ECCOMAS Congress 2016
VII European Congress on Computational Methods in Applied Sciences and Engineering
M. Papadrakakis, V. Papadopoulos, G. Stefanou, V. Plevris (eds.)
Crete Island, Greece, 5–10 June 2016

# AUTOMATIC VISUALIZATION AND CONTROL OF ARBITRARY NUMERICAL SIMULATIONS

**Jan P. Springer** and **Helen Wright**

University of Hull

**Keywords:** Computational Steering, Markup Languages, Schema, Data Binding, Visualization

**Abstract.** *Visualization of numerical simulation data has become a cornerstone for many industries and research areas today. There exists a large amount of software support, which is usually tied to specific problem domains or simulation platforms. However, numerical simulations have commonalities in the building blocks of their descriptions (e. g., dimensionality, range constraints, sample frequency). Instead of encoding these descriptions and their meaning into software architecures we propose to base their interpretation and evaluation on a data-centric model. This approach draws much inspiration from work of the IEEE Simulation Interoperability Standards Group as currently applied in distributed (military) training and simulation scenarios and seeks to extend those ideas. By using an extensible self-describing protocol format, simulation users as well as simulation-code providers would be able to express the meaning of their data even if no access to the underlying source code was available or if new and unforseen use cases emerge.*

*A protocol definition will allow simulation-domain experts to describe constraints that can be used for automatically creating appropriate visualizations of simulation data and control interfaces. Potentially, this will enable leveraging innovations on both the simulation and visualization side of the problem continuum.*

*We envision the design and development of algorithms and software tools for the automatic visualization of complex data from numerical simulations executed on a wide variety of platforms (e. g., remote HPC systems, local many-core or GPU-based systems). We also envisage using this automatically gathered information to control (or steer) the simulation while it is running, as well as providing the ability for fine-tuning representational aspects of the visualizations produced.*

Authors' preprint version as submitted to *ECCOMAS Congress 2016, Minisymposium 505 - Interactive Simulations in Computational Engineering*.

## 1 Introduction

Simulation and visualization of phenomena and processes has become an integral part in our approach to understand and manipulate the world around us. We increasingly rely on computer-based modelling and simulation in such diverse areas as engineering, fundamental research, economics, or even politics. For some time such simulations have been augmented by our ability to provide diverse visual representations based on simulation results improving comprehension of the underlying model, data, or processes. In some cases visualization is the crucial, enabling factor for comprehension and decision making. While for many problems simulation and just visualizing the results works fine, there are also problem domains where it is beneficial to change aspects of the simulation parameters while the simulation is running, likely based on an expert's analysis of visualized simulation results. In essence the unidirectional process of providing simulation results for visualization ideally should be complemented by the ability to control (or *steer*) the simulation, providing a bidirectional (or even multidirectional) flow of data.

The general approach to computational steering provides software infrastructure or an environment where simulations can be embedded to run experiments. Usually interfaces are provided that allow for transporting control or steering information to a simulation as well as returning results to the visualization component. The visualization component also contains encoded knowledge of how to represent steering controls. Past as well as current approaches for computation steering frameworks are based solely on software architectures and often tied to specific pieces of technology.

We are interested in how the connection between a simulation, its steering mechanism, and its visualization can be decoupled. We argue that software-interface approaches result in an unnecessary dependency between the intent of a simulation, its possible visualization(s), and the actual technology, i. e. the particular software API, used for realizing that coupling. We aim instead towards a flexible interface that is self-contained, platform-agnostic, and allows for longevity of simulation codes for future use. This can be seen as a generalized *glue* between the simulation and the visualization provider(s).

Our approach is motivated by the Distributed Interactive Simulation (DIS) protocol, developed since the late 1980s and standardized in IEEE:1278.1:2012 [14], which allows participation of independently developed software in a running simulation as well as the *replay* of captured events from a simulation session. Because DIS provides a well-defined protocol, including specification of extensions without breaking applications not enabled for specific extensions, it remains the standard of choice for large-scale training simulation. However, DIS does not allow to discover extensions without developing software for their appropriate evaluation. We argue that this difficulty can be circumvented by combining a data-based approach similar to DIS with a markup-language infrastructure. An approach based on XML Schema definitions [28, 29] provides for self-describing schemas and auto-generated data binding and in summary it offers all the advantages of a DIS-like approach with none of its fixed-format disadvantages.

The remainder of this paper is organized as follows. In section 2 we summarize the important role of software-oriented solutions in computational steering. Section 3 illustrates the idea behind a data-centric approach with an artificial example of a simplistic but generic computational-fluid dynamics application. In sections 4 and 5 we discuss the merits of our proposal, including comparing it with other recent work utilising XML, before summarising and offering suggestions for future work.

## 2 Related Work

Computational steering has a long and distinguished record in the wider computational science literature. Steering a computation interactively necessarily involves visualization, and the focus of early attempts at computational steering was on how to achieve the interworking of potentially diverse tools. For example, GRASPARC [4] linked computational and graphical elements via a data management system based on a tree structure (a *history tree*) that captured both the results of the calculation and the progress of the investigation. The user interface reflected this same structure: branches of the tree recorded snapshots of parameters and data that could be used to pause, modify, and restart the calculation without having to re-run the simulation from the beginning. Different visualization components could be connected and demonstrators of the system utilised both in-house developments and off-the-shelf commercial products in this regard. In CSE [23], a steering interface was built from 2d and 3d graphics objects whose properties were used both to control parameters of the simulation *and* furnish a visualization of results. Parameters were accessed in the simulation code using a data I/O library designed to be minimally invasive. As in GRASPARC, a data manager was central to the architecture, but in CSE it orchestrated steering by issuing event notifications to subscribed visualization, transformation, and calculation processes. An emerging theme of this and other steering research was the need to instrument the application source code which led in turn to the development of a number of steering libraries for this purpose. Notable contributions that particularly addressed steering in grid-computing environments were the gViz library [5] and the RealityGrid library [19].

While the aim of the majority of such steering frameworks and libraries was to enable interaction with existing computational codes, in contrast SCIRun [18] provided an environment to make combined compute-and-visualization ensembles from scratch. The architecture was based on the dataflow paradigm combined with visual programming, which had hitherto seen application in modular visualization environments (MVEs) such as AVS [8], IRIS Explorer [24], and DX (IBM Visualization Data Explorer, later OpenDX) [1]. Such systems allow the user to build a visualization application, dragging-and-dropping pre-packaged modules to accomplish the constituent steps of first reading in the data, then mapping to some graphical representation and finally rendering this to the screen. Whereas these latter supported steering by adding existing simulation codes into the system as new modules, the SCIRun module set [22] included algorithmic and mathematical elements as standard. As such it is as much a software workbench for building integrated computational and visualization applications, as it is a system for simply visualizing results.

The difficulty that besets all computational steering applications, whether based explicitly on the dataflow paradigm or not, is how to transmit control information upstream from the display to the simulation. The user and their interaction tools (e. g., panels hosting sliders, menu selections) are logically found at the output end of the pipeline, but the recipients of these tools' outputs (e. g., changes to parameters, pause and restart commands) are at the input end, all the while producing data. This becomes especially problematic when the steering-visualization pipeline is distributed [31]. The solution adopted depends on the architecture of the steering framework: GRASPARC [4] evolved a system of control structures, separate from its data pathways; CSE [23] adopted an event-based model; MVE-based systems generally abandoned strict adherence to dataflow principles in favour of architectures incorporating feedback mechanisms for implementing loop constructs, and sometimes caching data to preserve state (q. v. [1]). A novel approach described in [6], which also targeted an MVE, was to implement an input-oriented pipeline in parallel

with the more usual results-bearing pipeline. This allowed a description of the simulation's exposed parameters to be automatically searched and the correct scalar, vector, or positional interactor to be placed in the image of the output, with the correct position and appropriate value constraints already applied. Dubbed *multi-purpose image interaction*, the idea is closely allied to that described in the present paper, except the aim now is to capture the essence of both simulation parameters *and* results, independently of the software used. Using a self-describing, data-centric protocol, we assert that essential knowledge about a steering application can be preserved, for example in the absence of the original source code or, provided its outputs have been appropriately curated, potentially even after the binary can no longer be run.

## 3 Example

In the following we develop an artificial example for a simplistic but generic computational-fluid dynamics (CFD) simulation using XML Schema to define control and result types and derive possible source code based on those definitions.

### 3.1 Requirements for a Generic CFD

Computational steering of a generic CFD simulation can be separated into two elements: firstly, how to control the simulation and, secondly, what kind of results are returned. In the simplest case we assume that the simulation will be based on computing particles per simulation time step. Listing 1 shows pseudo code to encode this.

```
struct particle {
  unsigned id;
  float3   position;
  float3   velocity;
};

struct result {
  double          timestamp;
  list<particle> particles;
};
```

**Listing 1:** Result structure for a generic CFD simulation returning a simulation result for a particular time step and the computed particles placed in a list.

The `timestamp` attribute in the `result` structure contains the time in (local) simulation time for which the particle properties have been computed. The actual particle properties are returned in the `particles` attribute of the `result` structure. The `particle` structure contains the `position` and `velocity` as 3-component floating-point values and a unique `id` for each particle instance. This `id` maybe used to create traces for particles of interest. To control the simulation a `control` structure can be used as shown in listing 2.

```
struct control {
  enum etype { start, stop, update, };

  etype  type;
  float3 gravity;
  float3 force;
}
```

**Listing 2:** Control structure for a generic CFD simulation allowing for specification of starting, stopping, or updating the simulation.

The `control` structure allows for specifying if the simulation should `start`, `stop`, or if the `control` structure contains an `update` to the parameter set. This is encoded by the `type` attribute and expressed as an enumeration. In our example the simulation-control parameters are simply `gravity` and `force` expressed as 3-component floating-point values. These parameters can be set for the initial setup of the simulation, i. e. `type == start`, or when updating simulation conditions, i. e. `type == update`. In case the `type` attribute is set to `stop` the simulation should simply shutdown, though for more sophisticated simulations shutdown parameters might be used as well.

Separating the result description returned by a simulation from its possible control allows for using these structures for both communication between the simulation and visualization as well as within the visualization to represent their content. However, this is tightly coupled to a specific example. Any changes to accommodate different simulations or even just variants with slightly different control parameters will require some software-development effort for both the actual simulation as well as the visualization. Additionally, parameter sizes, intended meaning, or problem-domain-specific constraints are usually hard to express and to document for automated extraction.

### 3.2 XML Schema Definitions

An XML Schema definition (XSD) [28, 29] allows to define content structure to be instantiated in an actual XML document. While this could be any kind of document we will use the notion of an XML document here as a way of transporting data between interested parties, i. e. the simulation and visualization provider. XSDs use standard XML notation and can be evaluated using existing XML support infrastructure. An XSD can be interpreted as a class description from the view point of an object-oriented software approach. XML documents then play the role of objects, i. e. actual instances of a class at run time, although this comparison is only partially correct because XSDs only allow the definition of member attributes but not member functions.

XSDs support a variety of built-in data types, e. g., `string`, `int`, `float`, as well as creating compound structures, e. g., `simpleType`, `complexType`. Within a compound structure element definitions based on primitive or user-defined types may appear. These elements can be constrained to restrict their range or to represent a discontinuous range such as an enumeration. XSD also supports extending existing structures as shown in appendix A, listings 3–6. Listing 3 shows an XSD for n-component floating-point values which can be used to represent n-dimensional vectors. These n-component floating-point types are used in listing 4 to define a particle having position and velocity in 3d similar to the pseudo code in listing 1. XSD also allows including other XSDs, which enables sharing of definitions between separate XSD files. Listing 4 also provides an example for defining a list of particles. In addition aliases in the form of root elements to the actual compound definitions are provided, which are necessary to allow the automatic generation of serialization code. In listing 5 base types for the control and result structures are defined, which are independent of any specific simulation. The control structure contains an enumeration to express its three possible states, i. e. start, update, and stop, while the result structure contains a time stamp. Listing 6 shows the definition of the control and result structures for the generic CFD example in turn extending the general definition of control and result structures in listing 5, essentially expressing the same content as in listing 2.

XSDs provide a template for actual instances in the form of XML documents as shown in appendix B, listings 7–10. Listings 7–9 show sample control-structure instances for controlling start, update, and stop of the simulation, respectively. In the XSD for the `cfd_control` structure, the `gravity` and `force` properties are defined withinin an `xs:sequence`. This

necessitates that these properties appear in the XML document even if they are not required (e. g., in case of submitting a `cfd_control` instance of `type` stop). To make properties such as `gravity` optional the `minOccurs` attribute can be set to zero, which, for larger aggregations of properties, allows for reducing the actual size of the XML document. Listing 10 shows a sample `cfd_result` structure returned by the simulation for each simulation step. The `timestamp` property provides the local simulation time at which the result was produced while the `particle_list` property contains the list of particles and their state at that time. The `timestamp` property is defined as a `xs:time`, which is expressed in hours, minutes, and seconds format. Note that the seconds part of `xs:time` is based on a double-precision floating-point type and allows for arbitrary precise resolution in the sub-second range.

XSDs as well as XML in general are a complex topic and cannot be treated in full detail in the context of this paper. We refer the interested reader to one of the many books on this topic such as [12] or the standard specifications [26, 28, 29].

### 3.3 Code Generation

XML-based documents provide much flexibility. However, the actual processing of XML documents requires infrastructure for parsing and validation. Fortunately, there is a plethora of software available to accomplish this. Any programming language in use today provides XML processing capabilities either as part of the language (e. g., *Groovy* [11]) or as a library service. XML processing requires the use of a parser, which can be roughly categorized into stream-oriented (e. g., SAX [21]), tree-traversal or pull-parsing (e. g., DOM [30]), declarative transformation (e. g., XSLT [25, 27]), and data-binding parsing (e. g., for *C++* [32]). The data-binding parser category is of interest here because it allows to automatically create API bindings based on XSDs.

Similar to XML document processing, XML data binding using XSDs is also available for many programming languages and environments. However, we will concentrate on how to auto-generate *C++* source code from XSDs using [32]. A compiler is used to parse an XSD and produce language-dependent definitions. The structure of XSDs is easily transferable into object-oriented technology where XSD types, built-in or user-defined, are expressed as classes and XSD properties or attributes are expressed as class attributes. These auto-generated classes usually contain class-attribute access functions for getting or setting a specific attribute as well as for serialization of class instances. The XSD-to-*C++* compiler at [32] is also capable of extracting documentation and annotations as well as transforming this information for processing with documentation processing software such as `doxygen` [7]. Listing 11 in appendix C shows test driver code in *C++* for creating the XML documents in listings 7–9 and listing 10, respectively. Listing 12 in appendix C shows a test driver that reads XML documents created by listing 11 or manually constructed XML documents adhering to the XSDs in listings 3–6. It is worth noting that both test drivers are relatively small, support standard *C++* idioms for object creation and manipulation, and show that such auto-generated source code can be integrated into existing software with relatively little effort.

XML data binding also supports generation of parser mappings. The auto-generated source code using [32] is usually generated by setting the XSD-to-*C++* compiler into a tree mode. However, it also possible to run the XSD-to-*C++* compiler in a parser mode which allows for creating interfaces that connect to stream-oriented XML parsing infrastructure. The main difference here is that instead of using class abstractions to handle complex data types the stream-oriented interfaces allow for reflection on the data types usually by providing callback mechanisms to client code. This would allow for inspection of unknown entities and potentially

retrieving their definitions from URIs embedded in the XSDs itself. It could also be used to *interpret* the structure of an XSD to derive representational aspects such as building GUI elements on the fly.

## 4 Discussion

XSDs allow for describing complex data-structure types. These structure types are built from a rich set of built-in primitives as well as by introducing user-defined types by extending existing types. This is very similar to building abstract data types in the object-oriented paradigm. However, XSDs are independent from specific object-oriented programming languages, development environments, or execution platforms. Mature tools for many programming languages and environments exist to automatically generate source code for processing, serialization, etc. as well as documentation for a chosen target platform. This allows for using XML documents as instances of XSDs to exchange data in a truly platform-independent way. The relationship between an XSD and XML instances is similar to the class-and-its-objects relationship in the object-oriented paradigm with the restriction that XSDs do not support methods or messages between objects. However, this restriction is of little consequence because XSDs allow for encoding behavior using the command software-design pattern [9].

We acknowledge that our example in section 3, including the source code in appendices A–C, is rather simplistic. However, the example already captures the essence of computational steering at a high level, at least for particle-based simulations, and was developed in a very short amount of time. XSDs are already successfully used for defining structure in complex problem domains such as computer graphics (X3D [16, 17]) or geographic-information processing (GML [15]). In addition to automatic source-code generation XSDs also allow for interoperability between different sets of XSDs as well as run-time reflection of XML instances. Using the namespace feature of XSD it is possible to encapsulate and disambiguate which types are intended for use in a schema file. The namespace `xmlns:xs="http://www.w3.org/2001/XMLSchema"` introduces a short-hand prefix for the XSD namespace itself; in our example this is set to `xs` but could be any other unique string. A custom namespace can be built by providing a `targetNamespace` attribute. This allows reusing of already existing XSDs without the need for reinvention. Serialized XML instances from an XSD also provide information about their original definitions. By using the `schemaLocation` attribute, run-time parsing of some unknown element can be resolved by fetching the (missing) schema definition to be used for interpreting the XSD instance of the formerly unknown element.

We believe that our approach has the potential for resolving problems inherent to traditional approaches to computational steering, introducing greater flexibility for both the simulation provider as well as the visualization provider or improving the maintainability of infrastructure. We are aware that a large amount of computational-steering software is in existence and we do not propose to replace it by starting from scratch. However, there already seems to be awareness in the community that a more general and flexible approach to controlling (numerical) simulations in general is desirable. For example Biddiscombe et al. [2, 3] describe an approach where the virtual-file driver interface in HDF5 [13] was used to exchange information between a simulation and its desired visual representation. While rather application-specific in its solution Biddiscombe et al. [3] also propose a generic software API, embedded into the HDF5 customization, that would allow to transfer steering-control information from the visual interface to the simulation. Using our approach a simulation provider could simply add an XSD of the control elements and the structure of the simulation's result to existing software. Advertising the simulation-specific XSD would allow the development of independent visualization software

even in the event of not being able to recompile the simulation software or if no appropriate execution platform is available. The same is true for the visualization side. Experiments in finding new, useful, or problem-domain specific solutions could be carried out by concentrating on the data descriptions (and their meaning) while at the same time pseudo-simulators could generate randomized dummy data adhering to the simulation constraints to evaluate such visualization experiments under a variety of conditions. In addition, because XSD instances are communicated by serialized XML documents a large variety of simulation-visualization connection types are supported (e. g., within the same process and between processes on the same or separate machines).

Automatic source-code generation from XSDs provides a tight coupling between a simulation type and possible visualization and control interfaces. The alternative approach of XML data binding using parser mapping has the potential to automate this. Because the parser reports elements back to client code (via callbacks) not only unknown elements can be resolved (by fetching and evaluating their XSDs) but also automated transformations are possible based on the element types found. This is probably easiest to see in the case of automatically creating a GUI layout from some XSD. All built-in XSD types can be directly transferred to standard GUI elements for display or manipulation. Restricted elements such as enumerations can be easily expressed as selection elements in a GUI (e. g., pull-down list, list boxes). Compound structures may trigger grouping mechanisms usually also allowing collapsing or inflating their content. Assuming that some simulation-specific types are known a priori relevant visualization elements can be selected. The result structure in our example consists of a continuous value (`timestamp`) and a list of particles. This information could be used to provide a two-element layout containing a continuous slider for selecting a time step and a widget capable of processing and visualizing 3d data in a (cartesian) coordinate system. Similar approaches have been already explored in the gViz library [5] and the eViz framework [20] and may serve as a valuable starting point for experimentation as well as baselines for comparison. Intuitively, it seems that the more elements relevant to computational steering are known (and/or agreed upon) the more general such on-the-fly visualization-interface building might become. However, it would be naïve to suggest we can provide a full solution to this problem without deeper insights into the problem domain and its requirements. The only definite advantage our approach provides here is the potential for flexible run-time inspection of complex data types in contrast to traditional software APIs.

## 5 Conclusions and Future Work

We have presented a data-centric approach based on extensible self-describing protocols for communication between simulation and visualization providers in computational steering. Using XML Schema definitions we developed a protocol for the control and result elements of a simple generic CFD simulation. Such XSDs allow for automatic source-code generation in many programming languages and environments as well as the extraction of documentation. This in turn provides simulation providers with the ability to communicate how a specific simulation can be controlled as well as how its results should be interpreted. Visualization providers also benefit by being able to choose the most appropriate visualization and interaction techniques. XSD also enables decoupling of support infrastructure because only the XSD needs to be transformed into code for use in software while the XSD itself is completely independent from specific software APIs, technologies, or execution environments.

Our approach has the potential to describe and use computational steering in a much more flexible way than hitherto. This flexibility reaches into topics as varied as long-term maintain-

ability, connectivity, execution-platform improvements, or innovative approaches to simulation and visualization itself.

Our simple example is exactly that: simplistic. While we believe that the high-level separation into a control and a result structure is valid for a large number of (numerical) simulations, we are unable to provide a formalization here. In part this stems from the fact that computational steering does not have a formalized definition or standards document. We are aware of the efforts of the working group for a simple API for grid applications (SAGA) to define a core API [10] for general grid computing. However, while valuable in providing problem-domain analysis and terminology, SAGA only advocates a software-API approach. We believe that our approach is more suitable to standardizing communication in computational steering and suggest that the development of such a consensus includes discussion and experimentation within the whole community.

The increased use of GPU architectures provides a challenge for our approach. GPU-based simulations do have a great appeal in that they provide faster evaluation per time step below certain problem sizes as well as that the simulation results are already available for further graphics processing on the GPU. Essentially, the simulation provider and visualization component are the same and the use of a generic data-centric communication approach will require further in-depth contemplation.

Manual deduction of appropriate visualization techniques from numerical simulations requires a high degree of expert knowledge in both the specific problem domain of a simulation as well as in the domain of visualization techniques. The ideal solution would be to be able to automatically deduce the inherent structure of a simulation, for both the set of control parameters and the result(s), and to select appropriate visualization techniques. Our data-centric protocol already provides a general approach to the structure-deduction problem for simulation control and its results and we are interested in further investigating this problem domain with respect to automatic deduction of appropriate visualizations techniques.

We believe that our data-centric approach using a structured and self-describing format for communication in computational steering provides a good starting point for further development that enables true platform independence as well as future-proof reusability.

## A Example XML Schemas

In this section we provide the XSD sources for $n$-component floating-point types ($n \in \{1, 2, 3, 4\}$) using simple inheritance (listing 3), a generic particle type providing position and velocity in 3d as well as a unique id and a type for a list of such particles (listing 4), base types for control and result structures suitable in the context of computational steering (listing 5), and specializations of the control and result types for a generic CFD simulation (listing 6), which extend the previously defined base types for control and result structures.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="float1">
    <xs:sequence>
      <xs:element name="x" type="xs:float"     />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="float2">
    <xs:complexContent>
```

```
      <xs:extension base="float1">
        <xs:sequence>
          <xs:element name="y" type="xs:float" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="float3">
    <xs:complexContent>
      <xs:extension base="float2">
        <xs:sequence>
          <xs:element name="z" type="xs:float" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="float4">
    <xs:complexContent>
      <xs:extension base="float3">
        <xs:sequence>
          <xs:element name="w" type="xs:float" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

**Listing 3:** An XML schema definition for one-, two-, three-, and four-component floating-point types, which is stored in `floats.xsd`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:include schemaLocation="floats.xsd" />

  <xs:complexType name="particle_t">
    <xs:sequence>
      <xs:element name="position" type="float3"         />
      <xs:element name="velocity" type="float3"         />
    </xs:sequence>
    <xs:attribute name="id"       type="xs:unsignedInt"
                                  use="required"        />
  </xs:complexType>

  <xs:element name="particle" type="particle_t" />

  <xs:complexType name="particle_list_t">
    <xs:sequence>
      <xs:element name="particle" type="particle_t"
                                  minOccurs="0"
                                  maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="particle_list" type="particle_list_t" />

</xs:schema>
```

**Listing 4:** An XML schema definition for a generic particle and a list of particles, which is stored in `particle.xsd`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
  <xs:complexType name="control_t">
    <xs:sequence>
      <xs:element name="type">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="start"  />
            <xs:enumeration value="stop"   />
            <xs:enumeration value="update" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="control" type="control_t"/>

  <xs:complexType name="result_t">
    <xs:sequence>
      <xs:element name="time" type="xs:time" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="result" type="result_t" />

</xs:schema>
```

**Listing 5:** An XML schema definition for steering control and result base types, which is stored in `steering.xsd`. The base control type always contains an enumeration for starting, stopping, or updating the simulation. The base result type always includes a time stamp with respect to the simulation time.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:include schemaLocation="particle.xsd" />
  <xs:include schemaLocation="steering.xsd" />

  <xs:complexType name="cfd_control_t">
    <xs:complexContent>
      <xs:extension base="control_t">
         <xs:sequence>
           <xs:element name="gravity" type="float3" />
           <xs:element name="force"   type="float3" />
         </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="cfd_control" type="cfd_control_t"/>

  <xs:complexType name="cfd_result_t">
    <xs:complexContent>
      <xs:extension base="result_t">
         <xs:sequence>
           <xs:element name="particle_list"
                       type="particle_list_t" />
         </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="cfd_result" type="cfd_result_t"/>

</xs:schema>
```

**Listing 6:** An XML schema definition for steering a generic CFD, which is stored in `cfd.xsd`.

## B Example XML Documents

In this section we provide the XML sources for XSD instances of control structures of type start (listing 7), update (listing 8), and stop (listing 9), and a result structure (listing 10).

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<cfd_control xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="cfd.xsd">
  <type>start</type>
  <gravity>
    <x>0</x>
    <y>0</y>
    <z>-9.81</z>
  </gravity>
  <force>
    <x>0</x>
    <y>0</y>
    <z>0</z>
  </force>
</cfd_control>
```

**Listing 7:** An XML document representing a control structure of type start as defined by the XSD in listing 6 and produced by the program in listing 11.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<cfd_control xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="cfd.xsd">
  <type>update</type>
  <gravity>
    <x>0</x>
    <y>0</y>
    <z>-9.81</z>
  </gravity>
  <force>
    <x>0</x>
    <y>1</y>
    <z>0</z>
  </force>
</cfd_control>
```

**Listing 8:** An XML document representing a control structure of type update as defined by the XSD in listing 6 and produced by the program in listing 11.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<cfd_control xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="cfd.xsd">
  <type>stop</type>
  <gravity>
    <x>0</x>
    <y>0</y>
    <z>0</z>
  </gravity>
  <force>
    <x>0</x>
    <y>0</y>
    <z>0</z>
  </force>
</cfd_control>
```

**Listing 9:** An XML document representing a control structure of type stop as defined by the XSD in listing 6 and produced by the program in listing 11. Note that the gravity and force attributes may or may not be used in the shutdown process but the definition of the control structure in listing 6 requires their appearance in the XML document.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<cfd_result xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="cfd.xsd">
  <time>00:00:00.0005</time>
  <particle_list>
    <particle id="0">
      <position>
        <x>0</x>
        <y>0</y>
        <z>0</z>
      </position>
      <velocity>
        <x>0</x>
        <y>0</y>
        <z>0</z>
      </velocity>
    </particle>

        ⋮

    <particle id="9">
      <position>
        <x>0</x>
        <y>0</y>
        <z>0</z>
      </position>
      <velocity>
        <x>0</x>
        <y>0</y>
        <z>0</z>
      </velocity>
    </particle>
  </particle_list>
</cfd_result>
```

**Listing 10:** An XML document representing a result structure as defined by the XSD in listing 6 and produced by the program in listing 11.

## C Example Test Drivers

In this section we provide the *C++* source code of simple test drivers (listings 11 and 12) for creating the XSD instances serialized to XML documents in listings 7–10 and processing XML documents based on the XSD of generic CFD simulation in listing 6.

```
#include <cstdlib>
#include <fstream>
#include <iostream>

#include <cfd.hpp>

int main(int, char*[])
{
  int result(EXIT_SUCCESS);

  try {
    // required for de-serialization
    xml_schema::namespace_infomap map;

    map[""].name   = "";
    map[""].schema = "cfd.xsd";

    float3 const g (0,0,-9.81);
    float3 const f0(0,0, 0);
    float3 const fy(0,1, 0);

    {
```

```
      cfd_control_t const ctrl(type::start, g, f0);
      std::ofstream       ofs ("../cfd_control_start.xml");

      cfd_control(ofs, ctrl, map);
    }

    {
      cfd_control_t const ctrl(type::update, g, fy);
      std::ofstream       ofs ("../cfd_control_update.xml");

      cfd_control(ofs, ctrl, map);
    }

    {
      cfd_control_t const ctrl(type::stop, f0, f0);
      std::ofstream       ofs ("../cfd_control_stop.xml");

      cfd_control(ofs, ctrl, map);
    }

    {
      cfd_result_t::particle_list_type particles;

      for (unsigned i(0); i < 10; ++i) {
        particles.particle().push_back(particle_t(f0, f0, i));
      }

      using time_type = cfd_result_t::time_type;

      cfd_result_t const result(time_type(0,0,0.0005), particles);
      std::ofstream      ofs ("../cfd_result.xml");

      cfd_result(ofs, result, map);
    }
  }

  catch (xml_schema::exception const& e) {
    std::cerr << e << std::endl;

    result = EXIT_FAILURE;
  }

  return result;
}
```

**Listing 11:** Test driver using auto-generated source code, provided by the include statement for `cfd.hpp`, to produce the XML documents in listings 7–10.

```
#include <cstdlib>
#include <iostream>

#include <cfd.hpp>

int main(int argc, char* argv[])
{
  int result(EXIT_SUCCESS);

  try {
    if (2 <= argc) {
      bool processed(false);

      try {
        if (!processed) {
          cfd_control_t const ctrl(*(cfd_control(argv[1]).get()));

          std::cout << argv[0] << ": " << ctrl << std::endl;

          processed = true;
        }
```

```
      }

      catch (xml_schema::unexpected_element const& e) {
        std::cerr << e << std::endl;
      }

      try {
        if (!processed) {
          cfd_result_t const result(*(cfd_result(argv[1]).get()));

          std::cout << argv[0] << ": " << result << std::endl;

          processed = true;
        }
      }

      catch (xml_schema::unexpected_element const& e) {
        std::cerr << e << std::endl;

        result = EXIT_FAILURE;
      }

      if (!processed) {
        std::cout << argv[0] << ": " << "unable to process '" << argv[1] << "'" << std::endl;
      }
    } else {
      std::cout << argv[0] << ": " << "no file name supplied" << std::endl;
    }
  }

  catch (xml_schema::exception const& e) {
    std::cerr << e << std::endl;

    result = EXIT_FAILURE;
  }

  return result;
}
```

**Listing 12:** Test driver using auto-generated source code, provided by the include statement for `cfd.hpp`, for reading and displaying the content of XML documents containing control or result structures as defined in listing 6.

## References


[1] G. Abram and L. Treinish. An Extended Data-flow Architecture for Data Analysis and Visualization. In *Proceedings of ACM SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 17–21. ACM, 1995. DOI: 10.1145/204362.204366.

[2] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali. Parallel Computational Steering and Analysis for HPC Applications using a ParaView Interface and the HDF5 DSM Virtual File Driver. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 91–100. Eurographics, 2011. DOI: 10.2312/EGPGV/EGPGV11/091-100.

[3] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali. Parallel Computational Steering for HPC Applications using HDF5 Files in Distributed Shared Memory. *IEEE Trans. Vis. Comput. Graphics*, 18(6):852–864, 2012. DOI: 10.1109/TVCG.2012.63.

[4] K. Brodlie, A. Poon, H. Wright, L. Brankin, G. Banecki, and A. Gay. GRASPARC – A Problem Solving Environment Integrating Computation and Visualization. In *Proceedings IEEE Visualization*, pages 102–109. IEEE, 1993. DOI: 10.1109/VISUAL.1993.398857.

[5] K. Brodlie, D. Duce, J. Gallop, Sagar M., J. Walton, and J. Wood. Visualization in Grid Computing Environments. In *Proceedings IEEE Visualization*, pages 155–162. IEEE, 2004. DOI: 10.1109/VISUAL.2004.112.

[6] F. Chatzinikos and H. Wright. Enabling Multipurpose Image Interaction in Modular Visualization Environments. In *Visualization and Data Analysis 2003*, pages 203–214. SPIE, 2003. DOI: 10.1117/12.474026.

[7] Doxygen – Generate documentation from source code. URL http://www.stack.nl/~dimitri/doxygen/.

[8] J. M. Favre and M. Valle. AVS and AVS/Express. In C. Johnson and C. Hansen, editors, *Visualization Handbook*, pages 655–672. Academic Press, 2004.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith. *A Simple API for Grid Applications (SAGA)*. Open Grid Forum, 2013. URL http://www.ogf.org/documents/GFD.90.pdf.

[11] Groovy Programming Language. URL http://groovy-lang.org/.

[12] E. R. Harold. *XML 1.1 Bible*. John Wiley & Sons, 3rd edition, 2004.

[13] HDF Group. URL http://www.hdfgroup.org/.

[14] IEEE:1278.1:2012. *Standard for Distributed Interactive Simulation: Application Protocols*. IEEE, 2012. IEEE Standard 1278.1-2012.

[15] ISO:19136:2007. *Geographic Information – Geography Markup Language (GML)*. ISO/IEC, 2013. International Standard ISO/IEC 19136.

[16] ISO:19775-1:2013. *Information Technology – Computer Graphics, Image Processing and environmental data representation – Extensible 3D (X3D) – Part 1: Architecture and Base Components*. ISO/IEC, 2013. International Standard ISO/IEC 19775-1.

[17] ISO:19776-1:2015. *Information Technology – Computer Graphics, Image Processing and Environmental Data Representation – Extensible 3D (X3D) Encodings – Part 1: Extensible Markup Language (XML) Encoding*. ISO/IEC, 2015. International Standard ISO/IEC 19776-1.

[18] S. G. Parker, D. M. Weinstein, and C. R. Johnson. The SCIRun Computational Steering Software System. In *Modern Software Tools for Scientific Computing*, pages 5–44. Springer Verlag, 1997. DOI: 10.1007/978-1-4612-1986-6˙1.

[19] S. M. Pickles, R. Haines, R. L. Pinning, and A. R. Porter. A Practical Toolkit for Computational Steering. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1833):1843–1853, 2005. DOI: 10.1098/rsta.2005.1611.

[20] M. Riding, J. Wood, K. W. Brodlie, J. M. Brooke, M. Chen, D. Chisnall, C. Hughes, N. W. John, D. M. Jones, and N. Roard. eViz: Towards an Integrated Framework for High Performance Visualization. In *Proceedings of UK e-Science All Hands Meeting*, pages 344–351, 2005. http://www.allhands.org.uk/2005/proceedings/papers/344.pdf.

[21] SAX Project. URL http://www.saxproject.org/.

[22] SCIRun. URL http://www.scirun.org/.

[23] R. van Liere, J. D. Mulder, and J. J. van Wijk. Computational Steering. *Future Generation Computer Systems*, 12(5):441–450, 1997. DOI: 10.1016/S0167-739X(96)00029-5.

[24] J. P. R. B. Walton. NAG's IRIS Explorer. In C. Johnson and C. Hansen, editors, *Visualization Handbook*, pages 633–654. Academic Press, 2004.

[25] W3C:XSLT1.0:1999. *XSL Transformations (XSLT) Version 1.0 – W3C Recommendation 16 November 1999*. World Wide Web Consortium (W3C), 1999. URL http://www.w3.org/TR/1999/REC-xslt-19991116.

[26] W3C:XML1.1:2006. *Extensible Markup Language (XML) 1.1 (Second Edition) – W3C Recommendation 16 August 2006*. World Wide Web Consortium (W3C), 2006. URL http://www.w3.org/TR/2006/REC-xml11-20060816/.

[27] W3C:XSLT2.0:2007. *XSL Transformations (XSLT) Version 2.0 – W3C Recommendation 23 January 2007*. World Wide Web Consortium (W3C), 2007. URL http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[28] W3C:XSD1.1P1:2012. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures – W3C Recommendation 5 April 2012*. World Wide Web Consortium (W3C), 2012. URL http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/.

[29] W3C:XSD1.1P2:2012. *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes – W3C Recommendation 5 April 2012*. World Wide Web Consortium (W3C), 2012. URL http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/.

[30] W3C:DOM4:2015. *W3C DOM4 – W3C Recommendation 19 November 2015*. World Wide Web Consortium (W3C), 2015. URL http://www.w3.org/TR/2015/REC-dom-20151119/.

[31] H. Wright, R. H. Crompton, S. Kharche, and P. Wenisch. Steering and Visualization: Enabling Technologies for Computational Science. *Future Generation Computer Systems*, 26(3):506–513, 2010. DOI: 10.1016/j.future.2008.06.015.

[32] XSD - XML Schema to C++ Data Binding Compiler - Project Page. URL http://www.codesynthesis.com/projects/xsd/.