**Component Coordination in GLOO**

Jean-Guy Schneider and Markus Lumpe  
Faculty of Information & Communication Technologies  
Swinburne University of Technology  
P.O. Box 218, Hawthorn, VIC 3122, AUSTRALIA  
\{jschneider,mlumpe\}@swin.edu.au

**Abstract**

Incorporating components from a number of different sources into a given application is generally considered to be a non-trivial activity. Over the years, various coordination mechanisms have been proposed to tackle this problem. However, even today, the question remains how to best link these coordination mechanisms with an underlying programming paradigm without losing flexibility. A suitable technique to address this issue is language composition, enabling us to “fine-tune” a programming language on demand. In this compositional approach, we can add new features to the language as we go, within user-defined regions, and without polluting the underlying paradigm as a whole. To test the effectiveness of this technique, we explore a small stream processing framework and its corresponding coordination abstractions in this paper. More precisely, we report on our insights into using GLOO, a functional composition language, for the definition of extensible coordination abstractions that, through the composition of the concepts proxy, method pointer, and Pernici’s “objects with roles” constitute, collectively, an attractive means to capture and denote inter-component interactions in a user-centric and domain-specific way.

1. Introduction

The Webster dictionary defines coordination as “the harmonious functioning of parts for effective results.” When applied to software, coordination is the process of managing dependencies between computational entities [17]. There are two unique aspects to this process: components that encapsulate well-defined functionality and interface code that governs the interaction between these components [7]. The behavior of components can naturally be captured within the framework provided by mainstream, object-oriented programming languages [10]. But what are the effective ways to express interface code? We contend, interface code is best denoted in extensible domain sub-languages [10], [11], [12] that offer both a high-level, user-centric domain vocabulary and a mechanism to grow the language on demand in order to adapt to changing coordination requirements.

We have been developing GLOO, a functional composition language [10], [11], [12], to experiment and reason about compositional and extensible domain sub-languages. GLOO is based on the \(\lambda F\)-calculus [9], which aggregates dynamic binding, explicit namespaces [1], and a foreign code gateway to form the semantic foundation for an open-ended programming model [8] for software composition. However, rather than providing a rich set of predefined language abstractions, GLOO offers an extension mechanism that enables developers to define their own, user-centric constructs in order to enrich the underlying language on demand. In other words, GLOO arms component architects and component assemblers with a scaffolding giving them the means to define a domain model and the rules governing the component interactions in that domain. The result is a user-centric domain vocabulary that in itself is both composable and extensible.

In line with its open-ended programming model, GLOO does not offer any specific abstractions related to component coordination by default. However, the ability to define a user-centric domain vocabulary empowers us to acquire corresponding language support when needed. In particular, we require abstractions that enable us to represent components as first-class entities and language facilities to “glue” them together. We champion language composition as a technique to enrich a given programming language with additional features [12]. More precisely, rather than stockpiling an ever-increasing variety of concepts in a monolithic language, in a compositional extension model developers adjust the actual level of language support to the specific needs of the targeted domain on demand.

Defining language support for component coordination is no exception. We start with the Language of Java Services [11] and the Language of Namespaces and Traits [12]. The former defines mechanisms to seamlessly integrate both Java classes and Java objects into GLOO, whereas the latter defines an object-oriented vocabulary for a class-based programming model. These two domain sub-languages, together with GLOO’s extension model, allow us to construct a coordination vocabulary in which external components become first-class GLOO values. Access to external components is through proxy classes [6] that offer an exogenous control...
style [3] for the specification of the required coordination interface code.

The rest of this paper is organized as follows: in Section 2 we provide a brief overview of the core elements of GLOO as they relate to a suitable language support for component coordination and illustrate our approach of domain sub-language composition in Section 3. We continue by presenting a case study for a stream coordination scheme and its realization in GLOO in Section 4, followed by a discussion of possible interpretations and consequences in Section 5. We conclude this paper in Section 6 with a summary of our main observations and some pointers to future work.

2. The GLOO Programming Model

Conceptually, GLOO shares many similarities with Scheme [5], Haskell [4], CLOS [20], Ruby [21], Tcl [16], and the compiler description language CDL [8]. GLOO is a strict, statically scoped, functional programming language that provides a declarative programming paradigm with exogenous control style [2] to separate the computational from the compositional aspects of defined software abstractions [12].

The main programming entity in GLOO is a specification unit that provides a scope for the definition of a value (or component), which itself can be recombined with additional values (or components) defined in other specification units. Consider Listing 1, which shows the specification unit defining the coordination vocabulary used in this paper. In the declaration portion of the let-block, we import the core object-oriented language support as defined by the Language of Namespaces and Traits. We also include support for the definition of mini parsers [12] that allows us to define a new variant of the syntactic category method, which now has a provision for the abstract modifier. Finally, we define ExtendRoot, a parsing function that implements a partial class declaration and introduce a new modifier flag ABSTRACT.

The value of this specification unit is an explicit namespace [1], defining the desired ontology mapping for the coordination vocabulary. The namespace definition is preceded by two calls to addPermissibleState, a mini parser control function that enables occurrences of the keyword abstract to appear before a method declaration (LOCS.SUPER_SEEN and LOTS.IN_TRAIT are the corresponding explicit parser states [12] of the Language of Namespaces and Traits in which we wish to accept occurrences of abstract).

The actual namespace is constructed in two steps. First, we compose CoreLanguage with ObjectModel giving precedence to the bindings of CoreLanguage. As a result, we obtain a core nomenclature for a class-based, object-oriented programming model. Second, we define a set of ontology mappings that introduce the new keyword abstract, refine the syntactic category method, and translate the desired core coordination vocabulary to its corresponding object-oriented interpretation.

The body of method makes use of two additional GLOO features: computable binders and explicit lazy arguments. The former allows for a name discovery at runtime. For example, the term {LOCS.MODIFIER} evaluates to a label, whose name is defined in a sub-language of the Language of Namespaces and Traits. LOCS stands for Language of Classes, a basic Java-like programming model upon which the Language of Namespaces and Traits is built [12]. The lazy modifier $, on the other hand, converts the actual argument for parameter MethodId into an expression tree. The primary use of expression trees is to defer the evaluation of arguments to functions until their value is actually being required. Expression trees are like quoted expressions in

```
let
    load "System/Services.lf"
    @ObjectModel =
    load "Meta-Level/OpenExtensibleImpClass.lf"
    @CoreLanguage =
    load "Domain-Level/LanguageOfNamespacesAndTraits.lf"
    load "Domain-Level/MiniParsers.lf"
    ExtendRoot =
        (\AST::
            (\$Name:: Class AST Name super Object of Root))
    ABSTRACT = LOCS.PROTECTED + 1
    in
    addPermissibleState abstract LOCS.SUPER_SEEN;
    addPermissibleState abstract LOTS.IN_TRAIT;
    |
        (\ObjectModel # CoreLanguage ||,
            abstract =
                LOCS.modifier abstract ABSTRACT
                "Cannot redefine member abstract!"
                "Illegal abstract modifier specification!",
            method =
                \AST:: if (AST.{ LOCS.MODIFIER } == ABSTRACT)
                (\$MethodId::
                    method AST MethodId
                    (\():: error "illegal abstract method!")
                    (method AST)),
                Role = Trait,
                endRole = endTrait,
                BaseConnector = ExtendRoot,
                endConnector = endClass,
                Component = ExtendRoot,
                endComponent = endClass,
                null = 0
        )
end
Listing 1. Coordination Abstractions in GLOO.
```
Scheme [5]. In the case of method, the expression tree of Name provides us with the string representation for the name of the method being defined.

The feature that distinguishes GLOO from other language models the most is incremental refinement [9], a linguistic means that allows for the occurrences of unbound names in terms. Unbound names are not to be confused with free names, which refer to some binding occurrence in the current scope. GLOO’s unbound names act as hooks for additional behavior (much like MIME-types in Web-browsers). The current scope is not required to provide a corresponding binder. However, a receiving scope, say b, of a term a may provide some definitions for unbound names occurring in a. In such a case, b incrementally refines a, written a[b] [9].

Considering again the declaration section of Listing 1, we load the unit OpenExtensibleImpClass.lf (i.e., Open Extensible Imperative Class [12]) before the unit LanguageOfNamespacesAndTraits.lf (i.e., the Language of Namespaces and Traits) and assign their values to the names ObjectModel and CoreLanguage, respectively. The symbol @ in front of both names turns these names into binders for their respective specification unit, that is, we add the namespaces of OpenExtensibleImpClass.lf and LanguageOfNamespacesAndTraits.lf to the current scope and provide access to them through the declared binders.

One of the more subtle aspects of the Language of Namespaces and Traits is that its abstractions are defined without a predefined object model [12]. More precisely, the Language of Namespaces and Traits defines a set of open class-based programming artifacts that we have to compose with a suitable object model. We aim at a coordination model with exogenous control semantics [2]. This coordination style requires state, which is naturally supported by a Java-like, imperative object model. Although GLOO is a pure functional programming language, its extension mechanism allows us to define stateful programming abstractions [12]. However, the state-altering behavior is restricted to well-defined scopes similar to the way state monads work [22]. Using an appropriate stateful language extension, we can define an imperative object model [12], [13] and compose it with the Language of Namespaces and Traits to yield core language support for the specification of coordination interface code. More precisely, Open Extensible Imperative Class serves as an incremental refinement of the Language of Namespaces and Traits [12]. Therefore, the sequence

@ObjectModel =
load "Meta Level/OpenExtensibleImpClass.lf"
@CoreLanguage =
load "Domain Level/LanguageOfNamespacesAndTraits.lf"

yields a context

CoreLanguage[ObjectModel]

that establishes Open Extensible Imperative Class as the underlying object model of the Language of Namespaces and Traits for the definition of our coordination vocabulary.

3. Towards a Coordination Model

The specification of language support for new programming features requires different levels of abstraction. A typical scenario [13] comprises three layers: a meta-level, a domain level, and a user level. Collectively, all layers contribute to a clear separation of concerns in the definition of the resulting domain abstractions. However, the granularity of the abstractions within those layers differs from domain to domain.

At the meta-level we define the language support required to integrate new data types into the GLOO environment as first-class values. The computational entities whose activities we wish to coordinate are components, in particular, Java components. These entities are represented by Java objects at runtime. Incorporating these objects in the GLOO environment at the meta-level takes a Java wrapper and a GLOO wrapper, which together build a Java bridge to the underlying component. The Java wrapper encapsulates the component’s behavior and establishes an interface to the desired features, also including provisions for a proper exception handling. The GLOO wrapper, on the other hand, defines a language-neutral interface (through the Language of Java Services [11]) to the Java wrapper and hence yields an abstraction that enables us to treat components as first-class entities.

The domain level acts as a mediator between the meta-level and the user level and incorporates all required domain abstractions. In particular, we set out a desired component model, establish the rules governing connector specification and composition, and designate the newly-required core domain vocabulary. Component coordination can naturally be expressed using the object-oriented paradigm and the Language of Namespaces and Traits provides a corresponding programming model. Furthermore, as discussed in Section 2, we need to compose the Language of Namespaces and Traits with an object model that caters to the specific demands of component coordination. Such a model is Open Extensible Imperative Class [12] that defines an imperative object (or component) model supporting modular class extensions [14], [15], [23]. Modular class extension permit a non-invasive refinement of existing object-oriented domain abstractions. Using class extensions, we can assign roles [18] to component connectors that provide us with a suitable linguistic means to explicitly control connector capabilities.

The composition of the Language of Namespaces and Traits and Open Extensible Imperative Class constitutes the core domain vocabulary for component coordination in an object-oriented fashion. At the user level, we refine this vocabulary in order to obtain a nomenclature that facilitates the specification of interface code. In particular, we define
a set of ontology mappings that associate object-oriented features with coordination concepts. Such mappings range from simple translations between names to more complex mapping functions and mini parsers that introduce new syntactic elements and refine existing semantics definitions, respectively. The degree of the refinement at the user level varies with user needs and requirements. However, the refinements at the user level always aim at a user-centric domain vocabulary that offers the best possible match for the targeted domain.

4. Case Study – Stream Processing

To demonstrate the effectiveness of our approach of domain sub-language composition for component coordination, we developed a small stream processing framework for Java components in GLOO. In this section, we will first introduce the the basic building blocks of the framework (i.e., components, connectors, and roles) at a conceptual level, and then illustrate how the abstractions of the framework can be realized in GLOO. During this process, we also highlight the benefits of specific GLOO abstractions that substantially facilitate the implementation of the coordination abstractions.

4.1. A Stream Processing Framework

The basic idea behind the stream processing framework used throughout this section (cf. Figure 1) is to take the input data provided by a given data source, modify it in a sequence of transformation steps, and dispense the corresponding result at a given data sink. Data sinks, data transformers as well as data sources are the components of the framework. Each component is decorated with a connector that acts as a mediator for inter-component interaction and as such implements the required interface code. Roles [18] are assigned to each of the connectors. These roles allot the corresponding source and sink ends of connectors specific data-flow capabilities and encode the composition rules of the framework, that is, only the source end of a connector can be connected to the sink end of another connector. Finally, we capture components external to the GLOO runtime environment, that is, the Java utilities files and processes, in proxies. These proxies provide a basic security layer, moderate the component’s features, and promote the associated components to first-class GLOO values.

Two types of data sources are supported by the stream processing framework: input files and output processes, that is, system processes that only provide output, but do not require any input. Similarly, we support two types of data sinks: output files and the console to display the printable representation of data on the screen. Furthermore, the framework incorporates three types of data transformers: (i) Filter applies a boolean predicate to each data element, (ii) Transform maps each data element to a corresponding new element given some mapping function, and (iii) IOProcess delegates the data transformation to an external system process.

The supported connector roles are Source and Sink, which implement the source and sink ends of the associated connector’s specific data-flow capabilities. Whereas the sink ends are identical for all connectors in the framework, we have to consider two types of source ends: pull-push and pull-push*. For example, the components embodied in Filter and Transform only process data elements on demand, that is, when the sink they are linked to requires new input. We denote this behavior as pull-push. On the other hand, the sink-role coordinators for output files and the console output repeatedly pull an element from the associated input source and push it into the corresponding sink component until no further input elements are available. This behavior is

![Figure 1. Conceptual Model of Stream Processing Framework](image-url)
necessary as neither output files nor the console can initiate any interactions; they simply act as passive end-points for output. We denote this kind of behavior as pull-push*. Similarly, as the framework does not allow for call-backs from external system processes, the sink-role coordinator of IOProcess also requires a pull-push* behavior.

In order to encode the stream processing framework, we follow the approach outlined in Section 3: a meta-level that integrates external Java components as GLOO run-time entities, a domain level that defines the component model as well as establishes the rules governing connector specification and composition, and a user level that defines a set of ontology mappings that associate the features of the domain level with coordination concepts.

4.2. Meta-Level Abstractions

In order to integrate external Java components into the GLOO run-time environment, we have chosen a two-step approach: (i) wrap the functionality of Java components in a separate Java class that appropriately handles any exceptions thrown by the environment, and (ii) define a corresponding component proxy class in GLOO using the Language of Namespaces and Traits. Component proxies create and interact with the contained instance of the associated Java class using the Language of Java Services [11], which allows us to treat external components as first-class values in GLOO. This approach is illustrated in Listing 2 that provides a sketch of the Java wrapper for ExternalProcess and Listing 3 that depicts the corresponding GLOO component proxy, SystemProcess, for external system processes.

```java
public class ExternalProcess {
    // private instance variables

    public ExternalProcess (String command)
    ( */ process initialization */ )
    public String readline ()
    ( */ read a text line */ )
    public void writeLine ( String str )
    ( */ write a text line */ )
    public void closeInput ()
    ( */ flush and close input */ )
    public boolean EOF ()
    ( */ check for more data */ )
}

Listing 2. The Java bridge ExternalProcess.
```

```java
public class SystemProcess {
    // Object initializer
    protected method init (\Argss::
        let
            new_process =
                new ExternalProcess
                (| arguments = (| command:0 = Argss.process |),
                    signature =
                        (| command:0 = "java.lang.String" |))
            in
                process.set new_process
        end)

    method readLine (\()::
        send readLine (\|) to (process.get (\|))
    method EOF (\()::
        send EOF (\|) to (process.get (\|))
    method closeInput (\()::
        send closeInput (\|) to (process.get (\|))
    method writeLine (Message::
        send writeLine
        (| arguments = (| str:0 = Message |),
            signature = (| str:0 = "java.lang.String" |))
        to (process.get (\|))
    end)
}

Listing 3. The component SystemProcess.
```

The specification shown in Listing 3 also demonstrates local language composition. In order to define the GLOO wrapper SystemProcess, we need language support for two complimentary class models: one to represent Java objects as first-class values and one to expose the Java objects as elements of the Language of Namespaces and Traits. Therefore, within the scope defining SystemProcess, we import the specification unit LanguageOfJavaServices.lf that provides the necessary abstraction to “program” with Java in GLOO. For example, the application of new in the constructor init actually refers to the object initializer of the Language of Java Services. This overloaded meaning is safe and does not interfere with the Language of Namespaces and Traits’s own object initializer, as it is only visible within the scope defining SystemProcess. It is also noteworthy that the
Language of Java Services resides conceptually between the meta level and the domain level and services both. In case of SystemProcess, the Language of Java Services acts as an extension of the meta level.

Finally, the code in Listing 3 illustrates another GLOO feature: positions. By default, GLOO is oblivious to the particular location of a binding in a given namespace [11]. However, Java adheres to a position-dependent parameter passing style. As a consequence, we need to map given bindings to their required underlying location. There are two occurrences of position mappings in the specification of SystemProcess; in the constructor init and in the method writeLine. We write command:0 and str:0, respectively, to indicate that both bindings must be mapped to the first argument of their respective Java equivalent. This translation is automatic (i.e., a built-in feature of the Language of Java Services). Moreover, the GLOO semantics guarantees that only bindings with position annotations are propagated to Java, which allows for a coexistence of standard GLOO bindings and Java bindings within the same namespace.

4.3. Domain Level Abstractions

The domain sub-language Language of Namespaces and Traits, in combination with the Open Extensible Imperative Class, provides use with a variety of high-level object-oriented concepts like namespaces, classes, objects, and traits [19]. To facilitate the understanding of the abstraction building process, we have opted, therefore, to use an object-oriented approach to encode the required software artifacts at the domain level first. In the following, we briefly discuss the key abstractions of the implementation.

Roles as Traits. We implement roles as (stateless) traits [19] that define the required functionality for sources, sinks with pull-push coordination channels, and sinks with pull-push coordination channels as well as the corresponding coordination protocol as illustrated in Listing 4.

The trait Source, a role that complies with the semantics of the Sync channel type of Reo [3], defines the two methods: next and EOF that are used to retrieve the next element of a source and test whether there are any more elements left, respectively. EOF makes a self-call (denoted by self $$[\|\|]$$) to a method getInput that needs to be provided by any class Source is applied to. Hence, getInput is declared as being a required method for EOF. Since the specific implementation for next differs for different types of sources, Sink only specifies a common interface for next, but does not provide a concrete implementation. For this reason, next is defined as an abstract method, a feature that is not supported by the original traits model [19]. We can, however, easily amend traits with a suitable support for abstract methods as shown in Listing 1. This extension only effects scopes that import the defined coordination abstractions. Other scopes using the original Language of Namespaces and Traits are not affected. Language composition occurs within well-defined regions and does not accidentally escape the defining scope [12].

The trait PullSink defines the functionality of connectors with a pull-push coordination semantics and implements a single method compose. We use an approach in which the responsibility for sink-source compositions rests with the source role (i.e., when combining a given source with some sink, we write sink.compose source, where source is the subject and sink is the object in a connector composition).

Inside compose, we call setInput to save the source the associated sink is getting its input from and remove (or encapsulate) all methods associated with sink-source compositions from the resulting composite. This role change creates a final connector that cannot be further composed with other sources. Changing a role is not just a mere cast to a specific desired interface, but results in a physical removal of features. In object-oriented terms, changing the role of an object of some type T results in an object of some other type S, where T and S are not related. A useful metaphor is that

```java
StreamInterfaceNS =
Namespace StreamInterfaceNS
Trait Source // implements 'Source' role
abstract method next
method EOF (\{}:
  (self \{}\).getInput (\{}\).EOF (\{}\)
requires EOF getInput "Unit -> Component"
endTrait
Trait PullSink // implements 'Sink' with pull-push
  method compose (\Source::
    (self \{}\).setInput Source;
    encapsulate PullSinkRole (self \{}\) )
requires compose setInput "Object -> Unit"
endTrait
Trait PushSink // implements 'Sink' with pull-push
  method pushInput (\Source::
    apply (self \{}\).writeElement Source)
requires pushInput writeElement "Object -> Unit"
method compose (\Source::
  (self \{}\).open Source;
  (self \{}\).pushInput Source;
  (self \{}\).close Source;
  encapsulate PushSinkRole (self \{}\) )
requires compose open "Component -> Unit"
requires compose close "Object -> Unit"
endTrait
endNamespace
```

Listing 4. Implementing roles as traits in GLOO.
of a glass falling off a table and breaking into pieces. Once the glass is broken, it has changed its role; it has become pieces of glass, which cannot be fused again to build the original glass.

The trait PushSink also implements a method pushInput that allows for pushing all elements from the sink’s input into the associated component. To do so, it uses the abstraction apply that maps a given function (in this case, the required method writeElement) to all elements of the input. In order to enable an appropriate set-up and tear-down of the sink-end, compose also calls the required methods open and close that provide the corresponding facilities to open and close a connector end-point. The call to writeElement in method pushInput is expressed as an invocation of a method pointer. The expression (self (||)).writeElement yields a reference to the member function writeElement, whose receiver is bound to the value denoted by (self (||)).

The three traits Source, PullSink, and PushSink are all defined within the scope of the namespace StreamInterfaceNS. Again, neither Namespace, Trait, abstract, method, requires, endTrait, nor endNamespace are keywords in GLOO, but rather functions defined in the Language of Namespaces and Traits or are part of our coordination vocabulary.

Listing 5. Composite roles.

```
Trait PushSinkSource
   join (use PushSink of StreamInterfaceNS)
   with (use Source of StreamInterfaceNS)
endTrait

// Default implementation for PushSinks
Trait PushSinkDefault
   method open (\():: (||))
   method close (\():: (||))
   method writeElement (\():: (||))
endTrait

Trait EmptyPushSink
   join (use PushSinkDefault of StreamInterfaceNS)
   with (use PushSink of StreamInterfaceNS)
endTrait
```

Role Composition:. The traits Source, PullSink, and PushSink capture only the semantics of one end-point of a connector. We use role composition to assign the other end-point its semantics. The traits model offers various means to fuse traits [19], one of which is join, a trait composition operator to construct a new trait by merging the methods of its argument traits. The operator join also performs a number a sanity checks, which guarantee that composed trait does not contain any competing methods [19].

We define two composite roles for our stream processing framework: PushSinkSource and EmptyPushSink. The role PushSinkSource defines a data-flow connector behavior similar to Reo’s FIFO channel [3]. At the source-end, a PushSinkSource connector pulls data items repeatedly and pushes those items into its sink-end at once. Therefore, a connector implementing the PushSinkSource role exhibits a pull-push’ behavior.

The role EmptyPushSink, on the other hand, reproduces the semantics of a LossyFIFOn Reo channel [3] with n = 0. In other words, a connector adhering to the EmptyPushSink role is pull-push∗, where all data items are lost. A typical application of this role is Filter in which undesired data items are discarded.

Connectors as Classes:. Connectors of the stream processing framework are implemented as classes, that is, they define methods (that can be either public or protected) and state, but most notably can also include functionality that has been specified in a trait. To illustrate a connector implementation, consider the connector IOProcess given in Listing 6.

```
Class IOProcess super Object of Root
   var process_obj null
   var process null

   // Methods for ‘Source’ role
   protected method getInput (\())::
      process_obj.get (||)
   method next (\())::
      {((self (||)).getInput (||)).readLine (||) )
   method close (\())::
      {((self (||)).getInput (||)).closeInput (||) )
   protected method writeElement (Elem::)
      process_obj.writeLine Elem
endMethod

   // Methods for ‘PushSink’ role
   protected method open (\())::
      new SystemProcess of ProcessIONS
      (| process = process.get (||) |)
   protected method close (\())::
      {((self (||)).getInput (||)).closeInput (||) )
   protected method writeElement (\())::
      {((self (||)).getInput (||)).writeLine (||) )
endMethod

endClass
```

Listning 6. The connector IOProcess.
required by the connector end-point, that is, open starts a new system process that runs external to the GLOO runtime environment and close winds up the input channel to this process, respectively.

The class IOProcess also satisfies the Push-SinkSource role as shown in Listing 5, which combines the functionality of both, Source as well as PushSink. Furthermore, except next, all methods required by PushSinkSource are defined as protected methods in IOProcess. Although the original traits model does not provide any provisions for the specification of method qualifiers, there exists a sound and safe approach to account for them in the Language of Namespaces and Traits. In particular, we use a rule, which guarantees that the decision about the actual visibility of a trait method (i.e., protected or public) rests with the class that trait is applied to. Since IOProcess dictates that all methods but next are protected, we change the visibility of all PushSinkSource method to protected when composing PushSinkSource with IOProcess. This happens automatically as part of the composition logic defined in the Language of Namespaces and Traits.

4.4. User Level Abstractions

The composition of the Language of Namespaces and Traits and Open Extensible Imperative Class illustrated in this section defines the core domain vocabulary for component coordination in an object-oriented fashion. However, the abstractions of the core domain vocabulary do not allow us to directly express the corresponding coordination abstractions in a user-centric way. For example, although roles are implemented as traits in a more or less straightforward way, it would be much more intuitive to directly use the term Role instead of Trait to specify a corresponding coordination role. Similarly, it would be more intuitive to use Connector and Component instead of Class.

In order to overcome this shortcoming, we have redefined the core domain vocabulary at the user level (cf. Listing 1) in order to obtain a nomenclature that facilitates the specification of interface code in a more intuitive way. More precisely, we have defined a set of ontology mappings that associate object-oriented features of the domain level with coordination concepts at the user level. The corresponding mappings can take the form of a simple name translation from the core domain vocabulary to the coordination vocabulary such as, for example, a mapping from Role to Trait and Connector to Class. On the other hand, mappings can also consist of more complex partial class declarations or mini parsers that introduce new syntactic elements and refine existing semantics definitions, respectively. For example, the abstraction ExtendRoot used for the syntactic element BaseConnector is a syntax function that maps connector specifications to class definitions by adding the

---

Listing 7. Applying User Level Abstractions.

```latex
StreamInterfaceNS =
Namespace StreamInterfaceNS

Role Source // implements 'Source' role
  abstract method next
  method EOF {\(1\)}:
    (self {\(1\)}).getInput {\(1\)}).EOF {\(1\)} )
  requires EOF getInput "Unit -> Component"
endRole
...
// other roles omitted
endNamespace

FileStreamNS =
Namespace FileStreamNS

BaseConnector FileStream
  var name null
  var file null
// Object initializer
  protected method init \(\text{\texttt{\{\text{Parser}\}}}::\text{\texttt{\text{file.set (new InputFile of FileIONS \text{\texttt{\texttt{Parser}}}); \text{\texttt{file.get \{\{\}}}).open \{\{\}}} )\}
// Methods for 'Sink' role
  protected method getInput \{\{\}:: file.get \{\{\}
  method next \{\{\}:: \text{\texttt{\text{file.get \{\{\}}}).readLine \{\{\}}
endConnector

apply (use Source of StreamInterfaceNS) to FileStream
endNamespace
```

---

5. Discussion, Lessons Learnt

Both the design of stream processing framework and the associated coordination vocabulary provide a practical guideline for the definition of readily-available language
abstractions and the effective use of the declarative programming paradigm in GLOO. In addition, whereas many mainstream programming languages rely on accumulation of features, the design of GLOO highlights the incremental and compositional evolution of programming abstractions [12]. In particular, the composition of the two domain sub-languages Language of Java Services and Language of Namespaces and Traits, in combination with GLOO's extension model, allowed us to construct a user-centric coordination vocabulary in which coordination abstractions could be defined in a flexible and extensible way. Most notably, however, the definition of the required coordination abstractions was confined to a well-defined scope, including any extensions to underlying features. Hence, the approach of creating programming abstractions using composition rather than accumulation has proven to be successful for composing and coordinating components.

Moreover, our coordination language experiments have revealed a number of interesting insights into GLOO's capability of providing support for coordination abstractions:

- We have captured external components in proxies that encapsulate the Java utilities, "files" and "processes" in the case of the stream processing framework. The proxies themselves do not participate in the coordination directly, but rather provide a basic security layer, moderate the component's features, and promote the associated components to first-class GLOO values.
- We have assigned "roles" [18] to connectors. These roles allot the corresponding source and sink ends of connectors specific data-flow capabilities. We have modeled two types of connector roles: pull-push and pull-push*. The former corresponds to Reo's Sync channel type [3], a connector with a source and a sink, in which I/O operations at both ends succeed only simultaneously, whereas the latter implements variants of Reo's FIFO and LossyFIFOn channel types, connectors with an unbounded buffer which accept at their sink ends all data items.
- Traits [19] offer an attractive means to model roles and provide a compact specification format to capture the behavior of connectors. In addition, trait composition provides an appropriate mechanism to fuse roles that capture the interaction behavior at both ends of connectors. However, in our experiments we also encountered some shortcomings that made it necessary to amend the original trait model. Fortunately, these additions (i.e., abstract and protected method qualifiers) were easily to add and did not break the underlying protocol semantics of the traits model. We will explore additional refinements to the traits model in future work.
- For the composition of connectors, we have used method pointers which allow for a decoupling of method selection and method invocation, respectively. Due to the exogenous control style of the framework, there was the need to "push" data into the sink end of pull-push* connectors. However, the precise target (i.e., the component) attached to the sink was only known at runtime. In order to overcome this problem, we have used method pointers, first-class values consisting of two elements: the target object and a reference to the desired method. When composing a pull-push* connector with a concrete component at the sink end, we have constructed a method pointer for the component's output capability (e.g., a write method). Once the data becomes available at the source, the pull-push* connector uses the method pointer to push the data into the sink.

Our findings indicate that the induced compositional approach to language extension offers an attractive alternative to classical language extension resulting in a monolithic programming language. Defining a programming model for a new domain becomes much easier in the compositional approach and consequently more accessible to application developers. Compositional language extension is not concerned with the overall impact and effect on existing software artifacts. Extensions occur in well-defined scopes and do not pollute the surrounding application space. It is the user of the language that determines the actual domain vocabulary, not the language designer. Both, user and language designer contribute to the successful application of specific language concepts by defining their own, respective domain vocabulary. However, this approach mandates suitable expertise in standard principles and techniques of programming language design and implementation.

6. Conclusions and Future Work

In this work, we have reported on our initial insights into the use of GLOO, a functional composition language, for the definition of flexible and extensible coordination abstractions. We have illustrated that language composition is a viable option in defining suitable coordination abstractions without polluting the underlying host language with additional language features. Using the case scenario of a small stream processing framework, we have shown that a combination of the concepts of component proxy, method pointers, and Pernici’s objects with roles [18] formed the basis for implementing the underlying coordination model. Most notably, defining component connectors as compositions of roles that characterize Reo-style coordination channels [3] and implementing roles using traits [19] turned out to be one of the key factors in our approach. Finally, we have demonstrated how the vocabulary of the underlying domain can be explicitly represented in GLOO-code itself, allowing us to express the concepts of the coordination model in a much more intuitive way.
To further our understanding of domain sub-languages as a means to define flexible and composable coordination abstractions, we plan to apply the approach outlined in this work in other coordination areas. Most notably, we aim at the definition of a Language of Web Services. Particularly, we seek to explore the possibilities to (i) use GL to compose and coordinate existing Web Services, (ii) define Web Services in GL and (iii) promote GL itself to a Web Service. By using the Web Service approach, we expect to obtain a service-oriented programming model to integrate heterogeneous and distributed software artifacts in which GL acts as middleware. The results of the initial study presented in this work is an important milestone to achieve this goal.

Using traits as a means to represent roles in a coordination models revealed a number of shortcomings of the trait concept. Most notably, existing trait models do not support the notion of abstract methods nor method qualifiers. Hence, we plan to investigate how existing trait models can be extended with these two features in a seamless way.

References


