Resolving Data Mismatches in End-User Compositions

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

[1] Autonomous University of Zacatecas, Zacatecas, ZAC, 98000, Mexico.
pvelasco@uaz.edu.mx
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213, USA.
{vdwivedi,garlan,schmerl}@cs.cmu.edu
[3] IEETA/DETI, University of Aveiro, 3810-193 Aveiro, Portugal.
jfernan@ua.pt

Abstract. 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 in dealing with data mismatches, e.g., identifying the available converters and determining which of them meet their QoS expectations. In this paper we present an approach that eliminates this effort by automating the detection and resolution of data mismatches. Specifically, it 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 illustrate our approach in the neuroscience domain where data-mismatches can be fixed in an efficient manner on the order of few seconds.

1 Introduction

Computations are pervasive across many domains today, where end users have to compose heterogeneous computational entities to perform and automate their professional tasks. Unlike professional programmers, these end users have to write compositions to support the goals of their domains, where programming is a means to an end, but not their primary expertise [10]. Such end users, often form large communities that are spread across various domains, e.g., Bioinformatics [23], Intelligence Analysis [26] or Neurosciences.[4] End users in these communities often compose computational entities to automate their tasks and *in silico*[5] experiments. This requires them to work within their domain-specific styles of construction, following the constraints of their domain [8]. They often treat their computations and tools as black boxes, that can be reused across various tasks. Developers in these domains have been using approaches based on Service-Oriented Architecture (SOA) [9] to enable rapid composition of computations from third-party tools, APIs and services. There exist large repositories of reusable services such as BioCatalogue, BIRN and INCF,[6] and supporting domain-specific environments to compose them, e.g., Taverna [11] and LONI Pipeline.[7]

[4] http://neugrid4you.eu
[5] Tasks performed on computer or via computer simulation.
[6] www.biocatalogue.org, www.birncommunity.org and www.incf.org
[7] pipeline.loni.ucla.edu

Table 1: Common types of data mismatches.

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.

However, despite the popularity of such composition environments and repositories, the growing number of heterogeneous services makes composition hard for end users across these domains. Often end users have to compose computational entities that have conflicting assumptions about the data interchanged among them (as shown in Table 1).[8] That is, it is common for their inputs and outputs to be incompatible with those of the other computational entities with which they must be composed. This claim is supported by recent studies that have shown that about 30% of the services in scientific workflows are data conversion services [28]. Some composition frameworks today provide data mismatch detection facilities and special-purpose data converters that can be inserted at the point of the mismatch. In spite of this, data mismatch detection and resolution continues to be time-consuming and error-prone for the following reasons: (*a*) most current composition environments detect only type mismatches, while other mismatches are often undetected (e.g., format, content, structural, and conceptual), (*b*) due to the prevalence of converters in repositories such as BioCatalogue or BIRN, end users frequently have several converters to select from, often manually, (*c*) instead of a single converter, a solution might involve a combination of converters. This results in a combinatorial explosion of possibilities, and (*c*) among several repair alternatives, end users need to choose the best one with respect to multiple QoS concerns, e.g., accuracy, data loss, distortion. Today, this assessment is done by "trial and error," a time-consuming process often leading to non-optimal solutions.

The key contribution of this work is an approach that automates the detection and resolution of data mismatches, thus reducing the burden to end users. Specifically, our approach uses: (i) *architectural abstractions* to automatically detect different types of data mismatches, (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. We demonstrate the efficiency and cost-effectiveness of the approach for workflow composition in the neuroscience domain. The remainder of this paper is organized as follows. In Section 2 we introduce the

[8] We studied the literature in data mismatches and organized them in common types. However, this should not be considered as a complete list.

background and related work. In Section 3 we describe the proposed approach and in Section 4 we demonstrate it in practice via an example. In Section 5 we present a discussion and evaluation of the approach. Finally, in Section 6, we discuss the conclusions and future work.

2 Background and Related Work

Garlan et al. [14] introduced the term *architectural mismatch* to refer to conflicting assumptions made by an architectural element about different aspects of the system it is to be a part of, including those about data. Regarding data-related aspects, there is work focused on: (a) *categorizing and detecting (architectural) data mismatches* and (b) *automatically resolving them*. In this section, we relate our work to other literature in these two categories.

Categorizing and detecting data mismatches. There have been numerous efforts in the categorization and formal definition of data mismatches. Cámara et al. [5] defined the term "data mismatch" while in [3] Bhuta and Boehm defined "signature mismatch"; both mismatches highlight the differences that occur among two service components' interfaces with respect to the type and format of their input and output parameters. Similarly, Grenchanik et al. [19] defined "message data model mismatch" to describe differences in the format of the messages to be interchanged among components. Mismatch 42 in [13] refers to "sharing or transferring data with differing underlying representations." Previously, Belhajjame et. al. [2], Bhuta and Boehm [3] and Li et. al. [24] described mismatches for service compositions. Our data-mismatch resolution approach extends these previous efforts on categorizing data mismatches and formalizes them as rules to detect them amongst architectural components. In particular, we: (*i*) identify a set of relevant classes of data mismatches as constraint failures, (*ii*) use this error information to characterize the mismatches in an architectural style, (*iii*) build specific analyses to support the detection of the identified mismatches, and (*iv*) have constructed a prototype tool to detect them during system composition. In contrast to these works, we can detect more specialized data mismatches such as the ones shown in Table 1 using an architectural approach that is more suitable for automated formal analysis.

Resolving data mismatches. There exists some literature that addresses data-mismatch through automatic resolution approaches. The common approach across this work has been to use adapters, which are components that can be plugged between the mismatched components to convert the data inputs and outputs as necessary. Kongdenfha et al. [22] and Bowers and Ludascher [4] used adapters to convert among formats and internal structures of services' data. Several end-user composition environments today also use adapters for data mismatch resolution. For example, Taverna introduces *shims* that could implement data conversion services[9]. Similarly, LONI Pipeline provides the notion of *smartlines* [25] that encapsulate data conversion tools that resolve data format compatibility issues during workflow composition. However, unlike our approach, these works primarily focus on the automatic generation of adapters rather than on the selection and composition of existing ones. Besides that, these approaches work only

[9] www.taverna.org.uk/introduction/services-in-taverna/

for specific data types and formats (e.g., XML) and do not provide support for handling QoS concerns of end users to drive the selection of converters. Even when some environments provide selection support, they do not consider the scenario of having multiple adapters to choose from to solve the same data mismatch.

In the following sections, we describe how our approach addresses the shortcomings in the above discussed works.

3 Approach

As depicted in Figure 1, the approach presented in this paper is comprised of three main phases: (Data) Mismatch Detection, (Data) Mismatch Repair Finding and (Data) Mismatch Repair Evaluation. These three phases use (i) *architectural descriptions* for components and compositions to automatically detect different types of data mismatches, (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.

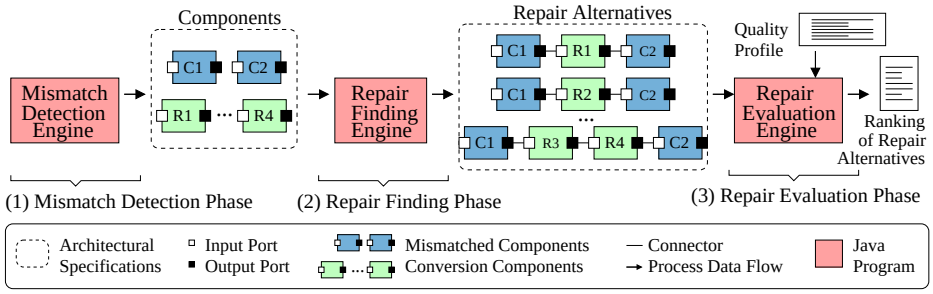


Fig. 1: The three main phases of the approach to data mismatch detection and resolution.

Note that it is not the end users who create such architectural descriptions; such descriptions already exist and are created by component developers and domain experts through means like SCORE [8] and SCUFL (from Taverna) [11]. We build on our previous work on the SCORE architectural style, which provides a *generic modeling vocabulary* for the specification of data-flow oriented workflows that comprises the following elements: *component types* –which represent the primary computational elements, *connector types* –which represent interactions among components, *properties* –which represent semantic information about the components and connectors, and *constraints* –which represent restrictions on the definition and usage of components or connectors, e.g., allowable values of properties and topological restrictions.

SCORE can be specialized to various domains through *refinement* and *inheritance*. This requires style designers and domain experts to construct substyles that extend the SCORE style and add properties and domain-specific constraints that allow end users to correctly construct workflows within that domain. In the example presented in this paper we use the FSL (Sub)Style, which includes components, properties, and constraints specific to neuroscience compositions. Figure 2 illustrates the specialization of some of SCORE's components types (i.e., Data Store, Service and User Interface) for the neuroscience domain via inheritance. The FSL (Sub)Style, shown on the left-hand side

of the figure, includes specializations of service components that provide the functionality of some of the tools offered by the FSL neuroscience suite.[10] In previous work we have also demonstrated the refinement of SCORE for the dynamic network analysis domain [8]. Figure 2 shows some of the components in the resulting substyles, i.e., Dynamic Network Analysis and SORASCS.

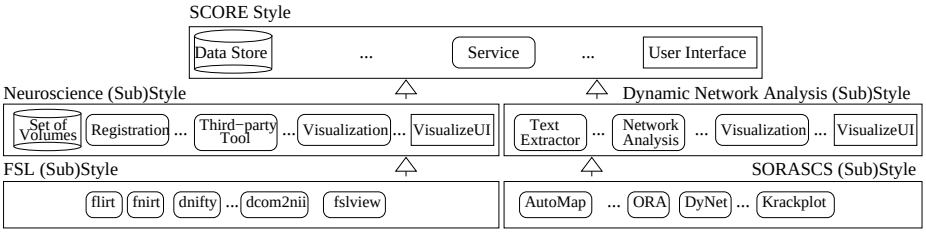


Fig. 2: Component refinement by inheritance.

Program 1 shows a snippet of an ADL-like[11] specification that illustrates specialization of FSL Style elements. Data *format* and data *structure* information are added as properties of the ports of the `flirt` service component.[12] Note also that the `flirt` service component inherits from the `Registration` service component in the Neuroscience (Sub)Style, which in turn inherits from the `Service` component in the SCORE Style as shown in Figure 2. The specialization of the SCORE style can be as detailed as needed in a particular domain. The resulting architectural specifications can be used to automatically check constraints to detect various types of violations in compositions. As we will show later, in this work we take advantage of all these aspects to detect data mismatches and construct legal repair alternatives.

Program 1 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;
 }
```

3.1 Mismatch Detection Phase

End users are often constrained by their domain-specific styles of construction while composing computations. By enforcing constraints that restrict the values of the properties of a composition, end-user compositions can be analyzed for data mismatches.

[10] http://www.fmrib.ox.ac.uk/fsl/

[11] We assume familiarity with Architectural Description Languages (ADL) syntax.

[12] In various architectural styles data ports are used to denote data elements produced (output) and consumed (input) by components.

Architectural specifications are particularly useful for such a verification, as they embed constraints that are evaluated at design time. In our approach, the *Mismatch Detection Engine* analyzes compositions with respect to the mismatches described in Table 1 by using the properties and constraints defined by SCORE (and the additional substyles). For example, this predicate can be used to define an analysis 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 of the same format (e.g., DICOM or NIfTI[13]), but the data must also have the same structural properties (e.g., Aligned or NotAligned). Predicates are implemented as type checkers that take end-user specifications and detect data mismatches. Once a mismatch is detected via the defined analyses, the Mismatch Detection Engine retrieves the architectural specifications of the pair of mismatched components and outputs this to the repair finding phase.

3.2 Repair Finding Phase

Selecting correct composition elements with appropriate properties, with right connections, has always been a tricky process, as people often make mistakes. In this phase, our approach attempts to solve this problem by taking declarative specifications of the pair of mismatched composition elements, along with the constraints in which they could be combined, and use a model generator to find a configuration that satisfies them.

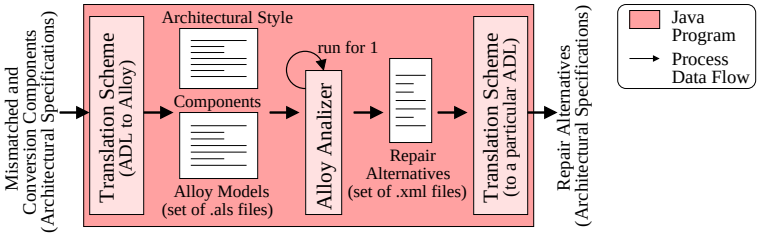


Fig. 3: The Repair Finding Engine.

Fig. 3 outlines how our approach uses the Alloy Analyzer [18] (as a model generator) to generate valid compositions that satisfy the domain-specific constraints. These form the repair alternatives for the compositions. The *Repair Finding Engine* takes architectural specifications of both the (pair of) *mismatched components* and a set of *conversion components* as input and translates them into Alloy specifications. For an accurate model-generation, our approach also requires an Alloy model of the *architectural style of the target system* to which the mismatched components belong, that includes the constraints in which the components can be used (as denoted in Fig. 3).

In recent years, various approaches to model architectural constructs in Alloy have been developed, e.g., [20, 17]. In our work, we have adopted the approach in [20] where architectural types are specified as signatures (`sig`) and architectural constraints are

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

specified as facts (`fact`) or predicates (`pred`). To provide a general idea of this translation method, consider the following ADL-like specification of the `dinifti` service component shown in the FSL (Sub)Style in Figure 2:

```
Component Type dinifti extends ThirdPartyTool = {
   in.format = DICOM;
   ...
}
```

The component extends the generic component type `ThirdPartyTool` and defines one port of the type `In` with a `DICOM` format value. Using the adopted translation method, results in the Alloy specification shown in Program 2. In this specification the `extends` keyword specifies style-specific types extending the signatures of generic ones, while the `format` and `in` relations model containment relations among types.

Program 2 A component specification in Alloy.

```
sig legalFormats {}
sig NIfTI, DICOM extends legalFormats{}
sig In {format: legalFormats}
sig ThirdPartyTool extends Service {  in: In,  ... }
sig dinifti extends ThirdPartyTool {}

fact {  dinifti.in.format = {DICOM}   ...  }
```

While generating the legal repair, we use the *constructibility of specific architectural configuration analysis* described in [20]. A simple version of this analysis can be performed by instructing the Alloy Analyzer to search for a model instance that violates no assertions and constraints within the specified scope number (using the `run for 1` command). The Repair Finding Engine thus finds all the valid instances of a repair alternative by having multiple runs of this command. As depicted in Fig. 3, the Alloy Analyzer stores these instances as XML files. These files are then automatically transformed to architectural specifications to be processed in the next phase of the approach.

3.3 Repair Evaluation Phase

Service repositories often have a large number of converters available that could lead to multiple repair choices for a data mismatch. In this phase, our approach automates a solution for such scenarios through a utility based strategy. We assume that most composition scenarios have some quality of service criteria such as speed, number of computation steps, quality of output etc., which can enable the selection of an appropriate repair strategy that maximizes the utility value of the resulting composition. Therefore, *architectural specifications of the set of repair alternatives* and a *QoS Profile* are inputs to the *Repair Evaluation Engine* (see Figure 4). This information is used to calculate an overall QoS value for each repair alternative by using *utility theory* [12].

We implemented a simple repair evaluation strategy using QoS profiles for compositions. A QoS Profile is a XML-based template that is meant to be filled in by the end user with two main types of QoS information: (i) *QoS expectations* for a repair alternative and (ii) *importance of each QoS concern* in the profile compared to other concerns. QoS concerns are defined as quality attributes and expectations on them are characterized as *utilities*. Here, *utility* is a measure of relative satisfaction –received by

the consumer of a service that explains some phenomenon in terms of increasing or decreasing such satisfaction. For instance, let x_1, x_2, x_3 be in a set of alternative choices. If the decision-maker prefers x_1 to x_2 and x_2 to x_3, then the utility values u_{xi} assigned to the choices must be such that $u_{x1} \leq u_{x2} \leq u_{x3}$. In utility theory, a utility function of the form: $u : X \rightarrow R$ can be used to specify the utility u of a set of alternatives, where X denotes the set of alternative choices and R denotes the set of utility values. For example, the "accuracy" quality attribute could have a utility function defined by the points $\langle$(Opt, 1.0), (Ave, 0.5), (Low, 0.0)$\rangle$ to represent that an optimal accuracy (Opt) gives an utility of 1.0, an average accuracy (Ave) gives the utility of 0.5, and a low accuracy (Low) gives no utility. An end user might need to specify preferences over multiple quality attributes to denote their relative importance. For example, in some situations the designer may require the urgent execution of the workflow. Thus, a repair alternative should run as quickly as possible, perhaps at the expense of fidelity of the result. Conversely, when converting among data formats, minimizing distortion can also be an important concern. In the QoS Profile this information is specified as weights.

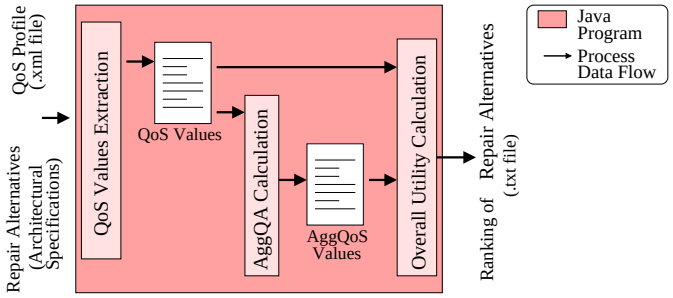


Fig. 4: The Repair Evaluation Engine.

To calculate the utility of a repair alternative, it is necessary to first calculate a set of aggregated quality attribute values ($aggQA$) for a repair alternative. These values, computed via a set of built-in domain-specific functions, are analogous to the quality attributes values exposed by each converter but they apply to a whole repair alternative. For example, suppose that a repair alternative comprises a sequence of three converters C exposing the following values for the distortion quality attribute: $Average$ (e.g., 0.5), $Average$ (e.g., 0.5) and $Optimal$ (e.g., 1.0). A distortion aggregated value for the whole repair alternative in this case could be $Average$ (i.e., 0.5) when using the following domain-specific function:[14] $aggQA_{Dist} : 1/m \sum_{i=1}^{k} = Dist(C_k)$, with $m = n + 1$. There is one function for each quality attribute in the QoS Profile. In this approach, converters must define values for the quality attributes to be considered in the QoS Profile in order to apply these functions.

Using the above information, and based on the ideas presented in [6], we have defined a straightforward way to compute the overall utility of a repair alternative. Given a set of repair alternatives, each defining a set of q quality attributes, a set of aggregated quality attributes values $aggQA$, a utility function u that assigns an utility value to each

[14] $Dist$ stands for distortion.

$aggQA$ and an importance value w for each one of these q quality attributes; a utility function U of the form: $U : \sum_{i=1}^{q} = w_i * u(aggQA_i)$, with $\sum_{i=1}^{q} w_i = 1$, can be used to calculate the overall utility for each repair alternative. The utilities for the alternatives are used to provide a ranking that the end-user can use to select the best repair alternative to the detected mismatch.

4 Example

In this section we illustrate our approach with an example of workflow construction in the neuroscience domain via a prototype tool called SWiFT [8], which provides a graphical workflow construction environment. The tool uses a simplified version of the SCORE architectural style to drive workflow construction and incorporates some analyses to verify their validity at design time. We have extended it, as described in Section 3, to allow for data mismatch detection. In this example we use both Data Services (to access data stores) and FSL Services.

4.1 The Neuroscience Domain

In the neuroscience domain, 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, as shown in Figure 5 (a).[15] 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, see Figure 5 (b). Thus, registration of brain volumes allows integrated brain-imaging views.

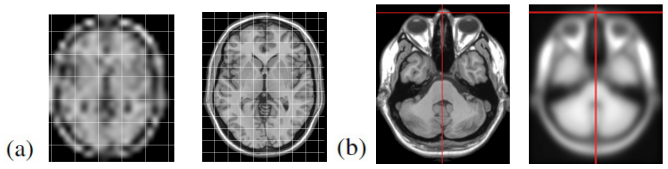


Fig. 5: (a) Volumes in voxels and (b) registered volumes with same brain reference.

Processing and visualizing data sets require scientists in this domain to compose 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 to data mismatches; thus, scientists also need to select conversion tools and services to resolve them.

4.2 Workflow Composition Scenario

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

[15] A voxel is a unit volume of specific dimensions, e.g. width, length and height.

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 2 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.

Table 2: 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' sameSubject='Yes|No' registered='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'

Table 3: Some brain-imaging tools to perform registration and format conversion.

Operation	**Description**	**Name**
LINEAR REGISTRATION	Align one brain volume to another using linear transformations operations, e.g., rotation, translations.	*flirt*
NON-LINEAR REGISTRATION	Extends linear registration allowing local deformations using non-linear methods to achieve a better alignment, e.g., warping, local distortions.	*fnirt*
FORMAT CONVERSION	Converts images from the DICOM format to the NIfTI format used by FSL, SPM5, MRIcron and many other brain imaging tools.	*dinifti, dcm2nii*

4.3 Data Mismatch Detection and Resolution

Figure 6 (a) shows how the data mismatch is presented to the scientist in our tool once it is detected by an analysis based on the predicate presented in Section 3.1. In order to compose these two services, the scientist should invoke the Repair Finding Engine by clicking on the "Resolve Data Mismatch" button in the tool interface (shown on the left hand side of Figure 6 (a)). We illustrate the case of a repair involving a combination of converters, see Table 3. Format conversion can be performed by using either the *dinifti* or the *dcm2nii* service converters. Registration can be performed by using the either the *flirt* or the *fnirt* FSL services. Part of the operations' parameter specifications of such services is also shown in Table 2. Based on these specifications and the corresponding Alloy Models, the Repair Finding Engine finds the following repair alternatives (RA):

RA_1: Set of Volumes - dinifti - flirt - Visualize Volumes
RA_2: Set of Volumes - dinifti - fnirt - Visualize Volumes
RA_3: Set of Volumes - dcm2nii - flirt - Visualize Volumes
RA_4: Set of Volumes - dcm2nii - fnirt - Visualize Volumes

All of these alternates are legal, as they obey the architectural style's constraints that restrict their structure and properties. However, because the constituent conversion services have different quality attribute values –see Program 3, the overall QoS of each

repair alternative is different. 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*: ⟨(Optimal, 1.0), (Average, 0.5), (Low, 0.0)⟩, *Speed*: ⟨(Optimal, 1.0), (Average, 0.5), (Low, 0.0)⟩ and *Distortion*: ⟨(Y, 0.0), (N, 1.0)⟩, with the 0.5, 0.1 and 0.4 weight values respectively.

Program 3 QoS specifications of the FSL services.

```
<QoSSpecification> <!-- dinifti -->
 <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>
<QoSSpecification> <!-- dcm2nii -->
 <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>
<QoSSpecification> <!-- flirt -->
 <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>
<QoSSpecification> <!-- fnirt -->
 <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>
```

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:[16]

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 U described in Section 3.3.

$$
\begin{aligned}
U_{RA_1} &= w_{Dist} * u(aggQA_{Dist}) + w_{Sp} * u(aggQA_{Sp}) + w_{Acc} * u(aggQA_{Acc}) \\
&= 0.5 * 1.0 + 0.1 * 0.5 + 0.4 * 1.0 = 0.95 \\
U_{RA_2} &= w_{Dist} * u(aggQA_{Dist}) + w_{Sp} * u(aggQA_{Sp}) + w_{Acc} * u(aggQA_{Acc}) \\
&= 0.50 * 0.00 + 0.10 * 0.50 + 0.40 * 1.0 = 0.25 \\
U_{RA_3} &= w_{Dist} * u(aggQA_{Dist}) + w_{Sp} * u(aggQA_{Sp}) + w_{Acc} * u(aggQA_{Acc}) \\
&= 0.5 * 1.0 + 0.1 * 1.0 + 0.4 * 1.0 = 1.0 \\
U_{RA_4} &= w_{Dist} * u(aggQA_{Dist}) + w_{Sp} * u(aggQA_{Sp}) + w_{Acc} * u(aggQA_{Acc}) \\
&= 0.5 * 0.0 + 0.1 * 0.50 + 0.4 * 1.00 = 0.45
\end{aligned}
$$

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

The obtained results are ranked and alternative 3, which has the highest utility, allows automatic generation of the workflow shown in Figure 6 (b). This mismatch resolution strategy not only generates a correct workflow, but it also alleviates the otherwise painful task of manual search and error resolution by the end users.

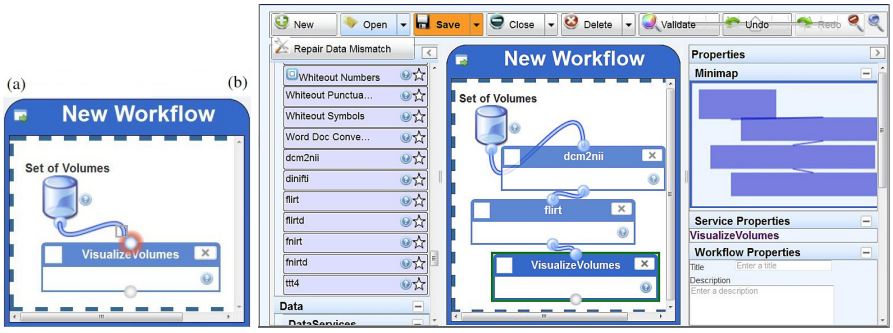


Fig. 6: (a) Data mismatch detection in our tool, (b) Workflow after mismatch resolution.

5 Discussion and Evaluation

In this section we discuss and evaluate our approach with respect to (*a*) its usefulness for the targeted end users, (*b*) its implementation cost and flexibility, and (*c*) the efficiency and scalability of the used techniques.

Usefulness for the targeted end users. Traditional composition requires low-level technical expertise, which is not the case for many end users in some domains. For software systems, architectural abstractions for components and composition help to bridge the gap between non-technical and technical aspects of the software. We exploit this to address the problems in end-user composition. Our approach is aided by architectural abstractions, which allow a generic system modeling vocabulary that does not deal with low-level technical aspects, and therefore can be more easily understood and used by non-technical users. Such abstractions are designed once (by experts and component developers) and can be reused multiple times by the end users.

Another aspect of our approach is the need for end users to specify multiple QoS values. Although end users do not think explicitly about QoS attributes, they certainly think implicitly about them. Informal discussions with end users highlights that they are concerned about how long an analysis will take (i.e. performance), whether information will leak (i.e. privacy), whether resulting images are suitable for a particular diagnostic goal (i.e. precision, data loss) and the like. Our approach asks them to think about and quantify these explicitly to help them identify better compositions for their requirements.

Implementation Cost and Flexibility. Because of the nature of our approach, its implementation cost can be significantly minimized by reusing or refining several artifacts such as the architectural styles, the analyses, the translation rules to Alloy, the domain-specific aggregation functions and the overall QoS utility function. Although some effort is needed for creating these artifacts, this effort is required *only once* by a style designer and a domain expert and the resulting artifacts can be *reused later many times* by end-users during workflow construction. Moreover, as discussed before, many of these artifacts can be reused through refinement. Note that the modeling constructs of languages such as BPEL or the domain-specific ones used by composition environments such Taverna and LONI Pipeline can be reused many times, but cannot

be refined to specific domains, like ours. Moreover, our approach is flexible enough to be integrated in composition tools; for example the SWiFT tool, used in the examples described in Section 4.

Efficiency and Scalability. A large number of languages today support the composition of computational elements. Examples include, BPEL, code scripts, and domain-specific composition languages (DSCLs) used by Taverna and LONI Pipeline. However, most of these provide very low-level and/or generic modeling constructs, and hence are not very efficient for end-user tasks [8]. Architectural specifications, in contrast, provide high-level constructs that can be reused and refined to address composition in specific application domains. The formal nature of architectural specifications enables various analyses to be performed *automatically*. We illustrated this by reusing and refining some architectural definitions in SCORE; specifically by adding properties to data ports and constraints on them, we were able to handle a bigger scope and tackle data mismatch detection in the neuroscience domain. Thus, as shown in Table 4, we claim that architectural specifications are more efficient and scalable than BPEL, code scripts or the mentioned domain-specific languages.

Table 4: Efficiency and scalability aspects for some composition specification languages.

	Architectural Specifications	**BPEL, Scripts, DSCLs**
Efficiency (in terms of): Automated Analysis	Robust	Limited
Scalability (in terms of): Refinement of abstractions	Robust	No support No

In comparison, several formal methods have been used to support the automated composition of architectural elements at design time. A majority of existing work in web-service automation focuses on using Artificial Intelligence (AI) planning techniques [1].[17] Although, many such AI planning techniques guarantee correctness of the generated compositions based on logic, a correct composition *might not be the optimal composition*, as it is recognized that planners tend to generate unnecessarily long plans [21] and little consideration is given to QoS aspects while selecting the services in a plan [1]. Additionally, AI planning based service composition tools such as SHOP2 [27] do not consider the scenario of having more than one service for a plan's action. Therefore, multiple composition plans cannot be generated. Another interesting line of work has been towards assisted mash-up composition using pattern-based approaches, e.g. [7] –despite the fact that not all the evaluation aspects presented in Table 5 apply for them. A mashup consists of several components, namely mashlets, implementing specific functionalities. Thus, pattern-based approaches to mashup composition aim at suggesting pre-existing "glue patterns", made of mashlets, in order to autocomplete a mashup. Most of this work relies on an autocompletion mechanism based on syntactic aspects of the signatures of the mashlets and the "collective wisdom" of previous users that have successfully use the glue patterns. Thus, optimal composition generation is limited. Moreover, the number of composition alternatives depends on the number of

[17] Service composition based on AI planning considers services as atomic actions that have effects on the state. Given a list of actions that have to be achieved and a set of services that accomplish them, a plan is an ordered sequence of the services that need to be executed.

existing patterns rather than the number of individual mashlets. Finally, approaches using ontologies based on description logic are also used to assist users in selecting and composing workflow components, e.g. [16]. However, most of these approaches offer limited support for resolving mismatches that require a collection of converters.

Table 5: Efficiency and scalability aspects of some approaches to automated composition, i.e. Model Checking with Alloy (MC), Artificial Intelligence Planning (AIP) and Pattern-based (PB).

	MC	**AIP**	**PB**
Efficiency (in terms of):			
- *Automated composition*	Robust	Robust	Robust
- *Composition correctness*	Robust	Robust	Robust
- *Optimal composition generation*	Limited	Limited	Limited
- *Multiple composition alternatives*	Robust	Limited	Limited
- *Translation to architectural constructs*	Robust	No Support	-
Scalability (in terms of):			
- *Processing large models*	Limited	Limited	-

Table 6: Results of the scalability experiment. All times are measured in milliseconds.

No. of Converters	**No. of Signatures**	**Translation Time (TT)**	**Solving Time (ST)**	**TT + ST**
4	13	256	47	303
10	21	827	141	968
15	26	1,077	234	1,311
25	36	1,575	453	2,028
50	61	9,376	2,215	11,591

We address the limitations of existing work in automated composition through model checking and model generation using Alloy. Two important aspects motivated its use in our work. First, by using the model finder capabilities of Alloy Analyzer it is easy to generate *multiple alternative compositions*. Secondly, Alloy provides a simple modeling language, based on first-order and relational calculus, that is well-suited for representing abstract end-user compositions. Additionally, we used several *ADL to Alloy automated translation* methods developed in recent years, e.g. [20, 17, 29].

One of the widely known problems of using model checking is the combinatorial explosion of the state-space that limit their scalability when working with large models. We believe that it is not a major concern in our case. To support this claim, we performed an experiment in which we increased the number of converters from 4 to 50 to work with bigger models.[18] Table 6 summarizes the results obtained, including those for the example presented in this paper with 4 converters. TT is the translation time, ST is the solving time, and the summation of TT+ST is the total time to generate the first possible solution –following solutions take negligible time.[19] Note that the time to generate a repair alternative in a scenario with 50 converters is about 11 secs. This time is a drastic improvement over the complexity of resolving such mismatches manually.

6 Conclusions and Future Work

Many composition frameworks today provide support for data mismatch resolution through special purpose data converters. However, as end users often have several con-

[18] The experiment was performed on a 2.67 GHz Intel(R) Core i7 with 8 GB RAM.

[19] TT is the time that the analyzer takes to generate the Boolean clauses; ST is the time it takes to find a solution with these clauses.

verters to select from, they still have to put significant effort in identifying them and determining which meet their QoS expectations. In this paper we presented an approach that automates these tasks by combining architectural modeling, model-generation and utility analysis. We demonstrated our approach with SWiFT –a web-based tool for workflow composition, using a simple data-flow composition scenario in the brain imaging domain. However, we have been working with other domains with different computation models [15].

Our future work includes exploring the integration of our approach with popular composition environments and performing usability studies on these environments. We also plan to study means to make QoS specification more approachable to end users by considering more real-life situations in specific domains, e.g. in the neuroscience domain, distortion could refer to a situation in which a converter obscures tumours of certain diameter. Similarly, as new converters and quality attributes may appear over time, we plan to define means to evolve the domain specific-functions and QoS profiles. We are also considering to explore applying the techniques used in this work to other forms of repairs. e.g. service substitution in workflows with obsolete services.

Acknowledgments

This material is based upon work funded by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Further support for this work came from Office of Naval Research grant ONR-N000140811223, the Center for Computational Analysis of Social and Organizational Systems (CASOS) and the FCT Portuguese Science and Technology Agency under the CMU-Portugal faculty exchange program. The authors would like to thank to Aparup Banerjee, Laura Gledenning, Mai Nakayama, Nina Patel, and Hector Rosas –MSE students at CMU, and Diego Estrada Jimenez –MSE student at the CIMAT for their contributions in development of the SWiFT tool and the integration of the engines into it respectively.

References

1. G. Baryannis and D. Plexousakis. Automated Web Service Composition: State of the Art and Research Challenges. Technical Report ICS-FORTH/TR-409, ICS-FORTH, 2010.
2. K. Belhajjame, S.M. Embury, and N.W. Paton. On characterising and identifying mismatches in scientific workflows. In *Proc. of the Int. Workshop on Data Integration in the Life Sciences*, volume 4075 of *LNCS*, pages 240–247. Springer-Verlag, 2006.
3. J. Bhuta and B. Boehm. A framework for identification and resolution of interoperability mismatches in COTS-based systems. In *Proc. of the Int. Workshop on Incorporating COTS Soft. into Soft. Syst.: Tools and Techniques*. IEEE Comp. Soc., 2007.
4. S. Bowers and B. Ludascher. An ontology-driven framework for data transformation in scientific workflows. In *Proc. of the Data Integration in the Life Sciences Workshop*, volume 2994 of *LNCS*, pages 1–16. Springer-Verlag, 2004.
5. J. Cámara, J.A. Martín, G. Salaün, C. Canal, and E. Pimentel. Semi-automatic specification of behavioural service adaptation contracts. *ENTCS*, 264(1):19–34, 2010.
6. S.W. Cheng, D. Garlan, and B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proc. of the Int. Workshop on Self-adaptation and Self-managing Systems*, pages 2–8. ACM, 2006.
7. S.R. Chowdhury. Assisting end-user development in browser-based mashup tools. In *Proc. of the Int. Conf. on Software Engineering*, pages 1625–1627. IEEE Press, 2012.

8. V. Dwivedi, P. Velasco-Elizondo, J.M. Fernandes, D. Garlan, and B. Schmerl. An architectural approach to end user orchestrations. In *Proc. of the European Conf. on Software Architecture*, volume 6903 of *LNCS*, pages 370–378. Springer-Verlag, 2011.
9. T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.
10. A.J. Ko et al. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21, 2011.
11. D. Hull et al. Taverna: A tool for building and running workflows of services. *Nucleic Acids Research*, 34 (Web Server Issue):W729–W732, 2006.
12. P.C. Fishburn. *Utility theory for decision making*. Pub. in operations research. Wiley, 1970.
13. C. Gacek. *Detecting architectural mismatches during systems composition*. PhD thesis, University of Southern California, Los Angeles, CA, USA, 1998.
14. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12:17–26, November 1995.
15. D. Garlan, V. Dwivedi, I. Runchkin, and B. Schmerl. Foundations and tools for end-user architecting. In *In Proc. of the Monterey Workshop*, volume 7539 of *LNCS*, pages 157–182. Springer, 2012.
16. Y. Gil, V. Ratnakar, E. Deelman, M. Spraragen, and J. Kim. Wings for pegasus: A semantic approach to creating very large scientific workflows, 2006.
17. K. Hansen and M. Ingstrup. Modeling and analyzing architectural change with Alloy. In *Proc. of the ACM Symposium on Applied Computing*, pages 2257–2264. ACM, 2010.
18. Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.
19. M. Grechanik K. Bierhoff and E.S. Liongosari. Architectural mismatch in service-oriented architectures. In *Proc. of the Int. Workshop on Systems Development in SOA Environments*. IEEE Comp. Soc., 2007.
20. J.S. Kim and D. Garlan. Analyzing architectural styles. *Journal of Systems and Software*, 83:1216–235, July 2010.
21. M. Klusch and A. Gerber. Evaluation of service composition planning with OWLS-XPlan. In *Proc. of the Int. Conf. on Web Intelligence and Intelligent Agent Technology*, pages 117–120. IEEE Comp. Soc., 2006.
22. W. Kongdenfha, H. R. Motahari-Nezhad, B. Benatallah, F. Casati, and R. Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Transactions on Services Computing*, 2:94–107, 2009.
23. C. Letondal. Participatory programming: Developing programmable bioinformatics tools for end-users. In *End User Development*, volume 9 of *Human-Computer Interaction Series*, pages 207–242. Springer Netherlands, 2006.
24. X. Li, Y. Fan, and F. Jiang. A classification of service composition mismatches to support service mediation. In *Proc. of the Sixth Int. Conf. on Grid and Cooperative Computing*, pages 315–321. IEEE Comp. Soc., 2007.
25. S.C. Neu, D.J. Valentino, and A.W. Toga. The LONI debabeler: a mediator for neuroimaging software. *Neuroimage*, 24:1170–1179, 2005.
26. B. Schmerl, D. Garlan, V. Dwivedi, M.W. Bigrigg, and K.M. Carley. SORASCS: a case study in SOA-based platform design for socio-cultural analysis. In *Proceedings of the Int. Conf. on Software Engineering*, pages 643–652. ACM, 2011.
27. E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for web service composition using SHOP2. *Web Semant.*, 1(4):377–396, October 2004.
28. I. Wassink, P.E. van der Vet, K. Wolstencroft, P.B. Neerincx, M. Roos, H. Rauwerda, and T.M. Breit. Analysing Scientific Workflows: Why Workflows Not Only Connect Web Services. In *Proc. of the .Congress on Services*, pages 314–321. IEEE Comp. Soc., 2009.
29. S. Wong, J. Sun, I. Warren, and J. Sun. A scalable approach to multi-style architectural modeling and verification, 2008.