

Transparent Replication Using Metaprogramming in Cyan

Fellipe Augusto Ugliara
FIT Tecnologia
Sorocaba, São Paulo, Brazil
fellipe.a.u@gmail.com

Gustavo Maciel Dias Vieira
DComp – CCGT – UFSCar
Sorocaba, São Paulo, Brazil
gdvieira@ufscar.br

José de Oliveira Guimarães
DComp – CCGT – UFSCar
Sorocaba, São Paulo, Brazil
jose@ufscar.br

ABSTRACT

Replication can be used to increase the availability of a service by creating many operational copies of its data called replicas. Active replication is a form of replication that has strong consistency semantics, easier to reason about and program. However, creating replicated services using active replication still demands from the programmer the knowledge of subtleties of the replication mechanism. In this paper we show how to use the metaprogramming infrastructure of the Cyan language to shield the application programmer from these details, allowing easier creation of fault-tolerant replicated applications through simple annotations.

CCS CONCEPTS

• **Computer systems organization** → **Redundancy**; • **Software and its engineering** → **Source code generation**; **Application specific development environments**;

KEYWORDS

replication, metaprogramming, code generation

ACM Reference Format:

Fellipe Augusto Ugliara, Gustavo Maciel Dias Vieira, and José de Oliveira Guimarães. 2017. Transparent Replication Using Metaprogramming in Cyan. In *Proceedings of SBLP 2017, Fortaleza, CE, Brazil, September 21–22, 2017*, 8 pages.
<https://doi.org/10.1145/3125374.3125375>

1 INTRODUCTION

Distributed computing offers the promise of increased reliability and performance compared to traditional, centralized computing. In particular, greater reliability can be achieved by *replicating* a service among many hosts to ensure availability of a service even in the presence of faults. Each copy of the service is called a *replica* and there are many strategies to create such replicated service that usually offer a balance between consistency and scalability. More scalable solutions tend to favor weak consistency guarantees, which makes reasoning about the correctness and programming such systems harder [20]. Solutions that favor consistency bear more similarities to centralized systems and are easier to reason

about and program [16]. However, even in this case, the programming of these applications still pose significant challenges [5].

Among the strong consistency techniques for replication, the more used and straightforward is called *active replication* [16]. The basis of operation of this technique is to consider the system being replicated as a deterministic state machine, that has its state changed only by well defined transitions. Put in a more object-oriented way, the system is modeled by a set of objects that only change state deterministically by calling a known set of methods. To replicate the service, we have to identify each transition before it happens, distribute the information about the occurrence of this transition and its data to all replicas and execute the transition in all of them. Based on our assumption that these transitions are deterministic, if we are able to distribute these transition among the replicas in a strict order, the replicas will progress along the exactly same states. These identical replicas will be able to provide the required service in an indistinguishable way from each other.

To make the task of creating a replicated service easier, frameworks such as Treplica [18, 19] and OpenReplica [2] were created. These frameworks help create a replicated system by taking care of the distribution, ordering and execution of the transitions selected by the application programmer. The integration of the application into these frameworks happens differently depending on the programming language used. In procedural languages the integration happens by function calls to the framework and callbacks from the framework placed by the programmer. In object-oriented languages the integration happens by creating the classes of the program by extending classes provided by the framework.

Current replication frameworks, however, only help with the communication and ordering of transitions required by active replication. Other requirements of this replication technique, such as a well defined set of mutator methods and the deterministic nature of these methods, are non trivial and completely left to the application programmer. This happens because the traditional procedural and object-oriented languages in which these frameworks are built are not suitable to enforce these non-functional requirements.

Traditional languages lack mechanisms to allow a program or framework to validate its own code. Languages that support metaprogramming are one approach for validations [8], allowing programs to inspect and modify its own code. Metaprogramming has been used to translate domain specific languages [15], implement design patterns [4], perform source code validations at compile time [6] and to detect defects in object-oriented programs [14].

In this paper we show how to use the metaprogramming infrastructure of the Cyan language [11] to transparently generate and validate integration code that uses the Treplica replication framework [18]. We were able to use *metaobjects* in a centralized object-oriented program to isolate the set of mutator methods that change

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SBLP 2017, September 21–22, 2017, Fortaleza, CE, Brazil

© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

ACM ISBN 978-1-4503-5389-2/17/09...\$15.00

<https://doi.org/10.1145/3125374.3125375>

the state of a set of objects, and to generate the appropriate extended classes to integrate with Treplica. The approach is similar in essence to OpenMP [7], OpenACC [21] and other systems that use compiler directives to guide the automatic generation of parallel code. However, our approach is much more direct to program and it is the first time metaobjects are used to create distributed code. Moreover, we were able to validate the generated code with respect to the presence of non-determinism in transitions by flagging mutator methods that would violate this requirement. For instance, our proposal is able to alert the programmer if it finds a call to a random number generator inside one of the mutator methods.

This paper is structured as follows. In Sections 2 we describe the Cyan language and give an introduction to its metaprogramming features. Section 3 describes the organization of the Treplica framework. In Section 4 we describe how to use the proposed metaobjects to turn a centralized Cyan program into a replicated one. In Section 5 we describe how Cyan metaobjects work. The paper ends with a review of related work in Section 6 and some concluding remarks in Section 7.

2 THE CYAN LANGUAGE

The language used in this paper is Cyan [11], a statically-typed prototype-based object-oriented language. Unlike most prototype-based languages, Cyan is statically typed as Omega [3], the language it was initially based on. That makes the design of Cyan much closer to the design of class-based languages such as Java, C++, or C# than to other prototype-based languages.

Cyan does not support classes as most object-oriented languages. Prototypes play a role similar to classes. Instead of using **class** to declare a class, we use keyword **object** to declare a prototype, such as **Building** shown in Figure 1. In this example, keyword **var** is used to declare an instance variable (field) and **func** to declare a method. In a field (instance variable) declaration, the type comes before the field name (**String** before **address** in Line 18). **self** refers to the object that received the message. The same as **self** in Smalltalk or **this** in other languages.

Each prototype is in a file with its own name (and extension **.cyan**). The package declaration should appear before the prototype (see Line 1). In this example, prototype **Building** is in package **main**. For lack of space we may show more than one prototype in the same figure and without the package declaration.

```

1 package main
2 object Building
3   func init: String name,
4         String address {
5     self.name = name;
6     self.address = address
7   }
8   func name: String name
9     address: String address {
10    self.name = name;
11    self.address = address
12  }
13  func getName -> String { return name }
14  func getAddress -> String {
15    return address
16  }
17  var String name
18  var String address
19 end

```

Figure 1: A prototype in Cyan

Methods are called by sending messages to objects. When an object receives a message, it decides which method to call. Thus, what is named “method call” in some languages is named “message sending” in Cyan (following Smalltalk [9]). There are three kinds of messages in Cyan: unary messages, binary operator messages, and keyword messages. Then, if **b** is a variable of type **Building**,

b getName

is the sending of *unary message* **getName** to object **b**.¹ Messages such as “-” in “-count” are also considered unary messages. In this example, message “-” is being sent to object **count**.

A binary operator message uses operators such as + or * as the message name. In **1 + 2**, message + with parameter **2** is being sent to **1**. A keyword message is composed by one or more *keywords*, each one followed by zero or more parameters. A *keyword* here is an identifier ending with a colon. So a keyword message sending may be

b name: "Dahlia" address: "21 Drive";

Consider **b** has type **Building**. If **b** refers to a **Building** object at runtime, the method called would be that declared in Line 8 of Figure 1.

Methods **init** and **init:** are constructors. They cannot be called directly by sending a message to a variable or expression (as the constructors of other object-oriented languages). To each **init** or **init:** method the compiler creates a method **new** or **new:** in the same prototype with the same parameters as the original method. This method creates an object and sends to it the corresponding **init** or **init:** message. Then the compiler adds a method **new:** **String name, String address** to prototype **Building** of Figure 1 (it is added to the compiler internal representation, the original source code is not changed). This method can only be called by sending a message to the prototype itself:

Building new: "Dahlia", "21 Drive"

This **new:** method creates a new object of **Building** and calls the **init:** method of Line 8. It is a compile-time error to send a message **new** or **new:** to anything that is not a prototype.

¹More specifically, it is the sending of message **getName** to the object referred to by **b**.

Building is the name of the prototype declared in Figure 1 and it is a reference to an object of this prototype when **Building** appears in an expression. That can be better understood by showing how prototype **Building** is translated to Java.

The Cyan compiler produces Java code. The Cyan prototype **Building** is translated to the Java class **_Building**. This class declares a static public final field called **prototype** whose type is **_Building**. This field is initialized with an object of **_Building** in its declaration. Since the field is static, there is only one field for all **_Building** objects. Since it is final, the **prototype** field refers to the same object during all the program lifetime.

When **Building** is used in an expression such as in

```
Building getName
```

the compiler produces code that uses the field **prototype**:

```
_Building.prototype._getName() // in Java
```

A prototype name can also be used as the type of a parameter or a variable as **String** in Lines 3 and 17 of Figure 1 or **Building** in the example below.

```
var Building b;
```

When used as a type, the prototype is translated to the corresponding Java class. Then the Java code for this example is

```
_Building _b;
```

let is used to declare a read-only variable to which must be assigned a value.

```
let b = Building new: "Dahlia", "21 Drive";
```

The variable name is **b** and its type, **Building**, is deduced from the expression.

A piece of code that uses prototype **Building** follows. It should be inside a method, which is not shown.

```
let Building b = Building new: "Dahlia", "21 Drive";
// print the name of the building
b getName println;
var String s;
// change the name and address
b name: "Gerbera" address: "260 Main Street";
s = b getName;
```

Variables that can change their values should be declared with keyword **var** as **s**. Instance variables that are not preceded by **var** or **let** are considered read-only (**let**) variables.

An anonymous function is a nameless function declared with the syntax:

```
{ (: T1 p1, T2 p2, ..., Tn pn -> R :)
  /* body */
}
```

Ti is the type of the parameter **pi** and **R** is the optional return type. If omitted, it is deduced by the compiler (if possible). An anonymous function, called just “function”, is a literal object. From it the compiler creates a prototype with an **eval:** method that has the **body** of the function. This method has a parameter for each function parameter. To each function a new prototype is created. The prototype of the above function inherits from the instantiated generic prototype

```
Function<T1, T2, ..., Tn, R>
```

Function is a generic prototype with a variable number of parameters. The last one is the return type of the function.

The value returned by a function should follow the symbol “^”.

```
let f = { (: Int a, Int b :)
  ^a*b
};
let result = f eval: 3, 4;
result println;
```

The declaration of **f** and the assignment of the function to it does not execute any code. When **f** receives message **eval:** at runtime the body of the function is executed returning **12**. The type of **f** is

```
Function<Int, Int, Int>
```

Figure 2 shows an example of metaobjects in Cyan, a feature that will be further explained in Section 5. In **Person** there is a metaobject annotation **@init(name, age)**. During the compilation, the compiler will load a Java class that represents this metaobject. This class is in package **cyan.lang**, a package automatically imported by every Cyan source file. An object of this Java class is created when the compiler scans prototype **Person** because of the annotation **@init(...)**. This object will be called “metaobject **init**”. A Java class is used for a metaobject because the compiler is made in Java.

```
1 object Person
2   @init(name, age)
3   func getName -> String { return name }
4   func getAge -> Int { return age }
5   String name
6   Int age
7 end
8
9 object Program
10  func run {
11    let Person meg =
12      Person new: "Meg", 2;
13    let Person doki =
14      Person new: "Doki", 4;
15    meg getName println;
16    doki getAge println;
17  }
18
19 end
```

Figure 2: Metaobjects and anonymous functions in Cyan

The Cyan compiler has several phases. In one of them it discovers the types of instance variables (fields) of all prototypes. After this phase the compiler calls a specific method of all metaobjects used in the code. In particular, it calls a method of metaobject **init** of the above source code. This method returns an object with the text to replace the metaobject annotation — it is a Java **StringBuilder** object.² The text returned by the method of metaobject **init** of this example is

```
func init: String name,
  Int age {
  self.name = name;
  self.age = age;
}
```

²The text in fact does not replace the annotation. But let’s assume that for now.

This code replaces the annotation in an internal (to the compiler) representation of the source code of **Person** (the original file with prototype **Person** is not changed). See Figure 3 for the resulting **Person** prototype. As a consequence, method **run** of **Program** can create a object of **Person** using the constructor added to this prototype.

Note that the Java class that represents the metaobject **init** knows the types of instance variables **name** and **age**. These types are necessary to generate the constructor. Metaobjects are somewhat related to annotations of Java [10] and other Java-based languages such as Groovy [1]. Unlike Java, metaobjects can change the source code in which they are used.

```

1 object Person
2   func init: String name,
3       Int age {
4       self.name = name;
5       self.age = age;
6   }
7   func getName -> String { return name }
8   func getAge -> Int { return age }
9   String name
10  Int age
11 end

```

Figure 3: *Person* generated during compilation

3 TREPLICA

Treplika [18] is a framework written in Java that provides an active replication structure for the development of replicated distributed applications. The active replication in Treplika uses the Paxos [13] algorithm to decide which transitions are applied to the replicas and to ensure they are performed in an ordered way. Treplika provides an object-oriented interface to facilitate the construction of applications that are as close as possible to conventional centralized applications [18], making the replication *mechanism* transparent to the developers. However, the framework requires some adaptation and applications using Treplika must be written following some steps. First, the prototype that contains the data to be replicated is defined. This prototype is serializable, which allows the data transmission between two distinct hosts. This feature allows data to be transformed in text, be transmitted and be transformed back in data that represent the same object.

Source code in Cyan is translated to source code in Java which is compiled to bytecode using the Java compiler. It is possible, using a metaobject, to insert Java code inside Cyan code. Figure 4 shows the prototype **Info**, which contains two variables: an **Int** and a **String**. These variables represent the *state* of this object, modified by the *actions* (transitions) which will be described later. The prototype **Info** also has two **set** methods used by actions to assign values to the private variables of **Info**.

```

1 object Info extends Context {
2   var String text
3   var Int number
4
5   func setNumber: Int number {
6     self.number = number;
7   }
8
9   func setText: String text {
10    self.text = text;
11  }
12
13  func getText -> String {
14    return self.text;
15  }
16 }

```

Figure 4: Prototype to be replicated

The prototypes of type *Action* represent the actions (transitions) responsible for deterministically changing replicated values of prototype **Info**. They are serializable and have the method **executeOn** that defines the methods to be called to effectively perform the transition and change the state of the replicated objects. Figure 5 shows the prototype **UpdateAction**, which implements a transition that performs calls to the methods **setNumber** and **setText**. When an object of prototype **UpdateAction** is passed to Treplika, it sends a copy of this object to the other replicas, properly ordered, and all of them call the method **executeOn**, passing a local copy of the object **Info** as a parameter. Therefore, all the copies will end up with objects **Info** with the same values.

```

1 object UpdateAction extends Action {
2   var String updateText
3
4   func init: String text {
5     self.updateText = text;
6   }
7
8   func executeOn: Context context {
9     var info = Info cast: context;
10    info setText: self.updateText;
11  }
12 }

```

Figure 5: Prototype that implements a transition

After the actions are created Treplika must be initialized, that is usually done in a method called **run** in a prototype **Program**. To perform a replicated operation the method **execute** of prototype **Treplika** must be called, which receives an **UpdateAction** object of Figure 5 and calls its **executeOn** method. Method **execute** applies the changes to the local application instance and forwards the action to the others replicas to be updated. Figure 6 shows how a Treplika state machine is declared and initialized in each replica. This is also an example of how an object of the prototype **UpdateAction** is passed as an argument to the method **execute** of the Treplika state machine.

```

1 object Program {
2   func run: Array<String> args {
3     let info = Info new;
4     let treplica = Treplica new;
5     treplica runMachine: info
6       numberProcess: 3
7       rtt: 200
8       path: "/var/tmp/magic" ++
9         args[1];
10
11     let action =
12       UpdateAction new: "text";
13     treplica execute: action;
14   }
15 }

```

Figure 6: Treplica configuration and execution

4 METAOBJECTS FOR REPLICATION

Applications using Treplica must be designed around its actions. They are the key component behind the active replication implemented by Treplica. In Cyan, Treplica actions implementation is made through a prototype that extends the prototype **Action**. Figure 5 shows the action **UpdateAction** that represents the method **setText** in the form of a prototype. An object of **UpdateAction** is a representation of a call to **setText**. This object is copied by other replicas, allowing them to call the same method with the same parameter. The prototype creation process that extends **Action** becomes costly and failure prone if many methods represent Treplica actions.

The prototype **UpdateAction** and the other prototypes that extend **Action** should not have application functional behavior and are developed in a standardized way. Their only purpose is to model in the form of prototypes methods that can be replicated. Since the creation of these prototypes and Treplica configuration can be standardized, the metaobjects **treplicaInit** and **treplicaAction** are designed to activate metaobjects that perform these activities during the compilation of the application.

The **treplicaInit** metaobject should be attached to declarations of variables whose type is a sub-prototype of **Context**.³ In Figure 7, this metaobject is attached to variable **info**, with the parameters of the desired Treplica instance. This metaobject changes the method **run** during compilation to create a new instance of **Treplica** and assign to it the object **info**, similar to the method **run** in Figure 6.

```

1 object Program {
2   func run: Array<String> args {
3     var local = "/var/tmp/magic" ++
4       args[1];
5     @treplicaInit( 3, 200, local )
6     var info = Info new;
7     info setText: "text";
8   }
9 }

```

Figure 7: Treplica configuration using metaobjects

The **Info** prototype in Figure 8 is similar to the one depicted in Figure 4, except that the **treplicaAction** metaobject is attached to the method **setText**. The metaobject associated with this annotation modifies the prototype **Info**, adding a new method to it and creating a new prototype that represents the method **setText** as a Treplica action. Prototype **UpdateAction** of Figure 5 is not necessary any more, since the metaobject **treplicaAction** adds an equivalent prototype to the program during compilation.

```

1 package main
2 import treplica
3 object Info extends Context {
4   var String text
5   ...
6   @treplicaAction
7   func setText: String text {
8     self.text = text;
9   }
10   ...
11 }

```

Figure 8: Replicated prototype using metaobjects

5 IMPLEMENTING METAOBJECTS

Metaprogramming is a paradigm that allows programs to manipulate other programs and change themselves in compilation or in execution time [8]. Metaprogramming has a broad meaning. In this paper we will consider it is the transformations and checks made **at compile time** by a meta level on a base program. The program that is changed or checked is called the *base program*. The code that does the changes or checks is called *the meta level*. The meta level may be just a set of classes or functions. The meta level acts as a plugin to the compiler. It may change how it parses, does type checking, generate code, and so on. Since we will restrict ourselves to compile time, runtime metaprogramming is not discussed. So we do not need to discover the methods of an object or replace a method for another at runtime.

Xtend [15], Groovy [1], Nemerle [17] and Cyan [11] are examples of languages with compile-time metaprogramming features. These languages allow walk in the abstract syntax tree (AST) to gather information at compile-time. All of them but Cyan allow the meta level to change the AST. In Cyan, changes are introduced by supplying source code in text form to the compiler. This is immensely easier than to supply an object of the AST that corresponds to a piece of source code. Since Cyan is used in this paper, we will describe metaprogramming in this language.

³The concept of a “sub-prototype” in Cyan is almost equal to that of “subclass” in class-based languages like Java and C++.

AspectJ [12] is a Java extension that supports *aspects*, which are composed by *advices*, ordinary Java code, and *pointcuts*. *Advices* can change the behavior of points of the user source code specified by *pointcuts*. For example, it can change instance variable access and method calls. Aspects are a kind of metaprogramming that is less general than that of Cyan. In this language code can be inserted in several compiler phases and in many places that are out of reach of aspects. Besides that, Cyan metaobjects have access to most of the information the compiler has at a specific point of the compilation, which aspects do not. Probably aspects can be implemented in Cyan just by creating new metaobjects, without any language modifications.

A metaobject protocol (MOP) is the interface between the meta level and the compiler in languages that support metaprogramming. In Cyan, the meta level consists of Java classes that declare methods that are called by the compiler in specific points of the compilation of the base program. Java classes are used because the Cyan compiler is made in Java. The compiler first calls methods of objects of classes of the meta level, then these methods can call some particular methods of the compiler. Which methods of the meta level are called and when is defined by the metaobject protocol.

We will give a simplified view of the MOP of the language because of lack of space. In Cyan, a Java class has to obey some prerequisites to be in the meta level. It has to be compiled with the Java compiler, be in a package called **meta**, and inherit from a class called **CyanMetaobjectWithAt**.⁴ Each such class is called a “metaobject class” and it has to override some methods inherited from **CyanMetaobjectWithAt** and implement some interfaces defined by the MOP. Both the class **CyanMetaobjectWithAt** and these interfaces belong to the package **meta** of the compiler.

It is important to note that the metaobject class has to be compiled with the Java compiler, but after that can be used with a regular Cyan compiler, one that does not know about this metaobject class. For example, the class **CyanMetaobjectTrepliaAction** is the metaobject class of **trepliaAction**. This class inherits from **CyanMetaobjectWithAt** and redefines its method **getName()** to return the string “**trepliaAction**”. After compilation, the “.class” file⁵ of **CyanMetaobjectTrepliaAction** should be put in a directory “meta--” of a package **treplia**, which is in a directory also called **treplia**, the same name as the package. When a source code imports package **treplia**, it also imports the metaobject **trepliaAction** and it is legal to use “**@trepliaAction**”. See Figure 8 for an example. Without the “**import treplia**” there would be a compilation error in “**@trepliaAction**” because the compiler would not be able to find the metaobject.

During parsing of the code in Figure 8, the compiler will create an object of **CyanMetaobjectTrepliaAction** when it finds “**@trepliaAction**”. This object is the real metaobject, the one whose methods will be called by the compiler. These calls are made in different points of the compilation. They depend on the interfaces that the metaobject class implement.

Class **CyanMetaobjectTrepliaAction** implements interface **IActionProgramUnit_ati** which allows the metaobject to create

a new method (**setText** in Figure 9), rename a method (the original **setText** is renamed to **setTextTrepliaAction**), and create a new prototype (**InfosetText** of Figure 10). The compiler knows **trepliaAction** can be attached to a method because method **maybeAttachedList** of base class **CyanMetaobjectWithAt** is redefined in **CyanMetaobjectTrepliaAction** to return a list of declarations to which the metaobject can be attached. This list has only one element, an enumerated value called **METHOD_DEC**. As shown in Figure 8, **trepliaAction** is attached to a method. Any attempt to attach it to a prototype or to any other declaration results in a compilation error.

```

1 object Info extends Context {
2     ...
3     func setText: String text {
4         var action = InfosetText new: text;
5         self getTreplia execute: action;
6     }
7
8     func setTextTrepliaAction:
9         String text {
10         self.text = text;
11     }
12     ...
13 }

```

Figure 9: Prototype Info modified

```

1 object InfosetText extends Action {
2     var String textVar
3     func init: String text {
4         textVar = text;
5     }
6
7     override
8     func executeOn: Context context {
9         var obj = Info cast: context;
10        obj setTextTrepliaAction:
11            textVar;
12    }
13 }

```

Figure 10: Prototype created by @trepliaAction

Interface **IActionProgramUnit_ati** declares methods
ati_CodeToAdd(ICompiler_ati compiler)
ati_renameMethod(ICompiler_ati compiler)
ati_NewPrototypeList(ICompiler_ati compiler)

The return value type is omitted and in each case consists of a list of changes demanded by the method. All of these methods are implemented in class **CyanMetaobjectTrepliaAction** which is declared to implement interface **IActionProgramUnit_ati**.

Method **ati_CodeToAdd** of **CyanMetaobjectTrepliaAction** creates the new method **setText** shown in Figure 9. Remember that we are using the example of Figure 8 which produces the code of Figure 9. Method **ati_CodeToAdd** does not use the AST of the current prototype. It produces a string with the source code of the new method **setText**. This is much easier to implement

⁴Other kinds of metaobjects, not presented in this paper, should inherit from other classes.

⁵The bytecode of the Java source file after compilation.

than to produce an AST object. This same observation is valid for any other method that changes the source code. The method **ati_renameMethod** of **CyanMetaobjectTrepliaAction** renames the old method **setText** to **setTextTrepliaAction**, while the method **ati_NewPrototypeList** creates prototype **InfosetText** of Figure 10.

```

1 object Program {
2   func run: Array<String> args {
3     var local = "/var/tmp/magic" ++
4       args[1];
5     var info = Info new;
6     var trepliainfo = Treplia new;
7     trepliainfo runMachine: info
8       numberProcess: 3
9       rtt: 200
10      path: local;
11     info setTreplia: trepliainfo;
12     info setText: "text";
13   }
14 }
```

Figure 11: Prototype Program modified

Both the interface **IActionProgramUnit_ati** and the base class **CyanMetaobjectWithAt** belong to the MOP of Cyan. There are many other classes and interfaces in the MOP, which is the way the Cyan compiler and a metaobject class communicate with each other. For example, interface **IActionVariableDeclaration_dsa** should be implemented by a metaobject class whose metaobjects should be attached to variable declarations. This is the case of metaobject **trepliaInit**, shown in Figure 7. This metaobject changes the declaration producing the code shown in Figure 11.

Parameter **compiler** of each of the **IActionProgramUnit_ati** methods provides a restricted view of the Cyan compiler. It supports methods that search for an instance variable, issue a compiler error, etc. Class **CyanMetaobjectWithAt**, inherited by all metaobjects of this paper, also provide some important methods. One of them, **getDeclaration()** supply the method to which the metaobject **trepliaAction** is attached — it is necessary to do a cast to the correct class of the Cyan AST, which is **MethodDec**. Then method **ati_renameMethod** can rename exactly that method. And method **ati_CodeToAdd** knows the method name it should create: it should be equal to the name of the method the metaobject is attached to. Metaobject **trepliaInit** is attached to a variable declaration. Throught **getDeclaration()**, it can get the AST object of the Cyan compiler that represents a variable declaration (after a cast of the returned value).

Class **CyanMetaobjectTrepliaAction** also implements the interface **ICheckProgramUnit_dsa2** and defines its methods, specifically method **dsa2_checkProgramUnit**. This method is called after the *base program* has been changed by all metaobjects and no further change is allowed. This method looks for non-deterministic method calls in a depth-first search starting in the method to which it is attached. That is, in the AST object of class **MethodDec** returned by method **getDeclaration()**.

```

1 object Info extends Context {
2   var String text
3   func ageInSeconds -> Long {
4     ...
5     // non-deterministic behavior;
6   }
7
8   @trepliaAction
9   func set: String text {
10    self.text = text ++ " age: "
11    ++ ageInSeconds;
12  }
13  ...
14 }
```

Figure 12: Example of validation

In the example of Figure 12, **trepliaAction** looks for non-deterministic method calls in **set**: and in any other method **set** may call. In this case, **ageInSeconds** and in any other methods that **ageInSeconds** may call. The method **dsa2_checkProgramUnit** of class **CyanMetaobjectTrepliaAction** checks which calls are considered as non-deterministic by querying a list of pairs of prototypes and methods. This list must be made by the programmer. If there is any possibility, even through polymorphism, that method **ageInSeconds** calls a non-deterministic method, the metaobject will ask the compiler to issue a compiler error. In this example, this is expected since **ageInSeconds** should get the current system time.

6 RELATED WORK

OpenReplica [2] is a framework to implement replicated services similar to Treplia [18]. Along with Treplia, OpenReplica represents the state of the art for easily creating replicated applications and both use a similar object-oriented approach that suffers from the same transparency and code verification problems. Both frameworks require an interface layer to encapsulate the methods implementing changes to the replicated state and neither allows code inspections that search for inconsistencies in the implementation of the interface. In this paper we use metaprogramming to tackle these challenges, similarly to the way metaprogramming has been used to attack similar problems.

Rentschler et al. [15] argues the use of domain specific languages (DSLs) to increase programmer productivity and quality and proposes the use of metaprogramming to translate these DSLs in other languages. They use the Xtend language [15] to transform a DSL using active annotations. We use a similar approach of automatic code transformation. But starting from centralized code written in a general purpose language, we arrive in distributed code written in the same language. Moreover, the metaprogramming infrastructure provided by Cyan allows for a more elegant implementation than the one obtained by using Xtend. Another similar work is the one by Blewitt et al. [4] that proposes the use of metaprogramming to automatically create components that implement design patterns.

Chlipala [6] shows a proposal for using metaprogramming to perform source code validations at compile time using macros. Inspecting the source code for problems during compilation increases

the application performance, because it is unhindered by run-time validations. Mekruksavanich [14] proposes a similar in which meta-programming is used to detect defects in object-oriented programs, by the use of software components capable of describe and identify such defects. Both these works tackle different problems from the ones described in this paper, but both show the benefits of the use of metaprogramming as an aid in the development of correct programs.

Compiler directives, have been successfully used to accelerate the creation of parallel programs. OpenMP [7] aims to ease the conversion of legacy centralized C++ and Fortran code into portable shared-memory parallel code. OpenACC [21] uses the same approach of compiler directive annotated code to offload some compute intensive tasks to accelerator devices such as general-purpose graphic processing units (GPGPUs). Both approaches simplify the task of producing parallel code, but still require a considerable knowledge of the programmer about how parallel programs work. We use metaobjects in a more simple way and aim to completely shield the programmer from details about the distributed programming model that is used. Currently we block the occurrence of invalid non-deterministic method calls and intend in the future to transparently replace this calls with deterministic counterparts.

7 CONCLUSION

We have shown how to use the metaprogramming infrastructure of the Cyan language to transparently generate and validate integration code that uses the Treplica replication framework. This way, programs written in Cyan can easily be converted from a centralized architecture to a replicated one by attaching metaobjects to mutator methods. Moreover, we demonstrated how to validate the generated code with respect to the presence of non-determinism, by alerting the programmer if unsafe operations are found in one of the mutator methods. We intent to expand this technique beyond simply flagging operations that violate the assumptions of active replication by replacing the non-deterministic operations with equivalent deterministic operations.

ACKNOWLEDGMENTS

This work was supported by the São Paulo Research Foundation (FAPESP) under grant #2014/01817-3 and by FIT - Instituto de Tecnologia (<http://www.fit-tecnologia.org.br>).

REFERENCES

- [1] 2017. The Groovy Language. <http://groovy-lang.org>. (2017). Accessed: 2017-04-21.
- [2] Deniz Altınbüken and Emin Gün Sirer. 2012. *Commodifying replicated state machines with OpenReplica*. Technical Report. Cornell University, Technical Report.
- [3] Günther Blaschek. 1994. *Object-oriented programming with prototypes*. Springer. <http://www.worldcat.org/oclc/29361389>
- [4] Alex Blewitt, Alan Bundy, and Ian Stark. 2005. Automatic verification of design patterns in Java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 224–232.
- [5] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 335–350.
- [6] Adam Chlipala. 2013. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 391–402.
- [7] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.
- [8] Robertas Damaševičius and Vytautas Štuikys. 2015. Taxonomy of the fundamental concepts of metaprogramming. *Information Technology and Control* 37, 2 (2015).
- [9] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [10] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.
- [11] José de Oliveira Guimaraes. 2008. *The Cyan Language*. Technical Report. Campus de Sorocaba da UFSCar. <http://www.cyan-lang.org>
- [12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. *Lecture Notes in Computer Science* 2072 (2001), 327–355. citeseer.nj.nec.com/kiczales01overview.html
- [13] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2 (2006), 79–103.
- [14] Sakorn Mekruksavanich, Preecha P Yupapin, and Pornsiri Muenchaisri. 2012. Analytical learning based on a meta-programming approach for the detection of object-oriented design defects. *Information Technology Journal* 11, 12 (2012), 1677.
- [15] Andreas Rentschler, Dominik Werle, Qais Noorshams, Lucia Happe, and Ralf Reussner. 2014. Designing information hiding modularity for model transformation languages. In *Proceedings of the 13th international conference on Modularity*. ACM, 217–228.
- [16] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.
- [17] Kamil Skalski. 2005. *Syntax-extending and type-reflecting macros in an object-oriented language*. Master's thesis. University of Wrocław, Poland.
- [18] Gustavo Maciel Dias Vieira and Luiz Eduardo Buzato. 2008. Treplica: Ubiquitous replication. In *SBRC '08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*. Rio de Janeiro, Brasil. <http://www.lbd.dcc.ufmg.br/bdbcomp/servlet/Trabalho?id=7450>
- [19] Gustavo Maciel Dias Vieira and Luiz Eduardo Buzato. 2010. *Implementation of an Object-Oriented Specification for Active Replication Using Consensus*. Technical Report IC-10-26. Institute of Computing, University of Campinas. <http://www.ic.unicamp.br/~reltech/2010/abstracts.html>
- [20] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (Jan. 2009), 40–44. <https://doi.org/10.1145/1435417.1435432>
- [21] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC—first experiences with real-world applications. In *European Conference on Parallel Processing*. Springer, 859–870.