An Architectural Approach to End User Orchestrations

Vishal Dwivedi[1], Perla Velasco-Elizondo[2], Jose Maria Fernandes[3] David Garlan[1] and Bradley Schmerl[1]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213, USA.
[2] Centre for Mathematical Research (CIMAT), Zacatecas, ZAC, 98060, Mexico.
[3] IEETA/DETI & Uni. of Aveiro, Campus Universitrio de Santiago, 3810-193 Aveiro, Portugal.

Abstract. Computations are pervasive across many domains, where end users have to compose various heterogeneous computational entities to perform professional activities. Service-Oriented Architecture (SOA) is a widely used mechanism that can support such forms of compositions as it allows heterogeneous systems to be wrapped as services that can then be combined with each other. However, current SOA orchestration languages require writing scripts that are typically too low-level for end users to write, being targeted at professional programmers and business analysts. To address this problem, this paper proposes a composition approach based on an end user specification style called SCORE. SCORE is an architectural style that uses high-level constructs that can be tailored for different domains and automatically translated into executable constructs by tool support. We demonstrate the use of SCORE in two domains - dynamic network analysis and neuroscience, where users are intelligence analysts and neuroscientists respectively, who use the architectural style based vocabulary in SCORE as a basis of their domain-specific compositions that can be formally analyzed.

1 Introduction

Professionals in domains such as scientific computing, social-sciences, astronomy, neurosciences, and health-care are increasingly expected to compose heterogeneous computational entities to perform and automate their professional activities. Unlike professional programmers, these end users write programs to support the goals of their domains, where programming is a means to an end, not the primary goal [7]. However, studies have shown that such users spend about 40% of their time on programming activities [5], meaning that a large community of people are spending a lot of their time on programming tasks rather than on tasks directly related to their domain.

While in some cases end users may find it sufficient to use a single tool to accomplish their goals, very often one single tool may not provide all functionalities. Hence, the end users must compose functions from a number of tools, libraries, and APIs. To define such compositions, they need to either write glue code in the form of executable scripts, or use special-purpose tools that provide GUIs that generate such code, both of which require significant technical knowledge that they often lack.

Today, there is a large variety of approaches to support the composition of computational elements; however, they can be classified into two main categories: i) code scripts, and ii) orchestrations. However, neither of these fit naturally to the end users' needs. For instance, a typical code script for neuroscience workflows (as shown in Fig. 1a) requires writing program calls to describe analyses. This not only requires knowledge of the scripting language, but also other technical details, e.g. the parameters used

by each program call. Orchestration languages such as BPEL (as shown in Fig. 1b) offer an improvement over scripts by providing higher level constructs (e.g., services as opposed to command-line parameters). However, they too have a low level of abstraction, and are still close to program code. For instance, such BPEL scripts require specification of control logic (e.g. Sequence, While), data assignment (i.e. Assign) and error handling constructs (i.e. Throw). As can be seen, both approaches are too low level for technically naïve end users and therefore tedious and error-prone. For both

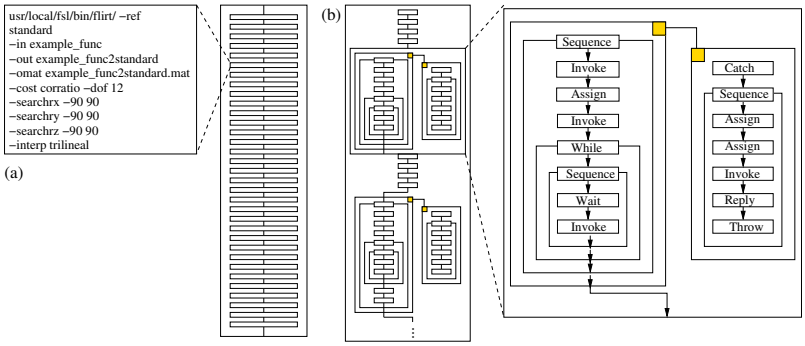

Fig. 1. Common modes of composition: (a) code scripts and (b) orchestrations.

these cases, detection of syntactic and semantic issues has to be performed manually. Although, at times GUIs and type-checkers aid syntactic verification, finding semantic issues that are more domain-specific is difficult because specifications written in terms of low-level code constructs are not convenient for describing semantic information. Additionally, the analysis of other relevant properties such as performance or deadlock is even harder to support on script code. This often leads to technically-nave end users resorting to opportunistic programming and copy-paste, wherein they make frequent mistakes [2]. In either case, creating compositions is difficult for end users because of:

- Complexity due to low-level details: Existing languages and tools require end users to have knowledge of a myriad of low-level technical detail such as parameters, file systems, paths, operating systems, etc.

- Lack of support for error resolution: Few mechanisms exist today for helping users detect syntactic and semantic problems with their compositions. Further, identifying and fixing quality attribute problems (such as security and privacy issues) in their specifications is difficult for end users.

- Conceptual mismatch: End users often think in terms of tasks they want to accomplish, while current composition mechanisms force them to think in terms of technology with which the task is implemented. For example, “Remove Image Noise” as opposed to calling the specific program(s) to perform this function.

We believe an architectural specification can alleviate the above problems by providing domain-specific abstractions that are close to the way that end users think about their problems, but that can still be mapped to code that can be executed on traditional platforms such as SOAs. In this paper, we propose how this can be achieved using architectural styles [14] that provide an abstract vocabulary of components (and the constraints that direct their usage) that can be used by end users to design compositions.

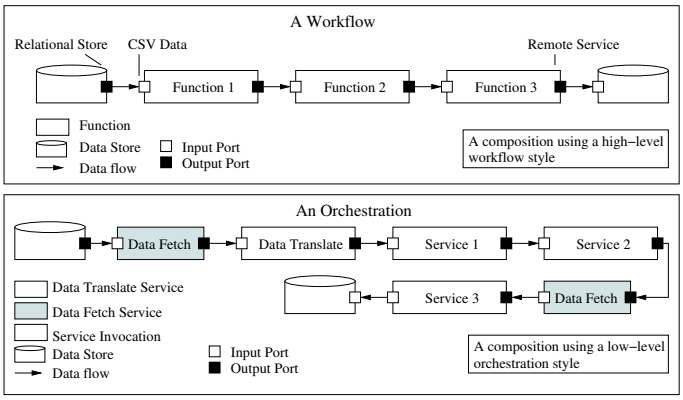

Fig. 2. An illustration of a mapping between a workflow to an orchestration style.

2 Design Approach

We propose a dataflow-based architectural style called SCORE (Simple Compositional ORchestration for End users) that can be used for assembling computations in various domains. SCORE provides a vocabulary that can be tailored for different domains and does not require writing low-level code. Instead of using directly executable scripts, we propose using multi-layered styles for representing workflows, where each layer handles different concerns. 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. The end users can specify their compositions in terms of an assembly of high-level functions. These functions can be translated into lower-level orchestrations using tool support.

2.1 Using Architectural Styles as a basis for Abstraction and Refinement

Software architecture provides the high-level structure of a system, consisting of components, connectors, and properties [14]. 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 [14] 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 [1] 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. We find this characteristic of Acme useful for many of the problems that we discussed in Section 1. Specifically, for mapping a functional concept to its technical solution, styles that determine functional vocabulary can be inherited and refined to the styles that consist of components and implement them.

For example, a high-level workflow, as in Fig. 2, can be mapped to a low-level orchestration if both of these are specified using architectural styles that follow inheritance

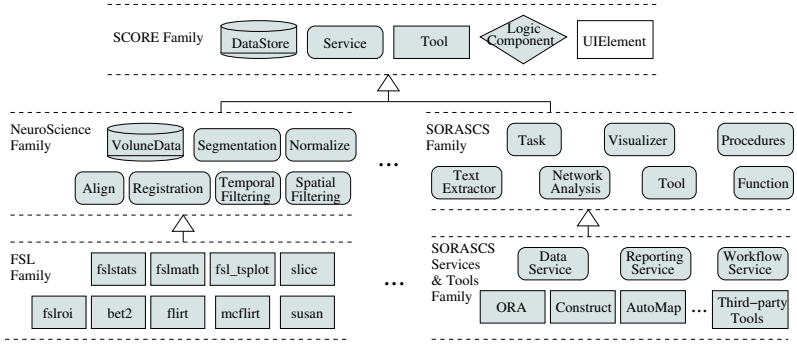

Fig. 3. Style derivation by inheritance.

relationships. The workflow in Fig. 2 composes three functions with different input and output data requirements and location constraints; its corresponding low-level orchestration includes services for individual functions and additional components for data translation and data fetching to compose a sound service orchestration. Note that this is not a 1-to-1 mapping between components, but it is derived from rules. For instance, the port properties of components DataStore and Function 1 in the workflow can point to a difference in data-type and location, leading to insertion of two components that can address the mismatch.

Although simple, this example gives a glimpse of how abstract models can be helpful to end users by providing just the necessary details allowing for a simpler end user specification. These abstract models can be translated into an executable specification using additional properties and constraints. SCORE is based on this approach, where end users can use a high-level style to compose functions that can be compiled into low-level orchestration. We call this functional composition an 'end user orchestration'.

3 SCORE

SCORE is an architectural style that provides a restricted vocabulary for the specification of workflows in a dataflow like specification. It abstracts the specification of workflows to the essential types and the properties of concern that match the computation model required by (end user) scientific communities. The SCORE style specifies rules that are evaluated at design time, enforcing restrictions on the kinds of components users can compose. Writing these rules involves some degree of technical expertise, but these are associated with the architectural style, which is written once by a designer, and then used by end users for modeling workflows based on the style.

3.1 SCORE Vocabulary

Table 1 shows SCORE architectural types, functions and constraints that are used to specify workflows using SCORE. These constrants 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. An important role of the SCORE style description is to define the meaning of semantic constructs in terms of the syntactic properties of the style elements. We achieve this, at least to a certain extent, by enforcing domain-specific constraints. These not only prohibit end users from creating inappropriate service compositions, but also promote soundness by ensuring feedback mechanism via marking errors when a component fails to satisfy any such constraints.

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 1. SCORE composition elements.

Properties and constraints on architectural elements can be used to analyze systems defined using SCORE. Table 2 displays some examples of analyses that are built using SCORE properties, such as analyzing a workflow for structural soundness, and various domain-specific analyses based on workflow properties. Some of the examples of such analyses written in Acme ADL are presented in [4]. The rules for these analyses are written as predicates that are analyzed for correctness while end users design workflows.

4 SCORE in Practice

As shown in Fig. 3, SCORE can be specialized to various domains through refinement and inheritance. This requires construction of sub-styles that extend the basic SCORE dataflow style by adding additional properties, domain-specific constraints, and rules that allow the correct construction of workflows within that domain. For our initial prototype we have defined sub-styles for a couple of domains - neuroscience and network analysis, that we use for modeling workflows in these two domains. For the neuroscience domain, we experimented with using SCORE for defining workflows that can

	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 2. Types of analyses

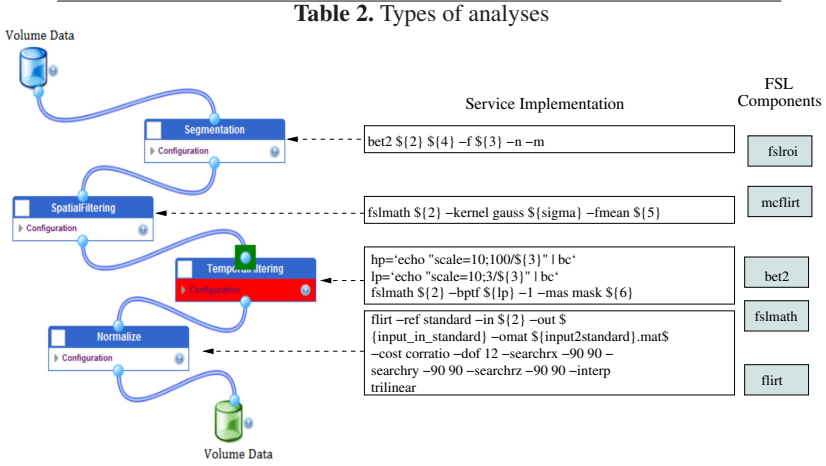

Fig. 4. A pre-processing workflow with an ordering problem.

automate FMRI [4] data pre-processing steps for which neuroscientists currently write detailed code-scripts (as shown in Fig. 1a), replacing them with a tool-assisted workflow (shown in Fig. 4) that is based on SCORE type system. SCORE provides the basic functional vocabulary for constructing workflows, while the low-level styles extend this dataflow-based vocabulary to include additional details about how tools like FSL [5] execute these high-level functions. Thus, not only does SCORE help to define neuroscience workflows at a functional level, it supports analysis such as checking for ordering, and security based on various domain-specific constraints. Fig. 4 for instance, gives an example where one such analysis has gone wrong because of the inappropriate ordering of services in the defined workflow.

Similarly, SCORE was also used to model workflows for dynamic network analysis - a domain that involves creating network models from unstructured data, and then use those models to gain insight about social phenomena through analysis and simulation. This was primarily used for our large SOA based platform named SORASCS [13] that provides an end user friendly SOA based platform to analysts to combine services from various tools in the intelligence analysis domain.

5 Related Work

SCORE can be characterized as providing an abstract vocabulary for composing computations, which can be analyzed for both syntactic and semantic errors, and reduces

[4] FMRI (functional magnetic resonance imaging) is a neuroimaging technique in the neuroscience domain to understand the behavior of the human brain.

[5] The FSL brain imaging tool-suite: www.fmrib.ox.ac.uk/fsl

the conceptual mismatch between end user's functional vocabulary and low-level code constructs (required by current composition mechanisms). We use this characterization to compare SCORE with the related work.

Abstraction: UML-based languages like BPMN have been widely used for documenting abstract compositions. However, their primary use-case has been documentation and not execution. They do not support analysis, and when used to capture details tend to be extremely complicated [10]. There have been other efforts such as SAS language [3] for modeling functional and QoS requirements by Esfahani et al at, and MDA based approaches [9] for composition using SOA profiles. However, such ontologies and profiles don't scale and lack the capability to be extended across different domains. SCORE in comparison, supports functional composition that can be refined, and compiled to low level specifications enabling an easier composition.

Error resolution: Most of the current composition languages provide type-checkers for syntactic verification, but they lack capabilities to resolve domain-specific errors. Almost all such composition languages have a relatively fixed schema that don't allow adding additional attributes that can be useful for expressing domain-specific constraints. In particular, the focus of most of these approaches has been to analyze soundness [11], concurrency [8] or control flow errors [15]. In comparison, SCORE provides support for adding properties and constraints, allowing designers to write domain-specific analyses that other languages cannot support.

Conceptual mismatch: Domain-specific compositions have been used for various scientific dataflow languages such as SCUFL in Taverna [6], and LONI Pipeline [12] for neurosciences. However, most of these approaches have a fixed type system that cannot be extended or refined as we do in SCORE. One of the benefits of such refinement is that the same set of high-level styles can be extended to other domains - for instance, in our case dynamic network analysis, and neurosciences share a common model of computation, but have no similarity in terms of design concepts.

6 Conclusions and Future Work

In this paper we proposed an architectural-style based approach for service composition using an end user specification style called SCORE. The goal of SCORE is to address the requirements of end users who are primarily concerned with composition of computational elements and the analysis of the resulting compositions, but have limited technical expertise to write detailed code.

The tool support constructed using SCORE component types, allows the visual composition of tools, services and data that can be executed by a run-time platform. Although, style-based composition helps to constrain the usage of the component types, it is still a challenge to design an optimal type system for a domain; however, such an upfront investment by style designers could be helpful for end users who can use such a family of component types in their tools. As a future work, we would like to extend SCORE to other domains, and support new types of analyses. We are also working on the problem of mismatch repair given the domain-specific constraints. These would require generating alternative compositions based on the constraints of the styles.

7 Acknowledgments

This work was supported in part by the Office of Naval Research (ONR-N000140811223), and the FCT Portuguese Science and Technology Agency (under the CMU-Portugal faculty exchange

program). Additional support was provided by the Center for Computational Analysis of Social and Organizational Systems (CASOS). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Office of Naval Research, or the U.S. government.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6:213–249, 1997.
2. J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, and S.R. Klemmer. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proc. of the 27th Int. Conf. on Human Factors in Computing Systems (CHI)*, pages 1589–1598, 2009.
3. N. Esfahani, S. Malek, J.P. Sousa, H. Gomaa, and D.A. Menascé. A modeling language for activity-oriented composition of service-oriented software systems. In *Proc. of the 12th Int. Conf. on Model Driven Engineering Languages and Systems*, pages 591–605, Berlin, Heidelberg, 2009. Springer Verlag.
4. D. Garlan and B. Schmerl. Architecture-driven modelling and analysis. In *Proc. of the 11th Australian Workshop on Safety Critical Systems and Software (SCS)*, pages 3–17, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
5. J. Howison and J.D. Herbsleb. Scientific software production: Incentives and collaboration. In *Proc. of ACM CSCW*, pages 513–522, March, 2011.
6. Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R. Pocock, Peter Li, and Tom Oinn. Taverna: A tool for building and running workflows of services. *Nucleic Acids Research*, 34 (Web Server Issue):W729–W732, 2006.
7. A.J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M.B. Rosson, G. Rothemel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43:21:1–21:44, April 2011.
8. M. Koshkina and F. van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *SIGSOFT Software. Engineering Notes*, 29:1–10, September 2004.
9. P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-driven service orchestration. In *Proc. of the 2th Int. IEEE Enterprise Distributed Object Computing Conference*, pages 203–212. IEEE Computer Society, 2008.
10. Harold Ossher, Rachel K. E. Bellamy, Ian Simmonds, David Amid, Ateret Anaby-Tavor, Matthew Callery, Michael Desmond, Jacqueline de Vries, Amit Fisher, and Sophia Krasikov. Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges. In *OOPSLA*, pages 848–864, 2010.
11. F. Puhlmann and M. Weske. M.: interaction soundness for service orchestrations. In *Proc. of the Int. Conf. on Service-Oriented Computing*, volume 4294 of *LNCS*, pages 302–313. Springer Verlag, 2006.
12. D.E. Rex, J.Q. Ma, and A.W. Toga. The loni pipeline processing environment. *Neuroimage*, 19:1033–1048, 2003.
13. B. Schmerl, D. Garlan, V. Dwivedi, M. Bigrigg, and K.M.Carley. SORASCS: A case study in SOA-based platform design for socio-cultural analysis. In *Proc. of the 33rd Int. Conf. on Software Engineering, (ICSE)*, pages 643–652, 2011.
14. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
15. W.M. P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In *Business Process Management*, volume 1806 of *LNCS*, pages 161–183. Springer Verlag, 2000.