# Integrating data-parallel and reactive constructs into Java

Yves ROUDIER Yuuji ICHISUGI
roudier@etl.go.jp ichisugi@etl.go.jp
(STA Fellow)

Electrotechnical Laboratory

**Abstract:** We introduce the EPP extensible preprocessor and explain how it permits to integrate new constructs into the Java language in a modular way. We present experiments of data-parallel frameworks and reactive objects programming realized with this system.

## 1 Introduction

The Java language [GJG96] has recently become very popular among programmers and researchers: it is rather simple and makes a good synthesis of many ideas previously proposed in isolation in the programming language field. Java has socket and thread libraries which can be used on various platforms. Some Java virtual machines already support shared memory type multi-processors; in the near future, many more will, and the language will thus become a more interesting platform for parallel programming.

On the other hand, Java lacks facilities for language extension, which are adopted by other object-oriented languages. For example, C++ provides a macro preprocessor, operator overloading and template facilities. Smalltalk and CLOS have closures and metaclass facilities. These features extend language constructs and operators and supplement the inheritance mechanism

which extends class, that is module definitions (as well as data types if class and type notions are unified) in a very restricted way. Without such extension possibilities, it is difficult to make parallel libraries which can be employed easily or seamlessly by end-users. In order to enhance Java on that matter, we developed an extensible Java preprocessor kit, EPP [Ich]. EPP can be used to introduce new language features, possibly associated with new syntax.

In this paper, we describe the EPP system and two example extensions programmed with it : Tiny Data-Parallel Java and reactive objects.

Tiny Data-Parallel Java (or TDP Java) is based on the same language model as Data-Parallel C [HQ91]. The source code of a TDP Java application is translated to standard Java code by a translator implemented using EPP.

Reactive objects are an extension of traditionnal objects: reactive primitives are added to their interface that enable them to react instantaneously to their environment, even when servicing a method. Such behavior can be necessary for programming clean synchronization between objects working in parallel (or pseudo-parallel). The extension, inspired from the model described in [CR96], gives the ability to add reactive primitives to a Java class.

Because the runtime system of both plugins is written in pure Java, high portability is guaranteed: they can be used off-the-shelf on every machine providing a Java virtual machine.

# 2 The Ld-2 language

We first introduce the Ld-2 language whose *system-mixin* feature is heavily used in the EPP system.

## 2.1 Rationale of Ld-2

*Ld-2* [Ich97] is an object-oriented description language implemented in Common Lisp [Ste90]. The usual inheritance mechanism of object-oriented languages makes it easy to implement extensible applications because all methods of objects can be considered as hooks for extensions. However, be it simple or multiple inheritance, this mechanism does not always allow for a very good extensibility when non orthogonal semantics must be combined.

The original purpose of Ld-2 was to provide an additional unit of organization for code in the form of a package. In addition to the traditional inheritance mechanism, Ld-2 provides a novel feature called *system mixin* which supports the development of more extensible and flexible software. This mechanism only targets a simple form of extension and thus does not require the sophistication of reflective systems which also solve this kind of problems (but require dealing with meta-levels). It is also very simple to learn for the programmer.

## 2.2 System Mixin

The program presented in Fig.1 defines a method named `:m` which contains nested `if` statements. This method is written in a traditional, "monolithic" way. When a programmer wants to add a new `else-if` clause to this method, the only way to do so is to edit and thoroughly modify the original source code. By using the inheritance mechanism, some editing can be avoided, but this changes the identity of the method. It is indeed impossible to extend the behavior of the method without defining a new subclass.

```
(class <foo> ()
  (defmethod :m (data)
    (if (eq data 'B)
        (do-B)
        (if (eq data 'A)
            (do-A)
            (do-default-behavior)))))
```

Figure 1: Monolithic definition in traditional object-oriented languages.

Ld-2 solves this problem by introducing the mechanism of system mixins. Bracha[BC90] showed the flexibility and reusability of *mixin-based* inheritance. System mixins goals are similar to mixins but the unit of inheritance is not a class but the whole program.

The second program presented in Fig.2 uses the system mixin feature and has the same behavior as the first one; however, its structure is much more modular.

```
(defsystem <<skeleton>> ()
  (class <foo> ()
    (defmethod :m (data)
      (do-default-behavior))))
(defsystem <<A>> ()
  (class <foo> (data)
    (method :m (data)
      (if (eq data 'A) (do-A) [:original data]))))
(defsystem <<B>> ()
  (class <foo> (data)
    (method :m (data)
      (if (eq data 'B) (do-B) [:original data]))))
(setup (list <<B>> <<A>> <<skeleton>>))
```

Figure 2: Modular definition in Ld-2.

The `defsystem` statement defines a system mixin, that is, a fragment of the program. The `setup` statement at the last line specifies the system mixins that should be composed into the complete program. The expression `[:original data]` is similar to the method invocation to the super class in traditional object-oriented languages.

When the method `:m` is called, the fragment of method `:m` defined in `<<B>>` will be invoked. If this fragment evaluates `[:original data]`, the next fragment defined in `<<A>>` will then be called.

## 2.3 Differential programming and beyond

The interest of the mechanisms that we presented in the previous section can be mainly related to extension. They permit what we call "differential programming": one programs a group of classes by creating differential versions with system mixins; the final result is the composition of these different parts. Our experience shows that these mechanisms can be used very successfully

for enhancing traditional program construction. They also make sense for evolving software under development, since new features can be introduced by difference from the previous version. Unrelated plugins can then be used and tested independently.

It is worthwhile to note that some differential programs can be aspect-oriented programs [AOPp] as well (although differential programming can also be used in other contexts). Existing AOP systems rely on a single component language that is enhanced by describing additional and separate aspects of the system with other non-component languages.

Contrary to these systems, the current version of Ld-2 offers only one programming language (the basic component language). But it allows for not only a single reference component layout, but a series of differential component programs, which are themselves potentially references for writing new differential parts, indeed composing new aspects. It is thus possible to program several aspects of the program separately, inside of system mixins; system mixins are not mixins and are thus not limited to one class definition: aspects description is not limited, because this framework allows system-wide descriptions. The main advantage of this technique is that each of these programs is connected by the class and method names, which provides a kind of implicit "join point", as the notion is described in AOP litterature.

# 3 An extensible preprocessor kit: EPP

Many tools have been proposed for preprocessor language extension; this remains however a difficult task. We first give some ideas of these problems and how EPP can help solve them.

## 3.1 Existing tools

To implement new languages or extend existing ones, preprocessors or translators are often used rather than native compilers. For instance, many language extensions and source level optimization tools for C/C++ are implemented as translators to C/C++. Recently, several extensions for the Java language have also been proposed which are again implemented as often simple preprocessors (or even only stub class generators, like [PZ97], for instance). Because of the simplicity and object-orientation of Java, it is also

relatively easy to provide extension support libraries.

The merits of this style of implementation are its easiness and high portability. Instruction level optimization can be delegated to the compiler of the target language.

Although there are potentially many useful language extension systems, a user must select only one for their own project. Generally, it is impossible to merge several such language extensions or to eliminate harmful features from the extended system.

Systems with a meta-object protocol (MOP) such as CLOS [KdRB91], MPC++ [Ish94], OpenC++ [Chi95], EC++ [CMP+97] or JTRANS [KK97] have solved this problem. These systems allow the implementation of language extensions as modules that can be selected by the users. Yet, extensibility of low-level constructs and of syntax, and modularity of extensions can be slightly restricted in these systems.

## 3.2 Description of EPP

The Java Extensible PreProcessor kit, *EPP*, is an application framework for preprocessor type language extension systems. Basically, EPP is itself a source-to-source preprocessor of Java. EPP enables a preprocessor programmer to write an extension as a separate module. We call such extension modules *EPP plugins*.

The parser of EPP is written in recursive descent style using the system mixin feature (see the Ld-2 section). Each function which parses a non-terminal has some hooks for extension. The hooks are actually methods which can be extended by other system mixins.

By using these hooks, the extension programmer can introduce new features (for instance through new `else-if` clauses), possibly associated with new syntax and without any global modification of EPP. Because all grammar rules are defined in a modular way, it is also possible to remove some original grammar rules from standard Java.

Once the parsed program has been transformed into a tree, the preprocessor programmers can easily manipulate it from their programs. The usefulness of such kind of tool has already been proven by Lisps and adapted to

C++ by various systems like Sage++ [ea94], MPC++ [Ish94] or OpenC++ [Chi95].

Direct modification of constructs in a parsing pass is a language-oriented extension; in addition, the programmer can also extend EPP in a more compiler-oriented way. EPP basically performs three passes: the parsing pass, the macro expansion pass and the code emission pass. The plugin programmer can add new passes before and after each of these passes: with this style, composing pass plugins corresponds to defining a graph of connections between passes inputs and outputs.

Plugins can be written very freely by the user. However following some guidelines (for instance, see [Ich97, IHT+96]) helps ensuring that no collision occurs between two different plugins. If no problem (same identifier names, etc.) arises such plugins can then be incorporated into EPP simultaneously. The user of the preprocessor can select any plugin that fits the characteristics of his project.

High composability of EPP plugins can be realized thanks to the description language *Ld-2* (see section 2) which is based on Common Lisp. We currently plan to rewrite the system directly in Java. Although the current target of EPP is only Java, the architecture of EPP is also applicable to other programming languages.

### 3.3 Example and usage of EPP plugins

The user of the preprocessor specifies the *EPP plugin files* that he uses at the top of each Java source code. Fig. 3 is a simple example program using an EPP plugin that defines a swap macro. The plugin file is actually an Ld-2 program file. It will be dynamically loaded by EPP before starting the preprocessing. The user can specify multiple EPP plugins at the top of the source file if they are compatible.

It is also possible to implement extensions that accept already existing Java programs without EPP command, that is, without explicitly writing `#epp "plugin"`. This was the implementation that we chose for a program scrambler which translates all private method and variable names into meaningless character strings for preventing reverse-engineering.

Although not currently implemented, it is also easy to introduce metaclasses that could then be used for extension in the program itself. With such

```
#epp load "swap"
public class test {
  public static void main(String[] argv){
    int a = 1, b = 2;
    swap(int, a, b);
  }}
```

Figure 3: A program using a swap plugin.

mechanism, only finer granularity constructs (loops, instructions, etc.) would need EPP commands for extension. Moreover, the metaobject part of such plugins would be programmed in the original language and not necessarily with Ld-2.

# 4 Tiny Data-Parallel Java

As an example of an EPP plugin, we explain our implementation of the *Tiny Data-parallel Java plugin*.

## 4.1 General principle and description of the plugin

Data-parallelism is generally opposed to control- or task-parallelism. In the latter, parallelism is introduced by adding a new control flow while in the former, it is introduced with new data types featuring intrinsic parallel properties. Data-parallelism generally permits to write very homogeneous programs where parallelism remains relatively implicit. To date, many evolved data-parallel frameworks have been proposed, for instance providing nested parallel constructs. SPMD frameworks can also be regarded as an evolution of data-parallelism towards coarser-grained parallel machines, and are now also evolving towards object-orientation. Other frameworks should be expected to appear, especially in the very active area of Java.

We want to define data-parallel classes for implementing dataparallel types: some methods of objects instantiated from these classes would then treated as data-parallel methods. From the programmer's point of view, these methods are executed on a large number of virtual processors. Our framework is very simple since our point is about extension and not about performance of the model programmed; we simply follow the Data-Parallel C model [HQ91].

```
#epp load "datap01"
public class CalcPi {
  public static void main(String argv[]){
    // Number of virtual processors.
    int num = 10000000;
    // Number of Java threads used for parallel execution.
    DpCalcPi.PE_n = 16;

    DpCalcPi.width = 1.0 / num;
    DpCalcPi p = new DpCalcPi(num);
    p.run();            // Execute the data-parallel method.
    System.out.println(DpCalcPi.result * DpCalcPi.width);
  }
}
parallel class DpCalcPi {
  static double result; // Mono variable.
  static double width;  // Mono variable.
  double x;             // Poly variable.
  public void run(){    // Data-parallel method.
    x = (VPID() + 0.5) * width;
    // Reduction. Calculate Σ 4 / (1 + x^2) .
    set_sum(result, 4.0 / (1.0 + x * x));
  }
}
```

Figure 4: A program which calculates $\pi$.

Fig. 4 shows a program which calculates the value of $\pi$. In that program,

ten million virtual processors are started at the same time, each calculating a part of the sum giving us the approximate value of $\pi$.

Class `DpCalcPi` is defined with the special modifier `parallel`: its `run` method is the *the data-parallel method* which is executed in parallel by virtual processors. EPP translates this method into a standard Java code which uses thread libraries and synchronization primitives.

In our (data-)parallel classes, there are two types of variables: mono- and poly-variables. Mono-variables are shared by all virtual processors and poly-variables are owned by each virtual processor. It is possible to access a poly-variable on another virtual processor by using the expression `x@[i]`. Mono- and poly-variables can be used in arbitrary expressions in the data-parallel method. Static variables defined in the `parallel` class are considered as *mono*-variables and instance variables are considered as *poly*-variables.

The current implementation imposes some restrictions on how to write data-parallel methods. Specifically, no local variable can be declared within the data-parallel method and remote variable accesses are not allowed in the middle of other expressions or method invocations like, for instance, in the expression `if(x@[i])...`; it is only allowed on the left- or right-hand side of assignment expressions. Currently, only one reduction function is provided: `set_sum`(*mono_var*, *expr*), calculates the value of `expr` on each virtual processor, and assigns the sum of these results to the mono-variable `mono_var`. To end with, current implementation performs no optimization, which is not our point, anyway.

## 4.2 Implementing virtual processors

The EPP plugin of Tiny Data-Parallel Java adds an extra pass to EPP, before the code emission pass. At this pass, the source code of Tiny Data-Parallel Java is expanded into plain Java code that contains additional statements for implementing virtuals processors.

Their first and most basic implementation is simply sequential: `for` statements can simulate virtual processors, and their synchronization.

Fig. 5 shows some source code and its translation into plain Java, using loops. This code makes use of the virtual processor identity primitive (VPID). Virtual processors can be allocated to several Java threads, `PE_i` being the ID of each thread; `vpn` indicates the number of virtual processors

allocated to threads. `vpi` is a loop counter used to simulate allocated virtual processors. Each poly-variable is represented as an array whose size is `vpn`. Therefore, the poly-variable `x` is translated to `(vp_x)[vpi]`. The method invocation `sync()` executes a barrier synchronization of all threads; it could however be redefined for specific purposes. This synchronization call is inserted in several places such as before and after remote data access or reduction function calls. Although not operative, it can be redefined (see below).

```
x = VPID();
System.out.println("VPID = " + x);
```

```
for (vpi = (0); (vpi) < (vpn); vpi++) {
  (vp_x)[vpi] = (((PE_i) * (vpn)) + (vpi));
  System.out.println(("VPID = ") + ((vp_x)[vpi]));
}

sync();
```

Figure 5: Source code and translated code of a TDP Java program.

## 4.3 Implementating parallelism on real processors

We implemented two other more interesting versions of the preprocessor : a multi-threaded one and a distributed one which both run on any platform supporting Java.

### 4.3.1 Multithreaded version

Using this multi-threaded version and a multiprocessor workstation, applications can be executed with real parallelism if the VM interpreter supports parallel execution of thread libraries.

In this version, a thread is normally created for each virtual processor required. However, it is possible to make use of the processor emulation by

loops in combination with threads, thus limiting the runtime to a reasonable amount of threads.

Thread synchonization is realized at each step of the computation. Method `sync` implements a synchronization barrier stopping all threads arrived at this method before others.

The synchronization scheme of that version already adopts a distributed point of view; the only difference with a really distributed architecture is that notifications of synchronization phases are achieved by writing to shared variables instead of sending real messages. Specifically, if one thread exits a synchronization phase first and gets a long enough time slice to enter the next synchronization phase, it must not modify the value of a variable needed by other threads to exit the previous synchronization. To avoid such locking of threads, the update of synchronization shared variables has to be very carefully done, either through a three-phase scheme or by locally keeping track of the current synchronization phase of the thread and storing the state for two successive synchronizations instead of one.

### 4.3.2 Distributed version

We also implemented a distributed version of TDP Java : it uses the HORB system [Hir97] to implement communications between each node. Using this distributed version, applications can be executed on heterogeneous distributed environments.

Our choice of HORB is due to its better performance figures compared to Sun's RMI (Remote Method Invocation system). HORB transparently provides us all the facilities needed for data transmission. However, implementing our system on top of RMI would be very straightforward as well; it also remains possible to use a different communication strategy, for instance an interface to PVM or MPI.

Our architecture defines a *virtual processor* as the combination of a HORB server and a HORB client (see Fig. 6). Schematically, the server part performs message reception while the client part performs the real computation and can send messages to other virtual processors. These two objects run simultaneously on different threads: the client part is always running, and HORB skeletons (which are invisible to the HORB user), which also run on threads, receive messages and activate the server part. Server and client communicate through shared variables.

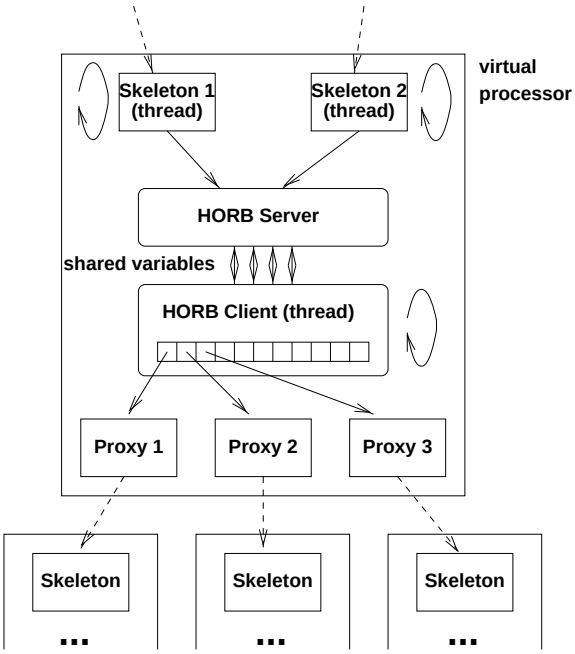


Figure 6: Distributed version

As previously, it remains possible to embed threads and sequential loops inside of these more heavy-weight processes. That design would give virtual processors a much finer granularity and would suppress waits during communications or I/O operations (interleaving of activities).

Before starting the program, a list of machines needs to be provided in a configuration file and in the current state of the implementation, we are only providing of cyclic allocation on these machines.

## 4.4 Performance

Parallel execution of multithreaded plus virtual processor loops version of Tiny Data-Parallel Java has been tested on the following environments:

1. JDK, Windows NT, Bi-Pentium Pro processors machine

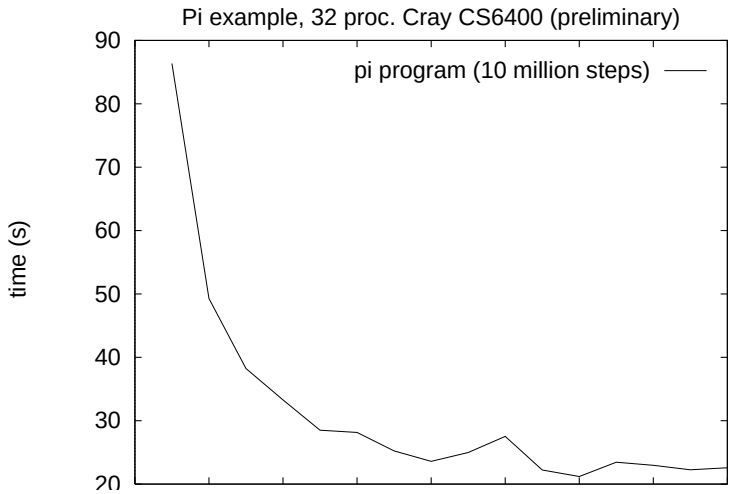


Figure 7: Performance of the $\pi$ program on the CS6400.

2. Toba (a Java-to-C translator which supports multi-threading), Solaris2.5, CRAY CS6400 (32 processors shared memory SPARC machine)

On the CS6400, Toba is used for translating the pure Java program. Fig. 7 shows the execution time of the program running on the second configuration. The program calculates $\pi$ with 10 million virtual processors and up to 16 threads (potentially scheduled on real processors, although no guarantee can be given about it).

## 4.5 Conclusions

The tiny data-parallel Java plugin currently does not have sufficient language features to support high performance parallel programs. This lack can be traced to the model of Data-Parallel C which is only adapted to SIMD machines. However, we think that it shows rather convincingly the effectiveness of the combination of the high extensibility of EPP and of the high portability of Java and their interest for parallel programming. Many data-parallel extensions are currently being proposed for Java (see for example [NPA] or

[LP96]), but none of these frameworks has been introduced by means of a general extension mechanism.

Furthermore, choosing this model was reasonable: it was simple enough for developing and above all debugging our runtime libraries which proved non-trivial. Working with threads is known to be difficult; the distributed version, even though it was developped on top of HORB [Hir97], which helped, was even more difficult to debug. In the future, we intend to apply all the techniques that we developed for programming other more efficient parallel models.

In spite of its current restrictions and inefficiencies, this tiny data-parallel Java can serve many interesting purposes:

- all Java libraries including the AWT can be used from the program.
- The implementation is as portable as Java. It can make use of the Java thread library, thus achieving much better speedups on multiprocessors (there are still only few multiprocessor platforms which can execute Java threads really concurrently, but things are getting better).
- Other EPP plugins remain applicable.
- It provides a simple framework for evolving quickly a sequential imperative program into a parallel one: even if performance is not getting better, programs that need much memory can very simply benefit from the potentially huge capacity of parallel architectures.

# 5 Reactive Objects for Java

We presented in [CR96] a reactive framework for Eiffel//, a parallel extension of the object-oriented language Eiffel. This section explains why a similar extension cannot be programmed in pure Java and with the original virtual machine without EPP. We present our implementation as an EPP plugin and perspectives.

## 5.1 Presentation of the reactive object model

[CR96] model introduced in a REACTIVE_OBJECT class, provided with several so-called *reactive* primitives:

- *abort* for abandoning the current service of a request by the object,
- *suspend* for temporarily suspending this service,
- *resume* for restarting a suspended service.

These primitives are called reactive because they can be called at any time during the life of an object, even during a service (i.e. the execution of a non-reactive method). This feature promotes the idea of separating the code performing the computational task from the code *reacting* to some input. Synchronization is introduced orthogonally with respect to the computationally active part of the program.

The reactive object model is interruption-driven, which is very different from traditional object-oriented message-driven communication and synchronization. Let us take the exemple of a web browser: asking for a specific URL starts a connection to a site. But if the connection is too slow, the user might press the stop button: the connection then has to be cancelled.

One can program this behavior by testing every now and then if the button has been pressed. This solution is not very good for the structure of the code: connection code and testing code are interleaved. It is not very efficient either: many useless tests take CPU time.

Alternately, it is possible to have the button, whenever pressed, send a reactive *abort* command to the object performing the connection. The connection service would then be abandoned as well. In that solution, the connection part of the code is not tangled any more with unrelated tests. In that sense, reactive programming is a form of aspect-oriented programming [AOPp].

## 5.2 Introduction of reactive primitives

Our implementation of reactive objects in the Eiffel// language was based on the special semantics of the communication. In Eiffel//, a message between two processes develops in 3 phases:

- interruption of the receiver process,
- rendez-vous between caller and receiver processes,
- asynchronous service performed by the receiver.

We were exploiting the interruption phase and the synchronous rendez-vous to suspend the active process.

In Java, however, we do not benefit from this very interesting semantics: communication is synchronous and threads, which are the main parallel feature for multiprocessors, communicate directly through memory ... The implementation that we adopted is thus impossible. Nevertheless, Java already provides some primitives like *suspend*, *stop*, etc. that permit to slightly control the execution state of threads. These primitives are very close to the basic primitives that we introduced in Eiffel// reactive objects. Since Java programmers already know about them, we want to use them and retain their name for the API. Very unfortunately, they cannot be redefined: they are *final* functions of Java. This issue is relatively easy to address with a class-based MOP or even with a simple stub generator: a proxy can be introduced for every reactive class whose only purpose is to introduce the new definition. But even worse, suspension cannot take place during a loop in a thread: this really represents a problem since it is a style that our model promotes.

We thus need to extend the language. The first solution to the loop problem is to interleave if statements in the output source code. The user is not aware of this implementation and manipulates cleaner libraries: this solution thus retains the separation of concerns between normal classes on one hand and reactive classes which provide additional interaction definitions on the other hand. It unfortunately degrades performances, as we noted earlier. Moreover, it necesitates an extensive recompilation of the program and libraries source code.

The other way to interrupt a thread in a loop with a reactive primitive is to simply insert a *yield()* instruction inside of this loop. This solution still requires a minimal amount of recompilation, but much less than with the previous solution; the integration with Java constructs also appears more logical. We retained this last choice for our implementation. Inserting *yield()*

```
class URL implements ReactiveObject {
 public void tryToConnect() { ... }
}
```

```
class TimeOut {
 ...
 void run {
   ...
   setPageLoaded(false);
   myURL.tryToConnect();
   wait(timeoutElapsed);
   if (! isPageLoaded())
     myURL.stop();
   ...
```

Figure 8: Source code for the URL and TimeOut.

statements inside loops is not feasible with class-based MOPs. EPP proves handy for this task since it can manipulate the program up to the instruction level and provides many hooks for doing so.

We are finishing the implementation of this extension as an EPP plugin. The plugin is very similar to TDP Java, except that it makes use of different hooks (like interface definitions, etc.). Our experimentations have shown that the implementation technique is indeed perfectly adequate and reactive as expected. The user adds a reactive behavior to a class by simply stating that it implements the ReactiveObject interface. For instance, in our browser example, the regular connection part of the code could be implemented by the class URL as shown in Fig.8. The control of the connection can then be left to a class TIMEOUT whose code could cancel the connection at some point.

## 5.3 Perspectives

In recent software bus component architectures like Sun's JavaBeans, programs carry out their task through the processing of events. In particular,

coordination of the different activities of components in such programs is based on the asynchronous exchange of events. However, control of this coordination is made very unclear by its complete decentralization: each component receives and knows how to handle some events, but the behavior of the complete system is not independantly stated and thus easily manageable.

This characteristic is even one of the strong points of the architecture, since new components can be added without reworking the entire application. Yet, given a declarative expression, additional centralized control would not compromise this property, and would be really useful for dealing with the complexity of a real-world system behavior. This is especially true if components run on top of threads or in different processes or machines: it is well known that asynchronous systems are inherently difficult to understand and debug.

[Rou96] defined not only the simple reactive object model that we presented in the previous sections, but also more comprehensive and event-driven synchronization frameworks based upon the Electre [CR95] language. We plan to program a similar system on top of our Java extension in the near future. This system, which would introduce a supplementary, explicit and centralized synchronization, should provide the programmer with a better understanding of the behavior of a system of components (or possibly the designer with a straightforward implementation). More importantly, formal tools have been developed for Electre and would be applicable. This should allow us to prove good behaviors or to detect situations that might cause failure and to suppress them.

As a side note, the semantics of the hierarchical model of reactivity that we designed in [Rou96] are based on what could be qualified of *synchronization by necessity*. Several loosely-coupled clocks are mapped onto a hierarchically structured system of reactive objects: these need only be resynchronized if they exchange events across the boundary of two clocks. This feature is interesting in a system of network-distributed components ands seems for instance suitable for World-Wide Web applications.

# 6 Conclusion and future work

We presented in this article the extensible preprocessor kit EPP, an application framework for preprocessors which makes use of Ld-2. We also explained succintly how to use it. A preprocessor programmer can implement EPP plugins which are reusable extension modules and which can add new grammar rules to Java.

EPP is currently written in the Ld-2 language which provides the system mixin feature for combining plugins; Ld-2 is itself implemented in Common Lisp. We discussed Ld-2 current and potential abilities (especially for AOP).

We are currently implementing an EPP plugin which adds system mixin features to Java. After this work is completed, we plan to rewrite EPP directly in Java with this plugin. It is worth noting that features supporting plugin programming such as pattern matching and Lisp backquote macro can themselves be implemented as EPP plugins. We are also studying a possible port of EPP to C++.

We then described two extensions programmed with EPP: a simple data-parallel framework for Java, and a reactive object extension. These frameworks demonstrate some of EPP capabilities at integrating new features into Java. We will also investigate other models of parallel programming in the future. Defining new extensions raises the question of their composability or compatibility: we are going to explore all the possibilities offered by the Ld-2 language in that respect and define a more complete set of guidelines for programming with the EPP system.

Java threads provide sufficient basic mechanisms for implementations, and in a portable way since all Java virtual machines provide this facility. However, generally speaking, the thread model is far too complex and error-prone for large-scale programming. Moreover, as we saw for the reactive plugin, manipulating some of their features in Java may need language extensions.

Adding distributed features adds more complexity to the application. We really believe that transparent, high-level frameworks are necessary for dealing with large-scale (should we say real-world ?) software development:

this is especially important if a team, and not only a single developer, is involved. The models and tools that we introduced might give some hints about how to obtain enough simplification for the programming of complex systems, especially for parallelism and distribution.

# References


[AOPp] XEROX Parc Aspect Oriented Programming project. Aspect-oriented programming home page.

[BC90] G. Bracha and W. Cook. Mixin-based Inheritance. In *Proc. of ECOOP/OOPSLA'90*, 1990.

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, volume 30(10) of *ACM Sigplan Notices*, pages 285–299, Austin, Texas, October 1995. ACM Press.

[CMP+97] Denis Caromel, Alistair McEwan, Jonathan Poole, Yves Roudier, David Sagnol, and the Europa group. Europa Parallel C++ version 2.1. In *Future Directions for Parallel C++*, June 1997.
see http://www.dcs.kcl.ac.uk/Future_Directions/ and the standard can also be found at http://www.etl.go.jp/etl/bunsan/~roudier/postscripts/europa.ps.gz.

[CR95] Franck Cassez and Olivier Roux. Compilation of the Electre reactive language into finite transition systems. *Theoretical Computer Science (TCS)*, (146):109–143, 1995.

[CR96] Denis Caromel and Yves Roudier. Reactive Programming in Eiffel//. In Jean-Pierre Briot, Jean-Marc Geib, and Akinori Yonezawa, editors, *Object-Based Parallel and Distributed Computation OBPDC'95*, number 1107 in Lecture Notes in Computer Science, pages 125–147. Springer-Verlag, 1996.

[ea94] F. Bodin et al. Sage++: A Class Library for Building Fortran and C++ Restructuring Tools. In *Proc. of Object-Oriented Numerics Confs., Oregon*, April 1994.

[GJG96] J. Gosling, B. Joy, and Steele. G. *The Java Language Specification*. Java Series. Sun Microsystems, 1996.

[Hir97] S. Hirano. HORB: Distributed Execution of Java Programs. In *Proceedings of WWCA'97 (International Conference on Worldwide Computing and Its Applications), Tsukuba, Japan*, March 1997. see also http://ring.etl.go.jp/openlab/horb/.

[HQ91] P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[Ich] Y. Ichisugi. EPP and Lods home page. http://www.etl.go.jp/etl/bunsan/~ichisugi.

[Ich97] Y. Ichisugi. Ld-2 users manual (DRAFT). Technical report, Electrotechnical Laboratory, 1997. can be obtained from [Ich]. In Japanese.

[IHT+96] Y. Ichisugi, S. Hirano, H. Tanuma, K. Suzaki, and M. Tsukamoto. Compiler Widgets — Reusable and Extensible Parts of Language System —. In *The 11th workshop of object oriented computing WOOC'96, Japan Society of Software Science and Technology*, March 1996. In Japanese.

[Ish94] Y. Ishikawa. Meta-level Architecture for Extendable C++, Draft Document. Technical Report Technical Report TR-94024, Real World Computing Partnership, 1994.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Dan Bobrow. *The Art of the Meta Object Protocol.* MIT Press, 1991.

[KK97] A. Kumeta and M. Komuro. Meta-Programming Framework for Java. In *The 12th workshop of object oriented computing WOOC'96, Japan Society of Software Science and Technology*, March 1997.

[LP96] P. Launay and J.L. Pazat. Integration of control and data parallelism in an object oriented language. In *Sixth Workshop on Compilers for Parallel Computers (CPC'96) Aachen, Germany*, December 1996.

[NPA] NPAC. HPJava reference and workshops site. http://www.npac.syr.edu/projects/javaforcse/.

[PZ97] Michael Philippsen and Matthias Zenger. JavaParty – transparent remote objects in java. In *ACM 1997 PPoPP workshop on Java for Science and Engineering Computation, Las Vegas, Nevada*, June 1997.

[Rou96] Yves Roudier. *Abstractions réactives pour les langages à objets parallèles : modèles et programmation.* PhD thesis, Université de Nice - Sophia Antipolis, December 1996.

[Ste90] G.L. Steele. *Common Lisp the Language, 2nd edition.* Digital Press, 1990.