Demonstration of Hyper-Programming in Java™

E. Zirintsis, G.N.C. Kirby & R. Morrison

School of Mathematical and Computational Sciences
University of St Andrews, North Haugh
St Andrews KY16 9SS
Scotland
{vangelis, graham, ron}@dcs.st-and.ac.uk

Abstract

We demonstrate the use of a hyper-programming system to build persistent Java applications in PJama, an orthogonally persistent version of Java™. This allows program representations to contain type-safe links to persistent objects embedded directly within the source code. The potential benefits include greater potential for static program checking, improved efficiency, and reduced programming effort.

1. Introduction

Persistent programming languages were developed in an effort to reduce the burden on the application programmer of organising the transfer of long-term data between volatile program storage and non-volatile storage [1]. Previously, application data to be retained between activations had to be written explicitly to a database or file system, and later read in again to the application space. This flattening and rebuilding of data structures involved a significant programming overhead, and an increased intellectual effort since the programmer had to keep track of a three-way mapping between program representation, database/file representation and real world. The introduction of orthogonally persistent languages meant that any program data could be made persistent simply by identifying it as such, with all transfers between memory hierarchy layers handled transparently.

The treatment of source programs as strongly typed persistent objects, which is made possible by the use of a Persistent Object System (POS) or Object Oriented Database as the support platform, permits a new approach to program construction. Hyper-programming involves storing strongly typed references to other persistent objects within a source program representation [2]. Thus the source code entity is represented by a graph rather than a linear text sequence. By analogy with hyper-text this is called a hyper-program. It may be considered as similar to a procedure closure, in that it contains both a textual program and an environment in which non-locally declared names may be resolved. The difference is that with a hyper-program the environment is explicitly constructed by the programmer, who specifies persistent objects to be bound into the hyper-program at construction time.

Figure 1 shows an example hyper-program, comprising a class definition containing two links. The first is to a class, *Person*, while the second is to an object that is an instance of that class. As illustrated by the link to the class, a hyper-programming system may support linking to non-first-class entities for convenience.

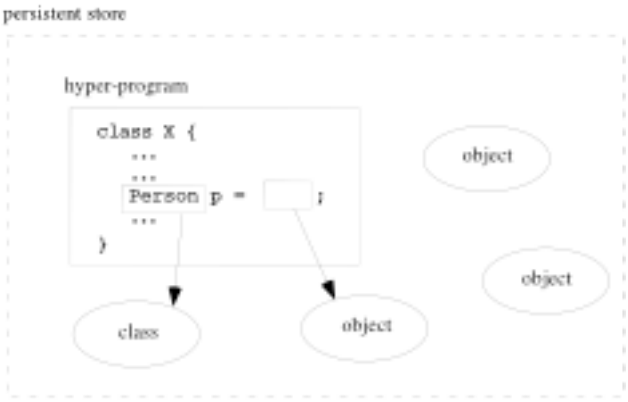

Figure 1. Example hyper-program

The support of hyper-program construction techniques by a POS provides a number of potential advantages:

- Program succinctness: textual descriptions of the locations and types of persistent components used by the program may be replaced by simple embedded references.
- Increased execution efficiency: checking of the validity of specified access paths to other components is factored out when they are embedded directly in

the source program. Checking of type consistency may be performed at compilation time rather than execution time.

- Reliable access to components: where a textual description of a component is replaced by a direct reference, the underlying referential integrity of the POS ensures that the component will always be accessible by the program. By contrast, where a textual description is used it may become invalid by the time the program executes, even if it was valid at the time the program was constructed.
- Complete closure representations: it is possible to fully represent code fragments that refer to existing values within their closure. This is not possible with purely textual representations, since the identities of those values may be significant.

2. Summary of Demonstration

Hyper-Program Composition and Viewing

The demonstration shows a prototype hyper-programming system [3] based on the PJama [4] persistent version of Java™.

Figure 2 shows an example of the hyper-programming user interface. It provides two tools: an editor in which hyper-programs are composed, together with an object/class browser that is used to locate persistent objects for linking. The browser is also used in conjunction with the editor for viewing hyper-programs: when the programmer clicks on a link in the editor, a representation of the corresponding linked entity is displayed in the browser.

The top window shows a hyper-program under construction in the editor. The body of the *main* method contains links to a static method, *Person.marry*, and to two instances of class *Person*. The lower window shows the object browser, currently displaying details of one of the object instances in the left pane, and the object's class in the right pane.

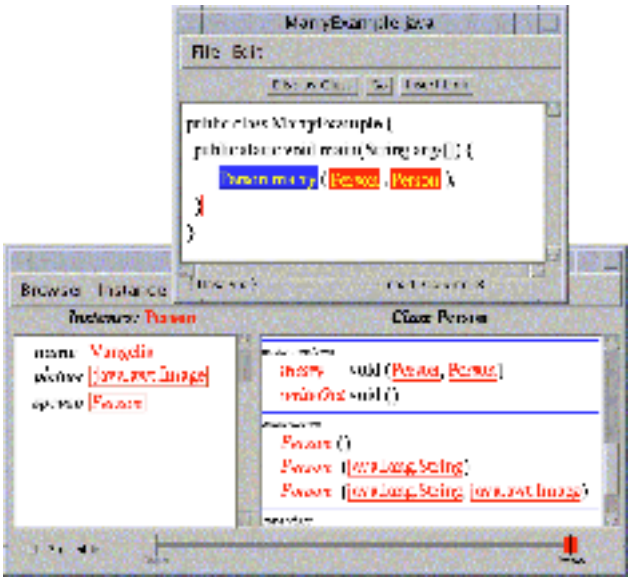

Figure 2. Hyper-programming user interface

The following activities are illustrated during the demonstration:

- locating persistent objects using the object/class browser;
- composing a hyper-program, including inserting hyper-links;
- viewing the hyper-linked entities in a hyper-program using the object/class browser;
- compiling and executing a hyper-program;
- dynamic invocation of object methods using the object/class browser;
- customisation of the editor and object/class browser displays, as described in the next section.

Customisation

By default, the editor displays hyper-links as textual labels. It is also possible to customise the display style on a per-object or per-class basis, achieved via user calls to a customisation API. The object/class browser can be similarly customised. Figure 3 shows an example in which both editor and browser have been customised to display instances of classes *Person* and *java.awt.Image* using an appropriate bitmap.

Figure 3. Example customisation

3. Design and Implementation Issues

The major issue in building hyper-programming systems concerns the semantics of the hyper-links, such as:

- what can a hyper-link refer to?
- what guarantees can be made about a hyper-link's referent data?
- how are hyper-links typed and when does type-checking occur?

The degrees of freedom regarding what a hyper-link can refer to depend upon the programming language semantics and the measure of open-ness in the system. Normally hyper-links will be able to refer to all first class language values. Second class entities not in the value space, such as classes or types, may also be conveniently hyper-linked depending on the flavour of the language. Update may be accommodated through hyper-links by linking to locations, which may or may not be first class values. More interesting is the extent to which hyper-links may refer to values created independently of the system, such as Web pages and DCOM objects. Furthermore the open-ness of the system can be extended by making the hyper-program representation open for other tools to manipulate.

In Java not all denotable values are first class values, for example methods, and there is no simple production rule to capture this in the syntactic definition. We define the denotable values that can be hyper-linked in Java as: objects; classes; interfaces; arrays; array elements; static members; non-static members; and constructors. Furthermore, links may be made to both values and locations that contain values (such as fields and array elements) where appropriate.

Table 1 shows the Java denotable values that can be hyper-linked in the hyper-programming prototype, and the corresponding syntactic productions [5].

Hyper-link To	Production
class	ClassType
primitive type	PrimitiveType
interface	InterfaceType
array type	ArrayType
object	Primary
primitive value	Literal
(static) field	FieldAccess
(static) method	Name
constructor	Name
array	Primary
array element	ArrayAccess

Table 1. Java hyper-links and productions

Referential integrity in a hyper-programming system means that once a hyper-link is established it is guaranteed by the system to exist and to be the same value when the hyper-link is executed. While this guarantee may be provided by a strongly typed persistent object store, it may also be expensive to provide in a distributed system. Variations therefore include the hyper-link being valid but not necessarily referring to the original value, and the hyper-link referring to a copy of the original. This may only be a problem where object identity is important such as in sharing semantics. A hyper-program may therefore display a range of failure modes from not failing to failure from the hyper-link being no longer valid.

The final issue is how hyper-links are typed, if at all. Assuming that they are, the interesting aspect of type checking is that the contract between the program and the referenced value may now take on a different agreement procedure. Instead of the program asserting the type of the hyper-link and the type checking system ensuring that the hyper-link has the correct type when it is used, the reverse may be used. That is the hyper-link knows its own type and therefore when it is used the program can be made to conform to this type. Statically this removes the need for type specifications for hyper-links in hyper-programs and dynamically it means that the program may be in error rather than the hyper-link.

4. Current research: hyper-code

Current research is directed at refining the concept of hyper-program to remove all distinctions between multiple program forms.

The goal is to maximise simplicity for the programmer. A single, uniform, program representation is presented at all stages of the software process. Since this

Figure 4. Hyper-code example

representation must be suitable for programs with closure, by implication it must be a form of hyper-program that can represent links to persistent values directly. To distinguish it from existing prototypes we refer to this representation as *hyper-code* [6].

The demonstrated PJama hyper-program system falls short of this goal in several respects. Different representations are used for creating new values and browsing existing values: new values are created by writing hyper-program source definitions, while existing values are displayed graphically in the object/class browser. Secondly, full representations are not provided for all existing values, in particular it is not possible to view method source code.

The hyper-code system will address these deficiencies by retaining all source code in the persistent store, and by providing the same single representation of values at all stages of the software process. As a consequence the distinction between editor and object browser will disappear: all programmer interaction with the system will take place via a single unified hyper-code editor. This will support a small set of simple operations that can be composed orthogonally to achieve all programming activities:

evaluate : this executes a selected fragment of hyper-code. If any result is produced this is returned as a further fragment of hyper-code.
explode : this expands a selected fragment of hyper-code to show more detail, while retaining the identities of any values referred to.
edit : this encompasses conventional editing facilities.

Implementation of this scheme in any particular language involves mapping the operations to the syntax of the language, and designing an appropriate hyper-code representation that can be used both to define new programs and represent existing values, in all possible cases. Figure 4 shows a simple example in ProcessBase [7, 8] in which the definition of a procedure *newPerson* contains various hyper-links, both exploded and unexploded. The exploded links, denoted by grey boxes, show detail of the linked entities in the form of more hyper-code.

5. Further Information

Further details of the hyper-program and hyper-code projects are available, with related publications, at the St Andrews web site:

`http://www-ppg.dcs.st-and.ac.uk/`

6. References

[1] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. “An Approach to Persistent Programming”. Computer Journal 26, 4 (1983) pp 360-365.

[2] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. “Persistent Hyper-Programs”. In **Persistent Object Systems**, Albano, A. & Morrison, R. (ed), Springer-Verlag (1992) pp 86-106.

[3] Zirintsis, E., Dunstan, V.S., Kirby, G.N.C. & Morrison, R. “Hyper-Programming in Java”. In **Advances in Persistent Object Systems**, Morrison, R., Jordan, M. & Atkinson, M.P. (ed), Morgan Kaufmann (1999).

[4] Atkinson, M.P., Daynès, L., Jordan, M.J., Printezis, T. & Spence, S. “An Orthogonally Persistent Java™”. ACM SIGMOD Record 25, 4 (1996) pp 68-75.

[5] Gosling, J. & McGilton, H. “The Java™ Language Environment: A White Paper”. Sun Microsystems, Inc (1995).

[6] Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Moore, V.S. & Morrison, R. “Unifying Interaction with Persistent Data and Program”. In **Interfaces to Database Systems**, Sawyer, P. (ed), Springer-Verlag (1994) pp 197-212.

[7] Warboys, B.C., Balasubramaniam, D., Greenwood, R.M., Kirby, G.N.C., Mayes, K., Morrison, R. & Munro, D.S. “Instances and Connectors: Issues for a Second Generation Process Language”. In **Lecture Notes in Computer Science 1487**, Gruhn, V. (ed), Springer-Verlag (1998) pp 137-142.

[8] Morrison, R., Balasubramaniam, D., Greenwood, M., Kirby, G.N.C., Mayes, K., Munro, D.S. & Warboys, B.C. “ProcessBase Reference Manual (Version 1.0.4)”. Universities of St Andrews and Manchester (1999).