Design and Prototype Implementation of an Evolutionary Environment for Concurrent Object-Oriented Programming

Ján ŽIAK

Slovak University of Technology
Faculty of Informatics and Information Technologies
Ilkovičova 3, 842 16 Bratislava, Slovakia
ziakjan@fiit.stuba.sk

Abstract. Simplicity and extensibility are important aspects of any programming environment because they aid clarity and long-term evolution. While in most programming languages the development of programmatic constructs stops after the language is designed, the central idea to this article is to start with a simple model of concurrent programming with no constructs and with the possibility to add them later as they are needed or invented. New constructs are created by implementing them as objects. The programmer then takes advantage of an object-oriented user interface which enables to write programs by direct insertion of constructs into the code.

1 Introduction

Programming in any language involves the usage of constructs. Each of these constructs, like for example iteration, message send, conditional execution or expression evaluation, have their own syntax and semantics. The knowledge of the semantics, which has usually nothing in common with the syntax, enables the programmer to express his or her thoughts. The knowledge of syntax, on the other hand, is an essential requirement for the ability to write down the thoughts already expressed in the semantics of the particular programming language. The syntax of a construct departs from its semantics in the sense that the syntax of the construct can be chosen almost at will, the only mandatory constraint being the existence of some kind of separate places or slots for the arguments of the construct. Thus, syntax of a

* Supervisor: doc. Ing. Martin Šperka, PhD., Institute of Applied Informatics, Faculty of Informatics and Information Technologies STU in Bratislava

construct plays a role equivalent to a symbol here, which has the consequence that there exists a magnitude of programming languages with almost identical constructs from a semantic point of view but which are being typed in numerous different ways [1]. Although this article is not meant to reconcile this problem, it seems rational that if programming languages were designed in a way that allows to change the look of their constructs and to seamlessly add new constructs into the language as they are needed or invented, there would exist at least a hope for the different syntaxes with the same semantics to converge into (ideally) one nice syntactic element so that competition between different languages becomes to be based on the ability of the language to express thoughts and to be less based on the ability to learn the subtle differences in syntaxes with the same meaning. This however does not mean that there would be no competition on the syntactic level.

An extensible programming language can have an advantage over a language defined by a fixed set of grammatical rules because it can respond more rapidly to changes. For example, a new syntax for an already existing construct could be tested to see how it performs and feels in respect to the already existing syntax and adopted if it performs well. New constructs (that is: new semantic elements) could be added to the language to extend its capabilities so that a change does not require to design a completely new programming language but only to extend an already existing one.

As evolution proceeds by creating new entities which replace some of the already existing ones not fit enough to survive, it is necessary that a programming language capable of evolution has the ability to get rid of constructs which are considered inferior. However, the present state of affairs in the style of development of new programming languages and of modification of existing ones makes this idea almost insane because almost all programming languages take for granted to retain all of their existing constructs if they are to be modified.

The next section introduces a model of concurrent object-based computation which is then in section 3 used as a basis for building new constructs. Section 4 gives a short overview of the graphical user interface allowing the user to program by direct insertion of objects, some of which are programmatic constructs, into program's code. The next section discusses the implementation of new constructs in Java and shows example constructs. This is followed by a comparison of programming in our environment with other approaches to extensible programming.

2 A model of concurrent object-oriented programming

The approach used in this article to design an evolutionary programming language is to start with a sufficiently general and minimalistic abstract model of computation. The basic presumption is that it is a concurrent object-oriented computation where objects communicate by messages. One object can process at most one message at a time. The response to received messages is determined by rules. The list of rules of an object specifies its behavior. Each object has its own behavior and its own state (fig. 1), although typically a single behavior would be shared between different objects of the same kind. The basic inspiration for the model came from Self and the Actors model of computation [3, 4].
The notion of a rule is similar to the notion of a rule of a Turing machine [2] or of a production system [8] - the only difference is that at most one rule gets executed per received message. Each rule has a condition-part and an action-part which executes only if the condition-part is satisfied. The rules are examined in the order in which they are listed and the examination stops when a rule with a satisfied condition-part is found.

The condition-part is a list of conditions which are checked (in the order in which they are listed) whether they are satisfied in the current context of object's state and received message. The condition-part can be empty in which case it is treated as being always satisfied. The action-part is a list of actions which are sequentially executed in the listed order. The customization points of the programming language are the conditions and the actions.

Object's state consists of a boolean value indicating whether the object currently processes a message. Because it is expected that many programming constructs will use variables, the state also contains structures for global and local variables. The local variables are ordered into a stack of frames each holding variable values for a rule. Object's global variables have the ability to survive over time while local variables cease to exist after message processing finishes.

As there already exist numerous programming languages, the model is not implemented from scratch but is designed and implemented in an already existing language and environment, namely Java. The design of classes in Java is based on the specification given in previous paragraphs. For example, a message sent asynchronously from one object to another is an instance of a Java class. A message
can be any object of the implementation language, although it is typically a list of Java-objects. Object’s response to a message is based on the analysis of the content of this message.

By default, messages sent to the object performing the send (i.e. to itself) are processed synchronously. This is similar to a method call in ordinary object-oriented programming languages. It should be noted that it impossible to known in advance whether an object is sending a message to itself or not. An object could receive a reference to itself in some message, store it in a variable and try to send a message to this received object. It is ensured that this always results into a synchronous message processing. This uses the stack of local variable frames to store information about values of variables when a rule is calling another rule.

3 Extending the model with new constructs

In the beginning, the programmer is allowed to specify the condition-part of rules as a list of message sends evaluating to a boolean value. Likewise, action-parts of rules are lists of message sends. Thus, the only available programming construct is a message send. While this can be considered to be general enough to specify any kind of computation, it is more like programming in an assembly language rather than in a higher-level one. Such a language lacks the constructs to express assignment of values to variables, iteration or conditional execution, and lacks the constructs for specifying patterns for categorization of incoming messages (i.e. selectors).

The graphical user interface shortly described in the next section allows one to directly add references to objects into the graphical representation of rules. These can refer to any kind of object of the implementation language. The basic idea is to implement new constructs in the implementation language, instantiate them and directly insert these instances into the right place in a rule.

The syntax of the constructs is specified by their graphical appearance and can be arbitrary within the rendering capabilities of the graphical interface. The constructs have no textual representation which makes pointless to speak of them as having a grammar. As mentioned in introduction, each construct has to contain some kind of separate places or slots where to put its arguments. For example, an "if-then" construct contains two slots: one for the condition and one for the statements to execute in case the condition is satisfied. The positions of these slots are part of construct's appearance within the graphical interface.

The semantics of a construct is implemented in the implementation language (i.e. Java) and is obtained from the object inserted into rule. The phase in which the semantics of constructs are being obtained is called compilation. During this phase, the construct typically checks for errors in its definition. The compilation omits syntactic analysis because the syntax of the constructs, that is their appearance, has a meaning only in respect to the programmer and is completely ignored by the compilation which works with Java-objects implementing a particular known protocol. It is in fact impossible for a syntax error to occur during compilation because there is no syntax to work with. (This does not apply to message sends which do require some parsing.)
The semantics of a construct is an instance of a class of the implementation language and conforms to a specific interface. The interface contains a single method `execute` which is responsible for performing the semantics of the construct when the object which contains a rule with this construct receives a message. The execution method is passed a reference to the Java-object representing the object on which behalf the execution is being performed. The method can use this reference to access object's state and behavior, and change them according to its semantics. For example, the semantics of the construct "assignment to a variable" is to replace the value of the given variable in the global or local variable frame by its new value.

As the model of computation is a concurrent one, one cannot know when the semantics of the construct finishes execution. For example, when the semantics entails waiting for a value which is the result of some computation on some other computer in the internet, the other objects cannot be blocked until this result is obtained. This issue is resolved by passing a continuation to the implementor of construct's semantics. When the execution of the semantics finishes, the continuation is used to report when the next construct enlisted in the condition-part or action-part of the rule can start to be executed. This mechanism allows to process messages sent to other objects while an object is waiting for an event to take place, enabling the concurrent model of computation to work on uniprocessor architectures.

4 Graphical user interface for manipulation with objects

Unfortunately, the graphical user interface used to perform programming cannot be discussed here in its full detail because it is too complex. Only principles relevant to programming will be considered here therefore.

Objects in the environment are represented by their visual appearance on the screen of a computer. The interface provides means to display the appearance of an object. Objects typically embed other objects. More precisely, the interface works with references to objects rather than directly with objects themselves and allows to grab a reference to a graphical object and to embed it into another object. It also allows to work with non-graphical objects by representing them as graphical objects.

The rules, condition-parts, action-parts and syntax of constructs are graphical objects. Rule is a composition of its condition-part and its action-part, each of which is a list of constructs and text. The syntax of a construct can be made to look like anything. For example, it would not be impossible to implement a construct which would look like a blue jumping ball. However, the syntax of a construct is typically a composition of some non-modifiable text and places where to put appropriate sub-constructs. The syntax of a construct is not influenced by other constructs in any way, which has the unpleasant consequence that there exists a possibility for two construct's with different semantics to look exactly identical.

The environment departs from a typical contemporary integrated development environment in substantial ways. It was inspired mainly by Squeak's [9] Morphic, but provides better support for direct manipulation with objects and should discourage the user from using text to express his or her ideas.
It should be noted that the idea of extensible programming presented in this article would have been impossible without the existence of this graphical environment for manipulation with objects.

5 Implementation and examples

The whole system together with the user interface is implemented in the Java programming language taking advantage of the existing infrastructure of its runtime environment and some additional libraries. However, the model of concurrent computation has been implemented from scratch because the rate of switching between threads was too coarse-grained, while the system requires a fine-grained level of parallelism.

The design of classes is based on the description of the computational model in section 2 of this article. Constructs which are to extend the language with new features implement a special interface containing a method transforming the construct into its compiled form. The compiled form, which can be an expression or a statement, is an object and conforms to an interface with an `execute` method which is invoked to perform the semantics of the construct in the context of a particular object. The difference between a statement and an expression is that the former does not by default return a value, while the latter does.

Results which are the products of some computation are communicated by means of sending messages. The typical pattern of returning a value which is the result of an expression evaluation is resolved by having a slot-object which accepts the result and then notifies any objects waiting for the result.

The implementation allows replace the behavior of any object by another behavior at any time. Likewise, the rules of a behavior can be dynamically added or removed. Thus, the environment is designed to be as much dynamic as possible. The consequence of this is a slow execution speed.

![Fig. 2. Two example rules](image)

Figure 2 shows two sample rules. The condition-part is in the top and action-part in the bottom. Variables are in bold, syntax of constructs is in italics, words in regular style designate message parts and the boxed text represents a symbol for a Java-class.
The first rule in the figure uses these constructs: message send, evaluation whether a variable is a global variable, declaration of a global variable, assignment, expressions which evaluates to the received message and reference to the receiver, and finally a send operation. The semantics of the rule says that its action-part is executed when variable `state` is not a global variable. The actions declare two variables as global (i.e., "static"), perform two assignments and then send the just-received message to the object itself for further processing.

The second rule is simple in its actions which just perform two message sends. Its condition part contains a construct called a selector which is able to determine whether the received message looks like the specified list of objects, in this case as a list of strings finished by an object which is assigned to the variable `aCollection`. The object assigned to `aCollection` is then checked for its conformance with a Java-interface. The condition-part of this rule simulates the checks which are performed when one invokes a method in Java. The main difference between this rule and Java is that Java uses static binding of method names rather than dynamic ones.

Some of the constructs which can be used to build programs are summarized in table 1. The last construct enables to process a specified message by a specified behavior on the behalf of the current object. This enables the programmer to represent patterns similar to inheritance.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>Statement</td>
</tr>
<tr>
<td>Selector</td>
<td>Expression</td>
</tr>
<tr>
<td>Reference to receiver (i.e. self)</td>
<td>Expression</td>
</tr>
<tr>
<td>Currently processed message</td>
<td>Expression</td>
</tr>
<tr>
<td>Instance-of check</td>
<td>Expression</td>
</tr>
<tr>
<td>Conditional execution</td>
<td>Statement</td>
</tr>
<tr>
<td>While-iteration</td>
<td>Statement</td>
</tr>
<tr>
<td>For-iteration</td>
<td>Statement</td>
</tr>
<tr>
<td>Global variable declaration</td>
<td>Statement</td>
</tr>
<tr>
<td>Global variable check</td>
<td>Expression</td>
</tr>
<tr>
<td>A variant of message sending</td>
<td>Statement</td>
</tr>
<tr>
<td>Interpretation of a message by a specified behavior</td>
<td>Statement</td>
</tr>
</tbody>
</table>

### 6 Comparison with other approaches

The model of computation takes its inspiration from various sources. The idea to treat object’s behavior as a set of rules with a condition-part and an action-part is similar to a rule of a Turing machine [2] or of a production system [8]. Some parts of the concurrent computation model and its implementation, like continuation and futures, came from the Actors computational model [4]. However, the model of computation was introduced only to provide a base for building an extensible programming language.
Contemporary approaches to extensible programming languages work with textual source files \[6, 7, 8\]. The constructs must be specified as text and must be clearly distinguishable from other constructs in a syntactic way. The approach of this article was to make a clear statement about the relation between construct’s syntax and semantics and to use graphical objects to represent the syntax in an arbitrary way. Syntax (i.e. visualization) of one construct does not influence other constructs in the same way as it being done in the case of textual representations.

The \textit{apt} tool \[6\] parses the source code and represents it by object-oriented structures which reflect the structure and elements of the source code. This representation is then processed by a procedural annotation processor modifying the Java source code and potentially generating new source files. Although the transformation of the source code is procedural (i.e. it is not a macro) the generated code has to be in the Java language. On the other hand, the approach used in this article makes no transformation of constructs into language X, but creates custom objects representing the semantics of the construct. This object-oriented representation of the constructs’ semantics is final because the environment in which the objects live is persistent.

7 Conclusions

This article described a method of extending a concurrent programming language with new constructs. The initial programming language is minimal. New constructs with new capabilities can be added to the language in order to extend it. The syntax of constructs is specified mainly by their graphical appearance and need not to be represented by text only.

The suggested method should enable the system to survive over a long period of time because new constructs with new meanings can be added to the language. The system does not initially contain constructs (except the constructs for simple message send and subexpression evaluation), so the set of constructs is in fact initially empty. Old-fashioned constructs which are no longer being used by programmers should be those that loose the struggle for survival among various different constructs with different semantics.

It should be noted that this article presented the implementation in an idealistic way and omits a precise description of the implementation.

There remains a lot of work which should be done in the future. First of all, the environment has not been tested in practice in a community of programmers. However, the odds that it would keep its promise of being evolutionary seem to be high. The environment should be populated with numerous new constructs – which would surely entail the necessity for some changes in the current prototypical implementation.

\textit{Acknowledgement:} This work was supported by Scientific Grant Agency of the Ministry of Education of Slovak Republic and the Slovak Academy of Sciences under grant No. VG 1/0161/03 (Information Processing in Distributed Environment of Intelligent Agents).
References