Transparent Replication Using Metaprogramming in Cyan

Fellipe A. Ugliara[a], Gustavo M. D. Vieira[a,*], José de O. Guimarães[a]

[a]*DComp – CCGT – UFSCar, Sorocaba, Brazil*

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.

Keywords: replication, metaprogramming, code generation

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

*Corresponding author
Email addresses: `fellipe.a.u@gmail.com` (Fellipe A. Ugliara), `gdvieira@ufscar.br` (Gustavo M. D. Vieira), `jose@ufscar.br` (José de O. Guimarães)

about the correctness and programming such systems harder [1]. Solutions that favor consistency bear more similarities to centralized systems and are easier to reason about and program [2]. However, even in this case, the programming of these applications still pose significant challenges [3].

Among the strong consistency techniques for replication, the more used and straightforward is called *active replication* [2]. 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 [4, 5] and OpenReplica [6] 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. Regardless of the approach employed, the linking of application and framework usually requires adding boilerplate code, intertwined with application code.

Current replication frameworks, albeit useful, 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 change and validate its own code. Languages that support metaprogramming [7] allow programs to inspect and modify their own code. Metaprogramming has been used to translate domain specific languages [8], implement design patterns [9], perform source code validations at compile time [10] and to detect defects in object-oriented programs [11].

In this paper we show how to use the metaprogramming infrastructure of the Cyan language [12] to transparently generate and validate integration code that uses the Treplica replication framework [4]. We were able to use *metaobjects* in a centralized object-oriented program to isolate the set of mutator methods that change 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 [13], OpenACC [14] 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. Our proposed set of metaobjects is able to replace non-deterministic methods with deterministic versions and alert the programmer if it still finds a call to a non-deterministic method inside mutator methods.

This paper is structured as follows. In Section 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 the proposed metaobjects, how to use them to turn a centralized Cyan program into a replicated one and how they work. The paper ends with a review of related work in Section 5 and some concluding remarks in Section 6.

2. The Cyan Language

2.1. Language Overview

The language used in this paper is Cyan [12], a prototype-based object-oriented language. Unlike most prototype-based languages, Cyan is statically typed as Omega [15], 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 [16], C++ [17], or C# [18] than to other prototype-based languages. Cyan programs are compiled to produce Java code, to be run in a Java virtual machine.

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 a field (instance variable) and `func` to declare a method. In a field declaration, the type comes before the field name (`String` before `address` in Line 17). `self` refers to the object that received the message. The same as `self` in Smalltalk [19] 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 (Figure 1, Line 1). In this example prototype `Building` is in package `main`. For conciseness, for now on we may show more than one prototype in the same figure and without the package declaration.

A variable or field can be declared using keywords `let` and `var`. `let` is used to declare a read-only field or local variable to which a value must be assigned. For example, a variable of type `Building` can be initialized as:

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

The variable name is `b` and its type, `Building`, is deduced from the expression. Variables and fields that can change their values should be declared with keyword `var` (Figure 1, Line 17). Fields that are not preceded by `var` or `let` are considered read-only (`let`) fields.

The syntax for message passing and method declaration is close to the Smalltalk syntax. Unary methods are those that do not take parameters, as

```
 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

`getName` of Line 13 of Figure 1. Assuming `aBuilding` is a variable of type `Building`,

```
aBuilding getName
```

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

A keyword method is declared with identifiers ending with `:` each of which taking zero or more parameters as method `name:address:` of Lines 8-12 of Figure 1. This method has two keywords. A keyword method may be called at runtime by keyword message passing, as in this example:

[1]More specifically, it is the sending of message `getName` to the object referred to by `aBuilding`.

```
aBuilding name: "Dahlia" address: "21 Drive";
```

In this code, message `name: "Dahlia" address: "21 Drive"` is sent to the object `aBuilding`. If `aBuilding` refers to a `Building` object at runtime,[2] the method called would be that declared in Line 8 of Figure 1.

The name of a method may be an operator such as `+` or `<`. Method `+` should take no parameters (for unary `+`) or two parameters (for binary `+`). These methods are called as usual: `1 + 2` is the sending of message `+ 2` to object `1`.

Cyan is a prototype-based language, which means each prototype such as `Building` is also an object and can receive messages:

```
Building name: "Gerbera" address: "78 main"
```

As a consequence, prototypes play a dual role: 1) they are types as are classes in Smalltalk, Java and C++, and 2) when used in expressions, they work like variables that refer to a fixed object of themselves. `Building`, when used inside an expression, refers to an object of prototype `Building`.

Object constructors are methods with names `init` or `init:` (if there are parameters). They cannot be called directly by sending messages. For each method `init` or `init:` found in a prototype, the compiler creates a method `new` or `new:` in the same prototype with the same parameters as the original method. This new method creates an object and sends to it the corresponding `init` or `init:` message. For example, the compiler adds a method `new: String name, String address` to prototype `Building` of Figure 1.[3] 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 appropriate `init:` method (Figure 1, Line 3). It is a compile-time error to send a message

[2]Even if `aBuilding` has type `Building`, the object referenced by it may be a subprototype of `Building`.

[3]These methods are added to the compiler internal representation, the original source code is not changed.

`new` or `new:` to anything that is not a prototype.

Keyword `extends` allows inheritance of a superprototype by a subprototype. Inherited methods can be overridden in the subprototype, as usual. Java-like interfaces can be defined by using keyword "`interface`" instead of "`object`" when defining a prototype.

2.2. The Cyan Metaobject Protocol

Metaprogramming is a paradigm that allows programs to manipulate other programs and change themselves in compilation or in execution time [7]. 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* or simply *program*. The code that does the changes or checks is called *the meta level* or simply *metaprogram*. The metaprogram may be just a set of classes or functions and it acts as a plugin to the compiler, potentially changing how it parses, does type checking, generates code, and so on. Since we will restrict ourselves to compile time, runtime metaprogramming will not be discussed in this paper.

Xtend [8], Groovy [20], Nemerle [21] and Cyan [12] are examples of languages with compile-time metaprogramming features. These languages allow to traverse the abstract syntax tree (AST) to gather information. In Cyan, changes are introduced by supplying, to the compiler, source code as text. 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.

A metaobject protocol (MOP) is an interface between the metaprogram, the program, and the compiler. It defines functions or methods of the metaprogram that should be called when a prototype is inherited or a method is overridden in a subprototype, when a field is accessed, a message is sent or when an annotation is found in the program by the compiler. For example, the MOP defines that a user-defined function should be called whenever a prototype is inherited. Cyan supports a Metaobject Protocol, but not all languages that support metaprogramming do.

In Cyan, the metaprogram consists of Java classes because the compiler is made in this language. That makes it trivial for the metaprogram and the compiler to communicate. During the compilation of a program, an *annotation* in the source code makes the link between the program (base level), the compiler, and the metaprogram. When the compiler finds an annotation in a certain compiler phase, it calls some specific methods of a metaobject tailored to that compiler phase. A metaobject is associated to each annotation and its Java class is part of the metaprogram. Before defining the MOP, let us see some examples of use of annotations and their associated metaobjects.

```
 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", 3;
13          let Person doki =
14              Person new: "Doki", 5;
15          meg getName println;
16          doki getAge println;
17      }
18
19  end
```

Figure 2: Metaobjects and anonymous functions in Cyan

In Line 2 of Figure 2 we can find a *metaobject annotation* or simply *annotation*: `@init(name, age)`. For each annotation, the compiler creates a *metaobject* of a *metaobject class*. This class is made in Java, as the compiler is, and is associated to a Cyan package. When the package is imported by a Cyan program the metaobject class is imported too, and the annotations whose name are equal to the string returned by method `getName()` of the metaobject class can be used in the source code. To simplify, we will say "metaobject `init`" and "class of `init`" for the metaobject associated to annotation `init` and the metaobject class of metaobject `init`.

Package `cyan.lang` is imported automatically by every Cyan program and this package keeps the metaobject class of `init`. Thus, this annotation can be used in any Cyan source code without explicitly importing a package, as is done in the example of Figure 2.

The Cyan compiler has several phases. For example, in Phase 2 it discovers the types of instance variables (fields) of all prototypes. After this phase completes the compiler calls a specific method of all metaobjects used in the code. In particular, it calls a method of metaobject `init` of Figure 2. This method returns an object with the text to be added after the metaobject annotation — it is a Java `StringBuilder` object. 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;
    }
```

This code is added after the annotation in an internal (to the compiler) representation of the source code of `Person` (the original file with prototype `Person` is not changed). Figure 3 shows the resulting `Person` prototype. As a consequence, method `run` of `Program` can create an 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.

In the remaining of this section we will give a simplified view of the MOP of the Cyan language. A Java class has to obey some prerequisites to be

```
object Person
    @init(name, age)#ati
    func init: String name,
               Int age {
        self.name = name;
        self.age = age;
    }
    func getName -> String { return name }
    func getAge -> Int { return age }
    String name
    Int age
end
```

Figure 3: *Person* generated during compilation

a metaobject class. It has to be compiled with the Java compiler and inherit from a class called `CyanMetaobjectWithAt`.[4] Each of such classes 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 as a class of the Cyan compiler, but after compiled it can be used with a Cyan compiler that does not known the source code of this metaobject class. When the package of the metaobject class is imported, the Java class is dynamically loaded by the compiler and the metaobject becomes available in the source code that imported the package.

The Cyan compiler has ten compilation phases. The first one is parsing and it builds the Abstract Syntax Tree (AST) of the whole program. Phase 2 is “type interfaces”, which associates types to prototype fields, method parameters and return values, to anything that is outside a method body (its statements) and has a type. Before that, the AST keeps only a string with the type name.

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

After parsing the compiler knows all the program prototypes.

In Phase 3 of the compilation, called “ati actions”, the compiler calls some methods of all metaobjects that implement interfaces `IActionProgramUnit_ati` and `IActionNewPrototypes_ati`. These interfaces are in the Cyan compiler. `IActionProgramUnit_ati` declares methods

```
    ati_codeToAdd
    ati_codeToAddToPrototypes
    ati_renameMethod
```

among others. The first one can return code that is added after the metaobject annotation and do checks in the source code. The second method returns code to be added to the prototype in which the annotation is.[5] The added code is in string format, a Java `String` or `StringBuffer` object. Since it is added to a prototype, it should consist of field and method declarations. Method `ati_renameMethod` allows the metaobject to rename a method of the prototype in which the annotation is. In Phase 3 only metaobjects associated to annotations that are outside a method body are allowed to generated code using method `ati_codeToAdd`. Otherwise, statements or expressions could be added to a method in this phase and this is not allowed.

All the methods of metaobject interfaces linked to phase “ati actions” receive as parameter an object of type `ICompiler_ati`. Through this object the metaobject can get information available in this phase, as the current prototype, the package name, the fields and methods of the prototype, and so on. Information can also be obtained from methods of the metaobject class itself, which should be, necessarily, `CyanMetaobjectWithAt`.

Metaobject `init` of Figure 2 generates the `init:` shown in Figure 3 in Phase 3. Note that the annotation is not removed, it can act in the following phases. String `#ati` is appended to the annotation to label it as used in this phase. Through a method inherited from class `CyanMetaobjectWithAt`, the

[5]It can also add code to other prototypes of the same package, but this needs a special compiler option and prevents separate compilation.

metaobject gets the object that represents the annotation and retrieves the annotation parameters. It then uses methods of the object of `ICompiler_ati` to check if every parameter is a field of the current prototype. This object is passed as parameter to all methods of all interfaces linked to Phase 3.

Fields and methods can be added to prototypes in the "ati actions" phase, changing the text of the source code in memory (the files are not touched). Because code could have been added, the program should be parsed again, which is made in Phase 4. This next phase is "type interfaces" again, the same as Phase 2. Phase 6 is the semantic analysis of methods, their statements and expressions are checked. The name of this phase is "calculate internal types". Now types are assigned to every element that is associated to a type. Methods of the following compiler Java interfaces are called in Phase 6.

```
    IAction_dsa
    IActionVariableDeclaration_dsa
```

Interface `IAction_dsa` declares a method for adding code after the annotation and performing checks in the source code. Annotations can be attached to prototypes, methods, fields, and local variable declarations. The metaobject associated to the annotation can get the AST of the element to which the annotation is attached. For example, an annotation `readOnly` attached to a method could check if any of the prototype fields receives a value in an assignment inside the method, issuing an error if any does.

The Cyan compiler ensures that metaobjects implementing the Phase 6 interface `IActionVariableDeclaration_dsa` can only be associated to annotations that are attached to a local variable declaration. Through this interface, a metaobject can check the annotated declaration and add code after it. It has access to the AST of the local variable and, if present, the expression assigned to it in the declaration.

Phase 6 together with Phase 5 comprise the semantic analysis of the program. Since statements and expressions can be added to the source code, in memory only, the code has to be compiled again. Phases 7, 8, and 9 of the compilation are equal to phases 4, 5, and 6 but with an important difference: the code cannot be changed anymore. However, the resulting source code can still be checked by metaobjects. Interfaces `ICheckDeclaration_ati3` and `ICheckDeclaration_dsa2` should be implemented by metaobject classes that need to do checks in Phases 8 (ati actions) and 9 (calculate internal types). The associated annotation should be attached to a declaration such as a method or prototype. Finally, Phase 10 is code generation to Java.

3. Treplica

Treplica [4] is a framework written in Java that provides an active replication structure for the development of replicated distributed applications. In this section we briefly describe how to program this framework through a binding developed for the Cyan language[6] with the purpose of characterizing the programming effort required to program a replicated application using the framework.

Active replication ensures that many copies of a single application known as *replicas* keep their state up to date and consistent even as changes are made as part of the application operation. It works by assuming the application behaves like a deterministic state machine, that only changes its internal state by deterministically executing transitions. If one is able to execute the same transitions in the same order in all replicas, they will end up with the same resulting state due to the deterministic nature of the transitions. As a consequence, an active replication framework must provide a reliable way of disseminating transitions in a ordered way among all replicas and it should provide a programming interface that allows regular applications to behave like deterministic state machines even if they are not programmed as such.

Treplica solves the problem of disseminating transitions by using the Paxos algorithm to ensure the transitions reach all replicas in the same order, even in

[6]More information in the Java interface of Treplica can be found in [5].

the presence of failures [22]. Treplica solves the programming interface problem by providing an object-oriented abstraction that defines the very simple notion of a shared state and well defined changes to this state. The resulting programming interface is as close as possible to conventional centralized applications [4], making the replication *mechanism* transparent to the developers.

The shared state of an application is defined by its *context*, a single object that stores the application data in its fields and references. In the Treplica framework this object should extend the `Context` prototype and be serializable. Serialization allows an object to be transformed to text, transmitted between different hosts and transformed back to a clone of the original object. Figure 4 shows the prototype `Info`, an example of an application context. This prototype contains two variables: an `Int` and a `String`, representing the state of this application.

```
object Info extends Context {
    var String text
    var Int number

    func setNumber: Int number {
        self.number = number;
    }

    func setText: String text {
        self.text = text;
    }

    func getText -> String {
        return self.text;
    }
}
```

Figure 4: Prototype to be replicated

The prototype `Info` has two `set` methods used to assign values to its private variables. More importantly, these two methods allow changing the state of the application context. The Treplica framework considers a message sent to an object that calls one of these mutator methods to be equivalent to a transition that changes the context state in a deterministic way. The framework then defines a way to capture and represent this message as an *action*.

The prototypes that extend prototype `Action` implement these actions. They contain as fields the parameters of the message to be sent and are serializable, allowing the record of this message to be sent to other hosts. Also, they must implement the method `executeOn:` that defines the keywords of the message to be sent by encoding an actual message send operation using the fields as parameters. The target object of the message sent by `executeOn:` is defined by a parameter received by this method. Figure 5 shows the prototype `SetTextAction`, which implements a transition that sends message `setText:` to an `Info` object.

```
object SetTextAction extends Action {
    var String updateText

    func init: String text {
        self.updateText = text;
    }

    func executeOn: Context context {
        type context
            case Info info {
                info setText: self.updateText;
            }
    }
}
```

Figure 5: Prototype that implements a transition

The actual firing of a transition is implemented by Treplica. When the application wants to change its state, it creates an appropriate action object and passes it to Treplica in a `execute:` message sent to an object of prototype `Treplica`. The framework then sends a copy of this object to the other replicas, properly ordered, and all of them send the message `executeOn:` to the received object, passing a local copy of the context as the parameter. Therefore, all the copies will end up with contexts with the same values.

For this to work, no changes to the context can happen without being represented as actions and the actions passed to Treplica for execution. The application places its state under care of the framework during the application initialization, when a `Treplica` object is instantiated. In Cyan that is usually done in a method called `run:` in a prototype called `Program`. Figure 6 shows how a Treplica object is declared and initialized in each replica. This is also an example of how an object of the prototype `SetTextAction` is passed as an argument to the method `execute:` of the Treplica object.

```
object Program {
    func run: Array<String> args {
        let info = Info new;
        let treplica = Treplica new;
        treplica runMachine: info
                 numberProcess: 3
                 rtt: 200
                 path: "/var/tmp/magic" ++ args[1];

        let action = SetTextAction new: "text";
        treplica execute: action;
    }
}
```

Figure 6: Treplica configuration and execution

The sequence diagram of Figure 7 shows the execution flow of the example in Figure 6. Both Replicas A and B start execution in the `run` method of prototype `Program`. The context of the application (`Info`) and the Treplica object (`Treplica`) are initialized during the execution of this method. Replica A wants to change the replicated state, so it creates an appropriate action object (`SetTextAction`) and sends an `execute:` message to Treplica with the object as parameter. Treplica will order and distribute the action object to both replicas and it will execute the action on both, independently, by sending an `executeOn:`

message to the local action object.

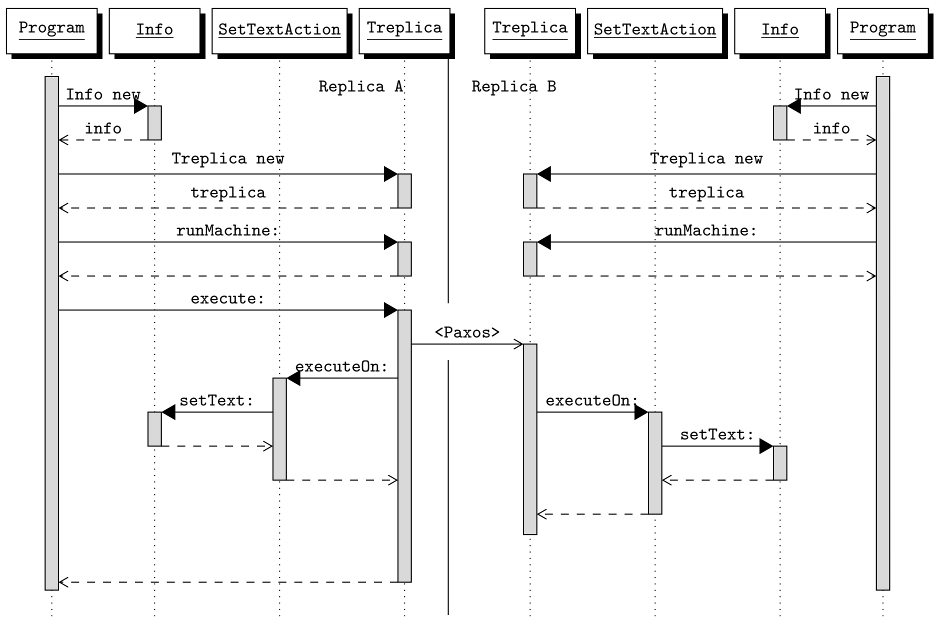

Figure 7: Treplica execution sequence diagram

The way Treplica encodes message passing is very simple and straightforward, but it requires the application programmer to create much boilerplate code in the form of action objects. Moreover, isolating message passing isn't enough to achieve replication, it is necessary that the method activated by the message doesn't create external effects besides changing the context state and that these changes are deterministic. It is the responsibility of the programmer to be aware of these requirements and avoid breaking them.

For example, in Figure 8 we introduce a small change to the `setText:` method of `Info` prototype to make it non-deterministic. The problem brought by the change is that every time the `setText:` method is called the value set will be different, even if the starting state of `Info` and the parameters of `setText:` are the same. This will make the replicas diverge, as the deterministic behavior of the transition will be violated.

```
...
object Info extends Context {
  ...
  func setText: String text {
    var date = System currentTime asString;
    self.text = text ++ date;
  }
  ...
}
```

Figure 8: Non-deterministic `setText:` method

4. Metaobjects for Replication

4.1. Overview and Use

The creation of a replicated application using Treplica requires the construction of a prototype representing the application context and as many actions as messages this context can receive. For each action it is necessary to define a new prototype with the correct number and type of fields, besides sending the correct message to application context when asked by Treplica. We have shown in the last section this task isn't hard, but it requires the tedious and error-prone creation of a lot of boilerplate code. Now we are going to show how programming of replicated application can be simplified by the use of Cyan metaobjects `treplicaAction` and `treplicaInit`. We first describe how to use the metaobjects and in the next section we describe how the metaobjects are created and how they work.

A good programming practice when using Treplica is to create action prototypes that do not have application functional behavior and limit themselves to send a single message with the correct parameters. Figure 5 shows the action `SetTextAction` that represents the sending of message `setText:` in the form of a prototype. It is interesting to notice that this prototype also has a constructor that initializes the fields of the created object with the same parameters used afterwards to send the encoded message. The creation of these prototypes can be standardized, and the metaobject `treplicaAction` will create an appropriate action prototype when attached to a method declaration of the application context.

For example, the `Info` prototype in Figure 9 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 sending of message `setText:` as a Treplica action. Prototype `SetTextAction` of Figure 5 is not necessary any more, since the metaobject `treplicaAction` adds an equivalent prototype to the program during compilation. Moreover, `setText:` messages sent directly to the application context will be "intercepted" and replaced by the creation of an suitable action object and the sending of this object to Treplica as an `execute:` message for replication and execution by the framework.

```
package main
import treplica
object Info extends Context {
    var String text
    ...
    @treplicaAction
    func setText: String text {
        self.text = text;
    }
    ...
}
```

Figure 9: Replicated prototype using metaobjects

Initialization of the application context and of the Treplica framework also happen in a standardized way as shown in Figure 6. The `treplicaInit` metaobject can be attached to declarations of variables whose type is a sub-prototype of `Context`. This metaobject has a double function: it marks a variable as holding the application context and it initializes Treplica. The explicit indication of the object holding the application context is important because only the methods belonging to the context prototype can be marked with the `treplicaAction` metaobject.

For example, in Figure 10 the `treplicaInit` 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.

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

Figure 10: Treplica configuration using metaobjects

Using these two metaobjects only the code in Figures 9 and 10 need to be written. This code is more compact, easier to read and is independent from any replication concerns or implementation details. This way we can avoid some of the pitfalls created by the programming interface of Treplica and create less complex applications.

4.2. Implementation of the Metaobjects

This section shows how the metaobjects `treplicaAction` and `treplicaInit` are implemented in Cyan. The compiled version of both classes, the ".class" file, are put in directory "`--meta`" of package `treplica`. When this package is imported, as in Figure 9, the associated annotations can be used.

`CyanMetaobjectTreplicaAction` is the Java class implementing the metaobject `treplicaAction`. This association happens because its method `getName()`

```
object Info extends Context {
    ...
    func setText: String text {
        var action = InfosetText new: text;
        self getTreplica execute: action;
    }

    func setTextTreplicaAction:
            String text {
        self.text = text;
    }
    ...
}
```

Figure 11: Prototype `Info` modified

returns `"treplicaAction"`. This class implements several interfaces, which are described below together with their role in the metaobject.

(a) `IActionProgramUnit_ati` from which methods `ati_codeToAddToPrototypes` and `ati_renameMethod` are defined. The first one adds a new method with the same name as the annotated method. In the example of Figure 11, the metaobject of Figure 9 adds method `setText:`. Method `ati_renameMethod` of class `CyanMetaobjectTreplicaAction` renames the original annotated method. In the example, `setText:` is renamed to `setTextTreplicaAction:`.

(b) `IActionNewPrototypes_ati` from which method `ati_NewPrototypeList` is redefined for creating a new prototype implementing the Treplica action. In the example, it is prototype `InfosetText` of Figure 12.

(c) `IAction_dsa` from which method `dsa_codeToAdd` replaces calls to non-deterministic methods by calls to deterministic ones. The project directory and the directory of each package may have a rule file called "deterministic" that describes how the replacement should be made. We will describe this non-determinism removal operation in more detail in the next section.

(d) `ICheckDeclaration_dsa2` from which method `dsa2_checkDeclaration` is redefined to check if there are any calls to non-deterministic methods in the

```
object InfosetText extends Action {
    var String textVar
    func init:  String text {
        textVar = text;
    }

    override
    func executeOn: Context context {
        type context
            case Info obj {
                obj setTextTreplicaAction:  textVar;
            }
    }
}
```

Figure 12: Prototype created by `treplicaAction`

final code. This should not be necessary since method `dsa_codeToAdd` replaces all possible calls to non-deterministic methods by calls to deterministic ones. However, only unary methods are replaced and `dsa2_checkDeclaration` checks keywords methods too. Besides that, other metaobjects may have introduced non-deterministic method calls in Phase 6 (if they add code).

Class `CyanMetaobjectTreplicaInit` is the Java class implementing metaobject `treplicaInit`. This class implements interface `IActionVariableDeclaration_dsa` and redefines its method `dsa_codeToAddAfter`. This method adds code after the variable declaration to create and initialize the `Treplica` object. For example, this metaobject takes the annotated code in Figure 10 and adds the initialization code in Lines 5–10 of Figure 13. The newly added code makes `info` reference `treplicainfo`, the treplica object, and vice-versa.

To further illustrate the transformations and code generated by metaobject `treplicaAction`, we will use an annotated method with a single keyword, shown in Figure 14. The code in this figure isn't valid Cyan code, but it is useful to show how the tokens of the method declaration are used by the metaobject to create new code.

```
object Program {
    func run: Array<String> args {
        var local = "/var/tmp/magic" ++ args[1];
        var info = Info new;
        var treplicainfo = Treplica new;
        treplicainfo runMachine: info
                     numberProcess: 3
                     rtt: 200
                     path: local;
        info setTreplica: treplicainfo;
        info setText: "text";
    }
}
```

Figure 13: Prototype `Program` modified

From the method in Figure 14, `treplicaAction` creates a prototype that represents the Treplica action shown in Figure 15. `UniqueId` is a unique temporary name that is different from every other identifier. In the original prototype (`Proto`), the method annotated with `treplicaAction` is renamed and a new method is created as shown in Figure 16.

From a declaration

```
@treplicaInit( processes, rtt, local )
var varName = ContextType new;
```

metaobject `treplicaInit` produces the code shown in Figure 17

4.3. Non-determinism Detection

Besides removing boilerplate code from a program, the proposed metaobjects offer an initial support for validating if the resulting code is indeed able to be replicated. As we have shown in Section 3, methods that change the state of the context must be deterministic and should not create external effects. In this work, we considered only the question of identifying non-deterministic methods and optionally replacing them with deterministic versions. In the previous section we have briefly described these tests and substitutions and now we describe

```
object Proto
  @treplicaAction
  func s1: T1 p1, T2 p2, ... Tn pn { ... }
  ... // other methods
end
```

Figure 14: `treplicaAction` attached to a generic method

```
object Protos1 extends Action
  var T1 p1Var
  var T2 p2Var
  ...
  var Tn pnVar

  func init: T1 p1, T2 p2, ... Tn pn {
    p1Var = p1;
    p2Var = p2;
    ...
    pnVar = pn;
  }

  override
  func executeOn: Context context {
    type context
        case Proto obj {
            obj s1TreplicaAction_UniqueId: p1Var, p2Var,
                ... pnVar;
        }
  }
end
```

Figure 15: New prototype that represents the action of calling a method

```
object Proto extends Context

  @treplicaAction
  func s1: T1 p1, T2 p2, ... Tn pn {
    var action = Protos1 new: p1, p2, ... pn;
    self getTreplica execute: action;
  }

  func s1TreplicaAction_UniqueId: T1 p1, T2 p2, ... Tn pn
      {
    // original method s1:
  }
end
```

Figure 16: Modified prototype `Proto`

```
var varName = ContextType new;
var treplicaVarName = Treplica new;
treplicaVarName runMachine: varName
                numberProcess: processes
                rtt: rtt
                path: local;
varName setTreplica: treplicaVarName;
```

Figure 17: The new declaration of a context

them in a bit more depth.

Identifying deterministic methods is complex and this work does not offer comprehensive solutions to this problem. Instead, *root* non-deterministic methods are defined by the developer in a rule file. These methods are usually operating system services that aren't deterministic by nature, such as reading the time or receiving a packet from the network. Once these root methods are defined as non-deterministic, any other method that uses a non-deterministic method is also flagged as non-deterministic.

Besides identifying a non-deterministic method, the rule file defines a replacement method. This method can be implemented by the application of by a supporting library and should provide a deterministic version of the indicated method. For example, it is possible to define a method that returns not the current time, but a timestamp prerecorded in the action object.

In the rule file, each rule is defined in a line and its format is shown in Figure 18. The rules are split in two parts by the symbol `-`. The first part defines the non-deterministic method, the prototype that defines this method and the package where it can be found. The second part defines a deterministic method that will replace the non-deterministic one, the prototype that defines this method and the package where it can be found. During compilation, the metaobject `treplicaAction` will replace `methodA` with a call to `methodB`, with the parameters of the original call used as parameters of this new call. The prototype that defines `methodB` is not instantiated, it is used in static form.

```
packageA,PrototypeA,methodA-packageB,PrototypeB,methodB
```

Figure 18: Example of non-determinism rule

The replacement of methods does not cover all possibilities of non-determinism removal in a program. Thus, after the initial identification and replacement that happens in Phase 6 of the compiler, another round of checking happens in Phase 9. In this final check, remaining non-deterministic methods will cause a compilation error. This procedure is a first tentative step in the direction of isolating replication related inconsistencies, but it shows it is possible to verify the source code based on the restrictions imposed by the programming environment.

As an example, consider the prototype `Info` defined in Figure 19. This prototype implements an application context and has a method `set:` marked as an action. During compilation, if the developer creates the rule file shown in Figure 20 the method `ageInSecNd`, that has non-deterministic behavior, will be replaced by the deterministic `ageInSec` method.

```
package main
...
object Info extends Context {
    var String text

    // non-deterministic behavior
    func ageInSecNd -> Long {...}

    func ageInSec -> Long {...}

    @treplicaAction
    func set: String text {
        self.text = text ++ " age: "
          ++ ageInSecNd;
    }
    ...
}
```

Figure 19: Example of non-determinism removal

```
main,Info,ageInSecNd-main,Info,ageInSec
```

Figure 20: Non-determinism rule for Info prototype

The class that defines the metaobject `treplicaAction` implements the interface `ICheckDeclaration_dsa2` and defines the method `dsa2_checkDeclaration`. This method performs a depth-first search starting from the method annotated with `@treplicaAction`, and passes through the methods calls recursively. All called methods are verified based on the rules file. The depth-first search returns when there are no more methods to be visited. In the example of Figure 19, `treplicaAction` looks for non-deterministic method calls in `set:` and in any other method `set:` may call. In this case, `ageInSecNd` and in any other methods that `ageInSecNd` may call. The method `dsa2_checkDeclaration` checks which calls are considered as non-deterministic based on the rules file.

5. Related Work

OpenReplica [6] is a framework to implement replicated services similar to Treplica [4]. Along with Treplica, 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. [8] 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 [8] to transform a DSL using active annotations. We use a similar approach of automatic code transformation. However, 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. [9] that proposes the use of metaprogramming to automatically create components that implement design patterns.

Groovy [20], Xtend [8], and Nemerle [21] support compile-time metaprogramming through annotations in the source code. The annotations are called *macros* in Nemerle and *active annotations* in Xtend. We will use the Cyan terms for all of them. In these languages, metaobjects can act in several compilation phases and they can add code, create new classes, do checks. Annotations can be attached to declarations such as classes and methods. Then metaobject `treplicaAction` could be implemented in Groovy, Xtend, and Nemerle except for one point: the checks related to non-deterministic methods. These checks are made in Phase 9 of the Cyan compiler, after every possible code change has already been made.[7] To our knowledge, these languages do not have a compilation phase in which all code changes are prohibited. It may be possible to implement non-deterministim checks using some clever compiler trick, but we are not aware of that.

Groovy, Xtend, and Nemerle do not allow an annotation to be attached to a local variable declaration. Then metaobject `treplicaInit` cannot be directly implemented in these languages. It could, probably, be implemented by indirect means: the annotation would be attached to a method and it would take a local variable name as parameter. Then the metaobject would walk in the method AST and add the code as the Cyan metaobject does.

Chlipala [10] 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 [11] proposes similar validations in which metaprogramming is used to detect defects in object-oriented programs by the use of software components capable of describing and identifying 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 [13] aims to ease the conversion of legacy centralized C++ and Fortran code into portable shared-memory parallel code.

[7]The code can be modified only until Phase 6.

OpenACC [14] 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 simpler 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 extend detection to other types of consistency violations.

Regarding the problem of separating the nonfunctional requirements from the functional ones, there are works on aspect orientation that try to solve the same problem. AspectJ [23] 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 which 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.

6. 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. The set of metaobjects created showed for the first time that it possible to automatically create replicated code using metaprogramming.

We also demonstrated the power and simplicity of the MOP of the Cyan language to create useful metaobjects in a very direct way. The programmer of a metaobject in Cyan does not have to deal with the AST to generate code, she only needs to insert textual source code directly to the original source code. However, the programmer can use the AST if necessary to know what the compiler knows about the code being compiled.

Moreover, we demonstrated the potential of using metaobjects to validate the generated code with respect to the presence of non-determinism, by replacing the non-deterministic operations with equivalent deterministic operations. We believe the technique presented to detect non-determinism can be extended to other types of violations of replication integrity, such as calling static methods outside the application context. In the future, we envision an application environment where distributed programming errors, one of the main factors limiting the use of this programming paradigm, can be directly found by the compiler. Also, the ability to create isolated, deterministic operations seems to be very useful in the creation of tests suites.

Acknowledgments

This work was supported by the São Paulo Research Foundation (FAPESP) under grant #2014/01817-3 and by FIT - Instituto de Tecnologia.

References

References

[1] W. Vogels, Eventually consistent, Commun. ACM 52 (1) (2009) 40–44. `doi:10.1145/1435417.1435432`.

[2] F. B. Schneider, Implementing fault-tolerant services using the state machine approach: A tutorial, ACM Computing Surveys (CSUR) 22 (4) (1990) 299–319.

[3] M. Burrows, The Chubby lock service for loosely-coupled distributed systems, in: Proceedings of the 7th symposium on Operating systems design and implementation, USENIX Association, 2006, pp. 335–350.

[4] G. M. D. Vieira, L. E. Buzato, Treplica: Ubiquitous replication, in: SBRC '08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems, Rio de Janeiro, Brasil, 2008.

[5] G. M. D. Vieira, L. E. Buzato, Implementation of an object-oriented specification for active replication using consensus, Tech. Rep. IC-10-26, Institute of Computing, University of Campinas (Aug. 2010).

[6] D. Altınbüken, E. G. Sirer, Commodifying replicated state machines with OpenReplica, Tech. rep., Cornell University, Technical Report (2012).

[7] R. Damaševičius, V. Štuikys, Taxonomy of the fundamental concepts of metaprogramming, Information Technology and Control 37 (2) (2015).

[8] A. Rentschler, D. Werle, Q. Noorshams, L. Happe, R. Reussner, Designing information hiding modularity for model transformation languages, in: Proceedings of the 13th international conference on Modularity, ACM, 2014, pp. 217–228.

[9] A. Blewitt, A. Bundy, I. Stark, Automatic verification of design patterns in Java, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM, 2005, pp. 224–232.

[10] A. Chlipala, The bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier, in: ACM SIGPLAN Notices, Vol. 48, ACM, 2013, pp. 391–402.

[11] S. Mekruksavanich, P. P. Yupapin, P. Muenchaisri, Analytical learning based on a meta-programming approach for the detection of object-oriented design defects, Information Technology Journal 11 (12) (2012) 1677.

[12] J. d. O. Guimaraes, The Cyan language, Tech. rep., Campus de Sorocaba da UFSCar (2008).
URL `http://www.cyan-lang.org`

[13] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE computational science and engineering 5 (1) (1998) 46–55.

[14] S. Wienke, P. Springer, C. Terboven, D. an Mey, OpenACC—first experiences with real-world applications, in: European Conference on Parallel Processing, Springer, 2012, pp. 859–870.

[15] G. Blaschek, Object-oriented programming with prototypes, Springer, 1994.

[16] J. Gosling, B. Joy, G. L. Steele, G. Bracha, A. Buckley, The Java Language Specification, Java SE 8 Edition, 1st Edition, Addison-Wesley Professional, 2014.

[17] B. Stroustrup, The C++ Programming Language, 4th Edition, Addison-Wesley Professional, 2013.

[18] C# language specification (Sep. 2014).
URL `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`

[19] A. Goldberg, D. Robson, Smalltalk-80: The Language and Its Implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[20] The Groovy language (2017).
URL `http://groovy-lang.org`

[21] K. Skalski, Syntax-extending and type-reflecting macros in an object-oriented language, Master's thesis, University of Wroclaw, Poland (2005).

[22] L. Lamport, Fast Paxos, Distributed Computing 19 (2) (2006) 79–103.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, Lecture Notes in Computer Science 2072 (2001) 327–355.