

Applications = Components + GLoo

Markus Lumpe¹

*Department of Computer Science
Iowa State University
Ames, USA*

Abstract

We are advocating a component-oriented software development approach that provides support for a clear separation between the computational and the compositional entities of an application. This principle is best captured by the application building paradigm “Applications = Components + Scripts.” However, the biggest obstacle for a successful use of this maxim originates from the choice of the scripting mechanisms being used to define applications as compositions of reusable software components. In this paper, we analyze GLoo, a novel component-oriented programming framework, which derives its expressive power from an extensible and *open-ended* scripting language. The design of GLoo aims at a higher-level, scalable, and *problem-oriented* software development approach, which enables simultaneously both *small-scale* and *large-scale* software development through the definition of *specially-designed domain sublanguages*.

Keywords: Component Composition, Language Design, Semantics

1 Introduction

The component-oriented software technology has emerged as a veritable incarnation of a general engineering principle in which the development of new products is based on accumulated, generally available system knowledge and experience. By factoring out certain *stable* entities as reusable components, whose quality is expected to improve with every reuse, the component-based software development approach offers a feasible solution for developing and evolving modern, high-quality software systems [5,24,29]. However, while our understanding of this technology has grown significantly over the last decade resulting, for example, in the definition of industrial-strength component models like Java Enterprise Beans [19] and the .NET framework [23], most existing component frameworks offer only limited support for the definition of higher-level, scalable, and domain-specific *compositional mechanisms* that reflect the characteristics and constraints of the components being composed [1,27]. The mismatch between the mechanisms offered by present-day component-oriented techniques and the methodology they are supposed to support

¹ Email: lumpe@cs.iastate.edu

is due to an unsuitable use of *general-purpose* programming languages, which are not tailored to software composition [16,20,21].

Furthermore, Aßmann [5] argues that a comprehensive component-oriented software development approach not only needs to provide abstractions to represent different component models and composition techniques, but it must also provide a systematic method for constructing large software systems. In particular, a specially-designed *composition language* is required [5,16,20,21] that (i) allows for an efficient integration of heterogeneous software artifacts, (ii) provides support for a problem-oriented software development approach, and (iii) enables software engineers to incrementally construct component-oriented domain abstractions on demand.

A language that satisfies these design criteria is GLoo, a novel component-oriented programming framework that is based on an *open-ended* scripting language, which allows for the specification of extensible *domain sublanguages* to capture a well-defined subset of a component-oriented software application. In earlier work [16], we have already demonstrated, how GLoo can be used to specify readily available language abstractions that serve as *syntactic* and *semantic* extensions to the GLoo framework. In this work, we present an approach for the integration of existing software artifacts. In particular, we discuss the refined GLoo programming model and illustrate an approach to define a *Language of Java Services* to incorporate Java software artifacts into the GLoo framework. The *Language of Java Services* not only provides the required language mechanisms to import, configure, and compose Java artifacts, but also represents these Java artifacts as native *first-class* entities within the GLoo framework. As a result, we obtain immediate interoperability between the Java and other, already present, heterogeneous software artifacts in the GLoo composition framework.

The rest of this paper is organized as follows: in Section 2, we briefly describe the main features and design rationale of the GLoo framework and present the design and implementation of the *Language of Java Services* in Section 3. In Section 4, we discuss both related work and its impact on the design of the GLoo framework. We conclude this paper in Section 5 with a summary of our main observations and outline future activities in the area of the specification and implementation of narrow-focused domain sublanguages.

2 The GLoo Approach

2.1 The Design Principles

One of the major challenges in designing a new programming language is to find the right balance between the features the new language must support and the features that would make the new language more versatile. General-purpose programming languages provide a good example for this dilemma. Rather than providing very high-level and readily available plug-and-play abstractions, general-purpose programming languages offer, in general, only medium-level language constructs to build reusable domain abstractions in form of libraries or API's.

We face a similar problem when designing a composition language. A general-

purpose composition language [17,20,27] should combine aspects of: (i) architectural description languages, allowing us to specify and reason about system architectures, (ii) scripting languages, allowing us to specify applications as configurations of components according to a given architectural style, (iii) glue languages, allowing us to specify component adaptation, and (iv) coordination languages, allowing us to specify coordination mechanisms and policies for concurrent and distributed components. However, combining these rather orthogonal aspects all in one single language is not a viable option, as the resulting language would become rather large and hard to master. As a consequence, such a language would miss one of its major design goals, that is, to provide a flexible, reliable, and verifiable component-based software development approach.

The design of GLoos builds upon a different strategy that has already been successfully tested in CDL [12], a language especially designed for the construction of compilers. CDL is an *open-ended* programming language with an *empty kernel*. The most unique feature of CDL is that it is a language that does not possess any predefined language constructs, except an extension mechanism to *borrow* new language abstractions defined in *macro libraries*, which serve, therefore, as *semantic extensions* to the core of CDL. Unlike CDL, however, GLoos already provides some basic support for defining applications as compositions of reusable software components. The core of GLoos is based on the $\lambda\mathcal{F}$ -calculus [15], a variant of the λ -calculus that hosts *dynamic binding*, *explicit namespaces* [2], *incremental refinement*, and a *foreign code gateway* as the primary tenants in a single formal framework. GLoos targets a *problem-oriented* software development approach that provides a *formal testbed* for modeling, reasoning, and verifying open-ended language mechanisms in the context of component-oriented software development. The design of GLoos aims at a *scalable* support for the definition of *first-class subject-oriented development artifacts* [22] that constitute narrow-focused *domain-specific sublanguages*, which are designed, for example, for subsets of a larger component-oriented software architecture.

GLoos is a *pure functional* scripting language. Due to its functional nature, GLoos provides a declarative programming paradigm favoring an exogenous control style [3] in which computation is encapsulated in components and control flow is encoded in connectors that serve as explicit composition operators [13]. In addition, GLoos, by means of its built-in gateway mechanism, allows for a simultaneous specification of Java code within the scope of a GLoos specification unit. The Java gateway code serves as *glue code* for both the specification of new data types and the configuration of components to adapt them to actual compositional requirements [27]. Thus, the GLoos gateway mechanism provides a crucial building block for the definition of new domain abstractions.

GLoos does not offer any support for concurrent composition by default. As a result, it seems that GLoos exhibits a serious deficiency in this area with respect to the requirements for a composition language [20]. Moreover, Benton et al. [6] argue that concurrency is a language feature and should therefore be made an integral part of the underlying language specification. However, the identification and definition of the right set of abstractions for concurrency within a given language is not an easy task. Consider, for example, the industrial-strength languages Java and C#. Both

languages offer a rich set of primitives to cope with concurrent activities within a software system. Nevertheless, these abstractions are not powerful enough to separate coordination from computation concerns, as, for example, synchronization has to be specified at the method level or the lifetime of object instances may change due to application-specific settings controlled by the lifetime management of the .NET Remoting infrastructure [31].

We face a similar problem in PICCOLA [1,17], which can be seen the predecessor of GLoO. PICCOLA is a small composition language in which support for concurrency is a primary tenant of the language model, as the semantics of PICCOLA is based on a variant of the π -calculus [18]. However, the linguistic elements of PICCOLA related to its support for expressing concurrent activities are very low-level, which gives PICCOLA the flavor of a “concurrent assembler”. Moreover, PICCOLA does not offer any kind of support for the encapsulation of a component’s cooperation patterns and the computation it is performing. As a consequence, communication and computation concerns occur mixed and interspersed in a typical PICCOLA specification impeding both a clear identification of underlying cooperation patterns and a meaningful reuse in a context other than the one the specification has been designed for.

As outlined above, the GLoO approach builds upon an extensible language model. Concurrency can be viewed as a *domain aspect*, which should be captured by a specific domain sublanguage that provides support for the denotation of concurrent activities and their coordination. This view offers a subtle, but intriguing variation of Benton et al. argument, as it adheres to the requirement that concurrency has to be made an integral part of the language. However, by capturing the required language abstractions in an extensible domain sublanguage, we obtain a better versatility of the resulting language support. A suitable model for the definition of such a sublanguage is, for example, the *Idealized Worker Idealized Manager* model (IWIM) [3]. A concrete realization of the IWIM model in GLoO could, for example, mimic Reo [4], a channel-based exogenous coordination language with first-class connectors that arrange component cooperation in a component-based application.

2.2 The GLoO Programming Model

Any methodology for a comprehensive component-oriented software development approach needs to enforce a clear separation between the computational and compositional entities constituting an application [27]. For this reason, Schneider has coined

“Applications = Components + Scripts”,

a maxim that not only captures the essence of component-based software engineering, but also stresses the fact that a component-based software development approach, in order to truly achieve its goals, must yield software systems that are *extensible*, so that new features can be added without breaking the existing functionality, and they must be *composable*, so that features can be recombined to reflect changing requirements on their architecture and design.

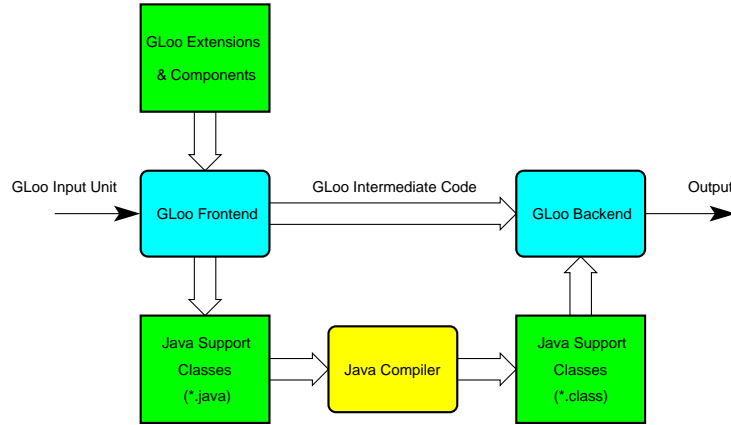


Fig. 1. The GLoo execution model.

The GLoo approach follows this maxim. In GLoo, applications are defined in *specification units* that define values or *components*, which can be composed with other values or components already present within the system. By using the GLoo framework, Schneider’s paradigm evolves to

“Applications = Components + GLoo.”

The GLoo framework builds upon an interpreter-based execution model as shown in Figure 1. The GLoo frontend analyzes a given input unit, imports required additional units from the GLoo extension library, and generates a *Java support class* for each GLoo unit being used. The support classes serve as *semantic extensions* to the GLoo runtime system, which are generated by the GLoo frontend that distills any specified gateway code into a corresponding Java abstraction within a given support class. The GLoo system uses `com.sun.tools.javac` to compile the support classes. When the compilation of the support classes finishes without errors, the backend loads the resulting Java classes into memory and starts evaluating the *intermediate code* derived from the input unit.

```

1  let
2    load "Extensions/LanguageOfJavaServices.lf"
3
4    Hashtable = JavaClass (| className = "java.util.Hashtable" |)
5    obj       = new Hashtable (|)
6    put       = service obj "put" (| key:0 = "java.lang.Object",
7                                     value:1 = "java.lang.Object" |)
8    size      = service obj "size" (|)
9  in
10   invoke put (| key = "uid", value = "Markus" |);
11   invoke put (| key = "home", value = "/Users/Markus" |);
12   invoke size (|)
13 end

```

Listing 1: Using `java.util.Hashtable` in GLoo.

As a first example, consider the specification shown in Listing 1. This unit illustrates, how the Java class `java.util.Hashtable` can be used in GLoo. The `load` statement in line 2 adds the *Language of Java Services* to the local scope of the current GLoo unit. The unit `LanguageOfJavaServices.lf` uses an additional third unit, called `JavaClass.lf`, which implements a set of abstractions to represent *Java classes* and *Java objects* as first-class values in GLoo. The declaration in line 4 imports the class `java.util.Hashtable`, which is being used in line 5 to create a

new `Hashtable` object. In its current version, the *Language of Java Services* does not provide any facilities to use generics. However, we will explore the possibility of specifying type parameters through the extension of the *Language of Java Services* in future work.

In lines 6 and 7, we define two services: `put` and `size`. Services behave like *method pointers* or delegates [23] that define an executable record structure consisting of a *target object* and a *method descriptor*. The specification of a service requires three elements: (i) a target object, (ii) a string denoting the method name, and (iii) the signature of the method. A signature defines a keyword-based argument list, where each argument is assigned a position and a type. GLoo uses dynamic binding to pass arguments to functions, that is, the position of an argument is irrelevant. Java, on the other hand, uses a position-dependent parameter passing mechanism. GLoo uses the position specification of a binding as a sorting criteria only. More precisely, if the arguments to a function possess explicitly specified positions, GLoo creates an argument list in which position-carrying bindings are placed in front of any position-independent bindings in the specified order. The reader should note, however, that position-independent arguments are ignored when calling Java methods. Thus, the method associated with service `put` requires two arguments: `key` at position 0 and `value` at position 1, both arguments being of type `java.lang.Object`, whereas service `size` has no arguments at all.

These services are then used in lines 10–12 in which we define a sequence of service invocations that yields the value of the last service invocation, that is, the integer value 2. The abstraction `invoke` takes a service and a set of arguments, which are encoded as a position-independent extensible record (i.e., a *form* [15]). Internally, the abstraction `invoke` uses the signature of the service to assign positions to the corresponding arguments and applies the resulting argument list to the associated Java method.

2.3 Building Domain Abstractions

GLoo does not offer any predefined abstractions to denote algorithms, except sequencing. However, GLoo allows for both *syntactic* and *semantic* extensions. Moreover, even though GLoo recognizes most Java operators, their actual semantics is undefined, that is, the user has to supply the GLoo system with an appropriate implementation.

The GLoo extension mechanism allows for the definition of arbitrary domain abstractions ranging from new data types to complex domain sublanguages that provide a user-centric view to a specific problem domain (e.g., a service-oriented interface to the Java API). However, common to all domain abstractions is that they require both a *meta level* and a *programming level*. The purpose of the meta level is to incorporate the new domain abstractions into the GLoo runtime system, whereas the programming level has to provide a set of higher-level programming abstractions that encapsulate the meta level to shield the application programmer from the underlying, potentially very complex, infrastructure of the defined domain abstraction.

The ability to define and integrate new domain abstractions in the GLoo frame-

```

package JavaSupport;
import LambdaF.*;

public class JavaClassValue extends LiteralValue
{
    private Class fClass;

    public Class getClassValue() { return fClass; }
    public JavaClassValue( Class aClass ) { fClass = aClass; }
    public String toString() { return fClass.toString(); }
}

```

Listing 2: Representing Java classes as first-class values.

work is of great importance in a comprehensive component-oriented software development approach. For example, even though GLoos does not offer any predefined concurrency and coordination abstractions, appropriate support for these concepts can be defined as syntactic and semantic extensions to the GLoos system, which mature over time to provide system architects with an ever-improving support for building reliable, verifiable, and robust component-based software systems.

3 The Language of Java Services

3.1 Classes and Objects as First-Class Values

In order to define the *Language of Java Services*, we need to integrate Java classes and Java objects in the GLoos framework as first-class values. More precisely, we need to define two *container value types* that encapsulate classes and objects, respectively, and a `JavaClass` meta level that defines the core abstractions to load classes, create objects, call methods, and map GLoos values to Java values and vice versa. Container value types are specified directly in Java as part of the package `JavaSupport` that is automatically loaded into the GLoos system when processing a specification unit.

The definition of the container value type for Java classes is shown in Listing 2. The type `JavaClassValue` defines a *read-only* value denoting a runtime instance of a Java class. Besides the getter method `getClassValue()`, we also have to redefine the `toString()` method to obtain a standard textual representation for the newly defined data type. In addition, by deriving `JavaClassValue` from the class `LiteralValue` we guarantee a sound integration of the new container value type in the GLoos framework and promote Java classes to first-class values in the GLoos system.

3.2 The Meta Level

Listing 3 provides an overview of `JavaClass.lf`² that implements the meta level of the *Language of Java Services*. The unit `JavaClass.lf` consists of three parts: (i) auxiliary glue code, (ii) the meta level gateway methods, and (iii) an export declaration. The auxiliary glue code defines additional behavior required by the gateway methods (e.g., the translation of GLoos values to Java values). The auxiliary code is copied verbatim to the underlying support class of `JavaClass.lf`. Both,

² We omit the presentation of the embedded Java code due to a lack of space.

```

%{ /* auxiliary glue code */ }%
let
  JavaClass = %{ /* load Java class */ }%
  newInstance = %{ /* reflection-based object instantiation */ }%
  invoke = %{ /* reflection-based method invocation */ }%
in
  (| JavaClass = JavaClass, newInstance = newInstance, invoke = invoke |)
end

```

Listing 3: A schematic view of the GLoo unit `JavaClass.lf`.

auxiliary code and gateway methods are enclosed in the delimiters `%{` and `%}`. The enclosed program text is treated as a single token by the GLoo frontend.

The gateway methods locally define the meta level for the *Language of Java Services*, which is composed from the method `JavaClass` to load Java classes, the method `newInstance` to create a new Java object for a given Java class, and `invoke` to call a Java method. The reader should note that `invoke` allows for both class-based and object-based method calls though analyzing the `receiver` argument of `invoke`. To publish the meta level, we create a new extensible record in which the gateway methods are bound to externally visible identifiers. Though not required by default, by convention we publish the gateway functions under the same names to retain to the connection between the local and external identifiers.

3.3 The Core Language

A domain sublanguage can be best described as a *many-sorted algebra* in which the component types are the sorts of the algebra and the grammar of the domain sublanguage is captured by the algebra’s signature [1]. This algebraic view of domain sublanguages provides us with a precise means not only to capture specific problem domains, but also to reason about their properties in a concise and elegant way.

When defining a domain sublanguage, we always aim at a solution with a small footprint, that is, an approach in which the discovery of the actual capabilities of a component is deferred until the corresponding information is actually being required. Using this technique, we can, for example, prevent the GLoo runtime system from creating expensive and potentially unnecessary adapter abstractions to bridge between GLoo and Java. Furthermore, we favor the application of the *continuation-passing style* (CPS) to define the elements of a sublanguage. Using this technique, function names take the role of *keywords* of the defined sublanguage, whose meanings provide a control context to evaluate a given programming abstraction. In other words, each sublanguage defines a *continuation-passing interpreter* in which the continuations *mimic* the parsing process of the underlying syntactic categories.

The core of the *Language of Java Services* is shown in Listing 4. The core exposes two abstractions: the type constructor `JavaClass` and the function `new` that takes a class container value and returns a function, which expects an extensible record denoting the constructor arguments to be passed to the meta level function `Java.newInstance`. The term `(|Args, class = Class|)` defines a so-called *binding extension* in which the record `Args` is refined by the binding `class = Class`. Again, the order of bindings in an extensible record is insignificant. However, in case of the occurrence of bindings with the same label, the right-most binding has

```

let
  Java = load "Extensions/JavaClass.lf"

  propagate_positions =
    (\Positions:: (\Arguments:: (| Positions # Arguments |)))
in
  (|
    JavaClass = Java.JavaClass,
    new = (\Class:: (\Args:: Java.newInstance (| Args, class = Class |)))
  |)
end

```

Listing 4: The core of the *Language of Java Services*.

precedence.

In addition, the core also defines the function `propagate_positions`, visible to the local scope of the unit only. The purpose of this function is to combine the position information specified in the signature record with the bindings in the argument record. The term `(|Positions # Arguments|)`, denoting the composition of the bindings of both records giving the bindings of `Arguments` precedence [15], yields a record that has the same bindings as `Arguments`, except that positions have been added if necessary. More precisely, if a binding in `Arguments` does not possess an explicitly specified position, then the record `Positions` is consulted to assign that binding a position.

3.4 A Smalltalk-like Programming Model

To add a *message* paradigm to the core language, we define the functions `send` and `to`, as shown in Listing 5. The function `send` takes a method name and an argument descriptor³ to return a new function that requires a continuation. This continuation is the function `to` that *sends* the method encoded in `Args` to the receiver object `Object`. The underlying meta level function `invoke` does not distinguish between different calling paradigms. It uses the record `(|Args, receiver = Object|)` to determine and call the desired method.

```

send = (\MethodName::
       (\Args::
        let
          sig      = Args->signature
          args     = Args->arguments
          new_args = propagate_positions sig args
          NewArgs = (| Args, arguments = new_args |)
        in
          (\Receiver:: Receiver (| NewArgs, method = MethodName |))
        end)),
to = (\Args:: (\Object:: Java.invoke (| Args, receiver = Object |)))

```

Listing 5: The Smalltalk-like extension.

To illustrate the use of the Smalltalk-like programming model, consider the specification given in Listing 6 that illustrates the setup of a Swing *Hello World* application. This example demonstrates not only the use of the abstractions `send` and `to`, but also shows that Java classes are first-class values (e.g., `JFrame`). The specification is clearly more verbose than its corresponding Java program, as we are required to explicitly specify the signatures of methods. However, the actual code

³ The argument descriptor encodes the actual arguments and the method signature as *nested records* that can be accessed using the dereference operator `->`.

```

let
  load "Extensions/LanguageOfJavaServices.lf"

  JFrame = JavaClass (| className = "javax.swing.JFrame" |)
  JLabel = JavaClass (| className = "javax.swing.JLabel" |)
in
  send
    "setDefaultLookAndFeelDecorated"
    (| arguments = (| defaultLookAndFeelDecorated:0 = true |) |)
  to JFrame;

  let
    frame = new JFrame (| arguments = (| title:0 = "HelloWorldSwing" |) |)
  in
    send
      "add"
      (| signature = (| comp:0 = "java.awt.Component" |),
        arguments =
          (| comp = new JLabel
            (| arguments = (| text:0 = "Hello World" |) |) |) |)
      to (send "getContentPane" (||) to frame);
    end
  end
end

```

Listing 6: Using the Smalltalk-like programming model.

portions are the same. Moreover, due to the small footprint of the underlying meta level, the required runtime overhead for setting up the Swing application in GLoos is negligible.

3.5 A Service-Oriented Programming Model

The message-oriented programming model offers a purely object-oriented view to Java software artifacts. Moreover, each method invocation requires an explicit signature, which makes the Smalltalk-like abstractions more suitable for configuration rather than control flow purposes. For this reason, we extend the *Language of Java Services* with a *service-oriented* paradigm. More precisely, we add the functions `service` and `invoke` (as shown in Listing 7) that serve as a minimal extension to the core language in order to provide support for a service-oriented programming model. The function `service` creates a record that denotes a method pointer in the service-oriented programming model. To call the method associated with the method pointer, the function `invoke` (i) propagates the positions defined in the signature of the service to the actual arguments, (ii) composes the resulting argument list with the method pointer, and (iii) calls the meta level function `Java.invoke` to execute the service. With the service-oriented programming model we obtain a component-oriented view in which components encapsulate computations and control flow is encoded in sequences of service invocations that act as explicit composition operators.

```

service =
  (\Obj::
    (\MethodName::
      (\Sig:: (| receiver = Obj, method = MethodName, signature = Sig |))))),
invoke = (\Service::
  (\Args::
    let
      args = propagate_positions Service->signature Args
    in
      Java.invoke (| Service, arguments = args |)
    end))

```

Listing 7: The service-oriented extension.

4 Related Work

The GLoo framework owes many of its design principles to PICCOLA, an experimental programming language that has already demonstrated the feasibility of a high-level composition language that provides a component-oriented view to software artifacts defined in a different host language [1,14,27]. Unlike GLoo, however, PICCOLA is based on the PICCOLA-calculus [1], a variant of the π -calculus, in an attempt to unite an asynchronous communication paradigm with the notion of *explicit namespaces* [2].

PICCOLA uses *composition scripts* as primary programming units, which denote a single component or application. Compiled scripts are stored in the PICCOLA *component library*⁴ that contains two kinds of information: *binaries* and *component interface definitions*. The component binaries are a kind of object files that need to be transformed into an executable agent image by a component linker. The interface definitions, on the other hand, are used by the PICCOLA compiler to perform static checks when a corresponding component is used within a script.

In order to seamlessly integrate external software artifacts, PICCOLA provides so-called *peer forms*, which define wrapper-like abstractions to mediate between both the PICCOLA representation and the host representation of services [1]. Peer forms rely on a built-in *Java-Piccola Brigade* that uses so-called *peer classes* to instantiate external host objects. Peer classes are an integral part of the PICCOLA runtime system and are the sole means to incorporate existing software artifacts into the PICCOLA framework. There is, however, a significant runtime overhead associated with the use of peer classes to perform service discovery. In order to reduce this overhead, PICCOLA uses so-called *lazy service evaluation*. Unfortunately, PICCOLA's lazy service evaluation strategy is not application transparent and is very costly, since it requires term duplication [1].

As a consequence, even though PICCOLA is an expressive language, it is far from providing the ease and flexibility required to build reliable component-based applications. The reason for this is twofold. First, PICCOLA still exhibits a conceptual gap between the mechanisms offered by language abstractions of PICCOLA and the component-oriented methodology that it is supposed to support. Secondly, the language abstractions provided by PICCOLA are still very low-level and there is no support for defining *syntactic* and/or *semantic* extensions to the language so that the encoding of higher-level software abstractions becomes unwieldy for real programming tasks.

GLoo offers a *problem-oriented* software development approach that allows for both *programming in-the-small* and *programming in-the-large* [10]. The GLoo model for the definition of narrow-focused domain-specific sublanguages is governed by the *subject-oriented programming approach* [22]. Subject-oriented programming is a generalization of the object-oriented paradigm. A *subject* is roughly equivalent to an entire program in an object-oriented language in which all code within that subject shares the same set of class and type hierarchies, operations, and object state. Moreover, to construct more powerful subject-oriented entities, various com-

⁴ This feature is only available in PICCOLA-1

position rules exist that allow for the combination of subjects. These rules specify mappings between class and type hierarchies in the composed subjects and describe how methods dispatch from within one subject impacts the other subjects in the composition. In other words, a subject can be viewed as a *compositional style* that encapsulates a *first-class development artifact* specific to some domain that assists developers to solve problems in that domain in a more efficient manner.

There is a renewed and intense interest in feasible solutions to the *extensibility problem*. In order to retain their usefulness in a real-world environment, software systems must be continually adapted. However, the key to a successful software evolution approach lies in acknowledging the existence of the extensibility problem and designing a system in a way, so that it can evolve on demand. Several approaches have emerged (e.g., Smalltalk [11], CLOS [28], Ruby [30], Tcl [32], Mixins [9], Traits [25], , or Classboxes [7]) that focus on a particular technique, known as *class extensions*. Class extensions allow for a modular addition of both new classes and new operations to an existing class hierarchy without relying on standard inheritance mechanisms. Moreover, class extensions provide a controllable mechanism to incorporate new behavior into existing applications and allow, therefore, for a reliable, verifiable, and robust software evolution approach. The extension mechanism offered by GLoo facilitates the definition of class extensions. Moreover, due to the ability to define both syntactic and semantics extensions, GLoo is not limited to one specific model for class extensions. New models can be introduced to the GLoo system by defining a corresponding domain sublanguage that offers the required programming abstractions, as if they were native to the GLoo language (e.g., *Language of Traits* [16]).

5 Conclusion and Future Work

Crucial to the success or failure of a software project is not only our understanding of the problem domain, but also the choice of the programming languages and their support for modeling the problem domain. General-purpose programming languages offer a reasonable support for the encapsulation of domain expertise in prefabricated software entities that can be reused by rearranging them in new composites [29]. However, general-purpose programming languages are less useful when specifying applications as compositions of reusable software components, as they exhibit a mismatch between the the abstraction level of the supported language constructs and the level of abstraction at which software composition takes place.

In this paper, we have analyzed GLoo, a novel component-oriented programming framework that allows for the definition of narrow-focused compositional domain sublanguages that provide a user-centric view of a given problem domain for the application programmer (i.e., the component assembler [29]). GLoo is a dynamic, open-ended composition language that rests upon dynamic binding, explicit namespaces, incremental refinement, and a foreign code gateway, concepts all crucial for a comprehensive component-oriented software development approach.

The foreign code gateway is the most important innovation of GLoo with respect to its predecessor PICCOLA and provides us with an effective means to incor-

porate Java code directly into the scope of GLoos specification unit to construct problem-oriented domain abstractions, as illustrated in the definition of the *Language of Java Services*. The GLoos programming model allows for a light-weight and scalable approach to define domain abstractions. Rather than building large, monolithic domain models, the GLoos approach fosters the definition of small, first-class subject-oriented development artifacts that can be composed and extended to build larger and more complex and possibly concurrent compositional programming abstractions.

However, the gateway mechanism is based on a very fragile technique. At the moment almost no compile-time checks are performed to verify the bridge between GLoos and the embedded Java code. This can result in the occurrences of runtime exceptions due to improper values bound to the expected arguments. Even though the GLoos system is robust enough to properly handle these exceptions, it is more desirable to perform some kind of assertion checking both at compile- and runtime. For this reason, we plan to revise the gateway mechanism by including means to denote *contractual specifications* [8]. Contractual specifications ensure that a given feature can be safely combined with other features or deployed in a new context. By adding the notion of *contract* to a gateway function, all conditions for its application are stated explicitly and formally as part of the gateway interface specification, which will enable the GLoos compiler to deduce a set of assertions that have to hold before and after the invocation of a gateway function. This technique in concert with a refined mechanism to pass values to and from a gateway function will provide us with a more robust extension mechanism in the future, while retaining its embodied flexibility.

The GLoos framework also provides support for *Model-Driven Engineering* [26]. Model-Driven Engineering (MDE) technologies combine domain-specific modeling languages with program synthesis tools for creating domain-specific models of large-scale systems. Models are considered first class entities. The GLoos programming paradigm does not impose any restrictions on definable development artifacts. In future work we plan, therefore, to use the GLoos as a *model-integrated computing platform* and perform feasibility studies to ascertain the effectiveness of representing domain models in the GLoos framework.

Acknowledgements. The author would like to thank the anonymous reviewers for their valuable comments and discussions.

References

- [1] Achermann, F., “Forms, Agents and Channels: Defining Composition Abstraction with Style,” Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics (2002).
- [2] Achermann, F. and O. Nierstrasz, *Explicit Namespaces*, in: J. Gutknecht and W. Weck, editors, *Modular Programming Languages*, LNCS 1897 (2000), pp. 77–89.
- [3] Arbab, F., *The IWIM Model for Coordination of Concurrent Activities*, in: P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, LNCS 1061 (1996), pp. 34–56, proceedings of Coordination ’96.
- [4] Arbab, F., *Reo: A Channel-based Coordination Model for Component Composition*, *Mathematical Structures in Computer Science* **14** (2004), pp. 329–366.
- [5] Aßmann, U., “Invasive Software Composition,” Springer, 2003.

- [6] Benton, N., L. Cardelli and C. Fournet, *Modern Concurrency Abstractions for C#*, in: B. Magnusson, editor, *Proceedings ECOOP 2002*, LNCS 2374 (2002), pp. 415–440.
- [7] Bergel, A., S. Ducasse, O. Nierstrasz and R. Wuyts, *Classboxes: Controlling Visibility of Class Extensions*, *Journal of Computer Languages, Systems & Structures* **31** (2005), pp. 107–126.
- [8] Beugnard, A., J.-M. Jézéquel, N. Plouzeau and D. Watkins, *Making Components Contract Aware*, *IEEE Computer* **32** (1999), pp. 38–45.
- [9] Bracha, G. and W. Cook, *Mixin-based Inheritance*, in: N. Meyrowitz, editor, *Proceedings OOPSLA/ECOOP '90*, ACM SIGPLAN Notices **25**, 1990, pp. 303–311.
- [10] DeRemer, F. and H. H. Kron, *Programming in the Large versus Programming in the Small*, *IEEE Transactions on Software Engineering* **SE-2** (1976), pp. 80–86.
- [11] Goldberg, A. and D. Robson, “Smalltalk-80: The Language,” Addison-Wesley, 1989.
- [12] Koster, C. H. and H.-M. Stahl, “Implementing Portable and Efficient Software in an Open-Ended Language,” Informatics Department, Nijmegen University, Nijmegen, The Netherlands (1990).
- [13] Lau, K.-K. and V. Ukis, *Automatic Control Flow Generation from Software Architectures*, in: W. Löwe and M. Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)* (2006), pp. 325–339.
- [14] Lumpe, M., “A π -Calculus Based Approach to Software Composition,” Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics (1999).
- [15] Lumpe, M., *A Lambda Calculus With Forms*, in: T. Gschwind, U. Aßmann and O. Nierstrasz, editors, *Proceedings of the Fourth International Workshop on Software Composition*, LNCS 3628 (2005), pp. 83–98.
- [16] Lumpe, M., *GLoo: A Framework for Modeling and Reasoning About Component-Oriented Language Abstractions*, in: I. Gorton, editor, *Proceedings of 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006)*, LNCS 4063 (2006), pp. 17–32.
- [17] Lumpe, M., F. Achermann and O. Nierstrasz, *A Formal Language for Composition*, in: G. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, Cambridge University Press, 2000 pp. 69–90.
- [18] Milner, R., “Communicating and Mobile Systems: the π -Calculus,” Cambridge University Press, 1999.
- [19] Monson-Haefel, R., “Enterprise JavaBeans,” O’Reilly, 2000, Second edition.
- [20] Nierstrasz, O. and T. D. Meijler, *Requirements for a Composition Language*, in: P. Ciancarini, O. Nierstrasz and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, Springer, 1995 pp. 147–161.
- [21] Odersky, M. and M. Zenger, *Scalable component abstractions*, in: *Proceedings OOPSLA '05*, ACM SIGPLAN Notices **40**, San Diego, USA, 2005, pp. 41–57.
- [22] Ossher, H., W. Harrison, F. Budinsky and I. Simmonds, *Subject-Oriented Programming: Supporting Decentralized Development of Objects*, in: *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, 1994.
- [23] Richter, J., “Applied Microsoft .NET Framework Programming,” Microsoft Press, 2002.
- [24] Sametinger, J., “Software Engineering with Reusable Components,” Springer, 1997.
- [25] Schärli, N., S. Ducasse, O. Nierstrasz and A. Black, *Traits: Composable Units of Behavior*, in: L. Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743 (2003), pp. 248–274.
- [26] Schmidt, D. C., *Model-Driven Engineering*, *IEEE Computer* **39** (2006), pp. 41–47.
- [27] Schneider, J.-G., “Components, Scripts, and Glue: A conceptual framework for software composition,” Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics (1999).
- [28] Steele, G. L., “Common Lisp the Language,” Digital Press, Thinking Machines, Inc., 1990, 2nd edition.
- [29] Szyperski, C., “Component Software: Beyond Object-Oriented Programming,” Addison-Wesley / ACM Press, 2002, Second edition.
- [30] Thomas, D., “Programming Ruby – The Pragmatic Programmers’ Guide,” The Pragmatic Bookshelf, LLC, 2005, second edition.
- [31] Troelsen, A., “Pro C# 2005 and the .NET 2.0 Platform,” Apress, 2005, Third edition.
- [32] Welch, B. B., “Practical Programming in Tcl and Tk,” Prentice Hall PTR, 1997, second edition.

Appendix: GLoo Syntax

<i>Script</i>	::=	[<i>Code</i>] 'let' <i>Declarations</i> 'in' <i>SingleValue</i> 'end'
<i>Declarations</i>	::=	{ <i>Declaration</i> }*
<i>Declaration</i>	::=	['@'] <i>Binder</i> <i>Binding</i> 'load' <i>String</i>
<i>Binding</i>	::=	'load' <i>String</i> <i>SingleValue</i>
<i>Form</i>	::=	'(())' <i>Identifier</i> '(' <i>FormContexts</i> ')'
<i>FormContexts</i>	::=	<i>FormDereferences</i> { '[' <i>Form</i> ']' }*
<i>FormDereferences</i>	::=	<i>FormRestrictions</i> { '->' <i>Label</i> }*
<i>FormRestrictions</i>	::=	<i>FormExtensions</i> { '\ ' <i>Form</i> }*
<i>FormExtensions</i>	::=	<i>PrimaryForm</i> { '# ' <i>Form</i> }*
<i>PrimaryForm</i>	::=	<i>FormBindings</i> <i>Form</i> [' ' <i>FormBindings</i>]
<i>FormBindings</i>	::=	<i>FormBinding</i> { ' ' <i>FormBinding</i> }*
<i>FormBinding</i>	::=	<i>Binder</i> <i>SingleValue</i>
<i>SingleValue</i>	::=	['\$'] <i>SeqValue</i> { '[' <i>Form</i> ']' }*
<i>SeqValue</i>	::=	<i>OrValue</i> { ' ' <i>OrValue</i> }*
<i>OrValue</i>	::=	<i>AndValue</i> { ' ' <i>AndValue</i> }*
<i>AndValue</i>	::=	<i>BitOrValue</i> { '&&' <i>BitOrValue</i> }*
<i>BitOrValue</i>	::=	<i>XorValue</i> { ' ' <i>XorValue</i> }*
<i>XorValue</i>	::=	<i>BitAndValue</i> { '^' <i>BitAndValue</i> }*
<i>BitAndValue</i>	::=	<i>EquivalenceValue</i> { '&' <i>EquivalenceValue</i> }*
<i>EquivalenceValue</i>	::=	<i>RelationalValue</i> { ('==' '!=') <i>RelationValue</i> }*
<i>RelationalValue</i>	::=	<i>ShiftValue</i> { ('<' '<=' '>' '>=') <i>ShiftValue</i> }*
<i>ShiftValue</i>	::=	<i>AddValue</i> { ('<<' '>>' '>>>') <i>AddValue</i> }*
<i>AddValue</i>	::=	<i>TimesValue</i> { ('+' '-') <i>TimesValue</i> }*
<i>TimesValue</i>	::=	<i>UnaryValue</i> { ('*' '/' '%') <i>UnaryValue</i> }*
<i>UnaryValue</i>	::=	['++' '--' '+' '-' '~' '!'] <i>PrimaryValue</i> ['++' '--']
<i>PrimaryValue</i>	::=	<i>LiteralValue</i> <i>PrimaryPrefix</i> { '->' <i>Label</i> }* [' ' <i>Label</i>] { <i>PrimaryValue</i> }*
<i>PrimaryPrefix</i>	::=	<i>Code</i> [':' <i>QualifiedId</i>] <i>Form</i> '(' '\ ' <i>Formal</i> '::' <i>SingleValue</i> ')' '(' <i>SingleValue</i> ') ' 'let' <i>Declarations</i> 'in' <i>SingleValue</i> 'end'
<i>LiteralValue</i>	::=	'epsilon' <i>Integer</i> <i>Float</i> <i>String</i> <i>Character</i>
<i>Formal</i>	::=	<i>Identifier</i> '(' ' ')
<i>Binder</i>	::=	<i>Label</i> '='
<i>Label</i>	::=	<i>Identifier</i> [':' <i>Integer</i>] '{' <i>SingleValue</i> '}'
<i>Code</i>	::=	'%{' <i>Java program text</i> '}'%
<i>Identifier</i>	::=	('a'-'z', 'A'-'Z') { ('a'-'z', 'A'-'Z', '0'-'9', '_') }*
<i>QualifiedId</i>	::=	<i>Identifier</i> { '.' <i>Identifier</i> }*
<i>Integer</i>	::=	('0'-'9') { ('0'-'9') }*
<i>Float</i>	::=	[<i>Integer</i>] '.' <i>Integer</i> <i>Integer</i> '.' [<i>Integer</i>]
<i>String</i>	::=	"" { <i>all characters except ""</i> }* ""
<i>Character</i>	::=	"" { <i>all characters except ""</i> }* ""