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, which are 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 [1, 2]. Among these techniques, a very straightforward and studied one is called *active replication* [3].

The principle underlying this technique is to consider the system being replicated as a deterministic state machine, which 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 transitions 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 to 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.

*Corresponding author

Email addresses: ugliara.fellipe@gmail.com (Fellipe A. Ugliara), gdvieira@ufscar.br (Gustavo M. D. Vieira), jose@ufscar.br (José de O. Guimarães)

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 nonfunctional 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], to detect defects in object-oriented programs [11], produce code at runtime [12], change the syntax of a language (as Lisp macros) [13], and implement pluggable type systems [14].

In this paper we show how to use the metaprogramming infrastructure of the Cyan language [15] 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 [16], OpenACC [17] and other systems that use compiler directives to guide the automatic generation of parallel code. However, our approach is easier to use, demanding just the addition of some annotations to methods and variable declarations, 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 nondeterminism in transitions by flagging mutator methods that would violate this requirement. Our proposed set of metaobjects is able to replace nondeterministic methods with deterministic versions and alert the programmer if it still finds a call to a nondeterministic 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. Section 5 shows the application of the metaobjects to a more complex program as a demonstration of the feasibility of our approach. The paper ends with a review of related work in Section 6 and some concluding remarks in Section 7. The source code of the Cyan compiler used, the set of metaobjects and example applications can be found at `https://bitbucket.org/gdvieira/cyan.treplica.git`.

2. The Cyan Language

2.1. Language Overview

The language used in this paper is Cyan [15], a prototype-based object-oriented language. Unlike most prototype-based languages, Cyan is statically typed as Omega [18], 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 [19], C++ [20], or C♯ [21] 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 the 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 `name` in Line 17). Fields can only be private in Cyan, it is optional to use keyword `private` before a field declaration. `self` refers to the object that received the message. The same as `self` in Smalltalk [12] or `this` in Java, C♯, or C++.

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 `Int` can be initialized 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

```
let counter = 0;
```

The variable name is `counter` and its type, `Int`, 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 `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 `-counter` are also considered unary messages. In this example, message `-` is being sent to object referred to by `counter`.

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:

```
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`.

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` or `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
```

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

[2] Even if `aBuilding` has type `Building`, it may refer to an object whose type is a subprototype of `Building`.

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"
```

It is a compile-time error to send a message `new` or `new:` to anything that is not a prototype.

In prototype-based languages, prototypes are objects and they can be used in expressions. For example, `Int` is an object whose value is `0`.

```
var Int two = (Int + 1)*2 + Int*Int;
assert two == 2;
  //  ++ transforms both arguments to strings and
  //  concatenates them
let String strZero = (Int asString) ++ 1;
assert strZero == "01";
```

However, In Cyan, only prototypes that declare an `init` method, a constructor without parameters, are considered objects with full rights. The `init` method is used to build the object that is accessed by the prototype name (as `Int` in the expressions of the last example). This assures that the prototype fields are correctly initialized. That would not be the case with prototype `Building` of Figure 1, which does not declare a method `init`. If the first message passing to `Building` is

```
Building getName
```

there would be an error: field `name` would not have been initialized. A prototype without an `init` can receive some kinds of messages such as `new` and `new:`.

In Cyan, prototypes play a dual role: 1) they are types as with classes in Smalltalk, Java, and C++, and 2) when used in expressions, they work like variables that refer to a fixed object of themselves. For example, `Int`, when used inside an expression, refers to an object of prototype `Int` (itself) whose value is `0`.

Keyword `extends` allows the 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 execution time [22] [7]. Metaprogramming has a broad meaning, it encompasses any kind of program handling at compile-time, loading time (when the program is loaded into memory), and at runtime. In this paper, we will limit ourselves to 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 can 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.

In this paper, we consider that 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 or Cyan prototypes. The compiler is written in Java, which makes it easy to write code in this language that interacts with the compiler. Since the Cyan compiler translates code to Java, the metaprogram can be written in Cyan too.

During the compilation of a program, an *annotation* in the source code links the program (base level), the compiler, and the metaprogram. An annotation is `@` followed by an identifier, optional parameters, and an optional DSL:

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

```
1   var Int sum = @eval("cyan.lang", "Int"){*
2       var Int count = 0;
3       for n in 1..10 {
4         count = count + n
5       }
6       return count;
7     *};
```

In this case, the optional DSL is given between {* and *}. It consists of Cyan code that is interpreted at compile-time. The result is the same as to replace the annotation by `55`. The metaprogram calls a Cyan interpreter at compile-time to produce the result. We will use just the identifier as the annotation name, as in "annotation `eval`".

```
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          Out println: ("This is method " ++
18             @compilationInfo(currentmethodname);
19      }
20
21  end
```

Figure 2: Metaobjects in Cyan

The example of Figure 2 uses two annotations: `init` in Line 2 and `compilationInfo` in Line 18. For each annotation, the compiler creates a *metaobject* of a *metaobject class*. This class is written in Java,[4] as the compiler is, and is associated with a Cyan package. When the package is imported by a Cyan program, the metaobject class is imported too. Then new behavior and new checks can be added to the code by importing packages and using annotations. To simplify, we will say "metaobject `init`" and "class of `init`" for the metaobject associated with 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 classes of `init` and `compilationInfo`. Thus, these annotations can be used in any Cyan source code without explicitly importing a package, as is done in the example of Figure 2. Then, the scheme the Cyan compiler uses for creating metaobjects is: a) when parsing the code and an annotation is found, look for a metaobject class in the imported packages. Method `getName` of the metaobject class should always return the annotation name; b) if there is no metaobject class, issue an error. Otherwise, create an object of the metaobject class and associates it with the annotation. For each annotation there is a unique metaobject and vice-versa.

During compilation, the Cyan compiler calls methods of the metaobjects associated with the annotations. They can insert code and do checks in the code based on information supplied by the compiler. In this example, metaobject `init` generates a constructor for `Person`. Figure 3 shows the resulting `Person` prototype.

[4]Cyan can be used as the metaprogramming language. However, in this paper, we will consider that all metaobjects are made in Java, which is the language used to code all metaobjects cited in the text.

The constructor code is returned by the metaobject method as a string (in fact, a `StringBuilder` object) and inserted in the source code by the compiler. Only the source code in memory is changed, the original file with prototype `Person` is not changed. Annotation `compilationInfo` produces the current method name, `"run"`.

Since the metaobject inserted an `init:` method in the prototype, method `run` of `Program` can create objects of `Person` using the `new:` method (constructor). Note that the Java class of metaobject `init` knows the types of fields `name` and `age`. These types are necessary to generate the constructor. This information is supplied to the metaobject by the compiler.

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

Figure 3: *Person* generated during compilation

In the remaining of this section, we will give a simplified overview of the Cyan MOP. The Metaobject Protocol can only be understood by studying the compilation phases of the Cyan compiler, shown in Figure 4. The compilation phases are parsing, resTypes (resolve types), afterResTypes (after resolving types), semAn (semantic analysis), afterSemAn (after semantic analysis), and code generation. The arrows indicate the data passed from one phase to the next. The data is described in the right-hand side of the Figure by the labels. Then, the arrow from parsing to phase resTypes is labeled with *(a)*, which represents the Abstract Syntax Tree (AST). Parsing uses only local information, available in the current source code. Therefore, after parsing there is no type information in the AST. For example, for a method parameter that has type `Int`, the AST keeps only the string "`Int`". In phase resTypes, the compiler looks for a prototype `Int` and assigns a reference to its AST to a field `type` of the AST object that represents the method parameter. All references to types outside method statements are resolved in phase resTypes.

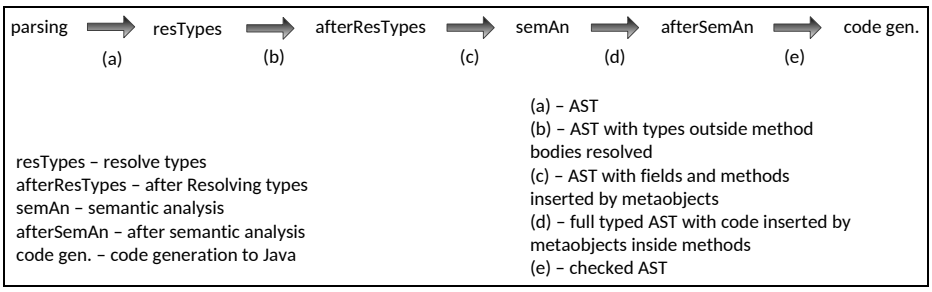

Figure 4: The Cyan Compilation Phases

In phase afterResTypes, metaobjects can add fields and methods to the prototype in which their annotations are. The metaobject associated with an annotation that is in a prototype cannot add code to another prototype. A method of metaobject `init` is called in phase afterResTypes. It returns, as a string[5], the code to be added, an `init:` method. A copy of the source code, in memory, is changed and therefore

[5] As a `StringBuffer` object, in fact.

the compilation starts again from phase parsing (but only for this file). Metaobject methods called in phase afterResTypes have access to the AST of the prototype in which they are, except method statements. Then, metaobject methods called in this phase know the prototype name, superprototype, implemented interfaces, the name and type of fields, and method signatures.[6] Metaobjects can then use this information to generate code, as does `init`.

Phase semAn is called "the semantic analysis" even though it only finishes this analysis, started in phase resTypes. In phase parsing, types are assigned to expressions inside methods. Every expression object of the AST has a "`type`" field that is `null` till the start of phase semAn. After this phase all "`type`" fields are resolved. Metaobjects can generate code inside methods only in this phase. This is what metaobject `compilationInfo` does in the example of Figure 2. It produces the current method name as a string.

After phase semAn, metaobjects cannot change the code anymore. But they can do checks in the code in phase afterSemAn. This phase is ideal for checks because there will be no further changes that can invalidate the checks. Currently, metaobjects cannot act in the last phase, code generation.

Metaobject methods can be called in phases parsing, afterResTypes, semAn, and afterSemAn.[7] But how the compiler knows which method it should call in each phase? The answer is in the interfaces implemented by the classes of the metaobjects. The Cyan MOP provides several Java (and Cyan) interfaces that should be implemented by metaobjects. Each interface is associated with a compilation phase. Whenever a metaobject class implements an interface of phase afterResTypes, for example, all metaobject methods that override the interface methods are called in phase afterResTypes. As a concrete example, the class of metaobject `init` is `CyanMetaobjectInit` that inherits from class `CyanMetaobjectAtAnnot` (as any metaobject class of this paper) and implements interface `IAction_afterResTypes`. This interface declares a method `afterResTypes_codeToAdd` that is overridden by `CyanMetaobjectInit`. This method is called in phase afterResTypes by the compiler.

During the parsing of prototype `Person` of Figure 2, the compiler creates an object of class `CyanMetaobjectInit` when it finds the annotation `init` in line 2. The name "`init`" is associated with this class because the compiled form of it, a Java "`.class`" file, is in a directory of package `cyan.lang` automatically imported by every Cyan source file. And method `getName()` of an object of this class returns `"init"`. The compiler knows how to associate an annotation to a metaobject class.

The `CyanMetaobjectInit` object created in phase parsing is the *metaobject* associated with annotation `init`. Method `afterResTypes_codeToAdd` of this metaobject is called in phase afterResTypes to produce the Cyan `init:` method, which is then inserted in the code by the compiler. All methods of interface `IAction_afterResTypes` are called on the metaobject. The other methods, in this case, do nothing.

The class of metaobject `compilationInfo` implements interface `IAction_semAn` which declares a single method, `semAn_codeToAdd`. During the semantic analysis, phase semAn, the compiler calls method `semAn_codeToAdd` of the metaobject associated with annotation `compilationInfo` of line 18 of Figure 2. This method produces the string `"run"` which is inserted in the code by the compiler.

Note that there is a one-to-one relationship between annotations and metaobjects. Two `compilationInfo` annotations, even with the same parameters, will cause the creation of two metaobjects in phase parsing. Annotations can be attached to declarations such as prototypes, methods, local variables, packages, and the program. Or they can be *free* as `compilationInfo`, which is not attached to anything.

There are several *other* interfaces associated with phases parsing, afterResTypes, semAn, and afterSemAn. They are used to:

1. parse the DSL attached to the annotation (parsing);
2. create new prototypes (parsing, afterResTypes, semAn);
3. intercept field access (semAn);
4. intercept message passings (semAn);
5. create code after a local variable declaration (semAn);
6. simulate the existence of fields (semAn);

[6] A signature of a method is composed of its name, name and type of parameters, and return value type.
[7] Call to metaobject methods in phase parsing will soon be deprecated.

7. simulate the existence of methods (semAn);
8. intercept the overridding of a method (afterSemAn);
9. intercept the inheritance of a prototype (afterSemAn);
10. check any kind of declaration (afterSemAn);
11. check message passing (afterSemAn).

Metaobjects are used, in this paper, for replication. We will show the details of metaobject implementation later on using the replication metaobjects.

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 with the purpose of characterizing the programming effort required to program a replicated application using the framework. The source code of this binding can be found in the project tree[8] under `lib/treplica`.

A complete description of Treplica is beyond the scope of this paper. A full description of Treplica, including its original Java programming interface, can be found in [5]. Nonetheless, we will briefly summarize two properties of Treplica that are important for understanding how to use the framework: its consistency guarantee and the way active replication works.

Replicated applications can be classified in two types reflecting the way updates are propagated: eager and lazy [1]. Assume a client is contacting a replicated server to execute a service, and that the execution of this service changes the state of the server. Eager replication propagates and ensures that all changes are stable in all replicas before returning to the client. Lazy replication executes and returns to the client immediately, propagating the changes at a later, more convenient time. Thus, in lazy replication the replicas can diverge immediately after an update is made and later converge back to a shared state, while in eager replication the replicas never diverge.

Lazy replication has the advantage that it exchanges fewer messages, resulting in lower consumption of system resources and less delay associated with message exchange. This is a consequence of the flexibility to decide when to propagate changes. Lazy replication protocols are therefore faster than eager protocols that must exchange messages before a change is made to a replica [1]. However, to achieve this higher efficiency lazy replication relaxes the consistency of the changes made to the replicas, leading to inconsistencies and requiring conciliation of the resulting data [1].

For example, suppose a bank system that uses lazy replication and stores a checking account with $100 in it. Two independent purchases of $100 can be made concurrently in different replicas of the bank system. At this moment the system is inconsistent, because the owner of the account has spent more than was available. But, eventually a process of reconciliation will happen and, for example, one of the purchases will be canceled for lack of funds. From the point of view of the system this is simple, but it may be a nuisance to both client and merchant. Worse still, if the goods were delivered, it may even be impossible to revert such transactions. Thus, the application programmer has to be aware of the need for reconciliation in every transaction, making the application more complicated.

Eager replication is more costly, but it does not suffer from the problem created by divergence and reconciliation of data. Data changes in a eager replication protocol usually guarantee a criterion of consistency called one-copy-serializability [23]. This criterion specifies that concurrent changes to replicated objects appear as a series of changes over a single logical copy. Solutions that use this type of consistency bear more similarities to centralized systems and are easier to reason about and program [3]. However, even in this case, the programming of these applications still poses significant challenges [24].

Active replication is a type of eager replication that 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

[8] `https://bitbucket.org/gdvieira/cyan.treplica.git`

application operation. It works by assuming the application behaves like a deterministic state machine, which 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 the presence of failures [25]. 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. In the Treplica framework this object should extend the `Context` prototype and be serializable.[9] Serialization allows an object to be transformed to text, transmitted between different hosts and transformed back to a clone of the original object. Figure 5 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.

```
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 5: 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 passing 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 6 shows the prototype `SetTextAction`, which implements a transition that sends message `setText:` to an `Info` object. Statement `type-case` is a safe way of downcasting in Cyan. `type` may be followed by one or more `case` clauses. The statement tries to cast the argument to `type` to the type that appears after `case`. In this

[9] In Cyan, every prototype is serializable.

example, the statement tries to cast `context` to `Info`. If this succeeds, the value is put in variable `info` and the `case` statements are executed.

```
object SetTextAction extends Action {
    var String updateText

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

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

Figure 6: 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`.[10] Figure 7 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[0];

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

Figure 7: Treplica configuration and execution

The sequence diagram of Figure 8 shows the execution flow of the example in Figure 7. 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

[10] This is the default that can be changed in the *project file*, a file that declares the packages and other configurations of a program.

object to both replicas and it will execute the action on both, independently, by sending an `executeOn:` message to the local action object.

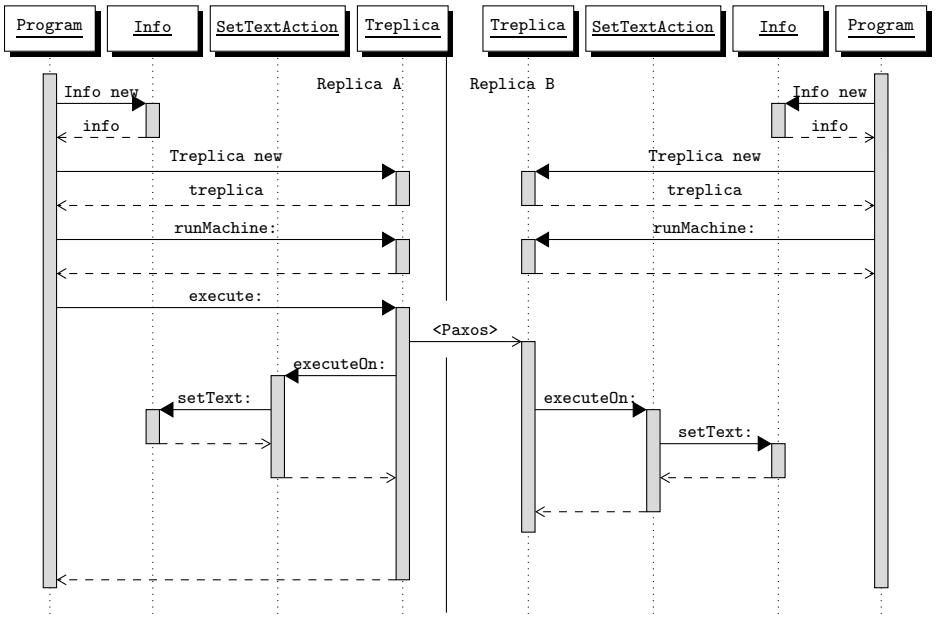

Figure 8: Treplica execution sequence diagram

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 9 we introduce a small change to the `setText:` method of `Info` prototype to make it nondeterministic. 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. Later in the paper, we show how this problem can be detected and solved.

```
1 ...
2 object Info extends Context {
3   ...
4   func setText: String text {
5     var date = System currentTime asString;
6     self.text = text ++ date;
7   }
8   ...
9 }
```

Figure 9: Nondeterministic `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 the 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. 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. Although Treplica does not impose these restrictions, by creating data-only action prototypes the coupling between the application and Treplica is reduced. Loosely coupled modules are easier to reuse [26] and align better with replication as a nonfunctional requirement. For example, Figure 6 shows the action `SetTextAction` that represents the sending of message `setText:` in the form of a prototype. Notice that this prototype, besides its `executeOn:` method, has a simple constructor that initializes the fields of the created object with the same parameters used afterwards to send the encoded message. Because of this regularity, the creation of these prototypes can be standardized. The metaobject `treplicaAction` will create an appropriate action prototype when its annotation is attached to a method declaration of the application context.

For example, the `Info` prototype in Figure 10 is similar to the one depicted in Figure 5, except that the `treplicaAction` annotation 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 6 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 a suitable action object and the sending of this object to Treplica as an `execute:` message for replication and execution by the framework.

Metaobject `treplicaAction` replaces nondeterministic message passings inside the annotated method by deterministic message passings. A list of replacements is given in files loaded at the start of the compilation — the details of how this is made will be given later on.

```
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 10: Replicated prototype using metaobjects

Initialization of the application context and of the Treplica framework also happen in a standardized way as shown in Figure 7. This initialization is automated by the `treplicaInit` metaobject, whose annotations can be attached to variable declarations whose types are subprototypes 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 11 the `treplicaInit` metaobject is attached to variable `info`, with the parameters of the desired Treplica instance. This metaobject adds code after the variable declaration, during compilation, to create a new instance of `Treplica` and assign to it the object `info`, similar to the method `run` in Figure 7. Note that the annotation `treplicaInit` can be attached to the declaration because the type of `info`, `Info`, is a subprototype of `Context`.

```
object Program {
    func run: Array<String> args {
        var local = "/var/tmp/magic" ++ args[0];
        @treplicaInit( 3, 200, local )
        var info = Info new;
        info setText: "text";
    }
}
```

Figure 11: Treplica configuration using metaobjects

Using these two metaobjects only the code in Figures 10 and 11 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. We describe the interface of metaobjects with the Cyan MOP in Java. The source code of these metaobjects can be found in the project tree[11] under `proj/meta/treplica`. 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 10, the associated annotations can be used.

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

Figure 12: Prototype `Info` modified

`CyanMetaobjectTreplicaAction` is the Java class implementing the metaobject `treplicaAction`. This association happens because its method `getName()` returns `"treplicaAction"`. This class implements several interfaces, which are described below together with their role in the metaobject.

(a) `IAction_afterResTypes` from which methods `afterResTypes_codeToAdd` and `afterResTypes_renameMethod` are defined. Method `afterResTypes_codeToAdd` adds a new method with the same name as the annotated method. In the example of Figure 12, the metaobject of Figure 10 adds method `setText:`. The code of `setText:` is produced as a string based on the data of the original method. The needed information is the method name, the parameter names and types, and return value type. The first line of method `afterResTypes_codeToAdd` is

```
WrMethodDec md = (WrMethodDec) this.getAttachedDeclaration();
```

`md` refers to the object of the AST that represents the method annotated with `treplicaAction`, `setText:`. Using `md`, we can get the method name, list of parameters, and return value type with the following method calls:

[11] `https://bitbucket.org/gdvieira/cyan.treplica.git`

```
1 md.getName()
2 md.getMethodSignature().getParameterList()
3 md.getMethodSignature().getReturnTypeExpr().getType()
```

This is all the information needed to create a new `setText:` method in string format, which is returned by method `afterResTypes_codeToAdd`.
Method `afterResTypes_renameMethod` renames the original annotated method. In the example, `setText:` is renamed to `setTextTreplicaAction:`. The information needed for this is obtained as in method `afterResTypes_codeToAdd`.

(b) `IActionNewPrototypes_afterResTypes` from which method `afterResTypes_NewPrototypeList` is redefined for creating a new prototype implementing the Treplica action. In the example, it is prototype `InfosetText` of Figure 13. The new prototype is created as a string. The information needed for that is got as in method `afterResTypes_codeToAdd`.

(c) `IAction_semAn` from which method `semAn_codeToAdd` replaces calls to nondeterministic methods by calls to deterministic ones. The list of nondeterministic methods and their replacements are put in files loaded before the compilation starts. We will describe this nondeterminism removal operation in more detail in the next section.

```
1  object InfosetText extends Action {
2      var String textVar
3      func init:  String text {
4          self.textVar = text;
5      }
6
7      override
8      func executeOn: Context context {
9          type context
10             case Info obj {
11                 obj setTextTreplicaAction:  textVar;
12             }
13     }
14 }
```

Figure 13: Prototype created by `treplicaAction`

(d) `ICheckDeclaration_afterSemAn` from which method `afterSemAn_checkDeclaration` is redefined to check if there are any calls to nondeterministic methods in the final code. This should not be necessary since method `semAn_codeToAdd` replaces all possible calls to nondeterministic methods by calls to deterministic ones. However, other metaobjects may have introduced nondeterministic method calls in phase semAn (if they add code, this code will not be visible to `treplicaAction` in this phase).

Class `CyanMetaobjectTreplicaInit` is the Java class implementing metaobject `treplicaInit`. This class implements interface `IActionVariableDeclaration_semAn` and redefines its method `semAn_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 11 and adds the initialization code in Lines 5–10 of Figure 14. The newly added code makes `info` reference `treplicainfo`, the treplica object, and vice-versa.

4.3. Nondeterminism Detection

Besides removing boilerplate code from a program, the proposed metaobjects offer 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 nondeterministic methods and optionally replacing them with deterministic versions. In the previous section we have briefly cited these tests and substitutions and now we describe them in a bit more depth.

```
object Program {
    func run: Array<String> args {
        var local = "/var/tmp/magic" ++ args[0];
        var info = Info new;
        var treplicainfo = Treplica new;
        treplicainfo runMachine: info
                     numberProcess: 3
                     rtt: 200
                     path: local;
        info setTreplica: treplicainfo;
        info setText: "text";
    }
}
```

Figure 14: Prototype `Program` modified

Identifying deterministic methods is complex and this work does not offer comprehensive solutions to this problem. 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. Nondeterministic methods and deterministic replacements for them are defined by the developer in *rule files*. Based on the information of these *rule files*, metaobject `treplicaAction` can replace nondeterministic calls in the annotated method by deterministic method calls.

Rule files are loaded in memory by metaobject `loadNonDeterministicFiles` whose annotation should be attached to the program in the *project file*. Every Cyan program should have a *project file* that describes its packages. The simplest example is just

```
program
    package treplica at "C:\projeto\lib\treplica"
    package cyan.math at "C:\Cyan\lib\cyan\math"
    package main
    package other
```

The compiler will consider that the program is composed of packages `treplica` and `cyan.math`, in the directories given after "`at`", and `main` and `other`, that should be in the same directory as the project file. The files of a package in Cyan should be in a directory whose name is the package name.

Annotations can be attached to the program and to packages:

```
import treplica at "C:\projeto\lib\treplica"
@loadNonDeterministicFiles(
 "C:\calculator\main\--data\nonDeter_to_DeterMethod.txt")
program

    package treplica at "C:\projeto\lib\treplica"
    package cyan.math at "C:\Cyan\lib\cyan\math"
    package main
    package other
```

Package `treplica` imported in the first line has metaobject `loadNonDeterministicFiles` used in line 2. This metaobject reads the rule files given as parameters (any number of them) and creates a data structure with the loaded data. This data structure is retrieved by metaobject `treplicaAction` in methods `semAn_codeToAdd` and `afterSemAn_checkDeclaration`. The first is responsible to replace a nondeterministic method by a deterministic one. The second checks if no other metaobject added a nondeterminitic call in phase semAn (this code addition would not be visible to method `semAn_codeToAdd`).

For each nondeterministic method, a 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 a rule file, each rule is defined in a line and its format is shown in Figure 15. The rules are split into two parts by the symbol `-`. The first part defines the package, prototype, and name of the nondeterministic

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

Figure 15: Example of nondeterminism rule

method. The method name of a *keyword method* is composed of each keyword followed by its number of parameters. Then, method

```
    func add: Int n, String s  to: Int n
```

has name "`add:2 to:1`".

The second part of the line, after `-`, defines the package, prototype, and name of the deterministic method that will replace the nondeterministic one.

During compilation, 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 object that receives message `methodB` is `packageB.PrototypeB`

For example, suppose the line below is in one of the rule files used by the program.

```
util, MyArray, add:2 to:1 - other, DetArray, always:2 theSame:1
```

And a method annotated with `treplicaAction` has the code:

```
  // package util was imported
var MyArray array = MyArray new;
array add: 0, ("aazero" substring: 2) to: (Fat fat: 5);
```

Then `treplicaAction` will replace the message passing of the last line by

```
other.DetArray always: 0, ("aazero" substring: 2)
               theSame: (Fat fat: 5);
```

If an expression that is an argument to this message passing has a nondeterministic call, it would be replaced according to the rules. That is, the replacement works as expected.

Annotation `nonDeterministic` can be attached to a method to mark it as nondeterministic. The associated metaobject does some checks, described next.

1. It checks if all superprototype methods with the same name are also attached to the same annotation.
2. Whenever the method is overridden in a subprototype, the metaobject checks if the overridden method is also attached to annotation
   `nonDeterministic`
   Then, by this item and the previous one, all methods with the same name in a hierarchy are either all marked as nondeterministic or none is.
3. Whenever there is a message passing in which the attached method may be called, the metaobject checks whether the current method, the method in which the message passing is, is annotated with "`@nonDeterministic`" or "`@treplicaAction`". If not, it issues an error because the method is also nondeterministic, not marked as such, and the message passing will not be replaced by a deterministic method call.

Metaobject `treplicaAction` uses the data from the rule files to replace some message passings by other ones. It does so even if the method to be replaced is not annotated with `nonDeterministic`. However, if a method is annotated with `nonDeterministic` and no replacement for it is specified in any rule file, `treplicaAction` issues an error.

There are limitations in the detection of nondeterminism described in this Subsection. It depends on humans to annotate methods as nondeterministic and to put the replacement method call in rule files. There is no help from tools other than metaobject `nonDeterministic`. A tool that does static analysis of the code could detect nondeterminism based on any direct or indirect use of some methods for input and output, time, network, etc. This will be future work.

Rule files may be inconsistent and this is not detected by the metaobjects. The inconsistent arises when the rule files describe the replacement of some methods of a hierarchy but not for others with the same name. Using the previous `MyArray` example, suppose we have the message passing:

```
1 var MySuperArray array = MyArray new;
2 array add: 0, ("aazero" substring: 2) to: (Fat fat: 5);
```

Assume that `MySuperArray` is superprototype of `MyArray` and that there is no replacement, in the rule files, for method "`add:2 to:1`" of `util.MySuperArray`. Then the message send of the last line will not be replaced, even though it is nondeterministic according to one of the lines of a rule file. Metaobject `treplicaAction` is not smart enough to detect this nondeterminism.

A complete solution to this kind of problem demands the knowledge of AST of the whole program. Then a metaobject would do a static analysis in the whole program and detect all nondeterminism based on the *root* nondeterministic methods pointed out in the rule files or annotated with `nonDeterministic`. For example, a method not cited in any rule file and not annotated with `nonDeterministic` would be considered nondeterministic if it may call a nondeterministic method.[12] Currently, that cannot be done because the MOP limits the information given to metaobjects. These limitations prevent some bad Software Engineering practices but also the implementation of some useful tools. See Guimarães [27] for more details.

5. Case Study: Warriors and Mages

Active replication allows us to create applications that share state among different instances without a central data repository. Peer-to-peer games, in particular turn-based "board" games, are the perfect match for active replication. These games have a complex state that need to be shared, composed of the game board, the pieces, the position of each piece, the current player, among others. This state changes after each player's turn as a consequence of her actions, and must be propagated to the other players before they take their turns. We created a turn-based game using Cyan and Treplica, to assess the suitability of creating a replicated application through the set of metaobjects presented in this paper.

The two-player board game *Warriors and Mages* simulates a battle for dominance of the board. Each player controls a group of three characters (two warriors and a mage), and players take turns moving or attacking. In her turn a player can make three of these actions before the other player's turn begins. The game ends when one player eliminates all the characters of the opposing player. As validation of the proposed metaobjects and as an example of how a designer would use them, we have implemented Warriors and Mages using Cyan and a very simple text-based user interface. In this section we will briefly describe the software architecture of our implementation as it was first designed: a centralized, nonpersistent application. Then, we will show the simple steps required to turn it into a replicated, fault-tolerant application. The source code of the final application can be found in the project tree[13] under `app/rogue`.

5.1. Implementation Outline

The implementation of the board game follows a very simple MVC software architecture. The model is implemented by the prototype `Board`. The player commands (move or attack) are sent as method calls to `Board`, that executes them accordingly to the rules of the game. For instance, a player is not allowed to make an illegal move. The execution of the move by prototype `Board` updates the internal state of the game, allowing the correct execution of further moves.

The prototype `Window` implements the view, using a simple Java Swing text field containing a visual representation of the board as shown in Figure 16. Each letter or symbol represents an element of the board. A (`.`) is free space, a (`?`) is a mage, a (`@`) the warrior and walls are represented by a (`#`). Besides showing the current state of the board, the prototype `Window` receives keyboard inputs and passes them to the controller. The commands are simple letters (S, M, A), converted to select, move or attack, respectively.

[12] Note that the definition of "nondeterministic method" is recursive: a method is nondeterministic if a) it is cited in a rule file or annotated with `nonDeterministic` or b) it may call a nondeterministic method.

[13] `https://bitbucket.org/gdvieira/cyan.treplica.git`

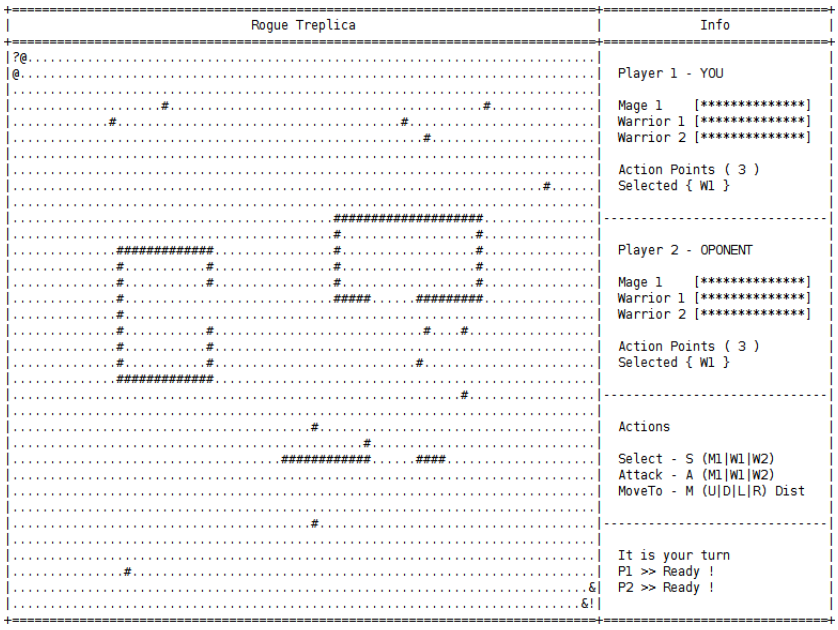

Figure 16: The game board

The controller is implemented by prototype `Program`. This prototype creates the board and the window, acting as a bridge between the two. It is responsible for the initialization of the application. Also, it receives the callbacks from prototype `Window` and calls appropriate methods of prototype `Board`.

To implement the complete game 14 prototypes were created, including these three central prototypes (`Board`, `Program` and `Window`). Figure 17 shows a simplified UML diagram of the main application's prototypes. To give a sense of the style and complexity of the application, we will briefly describe the most important prototypes and their relation.

Prototype `Board` holds the game state and implements its rules. The rules are implemented by key methods of `Board` that check if the a player move is valid and change the state of the board accordingly. These central methods are `startAction:`, `selectAction:`, `moveAction:` and `attackAction:`, which allows to reset the game to its initial configuration, select a piece, move a piece or perform an attack, respectively. Other accessory methods of `Board`, such as `getPlayer:`, `checkEndTurn:` and `checkValidTarget:`, allow checking game rules conditions without changing the local state.

The current state of the game is held by the `map` attribute of `Board`, an instance of prototype `Map`. This prototype holds a matrix with the positions of all elements present on the board, each of them a sub-prototype of `Entity`. The prototype `Entity` abstracts the type of each element and also houses the abstract `draw` method, responsible for creating a representation of each element on screen. There are two concrete sub-prototypes of `Entity`: `Figure` and `Tile`. The prototype `Figure` represents a piece on the board, either a mage of a warrior. This prototype holds information about the current status of each piece and its capabilities, providing methods to query and change this state respecting the games rules. The prototype `Tile` represents the floor and obstacles that define the playable board.

Finally, the prototype `Board` keeps an array of `Player` instances and an attribute `currentPlayer`, allowing it to keep track of which player is allowed to play at each turn.

5.2. Replication Using Metaobjects

Once the application was built, we wanted to enable two players in different machines using replication. We assumed the simple model where each player executes a local instance of the game, which stores a replica of the game state. Players take turns, and the moves each player executes give the flow of the game: after three moves are made by a player, it is the other player's turn. Each player executes her moves as operations

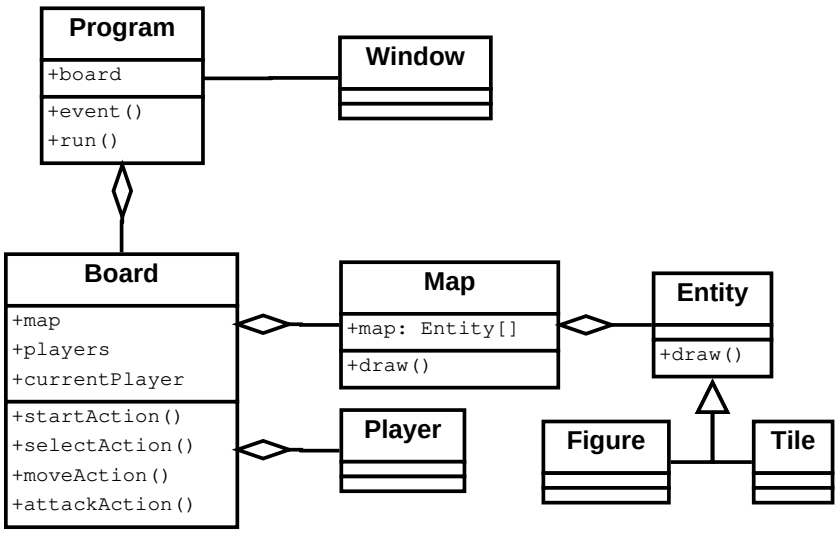

Figure 17: Game implementation main prototypes

that change the local state of the game, and these operations should be ordered and replicated among all replicas, ensuring the same game state for all players.

To replicate the game we are only interested in the model of the application. View and controller are local to a replica and their state does not need to be replicated. Thus, the prototype `Board` is the main locus of replication, defining the Treplica context. This prototype was made a sub-prototype of `Context` to signal this fact. Once the context is defined we must identify the mutator methods that change the state defined by this context. In this application these are the methods of prototype `Board` that implement player moves (`startAction:`, `selectAction:`, `moveAction:` and `attackAction:`). These methods must be wrapped in a prototype that extends prototype `Action` defined by Treplica. In our implementation that uses metaobjects this is accomplished by the use of the annotation `@treplicaAction` in each of these methods, as shown in Figure 18.

```
1  object Board extends Context
2    ...
3    @treplicaAction
4    func selectAction: String target { ... }
5
6    @treplicaAction
7    func attackAction: String target { ... }
8
9    @treplicaAction
10   func moveAction: String direction, String value { ... }
11   ...
12   @treplicaAction
13   func startAction { ... }
14   ...
15 end
```

Figure 18: Prototype `Board`

Once the context and actions are properly defined, it is necessary to instantiate and start the `Treplica` object responsible for replication, binding it to the context. Prototype `Program` is responsible for initialization of the application including instantiation of the `Board` object acting as context. At the point of creation of `Board` the use of the annotation `@treplicaInit` does all the required initialization and binding, as shown in Figure 19.

```
object Program extends Input
  ...
  func run: Array<String> args {
    Window build: self;

    var local = ("/var/tmp/magic" ++ args[0]);
    @treplicaInit( 2, 200, local )
    var data = Board new: args[0];
    data startBoard;
    ...
  }
end
```

Figure 19: Prototype `Program`

Once the binding of Treplica and context is done, all calls to the mutator methods of prototype `Board` tagged with `@treplicaAction` will be replicated. This fits the original behavior of the application, in which callbacks from prototype `Window` are received by `Program` and turned into calls to methods of `Board`. Thus, the main loop of the application remains the same, moves are made by the local player through events generated by the text window. Moreover, board state can also change though replicated method calls originating from the other player's replica. These moves are represented in the local replica window as they happen, through the calls to the `draw:` methods of `Map` and `Entity`.

This example shows that the proposed set of metaobjects is enough to allow the replication of a fairly complex application. Moreover, it is a demonstration of the simplicity of use of the proposed metaobjects. For a MVC application, which already has a clear defined model with a set of mutator methods, it is only necessary to add the proper annotations. Other type of applications may require some refactoring effort to define a clear bounded model. It is part of our ongoing research the evaluation of the ease of use of the proposed metaobjects for general software projects. This future work includes avoidance of some pitfalls, such as nondeterminism.

6. 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. 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.

Xtend [8], Groovy [28], Nemerle [29], Scala [30], Rust [31], Python [32], Java [33], AspectJ [34], Converge [35], Elixir [36] and Cyan [15] are examples of languages with compile-time metaprogramming features. Some of these languages (Xtend, Groovy, Nemerle, Scala, Java) allow the traversing the abstract syntax tree (AST) both to gather information and to change it. Others use a refined form of Lisp-like [13] quotes and unquotes, as Xtend, Nemerle, Groovy, Converge, and Elixir. Languages that only support quotes and unquotes are not directly related to the Cyan MOP.

Compilers of Scala, Java, and Rust support *compiler plugins* that play the same role as metaobjects in Cyan. The documentation about them is scarce except for Scala. Compiler plugins usually can register themselves with the compiler to be called at specific compilation phases. In Cyan, this registration is made by implementing interfaces associated with the phases. For example, a method `semAn_codeToAdd` is called in phase semAn because the metaobject class implemented interface `IAction_semAn`. In a compiler plugin, there may not be a direct association between annotations and the plugin. The plugin may take actions when it finds an annotation but, if the compiler plugin is not used, there is no action associated with it (and no compiler error).

AST transformations are similar to the Cyan metaobjects with one difference: they cannot be called when an operation, as subprototyping, method override, or field access is made. In Cyan, some metaobject methods intercept these operations.

The main difference between Cyan and languages with compiler plugins (Scala, Java, Rust) and AST transformations (Xtend, Groovy, Nemerle, Rust, Java) is the way new code is added to a prototype or class. In Cyan, the new code is returned, as a string, by metaobject methods. For example, method `afterResTypes_codeToAdd` of a metaobject can return a string with fields and methods to be added to the current prototype. The compiler is responsible for adding the new code and it tags the code with the annotation that created it. If there is an error, the compiler will point out exactly who made it. As an example, suppose two metaobjects ask for the insertion of field "`address`" to a prototype. Since there will be two fields with the same name, the compiler will issue an error. It will point exactly the two annotations that produced the code. In all other languages cited above, the code to be inserted should be an AST object.

The language may have macros or language mechanisms that transform strings into AST objects, a step that is not necessary in Cyan. In all other languages, code is added by AST handling. ASTs are a low-level structure of the compiler. They are subject to change, which would invalidate the compiler plugins or AST transformations, and they are not easy to understand. A typical compiler may have more than one hundred AST classes, each one with dozens of methods that can only be understood after a substantial understanding of the compiler. Cyan also allows the visiting of AST nodes, but that is made with the MOP AST, which is a read-only wrapped version of the compiler AST. This AST reflects the language structure, it does not have any specific details of the compiler. And there are security checks when accessing AST nodes. To cite one of them, a metaobject of an annotation in a prototype `P` cannot access the AST of statements of a method in another prototype `T` . This would make prototype `P` dependent on `T`, a dependency that is not registered by the compiler. If `T` changes, prototype `P` should be re-compiled. This breaks modularity because internal changes in a prototype are causing the recompilation of another prototype.

It is easy to make a mistake and invalidate the AST when handling it directly. In this case, the compiler may crash, generate wrong output code, or it may detect the invalid AST in a later compilation phase producing a confusing error message. If the code inserted causes a compilation error, the compiler will point out the error. But it will not give any information on which AST transformation or compiler plugin inserted the wrong code. In general, it is easy to code metaobjects in the Cyan MOP because they do not interact with the original compiler AST and they produce code as strings.

The metaobject `treplicaAction` could be implemented in languages that support compiler plugins and AST transformation. However, there is a feature of this metaobject difficult to implement: the checks made by metaobject `treplicaAction`, in phase afterSemAn, in the Cyan method annotated with `treplicaAction`. The metaobject looks for nondeterministic method calls that may have been added by other metaobjects. In this phase, the method code cannot be changed by any other metaobject.

In all other languages, in general there is no guarantee that compiler plugins or AST transformations will not change the code after a compilation phase. Then the checks can be made but, after it, another compiler plugin or AST transformation can add, in the annotated Cyan method, a call to a nondeterministic method. It may be possible to implement nondeterminism checks using some clever compiler trick, but we are not aware of that. A complete comparison between Cyan and other languages that support metaprogramming is made by Guimarães [27].

Regarding the problem of separating the nonfunctional requirements from the functional ones, there are works on AOP (aspect-orientated programming) that address the same problem. AspectJ [37] [38] is a Java extension that supports *aspects*. An *aspect* is composed of *pointcuts* and *advice declarations*. *Pointcuts*

pick out points of code called *join points* which may be method calls, object creation, field access, etc. An *advice* is composed of a pointcut and code. It uses the pointcut to select some *join points* that are wrapped with the advice code. For example, a *pointcut* can pick out all execution join points related to methods of a Java class called `Animal` that start with `get`. Then, the *advice* can wrap each method call with code to be executed before, after, or around the call. In AspectJ, this wrapping, called weaving, can happen at compile-time, post-compile time (using the binary files), or at loading time. With *Intertype declarations*, aspects can add fields and methods to classes and interfaces.

AOP and the Cyan MOP have largely different goals and this fact reflects in their features. In Cyan, metaobjects can only change the prototype they are associated with or, in a limited way, calls to methods of the prototype they are associated with. Aspects can change multiple files spread in the program, breaking modularity. It is not enough to read a source file in order to understand it because aspects, declared in other files, can change it. This is one of eight problems with metaprogramming that the Cyan MOP addresses total or partially [27]. AspectJML [39] is an AOP language that solves this problem by associating aspects with types. Each aspect only changes the subtypes thus limiting the affected code.

In Cyan, an annotation attached to the program or to a package (remember annotation `loadNonDeterministicFiles`) can add code to all prototypes of the program or to all prototypes of a package. The annotation is repeatedly applied to all prototypes of the program or the package. Using this mechanism, many features of AspectJ can be simulated in Cyan. A DSL attached to the annotation can even define *pointcuts* and code that should run on them.

Cyan metaobjects can visit the AST nodes of a prototype, method, statement, etc. This is not possible in AspectJ. Then, aspects cannot detect nondeterminism as metaobject `treplicaAction`. This detection depends on AST traversal. An aspect can add a superclass or add an implemented interface to a class. In Cyan, metaobjects cannot do that. This was on purpose, to prevent the MOP of making deep changes in the code.

Advice code can use variable `thisJoinPoint` to access, at runtime, information on join points. It can also be used to issue errors and warnings at compile-time. However, the information made available by `thisJoinPoint` is much more limited than in Cyan or any other language supporting compiler plugins or AST transformations. In Cyan, all the information the compiler has that is related to the language itself (and not related to internal compiler details) is available to metaobjects. This is necessary to the metaobject associated with annotation `treplicaAction`. Figure 12 shows the prototype `Info` modified by annotation `treplicaAction` attached to method `setText:` (see Figure 10). The new methods `setText:` and `setTextTreplicaAction:` of Figure 12 use information that is available in Cyan, at compile-time, but is not available in AspectJ: the parameter name `text` (line 3), the prototype and method name used to compose the prototype name `InfosetText` (line 4), and the method name `setText:` (used for composing the method name `setTextTreplicaAction` of line 8). In AspectJ, this information is available only when the final program, changed by the aspects, is running. Hence, in AspectJ, the generated code needs to use reflection and `thisJoinPoint` in order to access information available at compile-time in Cyan.

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 [16] aims to ease the conversion of legacy centralized C++ and Fortran code into portable shared-memory parallel code. OpenACC [17] 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 nondeterministic method calls and intend in the future to extend detection

to other types of consistency violations.

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. 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 produce source code as strings. However, the programmer can use the AST if necessary, for example, to visit the AST nodes of a method.

Moreover, we demonstrated the potential of using the metaprogramming infrastructure of a modern language to make the use of frameworks easier not only by removing boilerplate code, but also by making semantic validations that require integration with the compiler. In this paper we validate the generated code with respect to the presence of nondeterminism, by replacing the nondeterministic operations with equivalent deterministic operations. We believe the technique presented to detect nondeterminism 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

[1] J. Gray, P. Helland, P. O'Neil, D. Shasha, The dangers of replication and a solution, in: SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data, ACM Press, New York, NY, USA, 1996, pp. 173–182. doi:10.1145/233269.233330.

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

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

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

[13] P. Seibel, Practical Common Lisp, 1st Edition, Apress, Berkely, CA, USA, 2012.

[14] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, J. Noble, Javacop: Declarative pluggable types for java, ACM Trans. Program. Lang. Syst. 32 (2) (2010) 4:1–4:37. doi:10.1145/1667048.1667049.
[15] J. d. O. Guimaraes, The Cyan language, Tech. rep., Campus de Sorocaba da UFSCar (2019).
URL http://www.cyan-lang.org
[16] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE computational science and engineering 5 (1) (1998) 46–55.
[17] 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.
[18] G. Blaschek, Object-oriented programming with prototypes, Springer, 1994.
[19] 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.
[20] B. Stroustrup, The C++ Programming Language, 4th Edition, Addison-Wesley Professional, 2013.
[21] C# language specification (Sep. 2014).
URL http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf
[22] Y. Lilis, A. Savidis, A survey of metaprogramming languages, ACM Comput. Surv. 52 (6) (Oct. 2019). doi:10.1145/3354584.
URL https://doi.org/10.1145/3354584
[23] P. A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency control and recovery in database systems, Addison-Wesley New York, 1987.
[24] 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.
[25] L. Lamport, Fast Paxos, Distributed Computing 19 (2) (2006) 79–103.
[26] J. Eder, G. Kappel, M. Schrefl, Coupling and cohesion in object-oriented systems, Tech. rep., University of Klagenfurt (1994).
[27] J. d. O. Guimarães, The Cyan language metaobject protocol (2020).
URL http:www.cyan-lang.org
[28] The Groovy language (2017).
URL http://groovy-lang.org
[29] K. Skalski, Syntax-extending and type-reflecting macros in an object-oriented language, Master's thesis, University of Wroclaw, Poland (2005).
[30] L. Spoon, S. Tisue, Scala compiler plugins (jan 2020).
URL https://docs.scala-lang.org/overviews/plugins/index.html
[31] S. Klabnik, C. Nichols, The Rust Programming Language, 2nd Edition, No Starch Press, 2018.
URL https://doc.rust-lang.org/book/2018-edition/index.html
[32] L. Ramalho, Fluent Python: Clear, Concise, and Effective Programming, O'Reilly Media, 2015.
URL https://books.google.co.in/books?id=bIZHCgAAQBAJ
[33] Oracle, Interface plugin (Oct. 2019).
URL https://docs.oracle.com/en/java/javase/11/docs/api/jdk.compiler/com/sun/source/util/Plugin.html
[34] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of aspectj, in: J. L. Knudsen (Ed.), ECOOP, Vol. 2072 of Lecture Notes in Computer Science, Springer, 2001, pp. 327–353.
[35] L. Tratt, Compile-time meta-programming in a dynamically typed oo language, in: Proceedings of the 2005 Symposium on Dynamic Languages, DLS '05, ACM, New York, NY, USA, 2005, pp. 49–63. doi:10.1145/1146841.1146846.
URL http://doi.acm.org/10.1145/1146841.1146846
[36] Elixir (2018).
URL https://elixir-lang.org/
[37] 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.
[38] AspectJ, The aspectj language (2018).
URL https://www.eclipse.org/aspectj/doc/next/progguide/language.html
[39] H. Rebêlo, G. T. Leavens, Aspect-oriented programming reloaded, in: Proceedings of the 21st Brazilian Symposium on Programming Languages, SBLP 2017, Association for Computing Machinery, New York, NY, USA, 2017. doi:10.1145/3125374.3125383.
URL https://doi.org/10.1145/3125374.3125383