a PSP project, the team tracked implementation effort and provided cost estimates for developing not only the entire platform but also individual features. This data was extensively used to make estimates about time estimates without the framework. A summary of breakdown of implementation costs is shown in Appendix-B.

3. **Time estimates based on lines of code and other heuristics:** A third source of data was based on estimation rather than actual implementation for all the features. Especially, in the case of worst-case implementation, where it was not feasible to fully instantiate an entire composition platform. In such cases, we could only implement partial features and make ballpark estimates about the overall implementation effort for features that were not implemented.

Note that all three approaches provided a ballpark estimate for the overall implementation effort. We explored more formal estimation approaches like COCOMO [BCH+95], but those seemed impractical for a multiple of reasons. For one, building a regression based cost model based on historical execution and consideration of various project and developer attributes would be challenging for my research validation as that technique is probably suited better for a single instantiation over a longer time, as opposed to multiple instantiations where the worst case implementation included a partial set of features. And even this estimation would be imprecise. An informal task-estimate was therefore considered a better fit for ballpark estimates.

6.2.3 Results

Using the above breakdown of the tasks, we compared three instantiations of Halo. Key to this approach is breaking down the framework instantiation into tasks performed by the engineers, including: *domain analysis, model capture, design and implementation, and modifications* and comparing the best-case, average case and worst case effort required for those tasks.

The criteria for identifying best-case, average-case and worst case is as defined previously in Table 6.4. We notice that while in best-case and average-case, the Halo framework reduces the cost of implementation nearly by half; even in worst-case, Halo offers significant improvements to cost of development as compared to building the environment from scratch. Table 6.5 shows the overall breakdown of the costs for the best-case, average-case, and worst-case composition scenarios where the environment is implemented from scratch and with the framework.

In Table 6.6 we further breakdown the cost of development of key Halo features. As listed in Section 6.2.2, our estimation was based on three factors: (1) time estimates done via actual instantiations of the framework (2) time estimates based on similar environment implementation by external teams (and a comparison of features), and (3) estimation based on code and other heuristics, where the accuracy of estimations would decrease from actual calculations to estimations. As shown in Table 6.6, we had full instantiation of Halo to create an arithmetic expression and neuroscience environments. Moreover, even for worst case scenarios, we had partial implementations to understand the complexity and aid the esti-

	Best Case (Arithmetic Expression)		Average Case (Neuroscience Composition)		Worst Case (Widget Composition)	
Stage of Development	**Custom**	**Halo**	**Custom**	**Halo**	**Custom**	**Halo**
Domain Analysis	1 week	1 week	4 weeks	4 weeks	6 months	6 months
Model Capture	N.A.	1 day	N.A.	1 week	N.A.	1 month
Design/Implementation	2.75 months	1.25 weeks	6 months	3 months	2 years	17 months
Development Total	$\sim$3 months	$\sim$1.5 months	$\sim$7 months	$\sim$4.25 months	$\sim$30 months	$\sim$24 months
Update & Modification	$\sim$1 week	$\sim$1 day	$\sim$2 weeks	$\sim$2 days	$\sim$8 weeks	$\sim$2 weeks

Table 6.5: Overall costs for Halo instantiation tasks.

mation exercise [§]. We could also estimate costs for what would have taken to build such an environment from scratch by comparing data from a MSE project team that implemented a workflow tool for SORASCS [SGD^{+}11] [¶]. We compare the development costs from Arithmetic expression and Neuroscience composition examples, and do an analysis of custom solutions based on approximate effort it would take to implement similar functionality. In doing this analysis, our findings indicate that the majority of savings are towards implementing analysis support.

	Best Case (Arithmetic Expression)		Average Case (Neuroscience Composition)		Worst Case (Widget Composition)	
Key Feature	Custom	Halo	Custom	Halo	Custom	Halo
Repository Support	2 weeks	1 day	4 weeks	1 week	1.5 months	2 weeks § ‖
Analysis Support	2 weeks	2 hours	5 weeks	1 day	2.5 months	1 week §
Domain modeling	2 weeks	1 day	5 weeks	5 days	3.5 months	1 month §
Execution	1 week	2 days	2.5 weeks	1 week	5.5 months	3 months
Mapping	1 week	1.5 weeks	2 weeks	3 weeks	2 months	4 months
Visualization	3 weeks	3 weeks	6 weeks	6 weeks	8 months	8 months
Total	$\sim$2.75 months	$\sim$1.25 months	$\sim$6 months	$\sim$3 months	$\sim$24 months	$\sim$17 months
🟨	**Time estimates based on similar environment implementation by external members**					
🟩	**Time estimates based on framework implementations by the author**					
🟦	**Time Estimates based on lines of code and other heuristics**					

Table 6.6: Breakdown of design/implementation cost for key Halo features.

Although, while instantiating Halo, developers do need to incur some costs especially towards archi-

[§]Part of this partial instantiation involved implementing prototypes and domain modelling where Halo was instantiated partially without fully implementing all the adapters. The goal of this exercise was to understand the domain and make informed estimations of overall development costs.

[¶]More data for that is referenced in Appendix-B.

tecture modelling and writing the mappings to map the architecture vocabulary and UI and runtime, these costs are amortized by significant benefits in terms of lowering the costs to write analyses. Furthermore, while user interface and execution runtime are not key to the end-user architecting approach (given the wide range of design choices), Halo still reduces the cost of mappings that framework instantiators have to write anyways, often from scratch. By providing customizable APIs where common end-user composition commands are implemented as API methods and are further customizable, Halo provides significant benefits towards implementation of end-user composition execution functionality.

Overall, our experiments indicate that on an average, Halo reduces the cost of composition environment development by half. In the worst case, when the level of customization needs significant developer effort, even then Halo allows significant reductions in the cost of development. We believe this is a significant improvement over writing composition environments from scratch, especially given the large scale of functionality that is typically required for developing feature-rich, composition environments.

6.2.4 Limitations

One of the limitations of our approach is that the task framework provides an estimation but not exact measurement of the amount of it would take for various design and development activities associated with the composition environment development. Our estimates are based on multiple sources, including (i) data from the various case study implementations conducted as a part of this research, (ii) from other teams who have built similar tools, and (iii) based on lines of code based estimations. Not only can these time estimates vary based on the level of customization a framework instantiation may involve, but also the degree of reuse by a framework instantiator. For instance, reusing an existing adapter with limited customizations would need significantly less work than if an instantiator had to develop a net new adapter based on the modules provided.

Finally, another limitation of the validation approach is that the instantiations of the framework were done by the author. In an ideal world, we could have extended the validation to a controlled experiment where software developers with different roles instantiated Halo to their composition environments, and this could have been compared with another set of developers who built the environments from scratch. However, given the scope, budget and duration of the thesis research, such validation was not feasible. Despite these limitations, a careful attempt was made by the author to capture the data based on a task-based estimation model. While the composition environments vary considerably, the ball-park estimates were made based on the overall effort and often the lines of code needed to perform individual tasks.

6.3 Claim: General Purpose framework

To demonstrate generality of our approach, we performed a series of case-studies where we performed in-depth investigations of end-user architecting by instantiating Halo and developing various composition environments and their key components using the Halo framework. As described in the previous chapter, we demonstrated four instantiations of Halo, within the two computation styles of data flow and publish subscribe along with adapters to integrate them. We also gave examples of analyses based on end-user architectures.

In this section, we argue that this case-based generalization approach helps us demonstrate generality of the framework. Next, we describe the research question, our approach to demonstrate generality, and the limitations of our approach.

6.3.1 Research Question

In order to validate the claim that Halo framework is general-purpose, we address the following research question:

What is the evidence that Halo is a general-purpose framework?

To address this research question we use a case based generalization approach to demonstrate customizability of the Halo framework across multiple dimensions: including different computation styles, user domains, and various adapters needed to plug in different visual interfaces and execution environments. We described some of these instantiations in Chapter 5. In this section we describe how that evidence supports our claims around generality.

Observe case phenomena.
Explain the phenomena architecturally.
Generalize the theory to architecturally similar cases.

Table 6.7: Building architectural theory of case phenomena in the field [WD15]

6.3.2 Research method: Case based generalization

Wieringa and Daneva, in one of their works, discussed strategies for generalizing software engineering theories [WD15]. They argued that for the field of software engineering in general, it is often hard to define controlled experiments that capture all aspects of design and development. In such domains, we can use case-based generalization, where by studying individual cases, and generalizing about components and mechanisms found in a case, we can identify similarities across those cases. Furthermore, in such case-based research, variability is reduced by decomposing a single case into components with interactions, for example: people and roles in a project, layers of architecture, or components that are part of the architecture. These components and mechanisms may be recurrent across a large set of different cases, and hence contribute to the overall generalization.

Our validation is guided by this approach. Table 6.7 shows the overall steps of the approach.

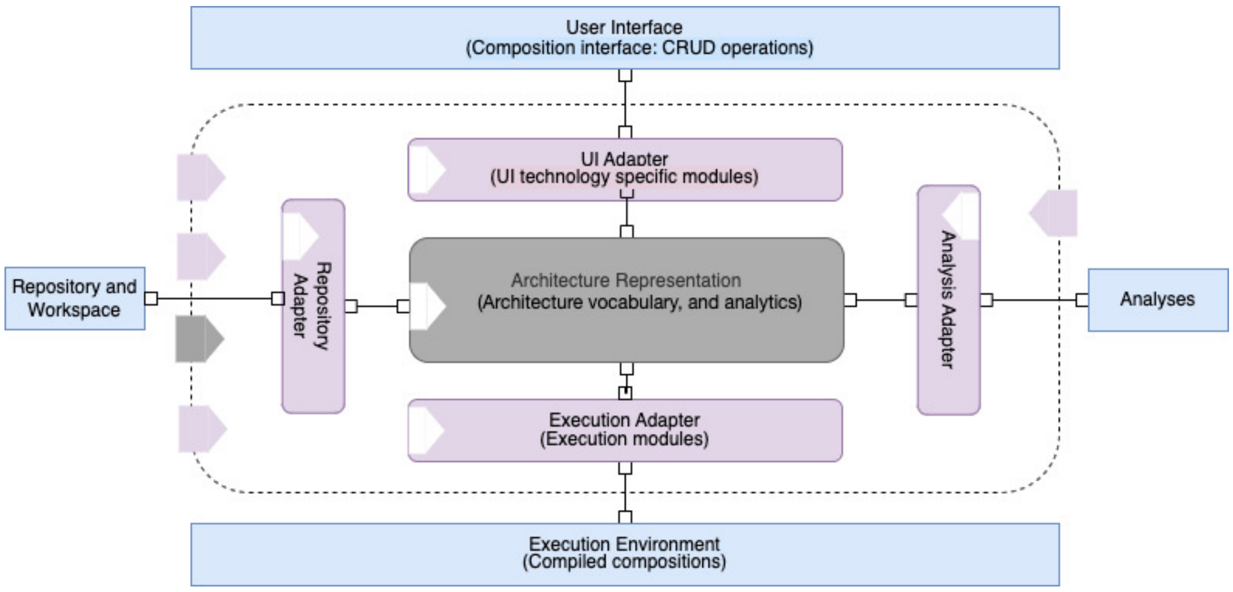


Figure 6.3: Halo framework architecture and customization points

Revisiting the general Halo architecture of Halo, as shown in Figure 6.3, we make a case that this general-purpose architecture can be instantiated in multiple domains, across multiple architecture styles, and integrates different user interfaces, execution environments, and analyses while the framework allows customizable plug in points to make this possible.

6.3.3 Results

In Chapter 5, we described various case studies that demonstrated multiple instantiations of Halo covering various dimensions of variability. Specifically, we demonstrated variability along the following dimensions:

1. Instantiations across 2 Computation Styles (Dataflow and Publish Subscribe)
2. Instantiations in 4 domains (Arithmetic expressions, Dynamic Network Analysis, Neurosciences, and Widget compositions)
3. Instantiations with 2 UI Adapters
4. Instantiations with 2 Execution Adapters
5. Instantiations with 2 Analysis Adapters
6. Instantiations with 2 Repositories

As we have discussed before, while the nature of composition may vary, Halo provides a general-purpose architecture that allows association of an architecture layer through the use of customizable adapters. Considering some of the key adaptation points:

- **User Interface:** While Halo does not specifically target or provide features for visual composition, it recognizes that a common model of composition consists of a drag and drop (or other kinds of drawing) interface where end-users can compose components in meaningful combinations. The style and vocabulary of such end-user compositions may vary from dataflows, to publish-subscribe widgets, to a mix of composition styles. Halo provides customizable APIs to translate these end-user compositions to architecture representation and execute them. While Halo supports some of the most common CRUD operations, it is possible to extend them if a specific interface calls for new types of user interaction.

- **UI Adapter:** UI adapters provide translation from user vocabulary to architecture vocabulary. While the specific implementations of UI adapters vary across composition environments, Halo provides a generic set of customizable APIs and modules to build these UI adapters. As a part of this research we implemented two example adapters. These can be adapted for new instantiations; or in a completely new domain, framework instantiators can implement new adapters from scratch.
- **Architecture Representation:** The architecture layer provides the key functionality of architectural representation, analysis and execution management. This layer provides the APIs to define compositions using domain-specific architecture styles, which specify the vocabulary of element types and constraints on compositions. Not only this layer is quite customizable, decades of academic research in software architecture have demonstrated how a variety of system architectures can be represented in runtime architectures and analysed in various ways. Acme as an architecture definition language and the associated tools and case studies provide significant support to model different types of architecture in an expressive way.
- **Execution Adapter** Execution Adapter is the layer that provides translation from architecture vocabulary to execution semantics. Like the UI adapter, the implementations of the execution adapter vary across different runtime environments. Halo provides a generic set of runtime APIs that can be further customized.
- **Analysis Adapter:** Analysis Adapter is the layer that allows integration of external analysis into Halo. Individual analysis can be based on different models or written with different assumptions, but the analysis adapter provides a generic mechanism to invoke the analyses.
- **Repository Adapter:** Repository Adapter is the layer that allows integration of external repositories into Halo. The repository adapter provides mechanisms to import a composition written into an external vocabulary to by transforming it into an architecture based vocabulary used by Halo. The degree to which such a layer can be customized, or is needed can vary. Some domains do not have an established component library; others don't have a need for it. But having such a repository allows environment developers an easy way for both internal and external sharing. Also, the translation mechanism may be different across domains but it can be customized to support different composition types.

6.3.4 Limitations

In the field of basic sciences, typically the studies aim to generalize the results from lab-to-lab, requiring the researchers to eliminate much of the variability of the real world in the laboratory. To accomplish this, often the research technique is based on idealized assumptions that may never be possible in practice (for example, constructs like Turing machine, point mass, perfect vacuum, and so on). In contrast, in the field of software engineering, such a generalization may not be practical. As an illustrative example, the number of variables involved for building quality end-user architecting environments could be quite large (e.g., they can vary across user interfaces, runtime environments, architecture models, the developer ecosystem, and so on). Also, given the complexity, there could be limitations on the amount of time needed to actually instantiate an environment. Unlike the scope claims in basic sciences, where one could limit variables via a controlled experiment, the scope claims for such an engineering field could be less about idealizing assumptions, but more about making patterns of behavior visible. [Wie14, Lay95]

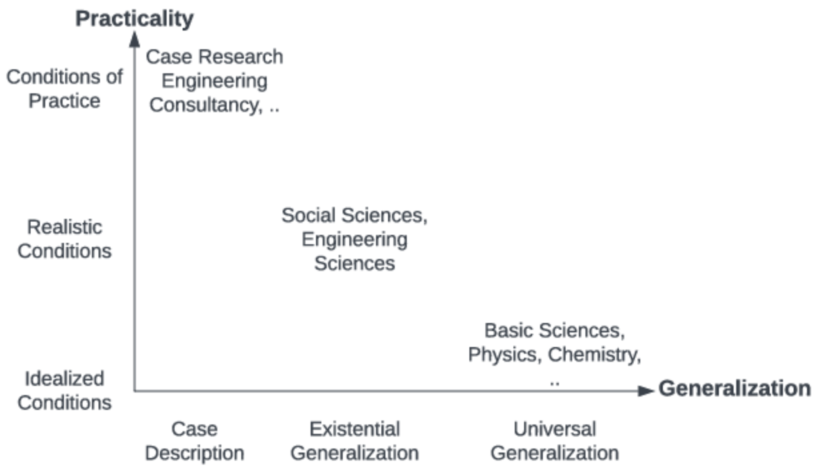

Practicality
Conditions of Practice
Realistic Conditions
Idealized Conditions
Case Research Engineering Consultancy, ..
Social Sciences, Engineering Sciences
Basic Sciences, Physics, Chemistry, ..
Generalization
Case Description
Existential Generalization
Universal Generalization

Figure 6.4: Trade-off between generality and practicality. [WD15]

Consequently, in the field of software engineering and information systems, where the main challenge is to deal with a variety of uncontrolled conditions of practice, rather than aiming for universal theories, the more practical approach is often to develop middle-range theories [Wie14]. Common examples of middle range theories include theories like the COCOMO model [BCH+95], the software engineering principles of Davis [Dav95] and some of the theories listed by Endres and Rombach [ER03]. Such theories are therefore called middle-range theories, and shown in Figure 6.4.

Our research predominantly fits this category. As shown in Table 6.7, by observing a case phenomena, and explaining the phenomena architecturally, we could generalize the theory to architecturally similar cases. In our research for example, for validating the generalizability of the end-user architecting approach I have considered multiple dimensions of variability. While these dimensions may not be exhaustive, or even applicable to "all the possible domains", such existential generalization is still valuable for a large number of domains. Practitioners who want to apply such a middle-range theory to their particular case could benefit from this technique and often adapt the research to their requirements.

6.4 Summary

In this chapter, we addressed how the examples and evidence collectively fulfil the thesis claims of **generality**, **cost-effectiveness**, and **quality**, summarized in the Table 6.8.

Table 6.8: Summary of claims and evidence

Claim	**Evidence**
Generality	Existential Generalization using a case-based approach, with instantiations across these dimensions: * 2 Computation Styles (Data flow + Publish Subscribe) * 4 Domains(Arithmetic expressions, Dynamic Network Analysis, Neurosciences, and Widget compositions) * 2 UI Adapters, 2 Execution Adapters, 2 Analysis Adapters and 2 Repositories
Quality	Support Search, Reuse, Construction, Analysis, Execution, and Debugging support
Cost	Cost reduction in the best-case, average-case, and worst-case for the example domains

CHAPTER 7

Design Choices and Limitations

In this chapter, we discuss some of the key design choices of the Halo Framework and the tradeoffs and limitations arising from those. In particular, we address the following design choices and limitations of each:

- Restricted to *End-User Architecting* frameworks (Scope)
- Managing the *open-world vs. close world* model trade-offs (Scope and Design Choice)
- Composition environments are implemented as an MVC-style application, where user interface and run-time adapters allow assembly and execution of the compositions (Design Choice)
- *Separation of concerns* across framework layers (Design Choice)
- Limitations to to using *Extensible Adapters* for bridging/integration (Limitations)
- Limitations to *using a model* for End-User Architecting (Limitation)
- Limitations to *expressiveness* of the model (architecture vocabulary) (Limitation)
- Limitations to *framework reuse* and cost-effectiveness (Limitation)
- Maturity of the domain (Limitation)

7.1 Focus on End-User Architecting

Our approach is centered on providing a generic architecture-based framework with reusable infrastructures than can be tailored to particular system styles and quality objectives, and further customized to specific composition scenarios. While it is a generic approach, its application is limited to tools and frameworks related to end-user architecting. Having an architecture model that represents the composition vocabulary and uses it for interesting analyses is key to the approach.

Today, composition environments support paradigms varying from free-form drawings to various specialized constraints. The user interfaces and runtime environments for these composition environments may vary so much that implementing them would be infeasible and therefore not the target of this thesis. There is significant work done by other researchers towards improving the quality of end-user composition and the performance of execution environments. This thesis instead focuses on how the user interfaces and execution environments can be integrated with an architecture layer in a cost effective manner, where interesting analyses can augment the composition activity.

7.2 Open world vs. closed world models

An important concern for end-user architecting is what types of composition vocabularies, design languages, and constraint systems can it address? Is it open to allowing new forms of composition technologies and libraries (that are developed in thousands every year)? How about handling computation models and compositions that we haven't seen yet (open world assumption)? Our approach to addressing these design issues is to split the design representations in two parts. At its core, Halo implements an architecture vocabulary, which is generic and rich enough to represent complex domain-specific constraints. We have designed approaches to specialize and extend architecture vocabulary to write complex vocabularies. In a way, at its core the rich architecture vocabulary follows a closed-world model using styles to represent the design elements and constraints.

However, Halo makes it possible to have external vocabularies and languages for compositions that are mapped to this architecture vocabulary through adapters. We understand that new user interfaces, runtimes, analyses or compositions may come up based on instantiation requirements. The design of adapters allows for this open-world assumption. This does come at a cost as framework instantiators may have to build their own adapters to suit their language or framework choices, but Halo provides templates and extension mechanisms to build these adapters at a low cost. In some cases, this reuse of adapters might be cheaper, while some other instantiations may require more development effort. Nevertheless, the total cost of building the composition environments is still lesser in comparison to developing everything from scratch.

7.3 Framework instances as an MVC-style application

While Halo can support various composition vocabularies, the environments instantiated using Halo follow a Model-View-Controller style architecture.

- **Model**: At it's core is the central architecture layer, which forms the model that defines the data-structure consisting of architecture styles and their refinements and specializations. These styles allow capturing of compositions and their constraints.
- **View**: While compositions are modelled in an architectural vocabulary, to an end user these are presented via different visual interfaces, where they can assemble them. Halo allows integration of various such interfaces, which form the view layer.
- **Controller**: While users interact with the user interfaces to assemble and execute compositions, Halo implements various controllers in the form of adapters that respond to the user input and perform interactions on the data model objects. This layer is responsible for various actions including type-checking, validations, import and export of compositions, and providing feedback to the user interfaces based on the execution status.

  While the exact implementation of the model-view-controller pattern varies across composition environments and architecture styles, MVC-pattern defines the mode of interaction for all composition environments built using Halo.

7.4 Separation of Concerns across framework layers

A frequent tradeoff for frameworks is deciding between generality versus power. A framework could be powerful but restricted to a specific domain or a task, or it could be quite generic and lose some of its power. At its core, Halo tries to address this tradeoff by using a generic architecture vocabulary that can be customized and refined using styles, but restricting the construction and manipulation of the architecture vocabulary through adapters. Halo used the principle of separation of concerns by creating a framework stack where individual components handle separate concern and the framework imposes reasonable limitations on how these components should or should not interact. For instance, the architecture layer builds and manipulates the vocabulary in an ADL. The UI and execution layer are responsible for representation of composition drawing and execution, respectively. While the Analysis and Reuse layers help with the reuse of analysis plugins and import and export of compositions. All these are integrated through adapters that provide the mapping and glue code for integration. As with most abstractions, interfaces must be added that define the intents for each layer. These interfaces can be extended and refined and thus allow framework instantiators easier integration by providing ready-made templates.

This separation of concerns allow independent reuse and modification of various framework components. In practice, however, there is some level of dependence across the components. For instance, what must be shown on the UI depends on how the execution runtime decides exposes those functionality. However, a clear separation of concerns allows the framework instantiators to avoid hardcoding those interactions. In the long run, not only it saves costs, but it also leads to better designed platforms.

7.5 Limitations to using Adapters as bridging/integration mechanism

As per Pree [Pre94], software frameworks consist of frozen spots and hot spots. Frozen spots define the overall architecture of a software system, which constitute its basic components and the relationships between them. These remain unchanged in any instantiation of the application framework. Hot spots represent those parts where the programmers using the framework add their own code to add the functionality specific to their own project. In Halo, the adapters and the architecture layer provides abstraction and sub-classing as mechanisms to implement the hot and cold spots. Instantiation of Halo involves extending the adapters by implementing the abstract classes, sub-classing and composing them when necessary to extend the interfaces, thus allowing integration of all the layers.

While the design of these adapters is fairly generic, there are limits to how much the hot spots of the framework can be extended or refined. For one, instantiator defined sub-classes must extend the abstract methods as defined in the framework. The inversion of control and the specific interaction pattern between the individual layers is fixed. While Halo provides some example adapters, if the instantiators need to deviate significantly from those framework-provided adapters, or the user-interface/runt-time technologies, they may need to develop their own adapters taking the current adapters as templates or examples.

7.6 Limitations to using (architecture) models

The end-user architecting approach uses architecture models to mediate design, analysis, verification, execution and reuse of compositions. This is based on assumption that architectural abstractions (and their manipulations) can capture the structures, relationships and constraints for end-user compositions. Halo uses the Acme ADL, which is well suited to capture design constraints in an architectural vocabulary. As long as compositions can be represented in a component-connector model, a model like Acme is a good fit to define the composing vocabulary and extend and refine it with properties, design constraints and rules. Furthermore, there have been been multiple case studies that have also augmented Acme to involve state-based and probabilistic specifications. However, while this holds true for a broad set of end-user architecting domains, it may be possible that there exist domains where a component-connector model may not be a good fit. If the inherent style for the target domain is something quite different or if it is hard to define a vocabulary (for any reason), Halo may not be suitable for such domains. While most of the common end-user architecting environments have vocabularies that can be represented as component and connector styles, one can imagine scenarios such as those involving free form compositions (with limited constraints), state-based compositions where components and connectors are not clearly defined, or some widget scenarios where connectors are adhoc. Using architecture models for such composition styles may not be an inappropriate model, and therefore, not a good fit for such use-cases.

7.7 Limitations to expressiveness of the model

Halo targets a general purpose ADL like Acme as its core building block, because it's generic and allows refinement and specialization through styles. As long as the modeling scenarios can be represented in component-connector models, Acme can be used to not only specify the vocabularies with the element properties, constraints and other rules around composition. While a generic ADL like Acme is a powerful tool to express a wide range of vocabularies, we can imagine that there may be some cases where the expressiveness of the Acme vocabulary can be limiting. As we discussed earlier, Acme makes it easy to define styles that can be represented in component-connector models. It is possible to represent other types of behavioral, probabilistic and state-based models by overlaying them with an Acme vocabulary, but the more we shift away from structural component-connector models, the expressiveness may get limited to the types of constraints and analyses that can be specified. We could potentially imagine domains where the major emphases are behaviors, state-based or even mathematical models where a component-connector vocabulary would be limited by its expressiveness. Halo may not be a good fit for those domains.

7.8 Limitations to framework reuse and cost effectiveness

Halo is helpful in saving costs and effort towards building end-user architecting environments. Because it's a generic framework, investment to develop generic infrastructure involving adapters and styles may initially be expensive, but costs to build such generic infrastructure is often amortized across many instantiations and would be much lower overall.

There are two types of reuse that are common for most frameworks:

1. First-order reuse of the generic framework components and APIs
2. Second order reuse via customizing artifacts used from one instantiation to next

Halo, like most frameworks, targets to strive a balance between power and generality. For the first order reuse, the architecture layer is extensible through style refinements, and the adapters can be customized by the framework instantiators. The second order reuse depends on the similarity of the instantiation domain to the previous one. For instance, if the styles are similar, it may be possible to reuse the base style. If the computation model and runtimes are similar, it may be possible to reuse the adapters to a greater degree. In our case-studies, we realized that for dataflow based computation models, it was possible to reuse much of the UI, analysis and storage adapters, with minor modifications to execution adapters when the runtime changes from BPEL to SCA. The greater the similarities exists across instantiations, the higher could be the second order reuse, and lower the cost.

There could be some small code-bloat because of using Halo to support this level of generality, but the added benefits of ease of reuse, saved costs from reduced development time, and efficiency derived from integration of analyses and reuse are worth this cost. Halo explicitly lowers this reuse cost by allowing easier extension mechanisms such as support for extensible interfaces, well-defined and flexible APIs, and an architecture layer that can be customized by styles that can be easily refined and specialized.

7.9 Maturity of the domain

Finally, another important factor that could potentially limit the adoption of an end-user architecting framework like Halo is the maturity of the domain itself. In domains that are new, complex or fast evolving, there may not be enough consensus among its domain experts about the architecture style for the domain and its computational constructs. If there is not much agreement to what are the key computational elements, their properties, and the constraints that guide their composition it may be hard for an end-user architect to define architecture styles that guide composition and analysis. Not only is a well-designed composition environment is hard for such domains, but it would be even harder is to reuse any artifacts that were previously designed by one user but may need significant modifications if the design rules or even the vocabulary has since changed. It would be challenging to use Halo in such domains as given the unknowns, the modification and integration costs would be much higher.

7.10 Summary

As we discussed in this Section, while end-user architecting as an approach is quite relevant to many domains where end users need analytic support for their compositions, there could exist domains where the very nature of compositions is free-form drawings without much constraints on how these visual elements must be composed together. For such domains not only it would be hard for an end-user architect to define a vocabulary but it could be hard to reuse any existing composition artefacts that were previously designed. Such unconstrained compositions are beyond the scope for Halo. However, for many mature domains where there is a common pattern of end-user composition, Halo can reduce the overall cost of creating composition environments through framework reuse and cost amortization through capabilities such as support for reuse, analysis and execution that don't need to be built from scratch.

CHAPTER 8

Conclusions and Future Work

In this chapter, we enumerate the thesis contributions and discuss research issues and future work to improve the capabilities of Halo in particular, and end-user architecting in general. We conclude with a summary of this thesis.

8.1 Thesis Contributions

This thesis advances the state-of-art in the field of *end-user software engineering* by providing an approach that supports easier construction and analysis of end-user compositions. We demonstrate that across a large number of domains, end-user composition and analysis is not only similar to architecture composition, but architecture-based tooling can support composition, debugging and execution for end users. Most importantly, all these capabilities can be provided through a framework that end-user composition platform developers can use to build high-quality composition environments at low cost for a wide variety of composition styles and domains.

Specifically, this thesis makes the following contributions:

- **A novel technique** for end-user architecting that reduces the time, cost and difficulty of building a significant class of end-user composition environments. This technique benefits composition environment developers as they can rapidly and incrementally customize composition environments at significantly lower cost than the existing hand-crafted environments.
- **A reusable framework for end-user architecting** that provides interfaces, libraries, controls structures and the necessary plug-in points for developing high quality end-user composition environments.
- **A set of analyses** that improve end-user composition experience. Examples include: ordering analysis, security and privacy analysis, performance analysis, and analyzing composition deployment while considering trade-offs such as performance vs. fidelity.
- **A collection of styles** that can be refined and specialized to model end-user compositions.
- A **qualitative study** that establishes some of the core composition requirements for end-user architecting in a few domains.
- A **demonstration** of the techniques to implement end-user architecting environments, across a few domains.

8.2 Future Work

In this section, we discuss few capabilities that are not currently realized by Halo, but that are logical next steps. We also discuss a few important research issues to further enrich our understanding of end-user architecting and advance the body of knowledge in software engineering. Specifically, we discuss:

- Short-term framework improvements, including tool support
- Medium-term framework improvements,
- Research issues beyond end-user architecting

8.2.1 Short-term Framework Improvements

For future work representing work that could be accomplished in several months, we can make these framework improvements:

- Providing an enhanced library of architecture styles across various domains
- Extending the error declaration model to define the standard error types and error codes
- Optimize the performance for the event processing within the Halo framework
- Extending the framework documentation to include further instantiations across more domains
- Improvements to the types of supported analysis, to include not only the data type and format based analyses but also support various other ad hoc analysis
- Support services that aid different modes of composition (editing, executing, debugging)

Most of the above enhancements would improve the adoption of the Halo framework by composition environment developers and ease of use of the existing APIs. By providing an enhanced library of architecture styles and analyses, the developers and end user architects could further reduce their development effort.

Finally, there are certain assumptions baked into the existing Halo framework, and while the adapters provide a general approach to framework instantiation, there is still scope to further improve the framework by providing generalized error codes, error feedback mechanisms, and support services that developers could integrate for much easier editing, execution and debugging.

8.2.2 Medium-term end-user architecture research issues

For future work from six months to a year, we can address a number of research issues that will enhance the Halo framework.

Supporting greater degree of automated corrections and design assistance and synthesis

Although, many composition frameworks today provide support for data mismatch resolution through special-purpose data converters, end users still have to put significant effort in dealing with data mismatches, e.g., identifying the available converters and determining which of them meet their QoS expectations is a hard problem. In our previous work we addressed this problem by automating the detection and resolution of data mismatches. Specifically, our approach uses architectural abstractions to automatically detect different types of data mismatches, model-generation techniques to fix those mismatches, and utility theory to decide the best fix based on QoS constraints. We applied this technique in the neuroscience domain, where we created a prototype composition environment to detect mismatches. Other tools like Taverna have taken a similar approach where the repositories provide shims to resolve mismatches, but the composition of those along with other components remains a fairly manual approach. We could extend and generalize our automated approach so that such design assistance and synthesis is easily available across multiple domains and it is easier to build tooling for such synthesis.

Improvements to the deployment architecture

Currently, Halo supports a Model-View-Controller style architecture where the environment is hosted locally on a Tomcat server, and where individual components need to be customized and deployed together. Instead of a monolithic application architecture, we could potentially break the deployment into a cloud based architecture where parts of the Halo platform could be available as a service. Parts of the platform could then be individually customized and offer more avenues for integration with web-applications.

8.2.3 Long-term research problems

Next, we discuss some of the long-term research projects in context of end-user architecting.

Systems with architecture + other models

For our current research, end-user architecting primarily focused on architecture models. However, the end-user architecting research can be further extended to domains and tasks that involve other kinds of modelling activities where architecture models can be combined with other types of domain models. Examples include:

- Security: Systems with architecture + other models (e.g., Attacker model) to analyze security properties
- Cyber-Physical systems: Various physical and control system models for domains such as industrial control systems, water systems, robotics systems, smart grid, etc. where there is a need for analysis
- Cloud systems: where various logical models define layers of functionality that can be analyzed for completeness, deployment and other use-cases

For end-user architecting domains we used architecture styles to guide such compositions. For many of these other domains additional models can be combined with architecture styles for analysis. Halo could be extended to support such domain-specific analyses.

Human-in-the-loop systems

Human-in-the-loop is a modelling technique that combines computing and human decision making for machine learning and other domains. In such domains, a human becomes part of the system where human activity represents a component of the overall system architecture. Not only does human activity influence the behaviour of the system, it is key to attaining the overall objectives.

An example of such a human-in-the-loop system for fraud detection would be a scenario where while automated techniques like ML-based algorithms, graph-based analysis and other tools can automate data generation and analysis, the human expert may be needed to interpret the data generated by various algorithms. While algorithms can be trained to continually improve their accuracy over time, the accuracy and bias of automated techniques is not guaranteed, where a human expert can mitigate those gaps by combining the human knowledge and experience with the computational speed of the AI and ML-based algorithms.

An example of this was from our work in resolving data-mismatches, where we defined an automated way for fixing the data mismatch (as shown in Figure 8.1). In our approach, we used (i) architectural descriptions for components and compositions to automatically detect different types of data mismatches in the composition, (ii) model-generation techniques to support the automatic generation of repair alternatives, and (iii) utility theory to automatically check for satisfaction of multiple QoS constraints in repair alternatives.

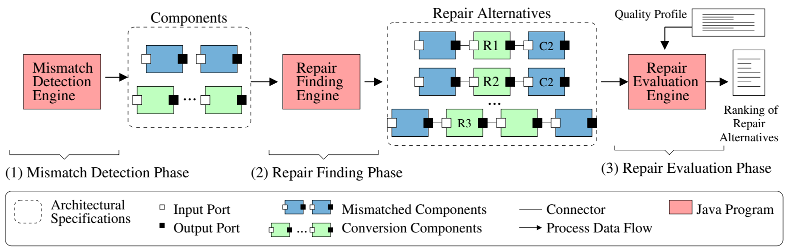


Figure 8.1: Automated Data mismatches with potential human interventions

While humans making end-user architecting decisions is common to compositions, where a human-in-the-loop kind of scenario may emerge is when a human expert may still be needed to define utility functions and various domain-specific tradeoffs that affect the automated selection of the components. For instance, a certain type of brain imaging analysis may only work with a given type of data. Or certain component compositions, while structurally feasible, may be incompatible because of the quality of the output they produce. Human activity in such cases then forms the part of the architecture itself. However, further research is needed to better understand how the end-user architecting approach can be extended in these domains to account for human-in-the-loop and build tooling to support various types of analyses.

Using machine learning for model generation

Further ahead, the end-user architecting approach advocated in this dissertation might benefit from various machine learning techniques for model generation and assistance. Machine learning is a widely used technique today that builds a model based on sample training data in order to make predictions or decisions without explicit programming for each of them. Across various domains, such data analysis can assist in a number of end-user composition tasks, including data pre-processing, component evaluation and automated compositions and fixes based on various properties. The key aspect of machine learning algorithms to perform accurately on new, unseen examples/tasks after having experienced a learning data set can be used for a wide class of end-user composition cases, for example:

- Design-assistance and synthesis based on user-selections

- Automatically identifying quality of service expectations for various components to make decisions about which much be composed together
- Identifying metrics to improve compositions that are customized for individual users

8.3 Summary

This dissertation demonstrates the effectiveness of end-user architecting and the Halo framework to support end-user compositions. We have argued that the computational activities of end users in many domains are analogous to that of software architects, and that rather than forcing end users to become programmers, we should instead provide architecture-based tools and techniques to support their tasks. Halo not only makes the development of composition environments cost-effective across these domains, it provides various analysis capabilities that can be leveraged by composition environment developers to improve the quality of composition environments.

In Chapter 1, we claimed the thesis that *It is possible to build an end-user composition framework that can be instantiated to provide high-quality composition environments at relatively low cost compared to the existing hand-crafted environments for a broad class of composition domains.* This thesis led to claims of generality, cost-effectiveness, and quality assurance through Halo.

In Chapter 2, we surveyed areas of related work and identified contributing disciplines, including end-user software engineering, software architecture, software product lines and DSL, and references to the end-user composition tools where composition activity can be guided by end-user architecting.

In Chapter 3 we described the end-user architecting approach followed by a description of Halo in Chapter 4 and the customization points in Chapter 5. In Chapter 6 and 7 we described how Halo meets the claims around generality, quality assurance and cost-effectiveness. In Chapter 8 we discussed the limitations of Halo and concluded with the future work.

In summary, this thesis fulfills the requirements as follows:

Generality Halo leverages architecture styles to characterize and define customization points to build and define various types of composition environments. It provides extensible adapters that help with the integration of various UI, execution environments and repositories. This provides a generic framework that allows a great degrees of reuse. The evidence for this reuse includes multiple prototype instantiations, which together demonstrate coverage using:

- 2 Computation Styles (Data flow + Publish-Subscribe)
- 2 UI Adapters
- 2 Execution Adapters
- 2 Analysis Adapters
- 2 Repositories

Quality Assurance through Halo In this dissertation, we identified that the end-users primarily used their composition environments for following "6 types" of tasks: (1) Search and explore, (2) Reuse, (3) Construction, (4) Analysis, (5) Execution, and (6) Debugging. Not only does Halo provide support for each of these roles, it improves the quality of the developed environments by supporting all the key end user tasks.

Cost Effectiveness Halo significantly reduces the cost of building end-user composition environments from scratch, through incremental customization of general-purpose components that can be extended by adapters. The evidence for this includes:

- Instantiations in 4 Domains (Arithmetic expressions, Dynamic Network Analysis, Neurosciences, and widget compositions)
- Task-based estimation of engineering effort for building end-architecting environments using Halo

Closing Remarks In this thesis we presented the Halo framework that supports all the key tasks performed by end-users using their composition environments. We presented evidence in support of Halo's generality to architecture styles, cost-effectiveness via reusable artifacts and adapters that allow development composition environments that support analysis, reuse and execution. However, such architecture based composition, analysis, and reuse could be further augmented and more tooling can be built to make this easier for end users across different domains. Further research can help not only to extend this approach but also incorporate other types of models besides architecture-based models.

Bibliography

[AAG93] Gregory D. Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *SIGSOFT FSE*, pages 9–20, 1993.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 187–197. IEEE, 2002.

[AGI98] Robert J. Allen, David Garlan, and James Ivers. Formal modeling and analysis of the hla component integration standard. In *SIGSOFT FSE*, pages 70–79, 1998.

[BCD$^+$03] Mariano Belaunde, Cory Casanave, Desmond DSouza, Keith Duddy, William El Kaim, Alan Kennedy, William Frank, David Frankel, Randall Hauch, Stan Hendryx, et al. Mda guide version 1.0. 1, 2003.

[BCH$^+$95] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models for future software life cycle processes: Cocomo 2.0. *Annals of software engineering*, 1(1):57–94, 1995.

[BCK07] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition*. Addison Wesley, 2007. ISBN 0-201-19930-0.

[BCS$^+$05] Louis Bavoil, Steven P. Callahan, Carlos Eduardo Scheidegger, Huy T. Vo, Patricia Crossno, Cláudio T. Silva, and Juliana Freire. Vistrails: Enabling interactive multiple-view visualizations. In *IEEE Visualization*, page 18, 2005.

[BCW$^+$18] Robert Bocchino, Timothy Canham, Garth Watney, Leonard Reder, and Jeffrey Levison. F prime: an open-source framework for small-scale flight software systems. 2018.

[BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[BGL$^+$09] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *CHI*, pages 1589–1598, 2009.

[BIN] BING - The Portuguese Brain Imaging Network Grid - IEETA. (BING). `http://www.brainimaging.pt`.

[Bio] Biomedical Informatics Research Network. (BIRN). `http://www.birncommunity.org`.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal.

Pattern Oriented Software Architecture: A System of Patterns. John Wiley & Sons, 1996.

[Car06] K.M. Carley. A dynamic network approach to the assessment of terrorist groups and the impact of alternative courses of action. *Visualizing Network Information Meeting, RTO-MP-IST-06, France*, 2006.

[CBB^{+}10] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond, Second Edition*. Addison-Wesley Professional, 2 edition, October 2010.

[Che08] Shang-Wen Cheng. Rainbow: Cost-effective software architecture-based self-adaptation, phd thesis. Technical report, School of Computer Science, Carnegie Mellon University, 2008.

[CKK01] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison Wesley, 2001.

[CN02] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002.

[CP12] K. M. Carley and J. Pfeffer. *Dynamic Network Analysis (DNA) and ORA*. Advances in Design for Cross-Cultural Activities (Part 2), D. D. Schmorrow, D.M. Nicholson (eds), CRC Press, 2012.

[Cyp93] Allen Cypher, editor. *Watch What I Do – Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[Dav95] Alan M Davis. *201 principles of software development*. McGraw-Hill, Inc., 1995.

[DEF^{+}11] Vishal Dwivedi, Perla Velasco Elizondo, José Maria Fernandes, David Garlan, and Bradley R. Schmerl. An architectural approach to end user orchestrations. In *The European Conference on Software Architecture (ECSA)*, pages 370–378, 2011.

[DGPS14] Vishal Dwivedi, David Garlan, Jürgen Pfeffer, and Bradley R. Schmerl. Model-based assistance for making time/fidelity trade-offs in component compositions. In *11th International Conference on Information Technology: New Generations, ITNG 2014, Las Vegas, NV, USA, April 7-9, 2014*, pages 235–240, 2014.

[DHG17] Vishal Dwivedi, James Herbsleb, and David Garlan. What ails end-user composition: A cross-domain qualitative study. In *End-User Development. IS-EUD 2017*, volume 10303 of *Lecture Notes in Computer Science*. Springer, 2017.

[DSS^{+}05] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia C. Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[EDG^{+}13] Perla Velasco Elizondo, Vishal Dwivedi, David Garlan, Bradley R. Schmerl, and José Maria Fernandes. Resolving data mismatches in end-user compositions. In *IS-EUD*, pages 120–136, 2013.

[EGR12] Sebastian Erdweg, Paolo G Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, page 7. ACM, 2012.

[EH13] R. Edwards and J. Holland. *What is Qualitative Interviewing?* The 'What is?' Research Methods Series. Bloomsbury Academic, 2013.

[Eid09] D. Eidelberg. Metabolic brain networks in neurodegenerative disorders: A functional imaging approach. *Trends Neurosci*, 32:548–557, 2009.

[ER03] Albert Endres and H Dieter Rombach. *A handbook of software and systems engineering: Empirical observations, laws, and theories*. Pearson Education, 2003.

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: library-based syntactic language extensibility. In *OOPSLA*, pages 391–406, 2011.

[EV06] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Eclipsecon Summit Europe 2006*, November 2006.

[FMR] FMRIB Software Library. (fsl). http://www.fmrib.ox.ac.uk/fsl/.

[Fow11] Martin J. Fowler. *Domain-Specific Languages*. Addison-Wesley, 2011.

[GCG+14] Daniel Garijo, Oscar Corcho, Yolanda Gil, Meredith N. Braskie, Derrek P. Hibar, Xue Hua, Neda Jahanshad, Paul M. Thompson, and Arthur W. Toga. Workflow reuse in practice: A study of neuroimaging pipeline users. In *10th IEEE International Conference on e-Science, eScience 2014, Sao Paulo, Brazil, October 20-24, 2014*, pages 239–246, 2014.

[GCS+09] David Garlan, Kathleen M. Carley, Bradley Schmerl, Michael Bigrigg, and Orieta Celiku. Using service-oriented architectures for socio-cultural analysis. In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE2009)*, Boston, USA, 1-3 July 2009.

[GDRS12] David Garlan, Vishal Dwivedi, Ivan Ruchkin, and Bradley R. Schmerl. Foundations and tools for end-user architecting. In *Monterey Workshop*, pages 157–182, 2012.

[GMW97a] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research (CASCON'97)*, CASCON '97, pages 169–183. IBM Press, 1997.

[GMW97b] David Garlan, Robert T. Monroe, and David Wile. Acme: an architecture description interchange language. In *CASCON*, page 7, 1997.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[Goo97] Howie Goodell. End-user computing. In *CHI '97 extended abstracts on Human factors in computing systems: looking to the future*, CHI EA '97, pages 132–132, New York, NY, USA, 1997. ACM.

[GRD+07] Yolanda Gil, Varun Ratnakar, Ewa Deelman, Gaurang Mehta, and Jihie Kim. Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. In *AAAI*, pages 1767–1774, 2007.

[GRH+05] B Giardine, C Riemer, R C Hardison, R Burhans, L Elnitski, P Shah, Y Zhang, D Blankenberg, I Albert, J Taylor, W Miller, W J Kent, and A Nekrutenko. Galaxy: a platform for interactive large-scale genome analysis. *Genome Res*, 15(10):1451–1455, October 2005.

[GRS+05] David Garlan, William K. Reinholtz, Bradley Schmerl, Nicholas Sherman, and Tony Tseng. Bridging the gap between systems design and space systems software. In *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop (SEW-29)*, Greenbelt, MD, 6-7 April 2005.

[GS03] Jack Greenfield and Keith 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*, pages 16–27. ACM, 2003.

[GS06] David Garlan and Bradley Schmerl. Architecture-driven modelling and analysis. In Tony Cant, editor, *Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems (SCS'06)*, volume 69 of *Conferences in Research and Practice in Information Technology*, Melbourne, Australia, 2006.

[GSD+11] David Garlan, Bradley Schmerl, Vishal Dwivedi, Aparup Banerjee, Laura Glendenning, Mai Nakayama, and Nina Patel. Swift: A tool for constructing workflows for dynamic network analysis. http://acme.able.cs.cmu.edu/pubs/show.php?id=333, 2011.

[HH11] James Howison and James D. Herbsleb. Scientific software production: incentives and collaboration. In *CSCW*, pages 513–522, 2011.

[HHN85a] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Human–Computer Interaction*, 1(4):311–338, 1985.

[HHN85b] Edwin L Hutchins, James D Hollan, and Donald A Norman. Direct manipulation interfaces. *Human–computer interaction*, 1(4):311–338, 1985.

[HV10] Zef Hemel and Eelco Visser. Pil: A platform independent language for retargetable dsls. In Mark Brand, Dragan Gašević, and Jeff Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 224–243. Springer Berlin Heidelberg, 2010.

[HV12] David B. Hellar and Laurian C. Vega. The Ozone Widget Framework: towards modularity for C2 human interfaces. In *Proceedings of SPIE conference on Defense Transformation and Net-Centric Systems*, Baltimore, Maryland, 2012.

[HWS+06] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole A. Goble, Matthew R. Pocock, Peter Li, and Tom Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web-Server-Issue):729–732, 2006.

[Joh97] G.W. Johnson. *LabVIEW graphical programming: practical applications in instrumentation and control*. McGraw-Hill School Education Group, 1997.

[KAB+11] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan F. Blackwell, Margaret M. Burnett, Martin Erwig, Christopher Scaffidi, Joseph Lawrance, Henry Lieberman, Brad A. Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21, 2011.

[KBK08] J.P. Kelly, A. Botea, and S. Koenig. Offline planning with hierarchical task networks in video games. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, Stanford, CA*, 2008.

[KOE12] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *OOPSLA*, pages 773–792, 2012.

[LAB$^+$06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew B. Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[Lay95] Ronald Laymon. Experimentation and the legitimacy of idealization. *Philosophical Studies: An International Journal for Philosophy in the Analytic Tradition*, 77(2/3):353–375, 1995.

[Let05] Catherine Letondal. Participatory programming: Developing programmable bioinformatics tools for end-users. *H. Lieberman and F. Paterno and V. Wulf, End-User Development*, pages 207–242, 2005.

[Lex] LexisNexis. http://www.lexisnexis.net.

[LNH03] Choonhwa Lee, David Nordstedt, and Sumi Helal. Enabling smart spaces with osgi. *IEEE Pervasive Computing*, 2:89–94, 2003.

[Mat04] Mari Matinlassi. Comparison of software product line architecture design methods: Copa, fast, form, kobra and qada. In *Proceedings of the 26th International Conference on Software Engineering*, pages 127–136. IEEE Computer Society, 2004.

[MCC11] David M. Moore, Portia Crowe, and Robert Cloutier. Driving major change: The balance between methods and people. *SOFTWARE TECHNOLOGY SUPPORT CENTER HILL AFB UT*, 2011.

[MG96] Robert T. Monroe and David Garlan. Style-based reuse for software architectures. In *Proceedings of the Fourth International Conference on Software Reuse*, April 1996.

[MH11] Amber Lynn McConahy and James D. Herbsleb. Platform design strategies: Contrasting case studies of two audio production systems. In *FutureCSD Workshop at CSCW*, 2011.

[MHS13] M.B. Miles, A.M. Huberman, and J. Saldaña. *Qualitative Data Analysis*. SAGE Publications, 2013.

[MKMG97] Robert T. Monroe, Andrew Kompanek, Ralph E. Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43–52, 1997.

[Mon99a] Robert T. Monroe. *Rapid Develpomentof Custom Software Design Environments*. PhD thesis, Carnegie Mellon University, School of Computer Science, July 1999.

[Mon99b] R.T. Monroe. Rapid development of custom software architecture design environments, phd thesis. Technical report, School of Computer Science, Carnegie Mellon University, 1999.

[MPK04] Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *Commun. ACM*, 47(9):47–52, 2004.

[MRT99] Nenad Medvidovic, David S Rosenblum, and Richard N Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st* *international conference on Software engineering*, pages 44–53, 1999.

[MS11] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from event-b models. In *Proceedings of the second symposium on information and communication technology*, pages 179–188, 2011.

[MT97] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *ESEC / SIGSOFT FSE*, pages 60–76, 1997.

[Mye90] Brad A. Myers. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.*, 1(1):97–123, 1990.

[NA00] Oscar Nierstrasz and Franz Achermann. Supporting compositional styles for software evolution. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 14–22. IEEE, 2000.

[Nar93] Bonnie A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT Press, 1993.

[Neu] NeuGRID for you CNRS. N4u - neugrid for you. http://neugrid4you.eu.

[OGA+06] Thomas M. Oinn, R. Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole A. Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip W. Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[Par79] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Software Eng.*, 5(2):128–138, 1979.

[Pek06] J.J. Pekar. A brief introduction to functional MRI. *IEEE Engineering in Medicine and Biology Magazine*, 25(2):24, 2006.

[PKR06] I. Parberry, M.B. Kazemzadeh, and T. Roden. The art and science of game programming. In *ACM SIGCSE Bulletin*, volume 38, pages 510–514. ACM, 2006.

[Pot12] Potomac Fusion. Ozone/Synapse download portal. http://widget.potomacfusion.com/main/home, 2012.

[Pre94] Wolfgang Pree. Meta patterns - A means for capturing the essentials of reusable object-oriented design. In Mario Tokoro and Remo Pareschi, editors, *Object-Oriented Programming, Proceedings of the 8th European Conference, ECOOP '94, Bologna, Italy, July 4-8, 1994*, volume 821 of *Lecture Notes in Computer Science*, pages 150–162. Springer, 1994.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT SOFTWARE ENGINEERING NOTES*, 17:40–52, 1992.

[RMT03] D.E. Rex, J.Q. Ma, and A.W. Toga. The LONI Pipeline Processing Environment. *Neuroimage*, 19:1033–1048, 2003.

[SAG+06] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7), July 2006.

[Sal15] J. Saldana. *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2015.

[SDK$^+$95] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.

[Seg07] Judith Segal. Some problems of professional end user developers. In *VL/HCC*, pages 111–118, 2007.

[SG96] Mary Shaw and David Garlan. *Software architecture - perspectives on an emerging discipline*. Prentice Hall, 1996.

[SG98] Bridget Spitznagel and David Garlan. Architecture-based performance analysis. In *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, pages 146–151, 1998.

[SG03] Bridget Spitznagel and David Garlan. A compositional formalization of connector wrappers. In *The 2003 International Conference on Software Engineering (ICSE'03)*, 2003.

[SGD$^+$11] Bradley R. Schmerl, David Garlan, Vishal Dwivedi, Michael W. Bigrigg, and Kathleen M. Carley. Sorascs: a case study in soa-based platform design for socio-cultural analysis. In *ICSE*, pages 643–652, 2011.

[SGT20] Gabriel Sebastián, Jose A Gallud, and Ricardo Tesoriero. Code generation using model driven architecture: A systematic mapping study. *Journal of Computer Languages*, 56:100935, 2020.

[Sim95] Charles Simonyi. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*, 1995.

[SKK$^+$12] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *ICSE*, pages 167–177, 2012.

[SRK$^+$11] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *SPLC*, pages 160–169, 2011.

[Str06] Stephen C. Strother. Evaluating fMRI preprocessing pipelines. *IEEE Engineering in Medicine and Biology Magazine*, 25(2):27, 2006.

[VAR09] Ferdinando Villa, Ioannis N. Athanasiadis, and Andrea Emilio Rizzoli. Modelling with knowledge: A review of emerging semantic approaches to environmental modelling. *Environmental Modelling and Software*, 24(5):577–587, 2009.

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[VEDG$^+$12] Perla Velasco Elizondo, Vishal Dwivedi, David Garlan, Bradley Schmerl, and Jose Maria Fernandes. Resolving data mismatches in end-user compositions. Submitted for publication, 2012.

[VS10] Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with mps. *Software Language Engineering, SLE*, 2010.

[WD15] Roel Wieringa and Maya Daneva. Six strategies for generalizing software engineering theories. *Science of computer programming*, 101:136–152, 2015.

[Wie14] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[YGS$^+$04] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. DiscoTect: A system for discovering architectures from running systems. In *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, Scotland, 23-28 May 2004.

[ZGM11] Ying Zhao, Shelley P. Gallup, and Douglas J. MacKinnon. Lexical link analysis for the haiti earthquake relief operation using open data sources. In *International Command and Control, Research and Technology Symposium*, Qubec City, Canada, June 21? 2011.

CHAPTER 9

Appendix A: SCORE architectural style refined for Brain imaging compositions

9.1 SCORE Style

```
Family SCORE-Fam = {

    Connector Type DataFlowConnector = {
        Role provider : providerT = new providerT extended with {
        }
        Role consumer : consumerT = new consumerT extended with {
        }
        rule onlyTwoRoles = heuristic  ! size(self.ROLES) > 2;
    }
    Component Type Service = {
        Port provide : provideT = new provideT extended with {
        }
        Port consume : consumeT = new consumeT extended with {
        }
        Port config : configT = new configT extended with {
        }

        Property operationName : SCORE-Fam.OperationName;
        Property location : SCORE-Fam.Location;
        Property function : SCORE-Fam.Function;
        Property owner : SCORE-Fam.Owner;
        Property toolOrigin : SCORE-Fam.ToolOrigin;
    }
    Property Type DataType = string;
    Role Type dataWriterT = {

    }
    Component Type Tool = {
        Port provide : provideT = new provideT extended with {
        }
        Port consume : consumeT = new consumeT extended with {
        }
        Port config : configT = new configT extended with {
        }

        Property operationName : SCORE-Fam.OperationName;
        Property function : SCORE-Fam.Function;
        Property owner : SCORE-Fam.Owner;
    }
    Role Type consumerT = {
    }
    Port Type provideT = {
        Property methodName : string;
        rule allValues = invariant forall p in self.PROPERTIES |
            hasValue(p);
    }
    Property Type OperationName = string;
    Port Type configT = {
        Property configFileName : string;
    }
    Property Type ToolOrigin = Enum {notSpecified ,Automap,Construct ,ORA, Pythia ,Other
        };
    Port Type readT = {
    }
    Connector Type UIDataFlowConnector = {
        Role provider : providerT = new providerT extended with {
        }
        Role consumer : consumerT = new consumerT extended with {
        }

        Property messagingProtocol : SCORE-Fam.MessagingProtocolPropT;
        rule twoRoles = invariant size(self.ROLES) >= 2;
        rule onlyTwoRoles = heuristic  ! size(self.ROLES) > 2;
    }
    Component Type DataStore = {
        Port readData : readT = new readT extended with {
        }
        Port writeData : writeT = new writeT extended with {
        }

        Property datatype : SCORE-Fam.DataType;
        Property location : SCORE-Fam.Location;
    }
    Component Type UIElement = {
        Port provide : provideT = new provideT extended with {
        }
        Port UIconfig : UIconfigT = new UIconfigT extended with {
        }

        Property datatype : SCORE-Fam.DataType;
        Property location : SCORE-Fam.Location;
    }
    Port Type consumeT = {
        Property methodName : string;
        rule allValues = invariant forall p in self.PROPERTIES |
            hasValue(p);
        rule somethingAttached = heuristic size(self.ATTACHEDROLES) >= 1;

    }
    Property Type Owner = string;
    Connector Type DataWriteConnector = {
        Role provider : providerT = new providerT extended with {
        }
        Role dataWriter : dataWriterT = new dataWriterT extended with {
        }
    }
    Property Type Function = string;
    Port Type UIconfigT = {
        Property configFileName : string;
    }
    Property Type Location = Enum {NotSpecified, localNetwork, externalNetwork};
    Connector Type DataReadConnector = {
        Role dataReader : dataReaderT = new dataReaderT extended with {
        }
        Role consumer : consumerT = new consumerT extended with {
        }
    }
    Role Type dataReaderT = {
    }
    Component Type LogicComponent = {
    }
    Role Type providerT = {
    }
    Port Type writeT = {
    }
}
```

9.2 Score style refined for Brain imaging

```
\begin{lstlisting}[language=acme]
import families/SCORE-FAM.acme;
Family BrainImaging extends SCORE-Fam with {
    Property Type referential = Enum {RAW,MNI,ICBM, Talairach ,NotKnown};
    Property Type alignedTo = Enum {NULL,MNI,ICBM, Talairach ,NotKnown};
    Property Type dimension = Enum {Three, Four, NotKnown};
    Property Type ordering = boolean;
    Property Type format = Enum {NIFTI,SPM,DICOM,NotKnown};
    Property Type modality = Enum {FMRI,MRI,NotKnown};
    Property Type cardinality = int;

    Property Type Structure = Record [
        dimension : Enum {Three, Four, NotKnown};
        nesting : boolean;
    ];

    Property Type Conceptual = Record [
        brainActivity : boolean;
        structure : boolean;
    ];


analysis mustBeOfType(e : Element,types : Set {type}) : boolean =
        exists t in types | declaresType(e, t);

analysis mustHaveValue(e : Element, v : string) : boolean =
        hasValue(e);

analysis portsOfType(c : Component,t : type) : Set {} =
       {select p : Port in c.PORTS | declaresType(p, t)};

analysis numberOfPorts(c : Component,t : type) : int =
       size(portsOfType(c, t));

analysis onlyConnectedTo (e : component, t : type) : boolean {
    forall c : component in connectedComponents (e) | declaresType (c,t)
}

analysis matched-Types (p1: Port, p2: Port) : boolean {
/*
    Iterate across port properties and check if p1 is a superset of p2 (or viceversa)
*/
}


analysis connectedComponents (e : element) : Component =
/* Iterate across components (port to connectors to components) and extract all
    connected components
*/
;
Component Type Acquisition extends DataStore with {
        Port readBIVolumes : readT = new readT extended with {
            Property aligned : BrainImaging.alignedTo << default = NULL; >>;
            Property format : BrainImaging.format;
            Property referential : BrainImaging.referential << default = RAW; >>;
            Property dimension : BrainImaging.dimension << default = Four; >>;
            Property ordering : BrainImaging.ordering << default = true; >>;
            Property modality : BrainImaging.modality;
           }
    }
Component Type ReferenceVolume extends DataStore with {
        Port readBIVolumes : readT = new readT extended with {
            Property aligned : BrainImaging.alignedTo << default = MNI; >>;
            Property format : BrainImaging.format;
            Property referential : BrainImaging.referential << default = MNI; >>;
            Property dimension : BrainImaging.dimension;
            Property ordering : BrainImaging.ordering;
```

```
                Property modality : BrainImaging.modality;

                Rule CheckReferential = heuristic self.referential != RAW;

            }
        }
Component Type SaveBIVolume extends DataStore with {
            Port writeBIVolumes : writeT = new writeT extended with {
                Property aligned : BrainImaging.alignedTo << default = MNI; >>;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential << default = MNI; >>;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
}

Component Type Align extends Tool with {
            Port InputBIVolumes : consumeT = new consumeT extended with {
                Property aligned : BrainImaging.alignedTo << default = MNI; >>;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential << default = MNI; >>;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
            Port referenceVolume : consumeT = new consumeT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
            Port OutputBIVolume : provideT = new provideT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
            Port motionCorrectness  : provideT = new provideT extended with {
                Property XCoordinates : string;
                Property YCoordinates : string;
                Property ZCoordinates : string;
            }
        }

Component Type Coregister extends Tool with {
            Port InputBIVolumes : consumeT = new consumeT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
            Port referenceVolume : consumeT = new consumeT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
            Port OutputBIVolume : provideT = new provideT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
        }
    Component Type TemporalAdjustment extends Tool with {
            Port InputBIVolumes : consumeT = new consumeT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
            Port parameters : configT = new configT extended with {

            }
            Port OutputBIVolume : provideT = new provideT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
        }
    Component Type Smooth extends Tool with {
            Port InputBIVolumes : consumeT = new consumeT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
            Port parameters : configT = new configT extended with {

            }
            Port OutputBIVolume : provideT = new provideT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
        }

    Component Type Normalize extends Tool with {
            Port InputBIVolumes : consumeT = new consumeT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
            Port referenceVolume : consumeT = new consumeT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
            Port OutputBIVolume : provideT = new provideT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
            Port TranformationOutput  : provideT = new provideT extended with {
            }
        }

    Component Type SPM-Function extends Tool with {
            Port InputBIVolumes : consumeT = new consumeT extended with {
                Property aligned : BrainImaging.alignedTo;
                Property format : BrainImaging.format;
                Property referential : BrainImaging.referential;
                Property dimension : BrainImaging.dimension;
                Property ordering : BrainImaging.ordering;
                Property modality : BrainImaging.modality;
            }
        Port conditions : consumeT = new consumeT extended with {
            Property conditionsFileName : string;
        }
        Port parameters : configT = new configT extended with {

        }
        Port OutputBIVolume : provideT = new provideT extended with {
            Property aligned : BrainImaging.alignedTo;
            Property format : BrainImaging.format;
            Property referential : BrainImaging.referential;
            Property dimension : BrainImaging.dimension;
            Property ordering : BrainImaging.ordering;
            Property modality : BrainImaging.modality;
        }
        Port Error  : provideT = new provideT extended with {
        }
    }
Connector Type readBIVoumeFromFile extends DataReadConnector with {
        Role consumer  = {
        }
        Role dataReader  = {
        }
    }
Connector Type writeBIVoumetoFile extends DataWriteConnector with {
        Role provider  = {
        }
        Role dataWriter  = {
        }
    }
Connector Type transferBIVolume extends DataFlowConnector with {
        Role provider  = {
        }
        Role consumer  = {
        }
    }


}
```

9.3 FSL Neurosocience tool style

```
import families/BING.acme;
import families/SCORE-FAM.acme;

Family FSL-FAM extends BING, SCORE-Fam with {

    Component Type mcflirt extends Tool with {
        Port provide  = {
        }
        Port consume  = {
        }
        Port config  = {
        }
    }
    Component Type slicer extends Tool with {
        Port provide  = {
        }
        Port consume  = {
        }
        Port config  = {
        }
    }

    Component Type fslstats extends Tool with {
        Port provide  = {
        }
        Port consume  = {
        }
        Port config  = {
        }
    }
    Component Type fslroi extends Tool with {
        Port provide  = {
        }
        Port consume  = {
        }
        Port config  = {
        }
    }
    Component Type fslmaths extends Tool with {
        Port provide  = {
        }
        Port consume  = {
        }
        Port config  = {
        }
    }
    Component Type susan extends Tool with {
...
    }
...
}
```

9.4 Example composition using Brain-imaging style

```
import families/BrainImaging.acme;
import families/FSL.acme;

System BINGWorkflow : BrainImaging = new BrainImaging extended with {

    Component Aquisition0 : Aquisition = new Aquisition extended with {
        Port writeData  = {
            Property data = FMRI;
            Property srcDirectory = "C://data/";
            Property volumeListName = "subjectA";
      }

        Property datatype = "volumeList";
        Property data = FMRI;
    }
    Component Align0 : Align = new Align extended with {
        Port consume  = {
            Property data = FMRI;
      Property methodName = "align()";
   Property volumeListName = "subjectA";
        }
        Port config  = {
            Property configFileName = "configAlign";
        }
        Port provide  = {
            Property data = FMRI;
            Property methodName = "normalize()";
            Property volumeListName = "subjectA";
        }

        Property function = "AlignVolumes";
        Property operationName = "align()";
        Property owner = "NotKnown";
        Property data = FMRI;
    }
    Component Normalization0 : Normalization = new Normalization extended with {
        Port consume  = {
         Property data = FMRI;
         Property methodName = "normalize()";
         Property volumeListName = "subjectA";
    }
        Port provide  = {

            Property data = FMRI;
            Property methodName = "spm()";
            Property volumeListName = "subjectA";
        }
        Port config  = {
            Property configFileName = "NormalizeConfigFile";
        }

        Property function = "normalizeVolume";
        Property operationName = "normalize()";
        Property owner = "NotKnown";
        Property data = FMRI;
    }

    Component SPM0 : SPM = new SPM extended with {
        Port consume  = {
            Property data = FMRI;
            Property methodName = "spm()";
            Property volumeListName = "subjectA";
        }
        Port provide  = {
            Property data = FMRI;
            Property methodName = "visualize()";
            Property volumeListName = "subjectA";
        }

        Property function = "spmAnalysis";
        Property operationName = "spm()";
        Property owner = "NotKnown";
        Property data = FMRI;
    }

    Component Visualization0 : Visualization = new Visualization extended with {
        Port consume  = {
            Property data = FMRI;
            Property methodName = "visualize()";
            Property volumeListName = "subjectA";
        }
        Port provide  = {
            Property data = FMRI;
            Property methodName = "store()";
            Property volumeListName = "subjectA";
        }
        Property function = "VisualizeData";
        Property operationName = "visualize()";
        Property owner = "NotKnown";
        Property data = FMRI;
    }

...

    Connector dataFlow0 : dataFlow = new dataFlow extended with {
    }
    Connector dataFlow1 : dataFlow = new dataFlow extended with {
    }
    Connector dataFlow2 : dataFlow = new dataFlow extended with {
    }
    Connector writeData1 : writeData = new writeData extended with {
  Property encryption = encrypted;
    }
    Attachment Aquisition0.writeData to writeData0.dataWriter;
    Attachment Align0.consume to writeData0.provider;
    Attachment Align0.provide to dataFlow0.provider;
    Attachment Normalization0.consume to dataFlow0.consumer;
    Attachment Normalization0.provide to dataFlow1.consumer;
    Attachment SPM0.consume to dataFlow1.provider;
    Attachment SPM0.provide to dataFlow2.consumer;
    Attachment Visualization0.consume to dataFlow2.provider;
    Attachment Visualization0.provide to writeData1.dataWriter;
    Attachment StoreData0.readData to writeData1.provider;
}
```

Appendix B: Cost estimates for a workflow composition environment (for SORASCS).

Feature Id	Feature Description	Points	Hours
FR01.01	The software shall support basic graphical editing capabilities - add, delete, connect, undo, redo, add text description for services	See break down	Hours
FR01.01.a	Add delete and connect	5	11.68
FR01.01.b	undo/redo	8	19.54
FR01.01.c	Text description is about changing the description of the service on the canvas.	3	6.37
FR01.01.d	Layout auto organize	13	33.13
FR01.01.e	The software shall allow the user to "snap" components on the canvas to a grid.	8	19.54
FR01.02	The software shall support orchestration of a workflow using services (thick and thin). The actual style of orchestration is to connect services' port to port using a single connection.	2	3.39
FR01.03	The software shall allow analysts to “drill down” a service[4/19/2010] workflow which is a nested workflow within the parent workflow.	13	33.13
FR01.04	The software shall show TBD[6] meta-information for each service.	3	6.37
FR01.05	The software shall allow analysts to add/append and delete meta-information for their own workflows.	5	11.68
FR01.06	The software shall provide searching/locating capabilities to organize the services	40	
FR01.06.a	The software shall categorize the services in the palette based on their meta-data.	13	33.13
FR01.06.b	The software shall provide search capabilities to find services on the palette based on their meta-data.	13	33.13
FR01.06.c	The software shall provide a filter capability (by name) on the palette that updates which services are currently displayed.	8	19.54
FR01.06.d	The software shall provide reorganization capabilities for the palette services that change their categorization hierarchy.	8	19.54
FR01.06.e	The software shall provide a framework for plug-ins to provide organization of the services on the palette.		
FR01.07	The software shall support replacing of the services within the workflow with the analogous services if available and if the analyst does not have permission to use existing services within the workflow	100	276.11
FR01.08	Software shall provide TBD[1] repair operations.	40	108.74
FR01.09	The software shall allow an analyst to add workflow repair operation plug-ins.	20	52.67
FR01.10	Software shall performTBD[1] syntactic checks for workflow validation	20	52.67
FR01.11	The software shall provide one TBD[1]example syntactic check	5	11.68
FR01.12	Software shall allow analysts to save the data at every connection or step of the workflow construction.	8	19.54

FR01.13	Software shall allow analysts to save and open a workflow regardless of its syntactic correctness.	3	6.37
FR01.14	Software shall allow analysts to set the value of parameters in services during workflow construction.	20	52.67
FR01.14.a	The software shall use default values for the service parameters and let the user manually set them using a configuration interface.		19.54
FR01.14.b	The software shall allow a service (including a "user interaction" service) to be connected to provide a value for a parameter.		19.54
FR01.14.c	The software shall provide an interface for complicated parameter types (e.g. records) as needed by the SORASCS services.		11.68
FR01.15	Software shall allow analysts to set the value of parameters in connectors during workflow construction.	5	11.68
FR01.16	Software shall allow analyst to show TBD[6]meta-information for each connector.	3	6.37
FR01.17	* Data service configuration (Revisit)	See break down	
FR01.17.a	The software shall retrieve data services from SORASCS.	5	11.68
FR01.17.b	The software shall allow the analyst to upload local data to SORASCS.	8	19.54
FR01.18	The software shall facilitate the transformation of compatible data types. When the user tries to connect ports with different but compatible data types, the tool shall provide a transformation solution if it exists on SORASCS.	20	52.67
FR02.01	The software shall I/F with SORASCS for TBD[2] execution modes	See break down	
FR02.01.a	End to end, this also include defining the UI perspective (change from composition to execution)	8	19.54
FR02.01.b	Debug (breakpoints)	20	52.67
FR02.01.c	Step by steps execution	3	6.37
FR02.03	The software shall support execution of both thick and thin services	See break down	
FR02.03.a	The software shall support execution of thin clients.	1	1.06
FR02.03.b	The software shall support execution of thick clients through the SORASCS Client interface.	8	19.54
FR02.03.c	The software shall support execution of a "user interaction" dialog service that can be connected to parameters or ports to provide their value during execution-time of the workflow.	13	33.13
FR02.04	The software shall I/F with SORASCS to display TBD[2] the execution progress of the workflow. graphically on the canvas.	See break down	
FR02.04.a	The software shall display the execution progress of the workflow. (UI)	13	33.13
FR02.04.b	The software shall interface with SORASCS to retrieve the progress information of an executing workflow. (Backend)	13	33.13
FR02.05	Before workflow can be executed it must first be deployed. The software shall deploy the workflow automatically if the analyst does not explicitly.	40	108.74
FR02.07	The software shall allow the user to view intermediate results of a workflow during execution.	13	33.13

FR02.08	The software shall allow the user to stop a workflow's execution while it is in progress.	5	11.68
FR02.06	The software shall allow analyst to “drill down” a service which is a nested workflow within the parent workflow.		
FR03.01	The software shall I/F with SORASCS to play back an already executed workflow on demand.	13	33.13
FR03.02	The software shall I/F with SORASCS to locate and display the executed workflows and the data associated with those execution histories.	5	11.68
FR03.03	The software shall access those services the analyst used through SORASCS in the past (the history service), and be able to import them onto the construction canvas as a new workflow.	13	33.13
FR04.01	The Software shall I/F with SORASCS to provide TBD[3]workflow analysis.	See break down	
FR04.01a	The software shall allow an Analyst to add additional semantic validation plug-ins.	40	108.74
FR04.01b	The software shall provide one TBD[1] example semantic validation.	5	11.68
FR05.01	The software shall allow analysts to package workflow to use it as a new service.	20	52.67
FR05.02	The software shall allow analysts to reuse their workflows. Deploying the parent workflow will also recursively deploy child workflows.	13	33.13
FR05.03	The software shall allow the analyst to do a “save as” option for saving the workflow.	2	63.39
F6R05.04	Workflow-to-Service Traceability	13	33.13
FR07.01	The software shall list the available services and workflows to the analyst.	8	19.54
FR07.05	The software shall validate user login through the I/F with SORASCS server. Actual authentication mechanism will be provided by SORASCS.	See break down	
FR07.05.a	Includes the user representation (classes) The software shall provide a class facade interface to SORASCS's authentication system.	8	19.54
FR07.05.b	The actual login The software shall authenticate the users through a login interface that uses the SORASCS authentication mechanism.	5	11.68
FR08.01	Error handling		
FR08.01a	The software shall provide an error framework to handle errors consistently application-wide.	3	6.37
FR08.01b	The software shall interface with SORASCS to display error results if the workflow execution fails.	2	3.39
FR08.01c	The software shall display an error message if a connection to SORASCS is required but not present. Operations that do not require a SORASCS connection should continue.		6.37

		Plan	Code							
	Release		1		2				3	
	Week	1	2	3	4	5	6	7	8	9
	Start date	17-May	24-May	31-May	7-Jun	14-Jun	21-Jun	28-Jun	5-Jul	12-Jul
FR01.01	Basic Graphical Editing Capabilities									
FR01.01a	Drag and Drop		30%		70%					
FR01.01b	Undo/Redo						10%		90%	
FR01.01c	Add Service Description								100%	
FR01.01d	Canvas Zooming				100%					
	Canvas Auto-Layout									
	Grid Snapping									
FR01.02	Workflow Orchestration		100%							
FR01.03	Construction Drilldown				100%					
FR01.04	Show Service Meta-Info		100%							
FR01.05	Edit Workflow Meta-Info		40%		60%					
FR01.06	Palette Capabilities									
	Palette Categorization		60%		40%					
FR01.06b	Palette Search				60%				40%	
FR01.06c	Palette Filtering		80%				20%			
FR01.06d	Palette Reorganization						60%		40%	
FR01.06e	Palette Plugin								100%	
	Analogous Services									
	Repair Operations									
	Repair Plugins									
FR01.10	Syntactic Checking		50%		50%					
FR01.11	Syntactic Check Example		100%							
FR01.12	Persistent Intermediate Results				40%				60%	
FR01.13	Save/Open Workflow						100%			
FR01.14	Service Parameterization									
FR01.14a	Set Manual Value		50%		50%					
FR01.14b	Connect Parameters						80%		20%	
FR01.14c	Complicated Parameters								100%	
	Connector Parameterization									
	Show Connector Meta-Info									
FR01.17	Data Service Configuration									

	SORASCS Data		30%				5%		65%	
FR01.17b	Local Data						30%		70%	
FR01.17c	View Data									
	Data transformation									
FR01.19	Project Management									
FR01.20	Multiply-Typed Ports								100%	
FR02.01	Workflow Execution									
FR02.01a	End-to-End Execution		90%		10%					
	Breakpoint Execution						60%		40%	
	Step-by-Step Execution								100%	
FR02.03	Service Execution									
FR02.03a	Thin Services		100%							
FR02.03b	Thick Services		70%		30%					
FR02.03c	UI Interaction Services								100%	
FR02.04	Execution Progress									
FR02.04a	Execution Progress UI		93%		7%					
	Execution Progress Backend		60%		40%					
	Workflow Deployment		30%		50%				20%	
FR02.07	View Intermediate Results				100%					
FR02.08	Stop execution								100%	
FR03.01	History Playback								100%	
FR03.02	Inspect History								100%	
	Implicit Construction									
FR04.01	Semantic Analysis									
FR04.01a	Semantic Analysis Plugin				30%				60%	
FR04.01b	Semantic Analysis Example						100%			
	Workflow Packaging		50%		50%					
	Workflow Reuse		40%				60%			
FR05.03	Workflow Save As						20%		80%	
FR05.04	Workflow-to-Service Traceability						100%			
FR07.01	List Services									
FR07.05	User Login									
FR07.05a	Authentication Infrastructure		30%		70%					
	SORASCS Authentication								0%	
FR08.01	Error Handling									
FR08.01a	Error Framework		100%							

FR08.01b	Execution Errors						100%			
FR08.01c	SORASCS Connection Error		100%							
QA01	Easy to Learn									
	Offer Suggestions									
QA03	Easy Re-Execution									
QA05	Local SORASCS									
QA06	Easy to Install		100%							
QA07	Sensible Organization									
QA09	Analysis Extension Framework									
QA10	Fast Load									
QA11	Palette Extension Framework						60%		40%	
QA12	Unavailability Feedback									
QA13	Easy Packaging and Sharing									
	Secured Communication									
QA15	Backend Interaction									
	Workflow Language Modification									
QA17	Non-Blocking execution						66%		34%	

Project Stage	**Date**
SRS Scope Version	6-Jun
Release1	6-Jun
Release2	4-Jul
Release3	18-Jun
Delivery Release	2-Aug