Hyper-Code Revisited: Unifying Program Source, Executable and Data

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

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

Abstract. The technique of hyper-programming allows program representations held in a persistent store to contain embedded links to persistent code and data. In 1994, Connor *et al* proposed extending this to *hyper-code*, in which program source, executable code and data are all represented to the user in exactly the same form. Here we explore the concept of hyper-code in greater detail and present a set of abstract language-independent operations on which various concrete systems can be based. These operations (*explode*, *implode*, *evaluate*, *root* and *edit*) are provided by a single user interface tool that subsumes the functions of both an object browser and a program editor. We then describe a particular implementation using PJama (persistent Java) and examine the impact of several language features on the resulting system.

1 Introduction

The *hyper-code* abstraction was introduced in [1] as a means of unifying the concepts of source code, executable code and data in a programming system[1]. The motivation is that this may ease the task of the programmer, who is presented with a simpler environment in which the conceptually unnecessary distinction between these forms is removed. In terms of Brooks' *essences* and *accidents* [2], this distinction is an accident resulting from inadequacies in existing programming tools; it is not essential to the construction and understanding of software systems.

Orthogonal persistence [3] brought about several similar simplifications of the programmer's task. One was to unify short-term and long-term data. Another was to unify data and code, in the sense that executable code became first-class (as a procedure or an object) and could be manipulated in the same way as other data. Hyper-code builds on these simplifications by further unifying source code and executable code. The result is that the distinction between them is completely removed: the programmer sees only a single program representation form throughout the software lifecycle, during program construction, execution, debugging, and viewing existing programs and data.

As a consequence, only a single programming tool is required to manipulate this uniform representation form, rather than the various program editors, data browsers,

[1] In that paper it was termed *hyper-source*.

debuggers, etc needed otherwise. Various processes such as compilation and linking are also accidental and hidden from the programmer.

In this paper we develop the idea of hyper-code further and attempt to separate the general issues from the language-specific. We describe it in the context of a framework comprising:

- Two abstract domains of *entities* and *representations*, with a set of operations over them. These are intended to be sufficiently general to be applicable to any programming system, and are used to aid description rather than being visible to programmers.
- Criteria to be satisfied by any candidate Hyper-Code Representation form (HCR).
- Criteria to be satisfied by any candidate set of operations over HCRs.
- A particular proposed language-independent HCR.
- A particular proposed set of (broadly) language-independent HCR operations.
- An example implementation of these in a concrete hyper-code system for a particular language (PJama [4]).

We then attempt to draw some conclusions from our experiences in mapping the general concepts to a specific language, and discuss the effect of certain language features on such an exercise.

2 Related Work

A number of programming environments and program editors have attempted to attack accidental complexities. Emacs [5] is a text editing tool that achieves some integration of the programming process, by allowing various operations such as compiling and linking to be invoked from within the editor. However, it does not mask the presence of such accidental operations, and it does not provide any integration of source code, executable code and data.

The Metrowerks CodeWarrior [6] and Microsoft Visual Basic [7] programming environments accelerate the development process by combining an editor, compiler, linker and debugger into a single application. This gathers source code, libraries, graphic resources, and other files into a *project*. An application can be built and executed by pressing a single button, thus the environments largely succeed in hiding accidental operations. They do not, however, integrate source code and data, since these are viewed and manipulated in completely different forms.

Smalltalk-80 is a graphical, interactive object-oriented environment [8] that hides most of the accidents of the traditional programming cycle. However, there are different tools for editing, browsing and debugging, and breakpoints cannot be set interactively. New code being constructed is represented differently from existing objects.

Hyper-programming, as developed in Napier88 [9] and PJama [10], forms the basis for the hyper-code program representation to be introduced in the next section. In those systems, however, there is again a distinction between source code and data, and there is no interactive debugging support.

Finally, the original hyper-code proposal [1] introduced the fundamental idea of hyper-code, that of completely unifying program and data. This paper extends that by developing a general model and describing an implemented system.

3 The Hyper-Code Model

3.1 Design Goals

Software systems may be programmed at various levels of abstraction. The concerns at each level are different, ranging from fine detail such as register values, memory accesses etc, to higher level concepts such as abstract data types, process models etc. The choice of an appropriate level of abstraction depends on the nature of the application. This in turn determines the appropriate tools and software entities to be used to construct the application.

Hyper-code is designed to support programming at a relatively high level of abstraction, which hides the existence of multiple program representations and tools. This requires a single program representation form, and associated tool, that are adequate to support all the activities necessary during the software development cycle. These operations include:

- constructing new programs, which may operate on existing data and programs;
- editing existing programs;
- browsing or viewing existing data structures;
- executing programs, in some cases with debugging and profiling support.

The goal, then, is that the programmer may carry out the entire program development cycle without knowledge of the underlying software tools that support it. The software is viewed in the single consistent hyper-code form in all contexts, whether it is being written for the first time, constructed from existing components, executed, debugged, etc. Where errors occur they are reported in terms of hyper-code; existing persistent data is viewed as hyper-code. This model gives the conceptual simplicity of direct source code interpretation, while retaining the obvious efficiency benefits of standard software tools.

Since hyper-code plays such a large part in the software development environment, two factors are particularly significant to the programmer: the form in which hyper-code is presented, and the operations that may be performed on it.

3.2 Hyper-Code Representations

It is possible to design hyper-code interfaces for various different languages. The precise form of the representation used will depend on the language, as will other aspects of the user interface. We can however identify certain general criteria, derived in part from the design goals listed above:

- To support the construction of new programs that operate on existing data values and programs, the Hyper-Code Representation (HCR) should incorporate denotations for such entities.
- To support browsing or viewing of existing data values, the HCR should allow their structure to be examined interactively.
- Any detailed view of an existing data value should have the same form as that in which an equivalent new value could be defined within a program (since there should only be one single representation form). As a consequence, the representation of an existing value should be source-code-based and syntactically valid. It may also be desirable, for interoperability, for the HCR to accommodate third-party source code. These factors preclude the use of a graphical data representation.
- It should be possible to initiate execution of a particular HCR instance, and to trace the execution path if desired.

One (and perhaps the only possible) HCR form that fulfils these criteria is the hyper-program [9], with the refinement that hyper-links may themselves have internal structure. A hyper-program may contain both text and hyper-links to existing entities, thus allowing entities to be bound at composition time, rather than the usual restriction to textual specifications that are resolved later at compilation, linking or run-time. For hyper-code, a given link may be expanded interactively to view the structure of the linked entity, without altering the link's meaning—the expanded link continues to denote the same entity. **Fig. 1** illustrates this HCR form, containing text, denoted by horizontal lines, and hyper-links, which may or may not be expanded.

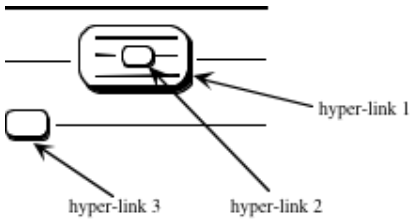

Fig. 1. General Form of an HCR

Here hyper-link 1 has been expanded, so that a representation of the linked entity is displayed within it in the same style. Hyper-links 2 and 3 have not been expanded. Nested links can be expanded to arbitrary degree.

3.3 Hyper-Code Operations

The operations provided by any specific hyper-code interface must support at least the activities identified earlier: editing new and existing programs; browsing data; and executing programs. Clearly there are many possible sets of such operations. In this section we will define one such set which has been designed with simplicity in mind. To aid the description of these hyper-code operations, we first introduce some termi-

nology concerned with programming language entities and representations of those entities.

Domains and Domain Operations

For a given programming language, let E be the domain of language entities, containing all the first class values defined by the language (the Universe of Discourse) together with various denotable non-first-class entities. Depending on the language, the non-first-class entities may include types, classes and executable code.

Let R be the domain of concrete representations of entities in E. These representations could be textual or have some more complex structure. Each entity in E has at least one representation in R. A simple example is the integer value *two* in E and its representation 2 in R. The programmer interacts with the programming system solely by viewing and manipulating representations in R. Entities in E are never dealt with directly, but only through their corresponding representations.

R and E are disjoint. In particular, a representation in R is not itself an entity. Of course, in some circumstances it may be necessary for an executing program to manipulate representations of entities (as is common in reflective programming). To extend the example above, the representation 2 could be manipulated by a program as a string containing the digit 2, and this string could itself be represented as "2". Thus we can distinguish the value, a second value that represents the first (both in E), and the two corresponding representations (in R).

E may be partitioned into a set of executable entities E_{exec} and a set of non-executable entities $E_{no\text{-}exec}$. The former contains programs and program fragments, while the latter contains first class values, types, classes, etc. E_{exec} may be further partitioned into $E_{exec\text{-}res}$, $E_{exec\text{-}no\text{-}res}$ and $E_{exec\text{-}err}$ depending on whether execution generates a result, completes with no result, or fails to complete due to an error. The error may be detected either statically or dynamically.

R may be similarly partitioned, following the structure of E, into R_{exec} (containing $R_{exec\text{-}res}$, $R_{exec\text{-}no\text{-}res}$ and $R_{exec\text{-}err}$) and $R_{no\text{-}exec}$.

We are now in a position to define four *domain operations* over E and R. Our hypothesis is that these domain operations are sufficiently general to describe any set of concrete hyper-code operations for any particular language, and we will illustrate their use in defining our chosen set of hyper-code operations. The domain operations map between and within the domains E and R, as illustrated in **Fig. 2**.

- *reflect* maps a valid representation to a corresponding entity ($R \Rightarrow E$).
- *reify* performs the reverse operation, mapping an entity to a corresponding representation ($E \Rightarrow R$).
- *execute* executes an executable entity, with potential side effects to the state of the entity domain. This may generate a result ($E_{exec} \Rightarrow E$), complete successfully with no result, or fail due to a static or dynamic error.
- *transform* maps one representation into another ($R \Rightarrow R$).

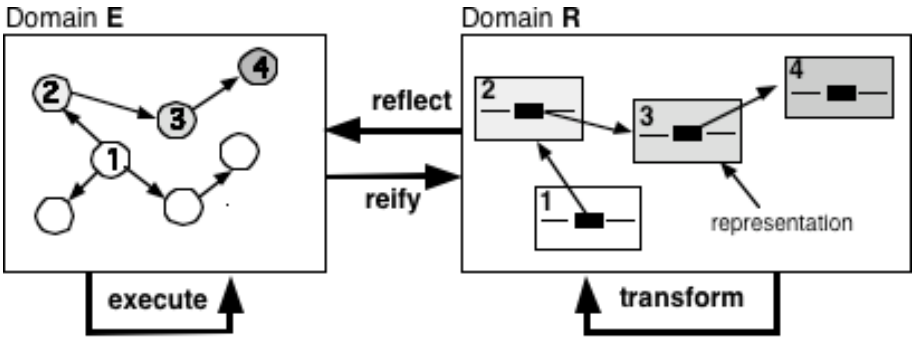

Fig. 2. Domain Operations

The rather abstract definitions of the domain operations above may be interpreted in various ways for different concrete hyper-code systems, thus imparting various semantics to the operations. For example, *execute* could involve strict, lazy or partial evaluation of an executable entity, depending on the model of computation supported by the language. The style of representation produced by *reify* could vary. The *transform* operation could be unconstrained, allowing any representation to be produced, or it could limit the possible representations as with a syntax-directed editor.

Specific Hyper-Code Operations

We now propose a small set of concrete hyper-code operations which is sufficient to fulfil programming requirements, and minimal. Later we will show how these operations may be mapped to a particular example language, together with a specific HCR form. The hyper-code operations are:

- *explode* expands a selected region of a hyper-code representation to show greater detail. This is itself expressed in the form of a hyper-code representation.
- *implode* performs the reverse of *explode*, hiding detail within a hyper-code representation.
- *evaluate* executes a selected hyper-code representation. If there is a result it is expressed as a new hyper-code representation.
- *edit* alters a hyper-code representation.
- *root* returns a selected persistent root as a hyper-code representation.

The actions of these operations can be characterised in terms of the domain operations:

- *explode* and *implode* both replace a selected representation with another, either more or less detailed. They involve a *reflect* operation to yield the entity represented, followed by a *reify* operation to yield a different representation. The result is in *R*.
- *evaluate* involves various sequences of operations, depending on the particular sub-domain of the representation being evaluated:
  - for $R_{exec\text{-}no\text{-}res}$, it involves a *reflect* operation to yield an executable code entity, followed by an *execute* operation to execute that code, which returns no result.

An example is the execution of a traditional "program" such as *gcc*, which is a self-contained sequence of statements that perform some action, with no result being returned directly.

- for $R_{exec\text{-}res}$, it involves a *reflect* operation to yield an executable code entity, followed by an *execute* operation to execute that code, followed by a *reify* operation to yield a representation of the entity returned as a result of execution. The result is in *R*. For example, evaluation of the code fragment represented by the characters *2+3* yields a result represented by the character *5*, as supported in interactive languages such as Smalltalk-80 [8] and Galileo [11].
- for $R_{no\text{-}exec}$, it involves a *reflect* operation to yield the entity represented, followed by a *reify* operation to yield a representation. The result is itself in $R_{no\text{-}exec}$. This is effectively a null operation defined for completeness; since the entity is not executable it is returned immediately as the result. Examples include evaluation of the representation *5* and of a hyper-link denoting an existing entity. Loosely, these correspond to manifest program literals.
- for $R_{exec\text{-}err}$, it involves a *reflect* operation to yield an entity, which is executed if the error is not detected statically. In either case an error is reported.

- *edit* involves a *transform* operation.
- *root* involves a *reify* operation on a persistent entity to yield a representation.

Thus traditional self-contained programs lie in $R_{exec\text{-}no\text{-}res}$, while fragments that return results lie in $R_{exec\text{-}res}$. The *evaluate* operation is defined over both forms.

Tracing Evaluation

The remaining design goal that has not yet been addressed is that of tracing the execution path through an HCR during its evaluation. This requires the HCR to display an indication of the current point of execution at any given time[2], and to provide some means of viewing the current state of any variables. More generally, it should be possible to view the entity bound to any identifier, whether mutable or not.

Since the HCR form supports links that may be exploded to view their structure, the same form can be used for identifiers. This is consistent with the requirement that a single consistent representation should be provided for data and code in all contexts. As a consequence, an HCR being evaluated is changed dynamically as evaluation progresses:

- When evaluation reaches the point at which an identifier is initialised, all subsequent occurrences of that identifier in the HCR, within the identifier's scope, are replaced by links. Depending on the particular language mapping, the links may be to a specific *identifier* entity, or directly to the entity bound to the identifier. In either case, the bound entity may be viewed by *exploding* a link. When evaluation leaves the identifier's scope, the original textual identifier again replaces the links.
- Where an identifier is bound to a mutable location (variable), the corresponding links are updated each time the location is updated.

---

[2] Multiple threads can be dealt with by displaying a separate copy of the HCR for each thread.

As a consequence, although the HCR is changed during evaluation, it returns to its original state after evaluation completes. Where an entity is produced as the result of an evaluation, its representation is returned to the programmer. Depending on the interpretation of *reify* chosen for a particular hyper-code system, the result may be a single unexploded hyper-link, an exploded hyper-link, or some other fragment of hyper-code. The programmer chooses whether the result should be returned as a new HCR or inserted into the original HCR being evaluated.

4 Design and Implementation of a Particular Hyper-Code System

In this section we give a brief overview of the design and implementation[3] of a prototype hyper-code system in PJama [4].

4.1 Mapping of Domains

In Java, and hence PJama, the domain of language entities *E* contains objects, arrays, primitive values, variables, classes, interfaces, array types and primitive types [12]. We also include code entities, comprising expressions, sequences of statements, or complete class definitions.

Every entity in *E* has a corresponding representation in the domain *R*, being a combination of text and hyper-links. We chose to allow hyper-linking to any entity that could be bound to an identifier in a Java program, giving a correspondence between links and identifiers. As a consequence, all entities can be hyper-linked except code entities. This is because the Java model (largely, strict evaluation) does not allow an identifier to be bound to a code expression: an identifier declaration or update causes the expression to be evaluated and the result, rather than the expression, to be bound. For example, evaluating the statement

```
int i = 1 + 2;
```

causes the value *3* to be bound to *i*, rather than the expression *1 + 2*.

Although unsatisfying in that not every entity in *E* can be denoted by a hyper-link, this restriction does not appear to be limiting in practice. Rather, it is intrinsic to the language model of strict evaluation.

Less obviously, we chose to omit methods and fields from *E* after experimentation with various designs. The fact that they are not first-class made it too awkward to design satisfactory exploded representations for them.

4.2 User Interface

Fig. 3 shows an example of an HCR displayed in a hyper-code window. The first hyper-link is to the class *Person* and the second to an instance of that class (classes and objects are distinguished by different colours).

---

[3] Details are available at *http://www-ppg.dcs.st-and.ac.uk/Research/HyperCode/*

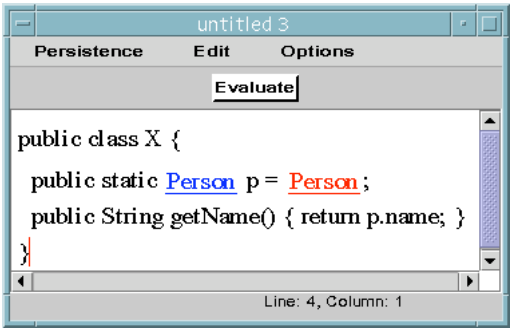

Fig. 3. Example Hyper-Code Representation

Fig. 4 shows the effect of performing the *explode* operation on the object link and then again on one of the object links revealed (to the person's address). Each exploded link shows a hyper-code representation of the corresponding object. The representation comprises a call to a newly generated constructor method[4] (*GeneratePerson.newPerson* and *GenerateAddress.newAddress* respectively), taking as parameters links to the object's field values.

This representation fulfils the two main criteria of allowing the structure of the object to be viewed, and being a syntactically valid code-based representation in precisely the same form that a programmer might write to construct the object initially. The only difference is that the hyper-code displayed in any exploded link cannot be modified by the programmer, since it is a representation of an entity that already exists. It can of course be copied into another window and the copy modified.

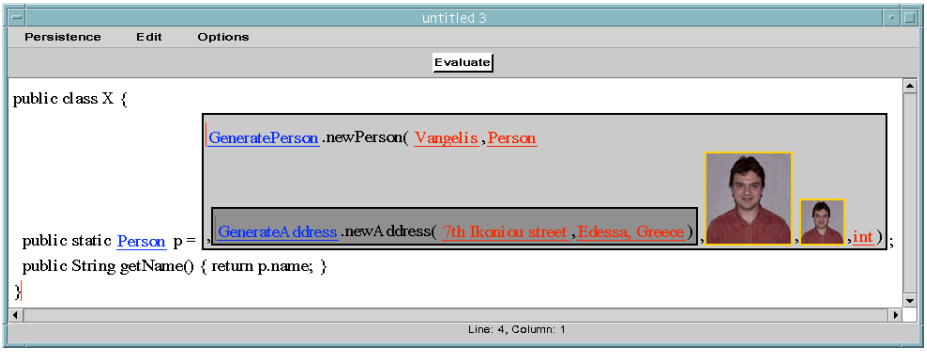

Fig. 4. Effect of Exploding an Object Link

Fig. 5 shows the effect of exploding a class link and several further links within it. Each exploded link displays the class's source code, which itself contains both text and links. This and the previous figure illustrate the unification of code (classes) and data (objects), since exactly the same representation form is used in both.

[4] This is done since there is no guarantee of a suitable constructor being available.

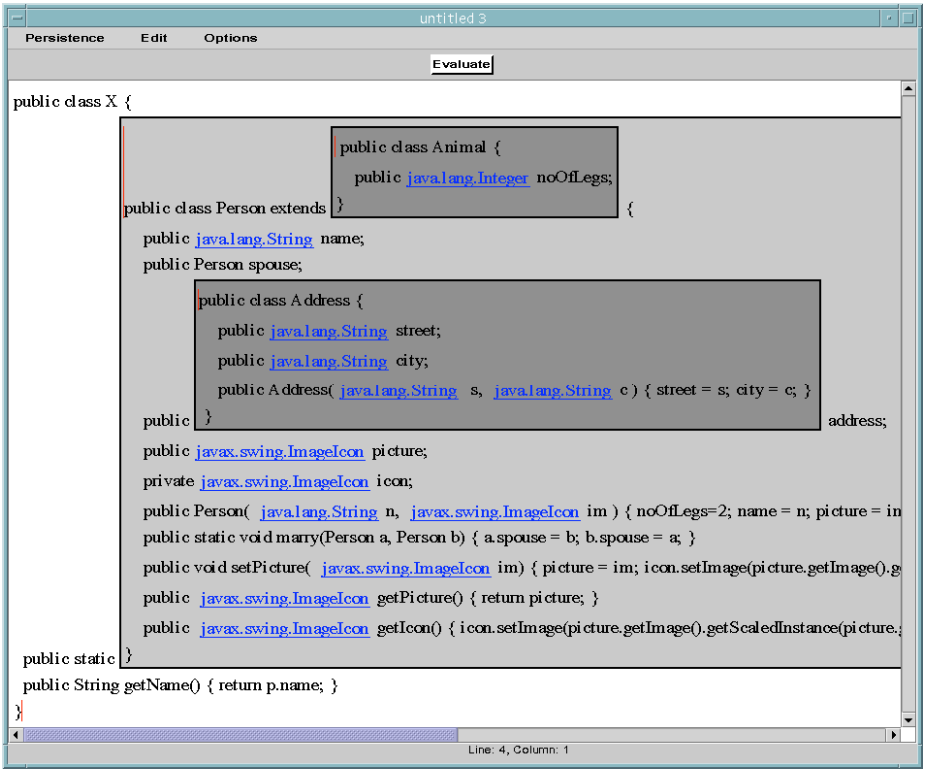

Fig. 5. Effect of Exploding a Class Link

To write hyper-code operating on existing data, the programmer needs access to the persistent roots. **Fig. 6** shows the result of the *root* operation: a window displays a set of hyper-links corresponding to the multiple PJama roots. To operate on one of these, a copy of the hyper-link can be dragged into any other window where it can form part of the hyper-code under construction. In the example, a hyper-link to an array has been exploded so that one of its elements can be dragged to another window.

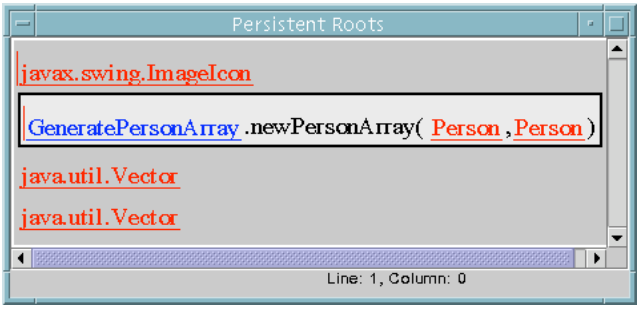

Fig. 6. Persistent Roots Window

At the time of writing (August 2000) the system as described thus far has been implemented. The tracing and control of execution paths using dynamic replacement of variables and links has not yet been completed. However it is possible to give an impression of how this will appear using simulated representations: **Fig. 7** shows snapshots at three successive points in the execution of method *m*, indicated by the arrows. In the first, the variable *p* has not yet been initialised and so is represented textually. In the second, after *p*'s initialisation, all uses of the variable are replaced by hyper-links to its current value, which can be viewed by exploding the appropriate link. Finally, after completion of the method execution leaves *p*'s scope and the variables return to being textual identifiers. Hyper-links to existing entities remain unchanged during evaluation.

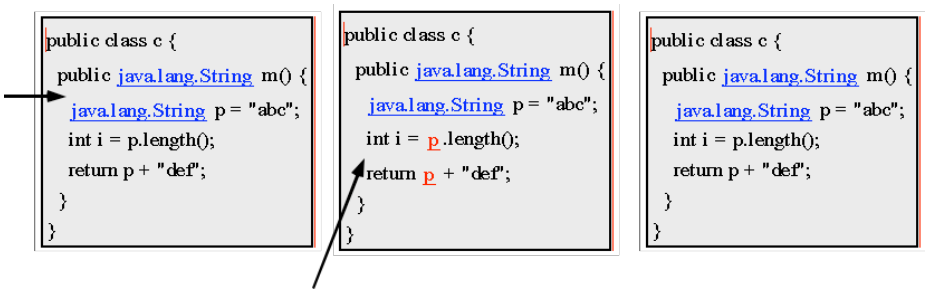

Fig. 7. Execution Tracing

4.3 Implementation

For simplicity, the hyper-code system has been implemented using source code transformation. When an *evaluate* operation is performed, each hyper-link is replaced by a textual expression to retrieve the linked entity from a hidden persistent data structure generated by the system. Fragments of code are transformed into complete class definitions. Instrumentation and control code is inserted in order to track execution paths, halt at breakpoints, and update variables/hyper-links in the display. The transformed code is compiled dynamically using the standard compiler, and the resulting classes loaded and invoked. Full details of the implementation are given in [13].

This scheme has the advantages of simplicity and portability. Alternative techniques such as byte code transformation at class loading time [14, 15] would probably give considerably better performance for the *evaluate* operation, at the cost of greater complexity. Although in other contexts such alternatives also have the advantage of not requiring class source code, this is not relevant here since the entire hyper-code scheme relies on source code being present.

5 Discussion

5.1 General Issues

The paper has described hyper-code in a general framework and given a flavour of a prototype implementation for PJama. Since that is not yet completed, it is not possible to draw any conclusions as to whether the hyper-code model, by removing some of the accidental complexities of programming systems, brings any worthwhile benefits to the programmer. This question could only be answered by conducting a full human factors analysis on target users.

We have claimed that the description of a programming system in terms of abstract entity and representation domains and operations is generally applicable. Similarly, we suggested that a hyper-program form for hyper-code representations is probably the only form possible given the original criteria for hyper-code. These claims need to be tested by designing mappings for other languages.

One obvious problem with the hyper-code representation form is its handling of cyclic data structures: since it imposes a hierarchy on a data structure by displaying each referenced object within the object referring to it, a cyclic structure can be expanded indefinitely without terminating.

5.2 Operation Sets

The set of concrete hyper-code operations described (*explode*, *implode*, *evaluate*, *edit*, *root*) is just one example of the many possible sets that could support the required activities. It does appear to be simple and minimal, at least in comparison with some of the earlier operation sets from which it evolved during this work. For example, in one version we distinguished between *inspection* of an entity, which generated a read-only representation, and *modification*, in which a representation could be edited and then reflected into a new entity that would replace the original. This scheme turned out to be unnecessarily complex and was too closely coupled with issues of mutability in a particular language. In a later version we defined separate operations for expanding a hyper-link in place and for expanding it to give a new hyper-code fragment. This was unnecessary given the ability to copy the hyper-code within an exploded link, so the two operations were replaced by the single *explode*.

It is not clear whether this operation set is suitable for all languages (without the *root* operation for non-persistent languages); it appears to be suitable for PJama and for another concrete hyper-code system that we have designed for the language ProcessBase [16]. One possibility for further work is to investigate alternative operation sets.

5.3 Language Issues

Focusing on implementation of particular concrete hyper-code systems, two issues may be of interest:

- how do the features of particular languages affect the mapping of those languages to hyper-code?
- what would be the features of a language ideally suited to hyper-code?

We now attempt to relate these issues to the experience of designing the PJama mapping and ProcessBase mappings.

Object Constructors

Since the exploded representation of a Java object takes the form of an expression to construct an equivalent object, it is problematic to decide which of the constructors of that class should be used. There is no way for the system to determine whether a given constructor initialises all of the fields or what other side-effects it causes. This issue is simpler in ProcessBase since a view (i.e. a record, the closest analogue to an object) may be constructed with an anonymous constructor that simply initialises all of the fields. The PJama version attempts to simulate this by generating a new method to construct the object (**Fig. 4**) but this is rather unwieldy.

Information Hiding

Java's encapsulation model of information hiding, specified by the **protected** and **private** modifiers, does not fit neatly with the desired operation of *explode*, in which exploding an object link should display a full representation of that object. If the object contains hidden members, the choice is to display only a partial representation, in which case the goal of representing the object by a valid expression is not met, or to attempt to subvert the protection using the *AccessibleObject* class available in recent Java versions.

In ProcessBase the only information hiding mechanism is the procedure closure, in which the access path to data used by the procedure may be hidden from general access. This means that the issue described above occurs for procedures rather than objects. The planned system behaviour is that, as with Napier88 [9], the full procedure closure may be viewed when a procedure link is exploded. Thus any data hidden in the closure will be revealed as a link in the hyper-code. This is both a powerful mechanism, and a potential problem since it precludes real information hiding. Clearly, facilities could be provided to apply additional protection mechanisms to the source code.

These are examples of the more general issue of whether a hyper-code system should allow the programmer to achieve anything that could not be achieved by writing conventional programs in the language. In the particular PJama case the system cannot, since it is itself implemented entirely in the language. More generally, where the implementer may have control over aspects of the target language implementation, this is a significant issue.

Mutable Locations

Some complexity is introduced by Java's standard treatment of variables, where an identifier may denote either a location or the current value of that location, depending on its context. For example, a special class of identifier link is required, which behaves differently from all others in that the linked value changes during evaluation

on each update. Similarly, the programmer can copy a link to the location of a temporary method variable held on the stack, and paste it in another window. What should be the semantics of that link after the method has returned? The pragmatic but unsatisfactory solution for PJama is that copying the link gives only the corresponding textual identifier.

Both of these problems are avoided if mutable locations are first-class, as in ProcessBase, which provides an explicit location constructor. This simplifies matters—in the first example, the value bound to a link now never changes, although if that value is a location its contents may. In the second example, the location will automatically persist beyond the method invocation, and so the link will continue to denote the same location wherever it is pasted.

Openness

The hyper-code scheme relies on source code being either recorded or generated as required for all entities, so that the *explode* operation can show details. This is feasible in a self-contained persistent system, in which all entities are originally derived from the evaluation of source code. Clearly however it does not work in an open system that has to deal with third-party code for which source is not available. This is true for much of the standard Java class libraries, as well as for most commercial Java software. It is possible to generate approximations to the source code by de-compilation of byte code, but often this will be rendered unusable by deliberate code obfuscation.

Reflection

Java provides good support for introspection over class structure, via the *Reflect* package. It does not, however, provide introspection over method code, even at the byte level, or dynamic access to the compiler. Both of these can be achieved, but implementation of the hyper-code system would be simpler if they were supported directly. Another issue is that current Java compilers work only at the granularity of complete class definitions, so there is considerable overhead involved in processing small expressions since they must be wrapped up into complete classes.

Desirable and Essential Features for Hyper-Code

Although the framework outlined in this paper is intended to be general enough to fit all languages, the following mechanisms, in some form, are essential for hyper-code:

- structural reflection over types
- graphical user interface programming

Several further language features are beneficial when designing a hyper-code system. Some affect the simplicity and elegance of the resulting system, while others impact on the ease of implementation:

- anonymous value constructors (not tightly bound to the value's type)
- information hiding by access path rather than encapsulation
- first-class locations
- dynamic compiler access
- structural reflection over code

The first four items above are provided by ProcessBase, and we intend to implement our hyper-code design as part of our research into compliant architectures [17].

6 Conclusions

This paper has described the following:

- a motivation for providing simpler programming systems;
- a proposed set of criteria to be fulfilled by any candidate system;
- a language-independent program representation form (hyper-code) and a set of operations over it;
- a particular mapping of the above to the Java language.

It is not yet possible to draw any conclusions as to whether hyper-code brings any worthwhile benefits to the programmer, but it appears that it may deliver a valuable simplification of the programming process.

The current status of the PJama hyper-code system, as outlined earlier, is that all of the operations have been implemented, but dynamic evaluation tracing has not (the implementation is currently underway). A design for ProcessBase has been completed and is described in [13].

7 Acknowledgements

This builds on earlier work carried out with Richard Connor, Vivienne Dunstan and Quintin Cutts. It is supported by EPSRC grant GR/M88938 'Compliant Systems Architecture Phase 2'. We thank the referees for their helpful comments.

References

1. Connor R.C.H., Cutts Q.I., Kirby G.N.C., Moore V.S., Morrison R. Unifying Interaction with Persistent Data and Program. In: P. Sawyer (ed) Interfaces to Database Systems, Proc. 2nd International Workshop on User Interfaces to Databases, Ambleside, Cumbria, 1994. Springer-Verlag, 1994, pp 197-212
2. Brooks F.P. No Silver Bullet – Essence and Accidents of Software Engineering. In: Proc. Information Processing 86, 1986, pp 1069
3. Atkinson M.P., Bailey P.J., Chisholm K.J., Cockshott W.P., Morrison R. An Approach to Persistent Programming. Comp. J. 1983; 26,4:360-365
4. Atkinson M.P., Daynès L., Jordan M.J., Printezis T., Spence S. An Orthogonally Persistent Java™. ACM SIGMOD Record 1996; 25,4:68-75
5. Stallman R. GNU Emacs Manual. Free Software Foundation, 1997
6. Metrowerks Inc. CodeWarrior Pro 5, 1999
7. Microsoft Corporation. Microsoft® Visual Basic® 6.0 Programmer's Guide. Microsoft Press, ISBN 1-57231-863-5, 1998
8. Goldberg A., Robson D. Smalltalk-80: The Language and its Implementation. Addison Wesley, Reading, Massachusetts, 1983
9. Morrison R., Connor R.C.H., Cutts Q.I., Dunstan V.S., Kirby G.N.C. Exploiting Persistent Linkage in Software Engineering Environments. Comp. J. 1995; 38,1:1-16
10. Zirintsis E., Dunstan V.S., Kirby G.N.C., Morrison R. Hyper-Programming in Java. In: R. Morrison, M. Jordan and M. P. Atkinson (ed) Advances in Persistent Object Systems, Proc. 8th International Workshop on Persistent Object Systems (POS8) and 3rd International Workshop on Persistence and Java (PJW3), Tiburon, California, 1998. Morgan Kaufmann, 1999, pp 370-382
11. Albano A., Cardelli L., Orsini R. Galileo: a Strongly Typed, Interactive Conceptual Language. ACM ToDS 1985; 10,2:230-260
12. Gosling J., Joy B., Steele G. The Java™ Language Specification. Addison-Wesley, ISBN 0-201-63451-1, 1996
13. Zirintsis E. Towards Simplification of the Software Development Process: The Hyper-Code Abstraction (PhD Thesis, University of St Andrews). *in preparation.*
14. Marquez A., Zigman J.N., Blackburn S.M. Fast Portable Orthogonally Persistent Java. Software - Practice and Experience, Special Issue on Persistent Object Systems 2000; 30,4:449-479
15. Chiba S. Load-Time Structural Reflection in Java. In: Proc. ECOOP 2000, 2000
16. Morrison R., Balasubramaniam D., Greenwood M., Kirby G.N.C., Mayes K., Munro D.S., Warboys B.C. ProcessBase Reference Manual (Version 1.0.6). Universities of St Andrews and Manchester, 1999
17. Morrison R., Balasubramaniam D., Greenwood R.M., Kirby G.N.C., Mayes K., Munro D.S., Warboys B.C. A Compliant Persistent Architecture. Software - Practice and Experience, Special Issue on Persistent Object Systems 2000; 30,4:363-386