Validation and mismatch repair of workflows through typed data streams

BY GAGARINE YAIKHOM1,*, MALCOLM P. ATKINSON1, JANO I. VAN HEMERT1, OSCAR CORCHO2 AND AMY KRAUSE3

1School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh EH8 9AB, UK
2Departamento de Inteligencia Artificial, Facultad de Informática, Universidad Politécnica de Madrid, Spain
3EPCC, University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, UK

The type system of a language guarantees that all of the operations on a set of data comply with the rules and conditions set by the language. While language typing is a fundamental requirement for any programming language, the typing of data that flow between processing elements within a workflow is currently being treated as optional. In this paper, we introduce a three-level type system for typing workflow data streams. These types are parts of the Data Intensive System Process Engineering Language programming language, which empowers users with the ability to validate the connections inside a workflow composition, and apply appropriate data type conversions when necessary. Furthermore, this system enables the enactment engine in carrying out type-directed workflow optimizations.

Keywords: type system; structural types; domain types; workflow optimization

1. Introduction

In workflow expression languages, computational activities are encapsulated inside processing-element instances. To build an application, these processing-element instances are connected with one another to produce an executable workflow. Not all processing elements can connect with one another, and these constraints are captured using a processing-element signature. While making connections, it is trivial to validate the cardinalities of the input and output interfaces using these signatures; however, untyped signatures are insufficient to ascertain the following:

— whether data produced by a source processing element is indeed consumable by the destination processing element;

*Author for correspondence (gyaikhom@gmail.com).

One contribution of 12 to a Theme Issue ‘e-Science: novel research, new science and enduring impact’.
— whether the enactment engine could transform data produced by a source processing element to match the data that are expected by the destination;
— whether data produced by a source can be pruned by removing data elements not required by the destination, thus reducing the communication costs;
— whether data can be split into multiple data streams so that each stream can be processed in parallel, thus improving the application throughput; and
— whether data can be compressed before long-haul data transfers, and which compression method to select to minimize the communication costs.

The Advanced Data-Mining and Integration Research for Europe (ADMIRE) framework (http://www.admire-project.eu) introduces various features that empower the enactment engine in making the above decisions possible. This helps the enactment engine in carrying out validation and optimizing transformations that improve the reliability and performance of workflows.

2. The Advanced Data-Mining and Integration Research for Europe framework

The ADMIRE project investigates the enactment of large-scale data-mining applications over distributed environments. The ADMIRE framework provides a higher level abstraction of data-mining applications, based on a streaming-data computation model. In this framework, computational objects (e.g. noise filtering algorithms) and data sources (e.g. seismology databases) are abstracted as processing elements, which are then connected over a communication network depending on the data flow requirement. Although the initial testing applications are data-mining applications, the ADMIRE architecture is relevant for a wide range of data-intensive applications.

(a) Abstraction and language

The ADMIRE abstraction model is realized using the Data Intensive System Process Engineering Language (DISPEL), which is a scripting language for the higher level expression of workflows. This language is similar to other workflow expression languages, such as ZigZag [1] from the MEANDRE framework, Web Services Business Process Execution Language (BPEL-WS) [2] from IBM and others listed in Deelman et al. [3] and Ludäscher et al. [4]. Using DISPEL abstraction constructs, the ADMIRE framework validates and optimizes execution (in other words, enactment) of data mining and integration applications.

While TAVERNA [5,6] has explored a two-level type system, the novelty of DISPEL lies in its three-level type system that captures language, structural and domain type information. In the TAVERNA architecture [7], the design decision was made to choose a loose typing of data defined by a collection of type constructors and list of Multi-purpose Internet Mail Extensions (MIME) types, which are reinforced with textual description (for human readability) and structured metadata in the form of Resource Description Framework (RDF) statements. While this type system is aimed towards a higher level typing of data that would belong to a set of well-recognized data formats (e.g. text in

Phil. Trans. R. Soc. A (2011)
rich text format), it does not allow typing of arbitrary formats. As discussed in TAVERNA [7], the open-ended nature of the various types that could exist inside various application domains would have made a strictly typed system extremely complicated for TAVERNA to manage.

In the DISPEL language, we avoid this pitfall by only providing a strict set of type definition constructs that will allow domain experts to define arbitrary domain types on-the-fly as they become relevant to their applications. Furthermore, as these domain types are associated with a strict structural type definition and defined using a limited set of constructs with strict type-propagation rules, the ADMIRE framework can validate data compatibility on both structural and domain levels.

Other domain-specific frameworks that provide domain-specific data types include Modular Visualization Environments such as IRIS EXPLORER [8], APE [9], APPLICATION VISUALIZATION SYSTEM [10], OPENDX [11] and SCIRUN [12]. These systems allow component interconnection with typed data streams (for instance, communication of data types geometry and lattice in IRIS EXPLORER). However, while these systems are designed for the specific purpose of scientific visualization, the ADMIRE framework is aimed towards a wider set of application domains that fall under the streaming-data computation model.

(b) Prototype implementation

The prototype implementation of the ADMIRE framework currently supports enactment over two platforms: Open Grid Services Architecture-Database Access and Integration (OGSA-DAI) [13] and Unified System Management Technology (USMT) from Fujitsu Laboratories of Europe [14]. OGSA-DAI is a platform solution for distributed data access and management. It allows data (for example, resource files, and relational or Extensible Markup Language (XML) databases) to be federated and accessible over web services. Through these web services, clients can query, update, transform and combine data from various sources. Further details are available from the OGSA-DAI home page [15]. USMT, on the other hand, is Fujitsu’s ubiquitous infrastructure for data centre management. It provides unified capabilities for the management of components in all layers of a data centre with web services.

(c) Semantic registry

The ADMIRE framework is a knowledge-based workflow management system. The schematic of the system architecture is shown in figure 1. Users communicate with the ADMIRE system using DISPEL scripts (which may be concealed by a Graphical User Interface (GUI)-based workbench). These scripts can carry out resource configuration (e.g. registration of computational activities, data source, etc.) and knowledge registration (e.g. type definition, function definition), and can submit workflow applications. To provide a consistent system for validation and optimization of workflow composition and execution, the ADMIRE system uses a semantic registry. The semantic registry provides interfaces for storage, analysis, validation and optimization of workflow applications. The DISPEL compiler and optimizer consult the registry by invoking these interfaces during the construction and enactment of workflow applications. As part of the prototype implementation, the ADMIRE framework has implemented a semantic registry.
using the ontology technologies RDF schema [16], Ontology Web Language (OWL) [17] and Simple Protocol and RDF Query Language (SPARQL) [18] query languages.

3. The Data Intensive System Process Engineering Language type system

DISPEL introduces a three-level type system for the validation and optimization of workflow applications. Using this type system, the enactment engine not only validates a workflow composition, but also validates the connections between processing elements. Furthermore, this type system exposes the lower level structure of the data streams that are being communicated through a valid connection, so that workflow optimization algorithms can re-factor and reorganize processing elements and their interconnections to improve performance.

The components of the three-level type system are:

— The language type system corresponds to traditional programming language types. It validates if the operations in a DISPEL sentence are
Typed data stream: validation and repair

properly typed. For instance, the language type checker will check if the control variables in a loop iterator, or indexes of an array, are of the correct type, and that the parameters supplied to a function invocation match the type of the formal parameters in the function's declaration.

— The *structural type system* describes the format and low-level interpretation of values that are being transmitted along a connection between two processing-element instances. For example, the structural type system could check if the data flowing through a connection are a list of tuples, etc.

— The *domain type system* describes how application-domain experts interpret, or abstract, the data that are being transmitted along a connection in relation to the application at hand. For instance, the domain type system will describe if the data flowing through a connection are a sequence of satellite images, or a collection of brain images.

In the current implementation of the ADMIRE framework, all of the above type checks are carried out statically at compile time. This happens when the workflow is assembled as the DISPEL script is being parsed. Except for the language type check, the remaining type checks are carried out in consultation with the semantic registry. Dynamic typing, therefore, is not allowed once a workflow has begun execution.

The following DISPEL example uses all of the above types:

```dispel
package eu.admire.type {
    namespace dcm "http://purl.org/dc/elements/1.1/";
    use eu.admire.activities.bibliography.FromDublinCoreToBiblio;
    Stype DCMeta is [String::dcm:title t; String::dcm:creator c; rest];
    Stype Bibliography is [String key, entry];
    Type DCMToBiblioPE is PE (<Connection:DCMeta input> => <Connection:Bibliography output>);
    DCMToBiblioPE DCMToBiblioConvertor(String hashAlg) {
        FromDCMToBiblio activity = new FromDCMToBiblio(hashAlg);
        return PE (<Connection input=activity.input> => <Connection output=activity.output>);
    }
    register DCMeta, Bibliography, DCMToBiblioPE, DCMToBiblioConvertor;
}
```

The above DISPEL script creates a wrapper function `DCMToBiblioConvertor` for converting Dublin Core Metadata (DCM) [19,20] to simple bibliography data. Processing-element instances returned by this function consume a list of tuples with DCM elements, and produce a list of bibliography data, where the key is generated using the checksum algorithm `hashAlg` (e.g. SHA1, MD5, etc.). The keywords `use` and `register` are DISPEL constructs for communicating with the registry: `use` for exporting definitions from the registry, and `register`
for importing definitions to the registry. The package keyword defines the scope of each registration. The rest of the example is discussed in the following sections.

4. Language type system

The language type system only applies to the validation and evaluation of DISPEL sentences. It is a statically checkable type system that corresponds to type systems as used in traditional programming languages.

The language type system consists of predefined base types, which are then extended using well-defined type constructors. The predefined language base types in DISPEL are Boolean, Byte, Integer, Real, Char, String and Connection. All of these base types are common to most programming languages, except for the Connection type, which represents a handle for inter-processing-element communications. These base types are further extended with user-defined language types, which are defined using DISPEL type constructors. There are three recursively applicable type constructors: Array, Tuple and Function; and a special type constructor PE.

Array. An array specifies a multi-dimensional arrangement of elements where each of the elements can be accessed uniquely using array indices. The array elements can be either base types, or user-defined extended types, and all of the elements of a given array must have the same type. The syntactic denotation of an array-type construction uses opening ‘[’ and closing ‘]’ square brackets that follow the type of the array elements. The number of these opening and closing square bracket pairs gives the dimension of the array type. The array bounds are specified when an instance of the array is created.

Tuple. A tuple specifies an unordered arrangement of elements that can have different types. This is in contrast to arrays, where all of the elements of the array must have the same type. Tuples, therefore, allow packing of data with different language types. The syntactic denotation of a tuple-type construction is based on the opening ‘⟨’ and closing ‘⟩’ angle brackets, where each of the enclosed types is listed within these brackets, and elements of different types are separated with semicolons; e.g. ⟨Real x, y; String label⟩ is a valid tuple.

Processing element. The PE type constructs processing-element types. This exposes the interface signature of the processing element, allowing validation and optimization of connections between processing-element instances. A PE type consists of two Connection tuples, where the tuple elements represent the input and output interfaces, e.g. PE ⟨Connection n, data⟩ => ⟨Connection[] outputs⟩.

Function. A function type constructor abstracts a composition of processing elements. A call to a function returns a composition of processing elements (or, composite processing element). This composition could comprise a single processing element (see example above), or consist of an entire application that manifests a complex network between processing-element instances. Functions allow insertion of a composite processing element within existing ones, as it encapsulates the composition inside a well-defined interface. The above example DISPEL script defines and registers a function DCMToBiblioConvertor, which returns a processing element of type DCMToBiblioPE that encapsulates the activity FromDublinCoreToBiblio, which is supplied by the enactment engine under the package eu.admire.activities.bibliography.
5. Structural type system

The structural type system is concerned with the structural representation of the data that flow through the connections between processing-element instances. During structural type checking, the parser validates the structure of the data stream.

The DISPEL structural type system allows annotation of connection instances to indicate fully (or in part) the structure of the values flowing along it. It allows propagation of structural type annotations across the enactment workflow (the data-flow graph) in accordance with propagation rules as stored in the registry or declared in the current DISPEL script. The structural type of a Connection handle is an important part of the PE type specification. It provides a means for the parser to validate structural type compatibility when connecting source and destination communication interfaces. If a connection is type incompatible, the parser could attempt to insert type convertors to validate the connection, similar to shim services in TAVERNA [21,22]. With the detailed logical information that is captured by the structural type system, the enactment platform is empowered with better error reporting, diagnostics and the opportunity to carry out data-stream directed optimizing workflow transformations.

Logically, every connection carries a stream of values that are ordered temporally depending on when the values were put into the connection. These values manifest an abstract structure, which is independent of the actual data implementation. In DISPEL, structural types capture this abstract structure, as shown in figure 2.

Within a stream, logical chunks are separated using delimiters, which are markers that separate the chunks. Depending on the manner in which the data are formatted, the choice of delimiters could vary from one enactment platform to the other. For instance, the values could be sent using the XML data interchange standard [23], or as binary representation using Data Format Description Language description [24], or JavaScript objects in JavaScript Object Notation (JSON) format [25], or when the source and destination share memory, the values are passed by object reference. During structural type checking, these implementation-specific representations of the data stream are of no concern to the DISPEL compiler, as it is only interested in validating the abstract structure of the values. The DISPEL compiler expects the enactment platform to provide appropriate activities that can process the actual data formats.
(a) The definition of structural types

The structural type system is constructed from the predefined base types that every DISPEL enactment system can handle, i.e. Boolean, Byte, Integer, Real, Char and String. Furthermore, this is extended with the partial structural descriptors `any` and `rest`, as discussed in §5b below. Using these as the basis, the structural type system is extended using structural type constructors, List, Array and Tuple.

*List.* The List type constructor defines a list structure that consists of a sequence of values of the same structural type, e.g. list of strings. The syntactic denotation of a list construction is based on the opening ‘[’ and closing ‘]’ square brackets, which enclose the structural type of the list elements. Thus, `[Real]` would denote all lists where every element of the list is a real number, and `[(Real x, y, z)]` could denote a list of tuples, where each tuple represents a three-dimensional coordinate.

*Array.* The Array type constructor defines a multi-dimensional arrangement of elements in the data stream, where each of the elements can be accessed uniquely using array indices. The array elements can be either base types, or user-defined extended structural types, and all of the elements of a given array must have the same type. The syntactic denotation of an array-type construction uses opening ‘[‘ and closing ‘]’ square brackets that follow the structural type of the array elements. The number of these opening and closing square bracket pairs gives the dimension of the array type. Since we are only interested in the abstract structure of the data stream, structural types do not capture array sizes as DISPEL considers these values to be implementation specific. Thus, `Real[][]` would denote a two-dimensional array of reals, perhaps a matrix, and `[Integer[]]` would denote a list of integer vectors.

*Tuple.* The Tuple type constructor defines an unordered arrangement of elements that can have different types. This is in contrast with arrays, where all of the elements of the array must have the same structural type. Tuples, therefore, allow packing of data with different structural types. The syntactic denotation of a tuple-type construction is based on the opening ‘⟨’ and closing ‘⟩’ angle brackets, where each of the enclosed types are listed within these brackets, and elements of different types are separated with semicolons. A generic tuple construct is

\[
(\text{type}_1 \; \text{id}_{11}, \; \text{id}_{12}, \; \ldots ; \; \ldots ; \; \text{type}_k \; \text{id}_{k1}, \; \text{id}_{k2}, \; \ldots ),
\]

where `\text{type}_i` denotes any structural type, and `\text{id}_{ij}` denote identifiers.

(b) Partial description of abstract structure

It is necessary to accommodate and use partial descriptions of the values, so that we can pass through the connections any domain-specific values by simple copy operations, until it reaches a domain-specific processing element that knows how to interpret and use the values. This is for practical reasons, since it is inconceivable that we can expand our repertoire of known types to handle every domain-specific requirement.
To accommodate partial description, DISPEL uses the following types:

- **Type `any`** denotes that a value can be of any type. This is akin to the types \(\langle\text{xsd:any}\rangle\) and \(\langle\text{xsd:anyType}\rangle\) in Simple Object Access Protocol (SOAP) packets. Thus, type `any` allows data streams with unspecified content type. For instance, \(\langle\text{Real } x, y; \text{ any detail}\rangle\) is a valid structural type where tuple element `detail` could be of any type.

- **Type `rest`** denotes remaining elements of a tuple that the structural type checker does not care about. Type `rest` should only appear once in a tuple, and should also be the last element of the tuple. For instance, \(\langle\text{String name; rest}\rangle\) is a valid construct, whereas \(\langle\text{String name; rest; Integer age}\rangle\) is invalid.

(c) **Structural type checking**

Structural type checking takes place during compilation when the workflow data-flow graph is assembled by parsing the DISPEL script. Whenever a connection is made between two processing-element instances, the compiler references the processing-element types as defined within the same DISPEL script, or the type information stored in the registry, in case, a PE type has been imported from the registry. If the types are compatible, the connection is made. If there is a structural type mismatch, however, the compiler has to make a decision between two alternatives. In the first instance, the compiler will check if the structural type of the data stream as produced by the producer can be converted to the structural type expected by the consumer. In the decidable case, where the compiler is certain that insertion of convertors can resolve the type conflict, the compiler inserts a network of convertor processing elements between the interfaces that are type incompatible.

In the undecidable case, where it is uncertain that the data will meet the input requirement, the compiler inserts a dynamic type checker (DTC). A DTC is a processing-element instance that takes two inputs: `requiredType`, which is a definition of the required type, and `dataIn`, which is the incoming data stream whose types are being validated. Based on these two inputs, the DTC instance produces two outputs: `dataAccepted`, which is a stream of values that match `requiredType`, and `rejects`, which are values that fail to match. These failures are communicated to the user with appropriate debugging information on why the value was rejected.

(d) **Unpack–convert–repack network**

To insert appropriate convertors, the compiler traverses the structural type abstract syntax tree (AST) that is formed by the recursive application of the structural type constructors, and enquires of the registry if a convertor is available at the given depth. If there is a convertor, an instance is created, and this convertor instance (a one-to-one processing element with single input and output interfaces) is made responsible for converting the data values at the given depth of the structural type tree. When a convertor cannot be found at the given depth, the parser continues looking for convertors by traversing deeper into the structural type AST. If no convertors are found at the lowest level.
(the leaf nodes of the structural type AST), the compiler flags the workflow description as type incompatible, and alerts the user to make the connections type compatible.

Type incompatibility alerts are raised at compile time when a workflow is being assembled from a DISPEL script. Since the current implementation of the ADMIRE framework does not support dynamic typing, these alerts are reported to the user before a workflow is executed. The manner in which the alerts are reported depends on the interface the user used for submitting the DISPEL script. On the GUI-based ADMIRE workbench, the alerts would come in the form of popup dialogues; on the command-line interface, the alerts will be displayed as console feedback.

(e) Insertion of type convertors

A convertor can be either a single processing-element instance, or a complex network of processing-elements instances, and this depends on the complexity of the abstract structure. Insertion of a single convertor instance is trivial; however, insertion of a complex convertor requires generation of a network of convertor instances that will unpack, convert and repack the data stream. To do this, the parser uses a recursive tree-traversal algorithm on the ASTs of the structural type specifications (both for the source and destination handles).

To illustrate this, consider a connection from a producer to a consumer, where they are communicating volumetric data at a given point at a given time (e.g. global ice formation survey). The convertor network is generated as shown in figure 3. This example demonstrates how data elements are transformed to meet the data types that are required by the consumer (e.g. conversion of cartesian to polar coordinates).
polar, metre to foot, etc.). Of course, we have assumed here that the relevant type and data convertors have already been defined and registered with the semantic registry.

While it is desirable to illustrate the insertion of type convertors in real applications, which the ADMIRE project is currently using as test cases (for instance, gene-expression analysis or flood forecasting), we have provided a simpler example here for ease of exposition. To use complex test cases as illustrative examples would be counter-productive without the required context, and establishing the context in this paper would be prohibitive given the amount of text such a context would require. For details on the test cases, please refer to Han et al. [26] for gene-expression analysis, Ciglan et al. [27] and Tran et al. [28] for flood forecasting and Trani et al. [29] and Spinuso et al. [30] for ambient-noise detection in seismology.

6. Inferring context: the domain type system

An abstract structure can represent more than one concrete data-stream format. For instance, the structural type specification \( \langle \text{Real } id_1, id_2, id_3 \rangle \) specifies a tuple with three real elements. Hence, this could represent the real axis of a three-dimensional cartesian coordinate, or the yaw, roll and pitch values of a three-dimensional orientation. Since conversion of data is only meaningful within a given context, structural type conversions cannot work directly by converting base types. Conversions are only legitimate within a context where the values are related; hence, the unpack–convert–repack network must be generated depending on a context.

The domain type system is concerned with how application-domain experts interpret the data that are being transmitted along a connection in relation to the application at hand. During domain type checking, the parser validates the compatibility of domain interpretation when establishing a connection between processing-element instances, and assists type conversion by providing an application context.

In the ADMIRE framework, the domain type system is realized using ontologies, and ontology-based annotations, using the standard ontology languages RDF schema [16] and OWL [17]. It is important to note that an application domain, in order to avoid conflict of interpretation, should define a unique name space for that domain. This is normally done by providing a Uniform Resource Identifier (URI) that hosts the ontology. In DISPEL, we use a URI-based namespace declaration. Domain types are specified with respect to a namespace declaration. This is illustrated by the example in §3, where the elements \( t \) and \( c \), while structurally abstracted as Strings, should only be processed within the context defined by the DCM specification, available at the URI specified using the namespace keyword.

At the registry, domain type checking is done using the SPARQL query language [18]. Whenever computational activities are registered with the enactment platform, the platform engineer must also supply the registry with the corresponding ontology-based annotations, as well as references to the ontologies that they use. During DISPEL processing, the parser exports relevant workflow construction directives by passing queries to the registry. For instance, before
connecting two connection handles, the parser queries the registry to ascertain if such a connection is type compatible; and if it is possible to generate a valid unpack–convert–repack network to resolve any type incompatibilities within the context specified by the domain types.

Such a system works in practice because the enactment platform relies solely on the registry for consistency and semantic interpretation, and no workflow can be enacted without prior registration of the required definitions with the registry.

7. Future work: workflow optimization

Using the type information available on the structural and domain interpretation of the data stream, a parser can carry out various workflow optimizations. We are currently investigating workflow optimization techniques where processing-element instances process a succession of lists. Some of these optimization techniques include redundant tuple element pruning that reduces communication costs; and permutation, sampling or splitting of a data stream for parallelization.

Figure 4 shows tuple pruning, which reduces communication costs by filtering out redundant tuple elements from the communicated data. It relies on consumers with input handles that use the rest structural type, e.g. \( \langle \text{String a, rest} \rangle \). Since rest signifies elements that are of no concern to the consumer, they could be considered redundant as long as the consumer does not forward the unprocessed input tuples to its successors (i.e. the optimization must not cripple successor processing elements).

Figure 5 shows tuple splitting, which increases parallelism by splitting tuples. In scenarios where the entire tuple is processed on the same resource by a composition of processing elements, tuple splitting allows tuple elements to be processed simultaneously by mapping each of the individual processing elements to various resources. Since higher level abstraction of processing element compositions (using DISPEL functions) requires correct typing, the optimizer can explore optimization opportunities by analysing the data-stream typing at the functional interface in relation to the structural and domain typing in each of the individual PE types encapsulated by that function.

The paper by Liew et al. [31], which also appears in this issue of Phil. Trans. R. Soc. A, discusses some aspects of the optimization of DISPEL workflows.

8. Conclusions

In this paper, we have introduced a three-level type system for typing workflow data streams. This type system is currently being used in the ADMIRE project, which investigates the enactment of large-scale data-mining

*Phil. Trans. R. Soc. A* (2011)
applications over distributed environments. Using the type system described herein, it is now possible for enactment engines to validate the interconnections between processing elements; thus increasing the reliability of workflow systems. Furthermore, we have discussed our ongoing work on using this type system for carrying out optimizing workflow transformations that improve performance by reducing communication costs and by exposing opportunities for data-stream-directed load balancing and resource scheduling.

The work presented here is part of the ADMIRE project, which is funded by the European Commission under Framework 7 ICT 215024.

References


7 TAVERNA. 2006 The TAVERNA architecture: working with data. See http://www.taverna.org.uk/developers/taverna-1-7-x/architecture/working-with-data/.


Type data stream: validation and repair


