Defining the Fluid Framework

Anthony Jones
Knowledge Engineering Group
School of Engineering and Applied Science
Aston University
Birmingham, B4 7ET, UK
d.cornford@aston.ac.uk

Dan Cornford

Abstract

In this position paper we present the developing Fluid framework, which we believe offers considerable advantages in maintaining software stability in dynamic or evolving application settings. The Fluid framework facilitates the development of component software via the selection, composition and configuration of components. Fluid's composition language incorporates a high-level type system supporting object-oriented principles such as type description, type inheritance, and type instantiation. Object-oriented relationships are represented via the dynamic composition of component instances. This representation allows the software structure, as specified by type and instance descriptions, to change dynamically at runtime as existing types are modified and new types and instances are introduced. We therefore move from static software structure descriptions to more dynamic representations, while maintaining the expressiveness of object-oriented semantics. We show how the Fluid framework relates to existing, largely component based, software frameworks and conclude with suggestions for future enhancements.

1 Introduction

Component based software development (CBSD) focusses on the use of existing independent binary 'parts' called *components* to build software applications[10]. CBSD maximises implementation reuse, amortizing development costs over many applications. CBSD is realised by component frameworks, which provide an environment for the development, deployment, and assembly of components. Component frameworks typically stipulate a component model, which defines the requirements for component implementations to be compatible with (and thus deployable within) the framework, and provide the means by which components may communicate at runtime. Composition languages allow component software developers to describe the structure of component software consisting of component deployments and their connections. The high-level composition languages, often providing an abstraction of an underlying component model, are used to drive the deployment, connection, and thus creation of component software.

We are developing a prototype component framework and composition language, collectively referred to as Fluid[5]. Fluid's composition language includes an expressive object-oriented type system, extending existing technologies first applied in the computer games industry, which focus on the reuse of component implementations and high-level composition descriptions.

This paper is organised as follows: we first define the Data Driven Programming (DDP) paradigm in Section 2, and then briefly describe in Section 3 how Fluid builds upon existing DDP concepts. Section 4 outlines our contributions to the field of stable and adaptable software. Finally, Section 5 presents a discussion of our work, including its relation to relevant research in the field of CBSD and possible future extensions.

2 Data Driven Programming

DDP is extensively applied to game development, where it provides a software structure that is flexible to changing requirements and is accessible to non-programmers via a variety of tools. Examples appear in related literature[8, 9]. In this paper, we refer to Data Driven Programming as *Object Oriented* Data Driven Programming (OODDP) in order to highlight its use of features from the object-oriented paradigm (such as type definition, inheritance, and instantiation), and to distinguish it from data-driven concepts used elsewhere (for example, in operating systems[7]). Fluid also uses an *object-centric* approach to implementing data driven programming, which has closer semantics to the

object-oriented paradigm, rather than the *data-centric* approach, which focusses on object attributes, and is often implemented using a database.

In describing his motivations for developing OODDP, Scott Bilas states[2]:

> To meet changing design needs, one can't just data-drive the object properties, one must data-drive the structure (schema) of the objects.

The term *schema* here refers to the objects, the objects' attributes, and the relationships between objects. In OODDP, this schema may be data-driven by replacing static inheritance relationships with dynamic composition relationships and then driving dynamic composition at runtime using data.

For example, consider the following simplified object oriented software structure, which provides the basis for a running example that will be referenced throughout the paper: A Vehicle type forms the root of an object oriented inheritance hierarchy, providing common behaviour (e.g. the ability to belong to a traffic network simulation, and to travel between locations in that network), and common attributes (such as engine size, average speed and so on). The PassengerVehicle type extends the Vehicle type by providing seats for passengers, plus the behaviour for boarding and alighting. The Car type extends PassengerVehicle in order to provide a concrete type for cars and similar vehicles, while the LightAircraft type provides a PassengerVehicle that can transport passengers by air. If this scenario were implemented in code, runtime objects would be instances of the child classes - that is, one would declare instances of the Car and LightAircraft types as part of the code, and each runtime object would be identified via its corresponding variable declaration.

OODDP replaces inheritance with composition in order to replace static relationships in the software structure with dynamic relationships that can be driven by data at runtime. In order to achieve this, related classes in the inheritance hierarchy must be represented using independent classes, which are then combined to provide the functionality previously represented by the static software structure. For example, the PassengerVehicle class no longer inherits from Vehicle, but is written as an independent class. If the parent (Vehicle) class from our earlier example provides any common behaviour to its children, then it will also be represented by an independent class that encapsulates this commonality. If the OODDP scenario were implemented in code, runtime objects would be represented by simple containers of instances of the independent child class types. For example, an instance of the Car class would be represented in the OODDP scenario as the composition shown in Figure 1, incorporating the functionality of the Vehicle class, plus that provided by the SeatingCapacity class. Each OODDP instance (that is, the container instance representing the corresponding class instance) is uniquely identified via a runtime identifier.

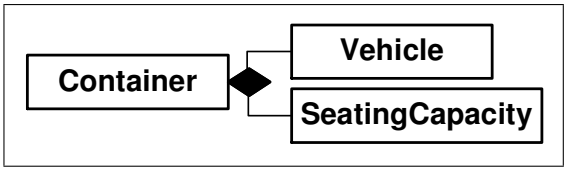

Figure 1. The PassegengerVehicle type is represented as an OODDP hierarchy of container and independent class instances.

In OODDP, each hierarchy of container object and contained class instances may be described using an OODDP *type description*. Type descriptions drive the runtime instantiation and composition of container and class instances in order to form an OODDP hierarchy as illustrated by the example in Figure 1.

Each type description includes an identifier for the OODDP type currently being described, and specifies which classes should be instantiated and inserted into the container. Each specified class instance is also given an identifier so that the various parts of an OODDP hierarchy may be uniquely identified. Class instance specifications may also include an optional configuration, which is used to customize the class instance and can be regarded as synonymous with class constructor parameterization (although in practice the configuration may not necessarily be used as such).

Figure 2, written in XML, describes the type PassengerVehicleType, which contains a Vehicle class instance and a SeatingCapacity class instance, both of which include configuration elements.

Furthering its incorporation of the object-oriented paradigm, OODDP re-introduces inheritance via manipulations

```
<Type id="PassengerVehicleType">
 <Vehicle id="vehiclePart">
  <!— Vehicle configuration —>
  <AverageSpeed>50mph</AverageSpeed>
  <!— etc —>
 </Vehicle>
 <SeatingCapacity id="seatingPart">
  <!— SeatingCapacity configuration —>
  <Seats>5</Seats>
  <!— etc —>
 </SeatingCapacity>
</Type>
```

Figure 2. An example OODDP type description for the OODDP hierarchy from Figure 1.

```
<Type id="LightAircraftType"
 parent="PassengerVehicleType">
  <!-- LightAircraftType inherits seatingPart from the
   PassengerVehicleType OODDP type description -->
  <!-- LightAircraftType provides its own description for
    the vehiclePart, which overrides that in
   PassengerVehicleType -->
  <Vehicle id="vehiclePart">
   <AverageSpeed>150mph</AverageSpeed>
  </Vehicle>
  <!-- LightAircraftType extends PassengerVehicleType by
   including an instance of the FixedWingFlight class -->
  <FixedWingFlight id="flightPart">
   <Ceiling>15000 ft</Ceiling>
  </FixedWingFlight>
</Type>
```

Figure 3. An OODDP type description for the LightAircraftType type, which makes use of OODDP inheritance to both include and extend the definition of the PassengerVehicleType type description given in Figure 2.

```
<Instance id="myCar" parent="PassengerVehicleType"/>
```

Figure 4. Declaring an instance of the OODDP PassengerVehicleType type description, as given in Figure 2.

on its type descriptions. New OODDP type descriptions can be described based on existing OODDP type descriptions, allowing for description re-use and thus more expressive power for OODDP software designers. In a similar way to traditional object-oriented paradigms, OODDP inheritance forms a relationship between a pair of parent and child type descriptions. For example, Figure 3 makes use of a parent type description (PassengerVehicleType from Figure 2) and describes a child type description (LightAircraftType). The LightAircraft type describes PassengerVehicleType type that uses flight as its mode of travel, as represented by its additional FixedWingFlight class instance.

An OODDP type description may be instantiated via the definition of an OODDP *instance description*. An instance description typically has a similar appearance to an OODDP type description, and provides an identifier for the instance. Figure 4 describes an instance of the PassengerVehicleType type given in Figure 2. After OODDP inheritance is applied, the 'myCar' instance description will include the contents of the PassengerVehicleType type description as given in Figure 2.

OODDP inheritance, as described above, may also be applied to instance descriptions. For example, Figure 5 describes another instance of the PassengerVehicleType type given in Figure 2, although this time we override the SeatingCapacity class instance with a custom instance having a different configuration.

```
<Instance id="anotherCar" parent="PassengerVehicleType">
 <SeatingCapacity id="seatingPart">
  <Seats>8</Seats>
 </SeatingCapacity>
</Instance>
```

Figure 5. Declaring another instance of the OODDP PassengerVehicleType type description, as given in Figure 2, although in this example the instance description overrides the PassengerVehicleType type description by providing a custom SeatingCapacity class description for a custom number of seats.

OODDP has become a popular methodology for game developers, who use OODDP to replace static inheritance relationships in software structures with ownership relationships that can be dynamically driven by data at runtime. However, OODDP remains an *applied* methodology, with no formal theory and no explicit connections to related fields. In the Fluid framework, we form a link between OODDP and CBSD by applying an adapted OODDP methodology to CBSD in order to provide an expressive composition language with high-level object-oriented semantics.

3 The Fluid Framework

The Fluid framework builds upon OODDP, in order to provide a component language which facilitates the definition of software structure, via the runtime **selection**, **composition** and **configuration** of components. We use the definition of component as given by Szyperski[10]:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

In the context of Fluid, a component is defined as a single class, synonymous with the OODDP classes (e.g. Vehicle) from Section 2, although such classes may provide a façade to more complex or multi-class components.

To facilitate **component selection**, we assume an existing repository of deployable components, where each Fluid component implementation is provided as a dynamically linked library (DLL) exporting two functions: a *create* function to create an instance of the wrapped component class, and a *destroy* function to destroy a given component instance. Each component implementation is accompanied

by a specification document, which describes the components's exposed (or *exported*) and required (or *imported*) functions and events, in order to facilitate inter-component communication and collaborative runtime behaviour.

For example, Figure 6 provides the Vehicle component's specification, which states that each Vehicle component instance exposes a function named 'fuelLevel' (returning how much fuel the given Vehicle instance has in its fuel tank). Additionally, each Vehicle component instance subscribes to an event named 'onRefuel' (which is triggered when a given Vehicle is told to refuel by the traffic network).

Exported function descriptions include an identifier, which allows other component instances to refer to, and thus make use of, the exported functionality. Imported function descriptions include an identifier by which the imported function is known to the component's implementation. Both exported and imported function descriptions include a type description providing information related to the function's signature, and this information is used to perform type-checking during component composition.

Event descriptions are similar to function descriptions, with the exception that events have publisher/subscriber semantics. A component instance publishes an event by exporting it. The event's element in the component specification document must provide a name for the event, plus the type of the event's *payload*. An event's payload is the single parameter passed to subscribing event handler functions. When the event is triggered by its owning component instance, its payload will be passed to the Fluid framework, which will in turn invoke each subscribing event handler, passing a copy of the payload as the function's only parameter. A component instance may subscribe to a published event by importing it; the subscribing component instance must include an appropriate event handler function.

Each component implementation must also be accompanied by a configuration specification, which describes what the component implementation may expect in terms of instantiation configuration. For example, a configuration specification may stipulate that a given configuration must contain an 'AverageSpeed' element providing the average speed of each Vehicle instance in the traffic network. Component configurations are validated against their corresponding configuration specification prior to component instantiation. As Fluid makes extensive use of XML, our configuration specifications are written as XML schemas, and we leverage XML schema validation to provide the required runtime checks. Figure 7 illustrates the selection of two components as part of an OODDP type description. The components in this example are identified by a 'component' attribute, and are located using a relative path to the required DLL implementation.

In Fluid, **component composition** is the runtime process of instantiating components and deploying interconnected

```
<Specification id="Vehicle">
 <Exports>
  <Function id="fuelLevel">
   <Signature>
    <!— type information for the "fuelLevel" function
    signature —>
   </Signature>
  </Function>
  <!— other exports, including additional functions and
   events —>
 </Exports>
 <Imports>
  <Event id="onRefuel">
   <Payload>
    <!— type information for an expected "payload"
    (parameter) to be delivered to this component when
    the "onRefuel" event occurs —>
   </Payload>
  </Event>
  <!— other imports, including additional
   functions and events —>
 </Imports>
</Specification>
```

Figure 6. A Fluid component specification for the Vehicle component. Vehicle exports one function and imports (or subscribes to) one event. Type information, and additional possible imports and exports, are not given due to space restrictions.

OODDP hierarchies. OODDP hierarchies are deployed by associating the container instance with the instance description's identifier attribute. Deployed OODDP hierarchies may be subsequently referred to via their identifier attributes. For example, the OODDP instance given in Figure 4 is associated with the identifier 'myCar', and its component parts may be referred to as 'myCar.vehiclePart' 'myCar.seatingPart' and so on.

OODDP hierarchies are interconnected by allowing component instances to refer to, and thereby make use of, the exported functions and events of other component instances. For example, given the component specification document in Figure 6 (and corresponding specification documents for any other components), Figure 8 illustrates the

```
<Type id="PassengerVehicleType">
 <Vehicle id="vehiclePart"
  component="components/vehicle.dll"/>
 <Passengers id="seatingPart"
  component="components/seatingCapacity.dll"/>
</Type>
```

Figure 7. A Fluid OODDP type description, illustrating component selection. The paths given for the component attributes correspond to component implementation DLLs in Fluid's component repository.

```
<Type id="PassengerVehicleType">
 <Vehicle id="vehiclePart">
  <Imports>
   <Event eventHandler="onRefuel"
    event="trafficNetwork.onRefuel"/>
  </Imports>
  <Configuration>
   <AverageSpeed>50mph</AverageSpeed>
  </Configuration>
 </Vehicle>
 <SeatingCapacity id="seatingPart">
  <Imports/>
  <Configuration>
   <Seats>5</Seats>
  </Configuration>
 </SeatingCapacity>
</Type>
```

Figure 8. A Fluid OODDP type description, illustrating component composition (Import elements) and configuration (Configuration elements).

connection of the Vehicle type's imported 'onRefuel' function to the exported function (similarly named 'onRefuel') of another component.

Component configuration takes place during component instantiation and composition. Each component selection described in an OODDP type description may include a number of configuration elements. For example, the PassengerVehicleType type given in Figure 7 provides a configuration for both of its Vehicle and SeatingCapacity components' selections. During component composition, but prior to the component selection DLL's create function being called, the given configuration elements are checked against the component selection's configuration specification document. If a conflict is found between the given configuration elements and the component selection's configuration specification document, then a component instance will not be created and component composition fails. Otherwise, the configuration elements are passed to the component selection DLL's create function to configure the construction of a component instance. The resulting configured component instance will form part of the OODDP hierarchy being generated by the enclosing component composition process.

4 Contributions to Stable and Adaptable Software

In many fields of software engineering, stability in one place provides adaptability in another. In object-oriented approaches, interfaces represent a fixed contract, while derived implementations provide polymorphic behaviour via type inheritance. In component based approaches, a common framework embodies a predetermined component model, while component composition may change in order to meet varying functional and non-function requirements.

Fluid offers stability via its component model and composition framework, and provides an expressive composition language for application adaptability. Using Fluid's OODDP manipulations, one can easily define a family of component selections, compositions and configurations. The resulting OODDP type system may be rapidly altered, via object-oriented overriding, to drive runtime software structure. We therefore move from static software structure descriptions to more dynamic representations, while maintaining the expressiveness of object-oriented semantics.

We feel that Fluid is particularly suited to situations where its OODDP representations provide a valuable additional level of abstraction. By encapsulating repeated patterns of component composition and parameterization, we isolate change to the definition of overriding deltas in derived OODDP types. The resulting information hiding may make adaptable software more accessible to non-specialist programmers and visual tools.

5 Discussion

Due to its use of an XML-based language for component description and composition, Fluid is perhaps most closely related to CoML [3]. Fluid's composition language shares CoML's design principles, particularly the use of an XML-based declarative language with connection-oriented programming model. Fluid builds upon these principles by introducing a composition language type system via OODDP manipulations, and instance configuration (parameterization) during component composition. Furthermore, CoML includes function invocation and event response as part of the composition language, while Fluid relies upon black-box components to determine runtime application behaviour. We have yet to explore the implications of including a behavioural aspect in Fluid's composition language.

Fluid's incorporation of OODDP emphasizes the use of component configuration to drive runtime behaviour. *Component hooks* [1] place a similar emphasis on parameterization, and operate at a similar level of abstraction using object-oriented traits such as inheritance and prototype-based operations such as instance extension. However, whereas component hooks facilitate white-box development time parameterizations, Fluid operates on black-box components at composition time.

Fluid's composition language may be considered similar in appearance to architecture description languages (ADLs) such as Darwin[6], which are used to describe architectural design as a communication tool for further development as part of the software development process. ADLs operate at a higher level than composition languages, which describe the deployment and connection of components in order to

drive the runtime structure of the software during application execution.

Additionally, Fluid's composition language provides the means by which dynamic composition relationships in the software structure may be described using object oriented inheritance semantics. While Fluid's composition language may be used during application execution to drive the *initial* structure of component software, the same representations may be used to dynamically manipulate the software structure at runtime at a high level of abstraction.

Because of its ability to modify software structure at runtime in response to changing application state, the Fluid framework may have a number of potential applications in Dynamic Data Driven Application Systems[4] (DDDAS). DDDAS use data obtained via measurements to inform an internal representation or simulation of a given domain. The simulation may be used to dynamically adapt the measurement process in order to predict how the domain may change and what its future state may be. While typical DDDAS adapt the measurement process via the modification of existing application parameters and algorithms, a Fluid DDDAS would be able to dynamically change or introduce new runtime behaviour by manipulating the composition of the application's software structure.

The Fluid framework is currently being developed, and so our immediate goal is to complete its implementation. We will apply the Fluid framework to visualization and simulation in the context of Geographic Information Systems as a proof of concept. In the longer term, we aim to enhance the Fluid framework with an additional level of abstraction that can be used to express evolving software requirements. We thus hope to be able to apply the Fluid framework to evolving and dynamic applications: for example, a traffic simulation that dynamically adjusts its software structure (via OODDP manipulations) to incorporate additional vehicle types and behaviours, for example governed by revised traffic regulations, that were not in the original design. A further consideration is how requirements could be represented in Fluid's OODDP descriptions, and if such representations would allow us to incorporate model-driven approaches.

Finally, we are also interested in looking at dynamic situations where the software structure evolves according to its own output, allowing the software to guide its own evolution according to predefined goals or requirements. In this case we extend methods from DDDAS allowing the measurement process to observe some defined aspects of the software reliability itself, and the run-time evolution then being guided to maintain software robustness in this dynamic setting.

CBSD focusses on the development of component software via the assembly of existing binary implementations with collaborative behaviour. The flexibility of component deployment, typically driven by the use of composition languages, provides component software developers with an opportunity to adapt an otherwise static software structure to changing application requirements during software design and component deployment.

By bridging the gap between CBSD and OODDP, we introduce additional flexibility and adaptability to component software by allowing software structure to be modified dynamically in response to changing application requirements at runtime.

Acknowledgements

We would like to thank Rami Bahsoon for his help with improving the communication of the ideas presented in this paper. We would also like to thank the reviewers for their constructive input regarding the paper content.

References

[1] U. Aßmann. Beyond generic component parameters. In *CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 141–154, London, UK, 2002. Springer-Verlag.

[2] S. Bilas. A data-driven game object system. In *Game Developers Conference Proceedings*, 2002.

[3] D. Birngruber. CoML: Yet another, but simple component composition language. In *Proceedings of First Workshop on Composition Languages*, Vienna, Austria, Sept. 2001.

[4] F. Darema. Dynamic data driven applications systems: A new paradigm for application simulations and measurements. In M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *International Conference on Computational Science*, volume 3038 of *Lecture Notes in Computer Science*, pages 662–669. Springer, 2004.

[5] A. Jones and D. Cornford. Advanced data driven visualisation for geo-spatial data. In V. N. Alexandrov, G. D. van Albada, P. M. Sloot, and J. Dongarra, editors, *Computational Science ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006. Proceedings, Part III*, volume 3993, pages 586 – 592, 2006.

[6] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[7] E. S. Raymond. *The Art of Unix Programming*, chapter 9, pages 249–250. Addison-Wesley, September 2003.

[8] B. Rene. Component based object management. In K. Pallister, editor, *Game Programming Gems 5*, chapter 1.3, pages 25–37. Charles River Media, February 2005.

[9] C. Stoy. Game object component system. In M. Dickheiser, editor, *Game Programming Gems 6*, chapter 4.6, pages 393–403. Charles River Media, Inc., Rockland, MA, USA, February 2006.

[10] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.