Jul 1st, 12:00 AM

Semantic validation and correction of scientific workflows

David Ratcliffe

Geoffrey Squire

Michael Kearney

International Environmental Modelling and Software Society (iEMSs)
2010 International Congress on Environmental Modelling and Software
Modelling for Environment's Sake, Fifth Biennial Meeting, Ottawa, Canada
David A. Swayne, Wanhong Yang, A. A. Voinov, A. Rizzoli, T. Filatova (Eds.)
http://www.iemss.org/iemss2010/index.php?n=Main.Proceedings

Semantic validation and correction of scientific workflows

David Ratcliffe, Geoffrey Squire, Michael Kearney
CSIRO ICT Centre, Canberra, Australia
firstname.lastname@csiro.au

Abstract: Scientific workflows describe steps for orchestrating the execution of a network of computational operators toward some goal, such as data transformation for analysis or visualization. Typically, these operators consume and emit transformed data, or cause some effect. In most scientific workflow systems, the operators are typed to enable compatibility checks for their composition that make up a workflow. However, type checking performed by most such systems today is still largely confined to syntactic checking with limited, if any, semantic type checking support. In this paper, we present a type system incorporating the W3C OWL ontology language to aid in representing the semantics of data and workflow operators. We show how this type system supports the detection of type incompatibility errors in workflow compositions, and how it facilitates a (semi-)automatic type correction procedure using type transformations. We have incorporated our solution into Kepler, enabling users to statically test the type-consistency of workflows typed using our type language, and demonstrate that inconsistent bindings between expressively typed operators can be automatically corrected via a procedure that seeks to compose adapter/shim functions, such as unit transformations, time series interpolations, or some other arbitrarily complex data transformations.

Keywords*: workflow; semantics; type system; owl; adapter; shim.*

1. INTRODUCTION

Scientific workflow technology is an emerging method used to create complex data processing procedures without the need to author much, if any, programming code. These systems are finding widespread use in diverse domains such as earth sciences, bioinformatics and astronomy. A wide range of these systems exist and are in use today, amongst the most popular being Kepler, Microsoft's Trident, Taverna (Hull et al. 2006), Triana (Majithia et al. 2004) and NASA's SciFlo (Wilson et al. 2005) which provide high-level graphical-based environments for the authorship of workflows.

Irrespective of the scientific workflow system used, fundamental similarities exist in the specification of data processing provided by these products. A scientific workflow describes the steps for orchestrating the execution of a linked network of potentially distributed computational operators toward some goal, such as the transformation of data for analysis or visualization. Such operators are often implemented to perform particular tasks over particular data. For this reason, composing heterogeneous operators in a workflow can be a challenging task if their interfaces are not directly compatible. In such a case, a workflow author is required to insert functions (known as *adapters* or *shims*; see Hull et al. 2004) that serve only to manipulate the data as output from one operator into a form appropriate for consumption by another operator with a different interface. The requirement to create adapters/shims in a workflow places a large burden on the workflow author, as they are required to cope with many data management issues at once, such as understanding the particular data formats involved, how to manipulate them appropriately, and how to specify a correct implementation; this is an error-prone process.

In our work, we seek to alleviate as much as possible of the data management issues surrounding the construction of adapters/shims by providing automated support for their generation. In order to achieve this, we are required to increase the level of expressiveness of data and operator types used in scientific workflow to a greater degree than that which is currently implemented by most systems, which is largely based on syntactic typing, with little to no support for semantic data and operator typing. Much work around the Kepler project (e.g., Bowers et al. 2005a) has recognized the limitations of such type systems, and various proposals have been put forward for more expressive type systems based on recognizing the syntax, structure and semantics of data and the operators that process them. In our work, we are attempting to extend the state of the art by generalizing the techniques presented in order to permit mediation directly between many fundamental types of data models, such as relational, nested, and multidimensional, and consider various syntaxes of each. We are also exploring the use of a language for describing abstract functions that can operate over such data, along with methods of describing how type information propagates through such functions by analysis of their specification and use in a workflow context.

We have designed and implemented a prototype system and have incorporated its use within Kepler. The system allows users to annotate the operators (actors) and their input and output port data types using an expressive type language which captures the syntax, structure and semantics of data. Using the expressions in this type language, we are able to perform automated tasks such as type consistency checking (i.e., testing if two types are compatible), which then allows us to detect type errors in bindings made between the output of an operator and the input of another operator. The detection of such erroneous bindings then provides us with a starting point for a search problem that seeks to rectify type inconsistencies by automatically inserting data transformation operators (adapters/shims), which are available to the workflow system and have also been described using our type language.

2. DATA TYPES

Most scientific workflow systems will recognize data types and provide type checking as a basic validation of a composition of operators. For example, the typed output of one function bound to the typed input of another in a workflow (as depicted in Figure 1) can be checked by a type system which compares the compatibility of the source data type (function output) with the target data type (function input). Commonly, the data types being compared are expressed in languages which are tightly bound to the *syntax* of the data, such as 'the XML document with schema X' or 'a Java array of Integer objects'.

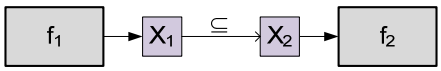

Figure 1. Function f_1 outputs data with syntactic type X_1 which is directly compatible to type X_2 as input of f_2.

The semantics of the data is typically not explicitly captured in types – instead, it is only implicit in the syntax, if at all. This can be a problem, as while types X_1 and X_2 may be compatible at a syntactic level, their meaning (semantics) may be incompatible. For example, X_1 and X_2 could be floating point numbers, but where X_1 represents values of speed in metres per second, and where X_2 represents temperature in degrees Celsius. A system that does not explicitly capture the semantics of a type cannot determine such semantic inconsistencies. In our work, we aim to increase the expressiveness of types in order to capture more about the structure and semantics of data, such that we can perform more sophisticated type checking. We consider three aspects to the data type problem – that of the surface structure, the underlying core structure, and semantics.

We refer to the *syntax* of data as being comprised of two aspects, that of *surface structure* which describes the way information is presented (e.g., XML), and that of *core structure*, which corresponds to the underlying arrangement of data (e.g., ordered set, nested

collection, record). We also consider transformational mappings between surface structures and core structures, which describe how to derive a core structure from a surface structure, and how to convert a core structure into a surface structure. We also consider the *semantics* of data, which we associate closely with the core structure of data, as the latter describes the underlying content of data instances in a more abstract way which may permit a range of surface structural representations.

Our aim is to develop a type system that is capable of representing and comparing many common data types as encountered in a scientific workflow system (such as those captured with XML, CSV, relational data, object data, logic databases, multidimensional data such as NetCDF, etc.) at their core structural and semantic types, in order to recognize type compatibilities at these abstract levels. Detecting that a set of types have compatible core structure and semantics can potentially suggest automated transformations between various surface structures. The following sub-sections explore some concrete examples of these three distinct aspects of type.

2.1 Surface Structural Type Differences

Consider the case where we are attempting to locate a transformation from XML source data to CSV target data, where both encode data with a tabular structure. Moreover, the columns in each record of this tabular data have the same semantics (in that they capture equivalent data).

Example source instance (XML):

```
<timeseries>
  <m ts="2010-01-05T09:00:00Z">0.501</m>
  <m ts="2010-01-05T09:15:03Z">0.442</m>
  ...
</timeseries>
```

Required target instance (CSV):

```
date,     time_utc, level_in
1/5/2010, 9:00,     5.01e-1
1/5/2010, 9:15,     4.42e-1
...
```

With an explicit model of the surface structural types (XML with a particular structure defined by an XML schema, and CSV with a particular grammar), core structural types (tabular data as a collection of records) and semantic types (defining the semantics of the collection, and records in tuples in the collection, e.g., a time series of rainfall measurements in inches, where records contain data, time and a magnitude value of rainfall in inches), we should be able to infer that a transformation between the XML and CSV based types is legitimate due to the equivalent core structural and semantic types, and possible if we have transformations from these core structural types to each surface structure (XML and CSV).

A procedure to transform the XML to CSV in this example would simply need to iterate over the underlying core structure in the source instance i.e., an ordered sequence of `<m>` elements) and emit, with relevant alterations and in the order visited, the data in each `<m>` element as a record in the CSV syntax.

2.2 Core Structural Type Differences

In this example, while the semantic type (e.g., a time series of rainfall measurements made in inches) may be the same between source and target types, the underlying data described in each has a different core and surface structure. For example, consider the following two XML fragments.

Example source instance (XML):

```
<timeseries>
  <m ts="2010-01-05T09:00:00Z">0.501</m>
  <m ts="2010-01-05T09:15:03Z">0.442</m>
  ...
</timeseries>
```

Required target instance (XML):

```
<ts>
  <day date="2010-01-05Z">
    <sample time="9:00:00Z" value="0.501"/>
    <sample time="9:15:03Z" value="0.442"/>
    ...
  </day>
  ...
</ts>
```

The source instance has a core structure of an ordered sequence of measurement items as flat records, while the target instance core structure is different, in that it nests measurement items into groups per day. While the latter is still an ordered sequence of items, the underlying core structure is different in that the items are not flat record structures as in the source instance, but instead contain ordered sequences themselves with measurements made at particular times per day. A procedure to transform data in the source type to data in the target type would require the construction of named groups which aggregate records per day. If this transformational operation (to group, or to flatten) is expressible over the language describing core structural types, then a mapping correspondence between the core structural types can be constructed. Since the semantic types are equivalent, the presence of such a mapping correspondence describing a grouping or flattening transformation will imply a transformation between the different surface structural types, if we have transformations from these core structural types to each surface structure.

Note that in this example, the semantic description of the target could have been tightened to reflect the grouped organization of data, in which case we would say there is also a semantic difference between the types. However, we consider this to be a core structural type aspect specifically defined to describe the arrangement of the data.

2.3 Semantic Type Differences

Continuing the time series example, consider the case where we are comparing two types that have exactly the same surface and core structure, as follows.

Example instance 1 (XML):

```
<timeseries>
  <m ts="2010-01-05T09:00:00Z">10.51</m>
  <m ts="2010-01-05T09:15:03Z">11.06</m>
  ...
</timeseries>
```

Example instance 2 (XML):

```
<timeseries>
  <m ts="2009-05-12T17:13:06Z">6.4</m>
  <m ts="2009-05-12T34:20:44Z">7.7</m>
  ...
</timeseries>
```

At a syntactic level, these two instances can both be valid under exactly the same XML Schema constraints. However, the first instance in this example encodes values of measurements of air temperature in degrees Celsius, whereas the second is encoding values of measurements of humidity in grams per kilogram. From a semantic perspective, these are instances of two completely different (disjoint) types, but this cannot be inferred from the instances themselves or the XML schema under which they are both valid. Without an explicit specification of the semantics of these types, a type system cannot automatically infer that they are incompatible.

2.4 Separation of Concerns

These examples show that by separating aspects of type into surface structure, core structure and semantics, we can identify type compatibilities through the inspection of each aspect. This provides a powerful capability to determine, for example, that two types are compatible despite their syntax (e.g., Section 2.1), or that they are incompatible even if they have the same syntax (e.g., Section 2.3).

2.5 Type Compatibility and Transformation

Figure 2 illustrates the aspects of type we wish to capture explicitly. Core structural types (S_1^C and S_2^C) have transformational mappings ($\mathcal{T}_1$ and $\mathcal{T}_2$) to surface structural types (S_1^S and S_2^S), and also logical correspondence mappings ($\mathcal{M}_1$ and $\mathcal{M}_2$) to semantic type descriptions (O_1 and O_2).

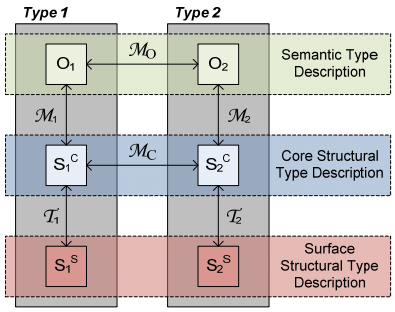

Figure 2. Surface structural, core structural and semantic aspects of data types, together with mappings and transformational correspondences.

We say that two types (*Type 1* and *Type 2*) are *directly* compatible only if each of their respective surface structural, core structural and semantic type aspects are compatible. Furthermore, since we permit subtyping, *Type 1* is only directly compatible with *Type 2* if all type aspects of *Type 1* are subtypes of the respective aspects of *Type 2*. There are certain conditions under which two types (*Type 1* and *Type 2*) are *indirectly* compatible:

- **Surface structural differences**. If the core structural and semantic types are compatible, then the transformational mappings ($\mathcal{T}_1$ and $\mathcal{T}_2$) allow us to interpret surface structural types as being compatible. This case was exemplified in Section (2.1), and is depicted in Figure 3.
- **Core structural differences.** If the semantic types are directly compatible, and there exists a mapping $\mathcal{M}_C$ that defines the transformation between different core structural types, then the transformational mappings ($\mathcal{T}_1$ and $\mathcal{T}_2$) allow us to interpret surface structural types as being compatible. This case was exemplified in Section 2.2.
- **Semantic, core structural or surface structural differences.** If any or all of the type aspects are not directly compatible, then we can still consider the types to be indirectly compatible if there exists a directed type transformation function f_T which has, as input, a type which is directly compatible with all aspects of *Type 1* and has an output type which is directly compatible with all aspects of *Type 2*. This case includes the example in Section 2.3, where we may have some function available (e.g., a software model) to compute transformations between air temperature values and humidity values in a certain context.

Figure 1 depicts what we consider to be a weak kind of type compatibility checking, in that it does not consider the semantic types which could cause a type incompatibility despite compatible syntax. We also consider this type compatibility checking to be unnecessarily strict, in that when determining if two types are compatible, both the surface structure and core structure must be deemed compatible. This kind of type of checking fails to recognize cases where core structure is compatible despite variations in surface syntax, so cannot indicate if a transformation from one surface structure to another could take place, as depicted in Figure 3.

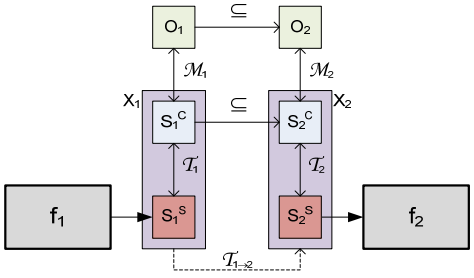

Figure 3. Function f_1 outputs data having syntactic type X_1 and semantic type O_1 which is indirectly compatible with data having syntactic type X_2 and with semantic type O_2 via transformation $\mathcal{T}_{1\rightarrow 2}$.

3. TYPE AND MAPPING LANGUAGES

3.1 Data Types

To explicitly capture the aspects of surface structural, core structural and semantic types, we require languages for each type, as well as a language to express mapping correspondences between core structural types and semantic types. Additionally, we need an interpretation that maps expressions of core structural types to surface structural types to define their transformation.

In our system, we use a predicate logic language which is expressive enough to capture the core structure aspect of syntax for many common data structures, such as relational and nested data. We also use another language, specifically the logical language W3C OWL, to define semantic types. We also employ the use of a first-order predicate logic language called iMaPl (Cameron et al. 2005) to define mappings between core structural types to semantic types in OWL, in order to assign semantics to core structural types. Specific details of these languages and our use of them in our type system are out of scope for this paper, as our focus is to describe the capability we have achieved in applying the language to type incompatibility detection and correction problems in scientific workflows.

3.2 Function Types

In addition to modelling data types, we also model data processing functions. We describe aspects of these data processing functions such as their inputs and outputs, together with a description of their category using OWL (e.g., we can describe a function which takes as input time series data and certain parameters such as axis labels, and generates a plot as a graph image on the output). From these abstract specifications, we are able to map to ground instantiations of the specification (e.g., such as the WSDL web service, Java function, etc. that performs the function and which has the inputs and outputs as described in the abstract specification). Additionally, we also have a simple model of function signature specifications over particular types, describing how input data may be propagated to output data, in order to facilitate the propagation of specific subtype information on invocations of the function on typed data.

4. IMPLEMENTATION

4.1 Current Implementation

We have currently implemented a prototype of our type system using Java, SWI-Prolog, OWL-API and the Pellet OWL reasoner. In particular, we have implemented the following:

- Language for specifying core structural types and mappings to semantic types described in OWL, and implementations of transformations between particular core structural types to particular surface structural types, such as relational schemas, tabular data including CSV, and a one way transformation from certain surface structural types capturing nested data (such as XML or object data) onto core structural types (allowing us to query existing instances, but not to generate new instances).
- A semantic type checker based on Description Logic subsumption reasoning implemented using Pellet via the OWL-API.
- A limited core structural type checker based on exact structural matching.
- A limited abstract function specification language for describing the propagation of typed input data to typed output data.
- A limited semantic type propagation capability that pushes certain constructions of semantic types through expressions manipulating core structural types.
- A type transformation composition planner implementing a search algorithm.

We have integrated our solution into the Kepler workflow environment for testing, where we take advantage of Kepler's ability to annotate workflow operators (Kepler actors) with type annotations, to which we attach specifications of our own type declarations (surface structural, core structural and semantic types).

When composing a workflow of Kepler actors which have been typed using our type language, our implementation of the underlying type system can detect binding errors (type inconsistencies between the output of one actor bound to the input of another with a directly incompatible type). The user then has the ability to send specifications of the workflow that produced errors to our type correction service, which seeks to determine if the two directly incompatible types are indirectly compatible via a type transformation procedure. Our underlying type correction service maintains an internal list of abstract operations and their grounded instantiations as Kepler actors. We have not implemented the function abstraction grounding feature as described in Section 3.2 in the Kepler environment, or the user selection of alternate options located by the type system for abstract grounding where alternate options exist (including alternate type transformation chains).

4.2 Future Work

We have not yet implemented the following:

- A language for specifying and interpreting bi-directional mappings between core structural types such as nested and multidimensional data and surface structures (e.g., XML, object data, NetCDF).
- Subtyping in the language for core structural types. We would like the ability to perform subsumption comparisons between core structural types.
- While we are capable of detecting type inconsistencies and resolving them by composing existing type transformation functions, we cannot yet automatically compute the transformations for converting between different yet indirectly compatible core structural types, such as aggregation (grouping) as described in Section 2.2.

5. RELATED WORK

The limitation of syntactic type checking alone in semantic workflow systems has been observed before. The most relevant and extensively published work on this subject surrounds the work related to the Kepler/Ptolemy project, as summarized in Berkley et al. [2005]. Our work closely reflects that of Bowers et al. [2004] where they describe a framework for *semantic mediation*, namely, the automated integration of workflow actors that operate over heterogeneous data. They identify core structural types of data and describe how to map such types onto a semantic model described by an OWL ontology (of a particular sub-language). When comparing two structurally incompatible but semantically compatible types, they seek to construct a transformation specification based on mappings between structural types and semantic types, as we also describe in Section 2.2. They describe an instantiation of their framework in terms of XML based structural types and how XQuery can be generated to transform between different XML structures which have been mapped to an ontology. In our work, we are attempting to further this capability by developing a language for core structural types that is abstract (and yet expressive) enough to capture many types of data, from nested to relational (and possibly also multidimensional), and transformational mappings from particular surface structures onto the abstract core structural types, in an effort to expand the applicability of such a system to one that can integrate directly across many combinations of heterogeneous data types.

Bowers et al. [2005a] present a hybrid type system that separates data structure from semantics, and links these two aspects using a first order predicate language (which they refer to as a hybridization constraint). In our work, we employ a similar scheme for mapping between core structural types and semantic types, in that we employ the use of iMaPl as defined by Cameron et al. [2005], which was designed for schema integration, in

a similar way as described by Bowers et al. [2004a] which describes mapping from relational predicates to an ontology schema.
The problem of propagating semantic type information through semantically typed actors in a workflow has been considered by Bowers et al. [2005] and Bowers et al. [2006]. In the latter of these works, algorithms for propagating semantic types over actors are presented, where the behaviour of the actors can described using relational queries (including selection, projection, product, union, etc). In our work, we are attempting to adopt similar strategies in order to propagate semantic types through operations that transform core structural types (e.g., nesting, or aggregation as described in Section 2.2). We recognize the importance of sound and complete propagation procedure which preserves semantic data descriptions when applied over functions with subsuming input and output types.

6. CONCLUSION

In this paper, we have described how an expressive type system that captures the semantic, surface structural and core structural aspects of type can be used to significantly improve the type checking capability of semantic workflow systems. In addition, we have implemented a procedure that takes advantage of the ability of the described type system to detect type inconsistencies, in order to perform type correction by composing existing type transformation functions. Our work extends earlier work in that we are achieving a mediation capability directly between data of different syntaxes, such as XML to CSV, or XML to relational. At present, our implementation currently stops short of generating type transformation functions on the fly (as described by Bowers et al. [2004] over XML types). We have shown how this capability can automate the potentially intensive and error-prone task of manually creating adapter/shim functions to mediate between incompatibly-typed workflow operators.

REFERENCES

Berkley, C., S. Bowers, M. Jones, B. Ludäscher, M. Schildhauer, J. Tao, Incorporating semantics in scientific workflow authoring, *In Proc. of the 17th International Conference on Scientific and Statistical Database Management (SSDBM'2005)*, 75–78, 2005.

Bowers, S., K. Lin, and B. Ludäscher, On Integrating Scientific Resources through Semantic Registration, *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, 349, 2004a.

Bowers, S., and B. Ludäscher, An ontology-driven framework for data transformation in scientific workflows. *In Proc. of the 1st Intl. Workshop on Data Integration in the Life Sciences (DILS)*, volume 2994 of LNCS, pages 1–16, 2004.

Bowers, S., and B. Ludäscher, Towards Automatic Generation of Semantic Types in Scientific Workflows, *In Proc. of the International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, WISE 2005, LNCS, 207–216, 2005.

Bowers, S., and B. Ludäscher, Actor-Oriented Design of Scientific Workflows, *In 24st Intl. Conference on Conceptual Modeling*, Springer, 2005a.

Bowers, S. and B. Ludäscher, A Calculus for Propagating Semantic Annotations Through Scientific Workflow Queries, *Current Trends in Database Technology, EDBT 2006*, 712–723, 2006.

Cameron, M. A., and K. L. Taylor, First-Order Patterns for Information Integration, *International Conference on Web Engineering (ICWE)*, 173–184, 2005.

Hull, D., R. Stevens, P. Lord, C. Wroe, and C. Goble, Treating shimantic web syndrome with ontologies, University, Milton Keynes, UK, 2004.

Hull, D., K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn, Taverna: a tool for building and running workflows of services, *Nucleic Acids Research*, vol. 34, iss. Web Server issue, 729–732, 2006.

Majithia, S., I. Taylor, M. Shields, and I. Wang, Triana: A Graphical Web Service Composition and Execution Toolkit, IEEE Computer Society, 514–524, 2004.

Wilson, B., B. Tang, G. Manipon, D. Mazzoni, E. Fetzer, A. Eldering, A. Braverman, E.R. Dobinson, and T. Yunck, GENESIS SciFlo: Scientific Knowledge Creation on the Grid using a Semantically-Enabled Dataflow Execution Environment, *Scientific and Statistical Database Management Conference (SSDBM 2005)*, 83–86, 2005.

APPENDIX A: MOTIVATING EXAMPLE

A.1 Workflow to Compute Wind Chill

In this use case, we show how a user may attempt to construct a workflow that helps to visualize a time series of wind chill values over time. The user has two data sources available, each an implementation of an Open Geospatial Consortium (OGC) Sensor Observation Service (SOS), where one produces air temperature measurement time series data, the other spatially co-located wind speed measurement time series data. The function available to compute wind chill does so by executing a function over every pair of air temperature and wind speed measurements made at the same time.

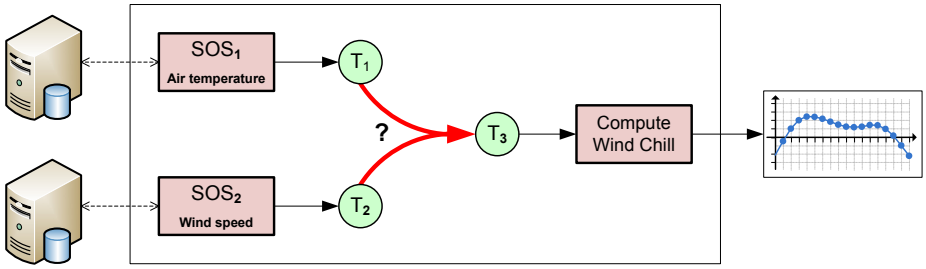

Figure 4. Incomplete workflow aimed at the retrieval and merge of two time series into one for the computation of wind chill time series data.

Figure 4 describes this incomplete workflow. The intention of the workflow author is to merge the SOS data appropriately for input to a function to compute a corresponding time series of wind chill values. In this deceptively simple example, there are several interoperability problems. At a semantic level, the wind chill function requires a time series of spatiotemporally co-located measurements of wind speed in kilometres per hour, and air temperature in degrees Celsius. However, the hourly air temperature measurements from SOS_1 are published in degrees Fahrenheit, requiring a unit transformation to Celsius. The wind speed measurements from SOS_2 have units of kilometres per hour and are made every 45 minutes, so do not align temporally with the air temperature values from SOS_1, even though both time series start at the same time (as specified in the queries to each SOS). To compound matters further, each SOS is generating time series data with different syntaxes, and the wind chill function requires an input of yet another different syntax.

In order to reconcile the workflow to one that will correctly achieve the desired intention, the following operations must be described within the workflow:

- Convert the units of air temperature measurements (of SOS_1) to degrees Celsius.
- Temporally align both time series before merging into one time series as input to the wind chill function (e.g., if we want hourly values for wind chill, this will require the interpolation of wind speed values over time to and sample corresponding values every hour to pair with the air temperature values).
- Generate a semantically and syntactically valid input document for the wind chill function which combines the aforementioned corrected data.

Typically, the author would proceed to manually reconcile the workflow by inserting appropriate functions and adapters/shims to perform these transformations. We will now show that, by encoding the types involved in the workflow as described in the earlier sections, how the type system can be used to automate tasks in completing the workflow in Figure 3 by assisting in the grounding of an abstract function specification to merge the

time series data, and to perform type incompatibility detection and correction by the semi–automated insertion of adapter/shim functions that perform type transformations.

A.2 Function Abstraction and Grounding

Since the workflow in Figure 4 is incomplete because the input data is not linked to the input of the wind chill function, we seek methods for joining them together to create a complete workflow. However, instead of constructing intermediate components for the workflow manually, we aim to take advantage of the system to infer how we can construct a correct sub-workflow by encoding our intension in the abstract as constraints, which the workflow system can resolve to a ground implementation.

Firstly, we consider the types T_1, T_2 and T_3 that we have in the incomplete workflow in Figure 1:

Table 1. Aspects of types T_1, T_2 and T_3

	Surface Structural	**Core Structural**	**Semantic Type (OWL)**
$\mathbf{T_1}$	XML Schema 1	$[(t, v_1)]$	A list of time (t) and value (v_1) tuples (time series), where v_1 are measurements of air temperature in °F, and where (t) values start at time X and have a time step interval of 60 minutes.
$\mathbf{T_2}$	XML Schema 2	$[(t, v_2)]$	A list of time (t) and value (v_2) tuples (time series), where v_2 are measurements of wind speed in km/h, and where (t) values also start at time X (as in those of $\mathbf{T_1}$) and have a time step interval of 45 minutes.
$\mathbf{T_3}$	CSV	$[(t, v_1, v_2)]$	A list of time (t) and value (v_1, v_2) tuples (time series), where v_1 are measurements of air temperature in °C, v_2 are measurements of wind speed in km/h, and where t values have some time step interval.

Firstly, we are required to describe the way the input time series data is to be merged into a single time series. This sort of operation can be described abstractly over core structural types as an anonymous function f_M which takes two lists of tuples each with a time and value vector, and combines this into a single list where value vectors are concatenated together where the times (t) are equal in tuples across both input lists (for example, the merge function signature may be: $f_M([(t, \boldsymbol{v}_1)], [(t, \boldsymbol{v}_2)]) \rightarrow [(t, \boldsymbol{v}_1 +\!+ \boldsymbol{v}_2)]$, where ++ is vector concatenation). Figure 5 demonstrates this specification in the current workflow, as follows:

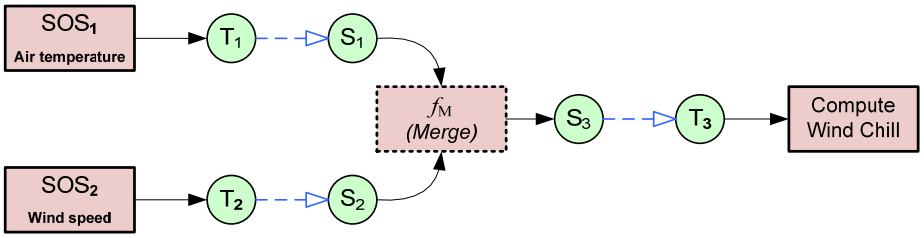

Figure 5. Incomplete workflow describing an abstract merge over time series data. The blue arrows represent bindings between typed operator outputs to inputs.

This partial specification of the workflow describes the necessary requirement to merge the two time series T_1 and T_2, but does not yet specify exactly how it is to be performed. In order to determine this, the system will need to ground the function abstraction (as described in Section 3.2) to a transformation which achieves the specified effect. Consider that the system does indeed have a transformation function already available, as shown below in Figure 6.

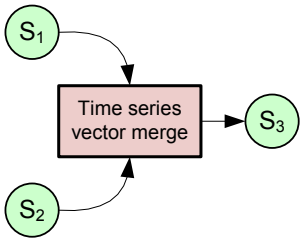

Figure 6. Generic, ground time series merge operator (Java implementation).

The time series vector merge function has the following input and output types:

Table 2. Aspects of types S_1, S_2 and S_3

	Surface Structural	**Core Structural**	**Semantic Type (OWL)**
S_1	Java Array	$[(t_1, v_1{:}vs_1)]$	A list of time (t_1) and values ($v_1{:}vs_1$) tuples (time series), with the same time step interval and start time as described in **S_2**.
S_2	Java Array	$[(t_1, v_2{:}vs_2)]$	A list of time (t_2) and values ($v_2{:}vs_2$) tuples (time series), with the same time step interval and start time as described in **S_1**.
S_3	Java Array	$[(t_3, v_1{:}vs_1 \text{ ++ } v_2{:}vs_2)]$	A (time series) sequence of time (t_3) and values ($v_1{:}vs_1$ ++ $v_2{:}vs_2$) tuples, with the same corresponding time step interval, start time and value types as described in **S_1** and **S_2**.

The system matches this transformation function to the merge function abstraction as it satisfies the constraints in the types and function signature. Specifically, this function strictly has input and output types which are subclasses of those described by the abstract merge function specification, in that it requires that all types have the same surface structural type (Java), and that the time step interval is the same in both inputs and output.

Individually, the type system finds that the semantic types T_1 and T_2 in the workflow are more specific subtypes of either S_1 *or* S_2. However, T_1 and T_2 cannot be considered to be compatible as the inputs to the merge function *together at the same time*, under the conditions against S_1 and S_2 which equate their time step values, since T_1 and T_2 have different time steps.

Similarly, when considering how values propagate through the merge function, we find that the values being transformed by the function (i.e., air temperature and wind speed) are not modified but are propagated to the output of the function, thus have the same units of measure in the output structure with type S_3. However, there is still a mismatch of units between the air temperature in degrees Fahrenheit values as coming out (and going in) of the merge function, against what is required in T_3 (air temperature in degrees Celsius).

Since the merge function is required in this workflow as shown in the specification in Figure 5, the system will attempt to reconcile the types T_1 and T_2 against S_1 and S_2, and S_3 against T_3, using type transformation functions, in order to use the time series merge operator in Figure 6.

A.3 Type error detection and correction

As described in the last section, the system has resolved the merge function abstraction to a particular ground function but has detected that the input and output types are not compatible with those as specified in the workflow to which they are to be bound.

In order to correct these type inconsistencies, the system will attempt to infer if types T_1 and T_2 are *indirectly* compatible to S_1 and S_2 (and S_3 with T_3) through some type transformational steps. Since no core structural transformational steps are necessary as these aspects of the types are already directly consistent, there are only semantic and surface structural inconsistencies left to resolve. Firstly, this section considers the semantic inconsistencies of types T_1 and T_2 against S_1 and S_2.

Consider the availability of a time series interpolation function as described in Figure 7. This function takes a list of time value pairs (representative of a time series which can be interpreted as having a continuous interval), and attempts to interpolate between the given points in some specified way, such as using curve-fitting for linear interpolation, polynomial, etc. in order to resample the values using a different time interval.

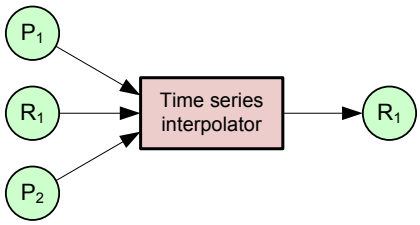

Figure 7. Generic, ground time series interpolation operator (Java implementation).

The input and output types of this function are summarised in Table 3.

Table 3. Aspects of types R_1, R_2, P_1 and P_2

	Surface Structural	**Core Structural**	**Semantic Type (OWL)**
R_1	Java Array	$[(t_1, v_1)]$	A list of time (t_1) and value (v_1) tuples (time series), with some time step interval and start time.
P_1	URI	x	An interpolation type x (e.g., linear, polynomial, spline, etc.)
P_2	(Java Float, URI)	(ts_{period}, ts_{uom})	A time step interval which is type of time period with some factor ts_{period} and unit of measure ts_{uom} (e.g., 5.0 seconds).
R_2	Java Array	$[(t_2, v_1')]$	A list of time (t_1) and value (v_1') tuples (time series), where v_1' are of the same type as those defined in **R_1**, and with time step interval described by **P_2**.

In the presence of such a function, the system will recognize that by transforming one (or both) data with types T_1 or T_2 through the interpolation function that the resultant data type(s) R_2 will be compatible with both S_1 and S_2 (when considered together) as input to the merge function. As the time interval in the semantic type description for R_2 is not constrained to be any particular value (must be defined by P_2), and does not propagate through the function from the input. However, input data of types P_1 and P_2 are as yet unbound, and no other information is available to the system as to how to resolve these. Therefore, while the system can suggest that the application of this interpolation function may be appropriate based on the limited information it has about the types, it will be left to the user to decide on the applicability of the function and if so, to define all unbound input parameters manually.

By binding each of T_1 and T_2 as input to time series interpolation functions, we transform them to new types, that which we will define as R_2' and R_2'' (respectively), which capture the propagated type of values being interpolated and re-sampled over, namely, that of air temperature and wind speed measurements, respectively. Similarly, when considering the output of the merge function, we find that the output is in fact a subtype of S_3 since this function also propagates input values with particular semantic types that it merges into one

time series; we will refer to this type as S_3', where $S_3' \subseteq S_3$. In comparing type S_3' to the required input type T_3 of the function to compute wind chill, we find that the types are not directly compatible, as summarised:

Table 4. Aspects of types S_3, S_3' and T_3

	Surface Structural	**Core Structural**	**Semantic Type (OWL)**
S_3	Java Array	$[(t, v_1, v_2)]$	A (time series) sequence of time (t) and value (v_1, v_2) tuples, with the same time step interval as defined by both **S_1** and **S_2**.
S_3'	Java Array	$[(t, v_1, v_2)]$	A (time series) list of time (t) and value (v_1, v_2) tuples, where v_1 are measurements of air temperature (°F), v_2 are measurements of wind speed (km/h), and where t values have a time step interval as defined by **S_3**.
T_3	CSV	$[(t, v_1, v_2)]$	A (time series) list of time (t) and value (v_1, v_2) tuples, where v_1 are measurements of air temperature (°C), v_2 are measurements of wind speed (km/h), and where t values have some time step interval.

S_3' is not directly compatible with T_3 as while their core structures match, S_3' defines measurements of air temperature in °F (v_1), whereas the same air temperature measurements (v_1) in T_3 are described as having units in °C; additionally, the surface structures differ (Java Array as opposed to CSV).

The system admits unit transformation functions as depicted in Figure 8 which are capable of transforming from one type (V_1) to another (V_2) that preserve surface and core structural types (applicable to a common surface structural type) but which translates semantic types. With such a function, the translation between the semantic types only occur in parts of the type expressions that describe compatible primitive data in the core structural type as having differing but compatible units of measure. In determining which units of measure are compatible, such functions rely on the availability of individual functions such as the one depicted in Figure 9 which implement transformations between primitive values of differing units.

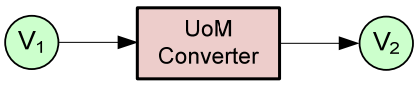

Figure 8. Type transformation function (f_T) applying unit conversion functions over primitives in any core structure permitted by the surface structure (Java implementation).

Table 5: Aspects of types V_1 and V_2

	Surface Structural	**Core Structural**	**Semantic Type (OWL)**
V_1	Java	**T**	**O_1**, which describes primitive values within **T** as having particular units of measure
V_2	Java	**T**	**O_2**, which describes primitive values within **T** as having a different unit of measure to those defined in **O_1**, where the different units are compatible and there exists type transformations f_T to convert between all the specified different units of measure

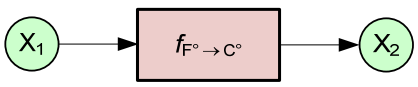

Figure 9. Type transformation function (f_T) to perform a transform from values in °F to °C (Java implementation).

Table 6. Aspects of types X_1 and X_2

	Surface Structural	**Core Structural**	**Semantic Type (OWL)**
$\mathbf{X_1}$	Java Number	v_1	v_1 is a Temperature (°F)
$\mathbf{X_2}$	Java Number	v_1	v_1 is a Temperature (°C)

Since S_3' is not directly compatible to T_3, in testing for an indirect compatibility relationship, the type system will locate the type transformation function in Figure (8) which can apply the type transformation in Figure (9) on data of type S_3' in order to construct a new type, V_2':

Table 7. Aspects of type V_2'

	Surface Structural	**Core Structural**	**Semantic Type (OWL)**
$\mathbf{V_2'}$	Java Array	$[(t, v_1, v_2)]$	A (time series) list of time (t) and value (v_1, v_2) tuples, where v_1 are measurements of air temperature (°C), v_2 are measurements of wind speed (km/h), and where t values have an interval of that defined by $\mathbf{S_3'}$.

The last type V_2' now only differs in surface structure to T_3, but the system can either generate, or may already have available as shown in Figure 10, a function to render CSV data from core structural types matching lists of flat record structures:

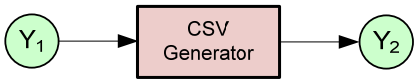

Figure 10. CSV generator function (Java implementation).

Table 8. Aspects of type Y_1, and Y_2

	Surface Structural	**Core Structural**	**Semantic Type (OWL)**
$\mathbf{Y_1}$	Java Array	$[(x_1,\ldots,x_n)]$	$\mathbf{O_1}$ (any type)
$\mathbf{Y_2}$	CSV	$[(x_1,\ldots,x_n)]$	$\mathbf{O_1}$

By composing the CSV generator in Figure 10 to data of type V_2', we achieve data with a type of Y_2', which is directly compatible with type T_3, thus completing a full (yet still only partially grounded) workflow as shown below in Figure 11:

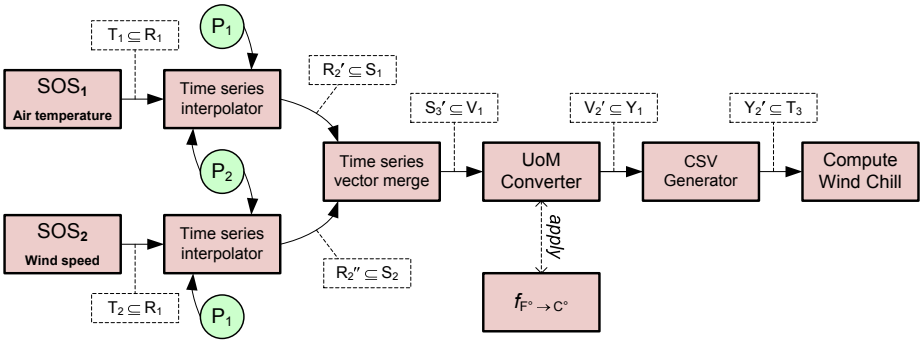

Figure 11. Partially grounded workflow with undefined parameters P_1 (interpolation type) and P_2 (time series interval).

At this point, the user is required to ground all unspecified parameters of the functions composed to complete the workflow before it can be executed; in this case, the user is required to specify an interpolation type P_1 as input to both time series interpolation functions operating over the time series of types T_1 and T_2 (e.g., linear interpolation). Also, the user is required to select a common time step interval P_2, also as input to both time series interpolation functions, which describe how to re sample the time series T_1 and T_2 such that they are temporally aligned before applying the time series merge operation.

In summary, this example has shown the following:

- Abstract function specifications with mappings to ground implementations, and their semi automatic insertion into a workflow.
- Sophisticated type checking capabilities, with type inconsistency detection and correction via type transformation composition and insertion.