Using generative programming to visualise hypercode in complex and dynamic systems

Katherine Mickan **Ron Morrison** **Graham Kirby**
Dharini Balasubramaniam **Evangelos Zirintsis**

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

Abstract

The research presented here takes place in the context of the EC Funded ArchWare project which focuses on innovative architecture-centric languages, frameworks and tools for engineering evolvable software systems. Of particular interest are complex and dynamic systems characterised by the need to evolve to meet changing requirements without total shutdown or the loss of state information. The ArchWare approach uses the unique combination of a pi-calculus based architecture description language, persistence and hypercode. Hypercode provides the essential base technology for composing and decomposing system components without losing state. The contribution of this work is an implementation of hypercode using generative programming techniques to produce different hypercode visualisations.

Keywords: hypercode, structural reflection, generative programming, system evolution.

1 Introduction

Hypercode was introduced by Kirby et al., (Kirby, Connor, Cutts, Dearle, Farkas & Morrison 1992), in work motivated by the search for better programming language support for the software engineering process. By unifying the concepts of source code, executable code and data in a programming system, hypercode eases 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, (Zirintsis 2000), this distinction is an accident resulting from inadequacies in existing programming tools; it is not essential to the construction and understanding of software systems. In a hypercode system the user composes hypercode and the system executes it. The user only sees a single view of the system and underlying operations are abstracted over.

A hypercode program is constructed from a mixture of text and hyperlinks. The text is normal program source code and the hyperlinks point to existing values. In a hypercode system the user can compose programs interactively, navigating the environment and selecting data items, including functions, to be incorporated into their program as hyperlinks, (Kirby et al. 1992). Clicking on a hyperlink allows the user to see a hypercode representation of the value. The artificial distinction between source and executables is removed, therefore a hypercode view can be generated for any value in the system, (Morrison, Connor, Cutts, Dearle, Farkas, Kirby, McGettrick & Zirintsis 1999).

The first hyper-programming system, implemented for Napier88, (Kirby et al. 1992), demonstrated how the technique could ease the task of reflective programming and provide support for source representations of procedure closures. Farkas & Dearle (1994), presented a mechanism called Octopus, that permitted the types of values to be abstracted over and values to be manipulated in a type independent manner. Octopus comprised a set of operations over a dynamic infinite union type, essentially providing higher level tools based on the structural reflection in the langauge. Another aspect of their work was partially resolved hyper-programming, which enabled the production of templates. The templates allowed programs to be constructed and compiled without the requirement that the values used by the program be present. In this manner, individual components could be constructed independently and later assembled to form a complete application. Zirintsis, Dunstan, Kirby & Morrison (1999), constructed a hypercode system for Java and established the hypercode operations, through which a user interacts with hypercode.

The arena of complex and dynamic systems presents itself as a new application for hypercode technology. Greenwood, Robertson & Warboys (2000), use the term *co-evolution* to describe the symbiotic relationship between dynamically changing commercial environments and the software that supports them. In these systems there is an ever present demand to accommodate change over time. As requirements change, software needs the capacity to adapt to the altered environment in which it is used, in order to avoid increasing redundancy. This evolutionary potential is particularly relevant to large, long-lived systems which are expensive to build and deploy.

The conventional path to evolution involves editing source code, recompiling and rebinding. However, this may not always be acceptable. In large or long-running systems the source code, for the components which are to be changed, may no longer be available. Even if the source code is obtainable, it may be impossible to rebind the system without other components' sources. Perhaps more importantly this evolution by redefinition style may lose valuable data. When the current version of a component is replaced by its new redefined version, local data, representing the current system state, can be lost, (Morrison, Balasubramaniam, Greenwood, Kirby, Mayes, Munro &

Copyright ©2004, Australian Computer Society, Inc. This paper appeared at the 27th Australasian Computer Science Conference, The University of Otago, Dunedin, New Zealand. Conferences in Research and Practice in Information Technology, Vol. 26. V. Estivill-Castro, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

This work is supported by the EC Framework V project ArchWare (IST-2001-32360) and the ORS Award Scheme.

Warboys 2000*a*).

Applying hypercode to evolution results in desirable properties lacking in the traditional approach. Firstly, hypercode is able to capture closure and consequently, a hypercode program can be evolved without total system shutdown or loss of state. Secondly, it abstracts over the distinction between source code and values, thereby guaranteeing that a component's source code will be accessible.

The work here is part of the ArchWare project which considers evolution from the perspective of software architectures - a context in which a system is constructed from a set of components bound together by connectors. This compositional nature is reflected in the evolutionary process, where evolution is based on the system's decomposition into components, the replacing or modifying of those components, and the recomposition of the evolved system. The *compose* and *decompose* operations have been defined to structure this process, (Morrison et al. 2000*a*).

1.1 Generative Technology

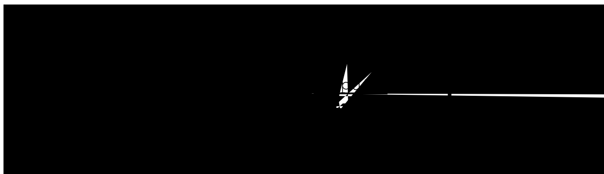
We have implemented a hypercode system using generative technology, (Czarnecki & Eisenecker 2000), to realise a set of hypercode operations for hypercode programming. Employing generative techniques, rather than making changes to the software platform, separates the orthogonal concerns of the programming language and hypercode.

The *evaluate_hypercode* operation transforms, compiles, binds, executes and produces a visualisation for a hypercode program. Using generators in the implementation implies a set of transformations which map the original hypercode onto some target code according to a set of rules. Compiling and executing the target code completes the *evaluate_hypercode* process. Different generators can operate over the same piece of hypercode to produce different target code. This paper describes two generator algorithms used to produce different aspects of the visualisation of hypercode evaluation.

The first generator produces a program equivalent to the hypercode which can be compiled and executed. Execution of the program effects a visualisation of *evaluate_hypercode*'s result. The second generator produces target code to display an animation of the evaluation's progress, a process known as *identifier tracking.* During identifier tracking the user can view values in the closure of the executing code.

2 Hypercode

A sample of hypercode using the language ProcessBase, (Morrison, Balasubramaniam, Greenwood, Kirby, Mayes, Munro & Warboys 1999), is shown in Figure 1, where a function, *processor*, is defined which contains four hyperlinks. The mixture of text

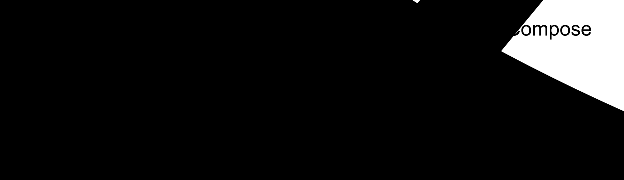


The ⇒ sy… which operates und… compiler may merge a number… program to be compiled. The merging fa… binder and the rules under which it operates deter-mine the order and scoping of the included files. Decomposition is defined as the reverse of composition, giving the reversible equation:

$$P \circledR Q \Leftrightarrow S_{P \circledR Q}$$

Hypercode provides the mechanism to realise the reversible composition operation, since it offers introspection and the preservation of shared state and data …hroughout the evolutionary process. Normally, en…psulated data in a program is difficult to access and …aking the encapsulation is an irreversible opera-

… further explain the *compose* and *decompose* op… a scenario is modelled in Figure 3. The orig…m, in the bottom left of the figure, can be … a composition of three filter components … two pipes. Two of the filter componen… …ta outside of the connected co… …by the star shapes.



it into its … still ma… …nks to the data. Changes may then … the components so that two of the filt… …d into a single filter. This new filter … maintains links to the data referred … …nal filters. The three components can … to form a new system.

… can be interpreted from both an architec- … …ocess p…rspective. From the architec- … …rsp…ctive th… diagram c…ptures the structure

first class functions Functions are first class values in the language.

All the hypercode operations have been implemented in ProcessBase, forming an environment in which to program hypercode. ProcessBase is also the language in which the user composes hypercode programs. The programming environment consists of a hypercode system which accepts a request to perform an operation on some hypercode and returns the result of the operation if applicable.

Figure 5 shows a short hypercode program as it appears to the user.

```
let filter <- fun(in_stream: string) -> string
  spacer ++ in_stream ++ spacer

filter(in_pipe)
```

Figure 5: Hypercode representation viewed by the user

The same hypercode is represented as an XML string marked up with hyperlinks in Figure 6. The

```
let filter <- fun(in_stream: string ) -> string
  <hl id="1">spacer</hl> ++ in_stream
            ++ <hl id="1">spacer</hl>

 filter(<hl id="2">in_pipe</hl>)
```

Figure 6: Hypercode representation encoded as XML

code defines a function, *filter*, which concatenates its input with a *spacer* on each end and returns the resulting string. Both *spacer* hyperlinks point to the same data value. In the final line, *filter* is called with a parameter *in_pipe*, which is a hyperlink. In the XML, the hyperlinks are marked up with <hl> tags which label them with ID strings. Each hypercode program is associated with a list of values in which the ID string is used to locate the value pointed to by this hyperlink.

4.1 The evaluate_hypercode Operation

The conventional evaluate operation is comprised of compilation and execution, whereas *evaluate_hypercode* involves transformation, compilation, binding, execution and visualisation. During the operation some hypercode is executed and hypercode is returned as the result. Figure 7 shows how evaluating the *filter* function produces a hyperlink to the result of execution.

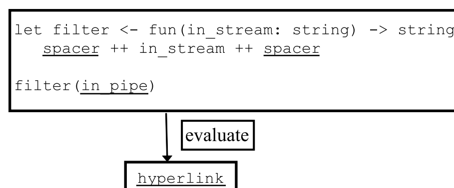


Figure 7: *evaluate_hypercode* operation

Figure 8 shows how *evaluate_hypercode* can be defined in terms of the functions applied to the hypercode, where *h* is some hypercode. The compiler is used to reflect a transformed hypercode program; the

```
evaluate_hypercode ( h ) =
  visualise ( execute ( compile ( transform ( h ) ) ) )
```

Figure 8: *evaluate_hypercode* definition

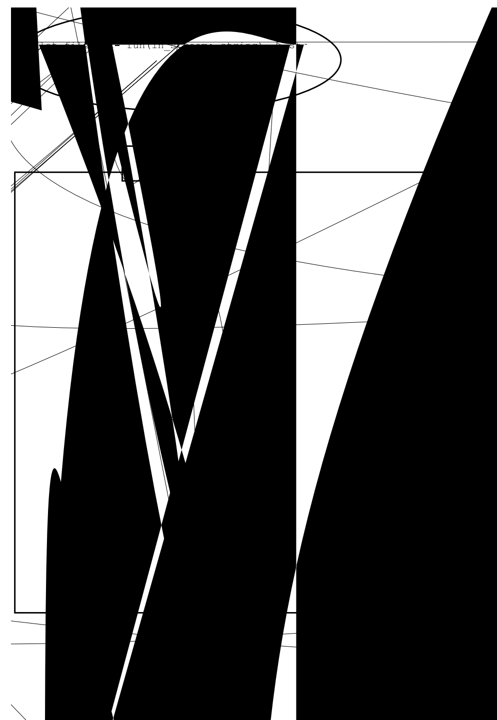
result of a successful compilation is then executed; and the result of the execution reified, which produces a visualisation of the result for the user to view.

There are two generators used in the implementation of *evaluate_hypercode* which will be focused on in the next sections. In order to display the result of *evaluate_hypercode* to the user, the hypercode must be compiled and executed. The task of the first generator is to transform the hypercode into ProcessBase source so it can be processed by the conventional compiler. The second generator carries out a transformation for *identifier tracking*, where the hypercode execution is reified, and which can be used for debugging hypercode. During *identifier tracking*, identifiers that come into scope during the execution turn into hyperlinks, enabling their values to be examined by the user.

4.2 Compiling Hyperlinks

Part of evaluating hypercode is linking the existing values pointed to by the hyperlinks into the executable code. The new implementation of *evaluate_hypercode* for ProcessBase uses the generative programming technique known as source-to-source transformation to do this. It makes no changes to the standard compiler. A function is added which encloses the hypercode and takes a list of values as its parameter. When the transformed hypercode is executed, a list of values pointed to by the hyperlinks is passed to the function. The example code in Figure 5 will be used to show how the transformation progresses.

An overview of *evaluate_hypercode* is depicted in Figure 9. A hypercode program, shown as it would be



rendered for the user, is input to *evaluate_hypercode*. The program has three hyperlinks in it pointing to two values. *evaluate_hypercode* performs the following steps:

1. Source-to-source transformation on the input produces transformed code and a list of hyperlinks.
2. Compiling the transformed code generates a function.
3. Projecting and executing the function with the list of hyperlinks as its parameter gives a visualisation of *evaluate_hypercode*.

The final outcome of evaluating the hypercode is output and rendered as a hyperlink.

In the first part of the transformation, Figure 10, each hyperlink in the hypercode is replaced with a new unique name: *hl1* replaces *spacer* and *hl2* replaces *in_pipe*. Introducing these new identifiers into the code requires some type coercion. Most of the code generated by the following transformations is concerned with this task.

```
let filter <- fun(in_stream: string) -> string
  hl1 ++ in_stream ++ hl1

filter(hl2)
```

Figure 10: Hyperlinks replaced by names

The values pointed to by the hyperlinks are associated with their new names in the list of hyperlinks which was referred to in Figure 9. This list is generated by the hypercode system as part of step 1. In step 3, the compiled and transformed hypercode is called with the list as its parameter. Figure 11 shows the type of the list, which is a record. It has *id* field to hold the name; and an *entity* field, of the infinite union type *any*, to hold the value. Any data structure could have been used. The use of a list is an implementation decision.

```
rec type list_type is view[id: string;
                          entity: any;
                          next: loc[list_type]]
```

Figure 11: Type of the list of hyperlinks

The code fragment in Figure 10 is made into legal ProcessBase code by including declarations for the new names, *hl1* and *hl2*. The values associated with these identifiers will be part of the list of hyperlinks, and therefore have type *any*, so their specific types are also declared here. *hl1* and *hl2* are cast onto their types as they are declared.

In Figure 12 the type of the parameter list, *list_type* is defined; followed by the definition of a function, *getFromList*, which will extract a value from the list, given its name. Next, the types of the hyperlinks are defined; in this case they are both string. This type information is acquired using the *typerep* library function in ProcessBase. Subsequently a project clause performs a cast from the infinite union type, which is the type of the value in the list, onto *hl1_type*; the same is done for *hl2*.

In Figure 13, the *generator* function is added around the hypercode; it takes as a parameter the table of hyperlinks, *list*. The *generator* function is so called because it generates an executable version of the hypercode.

```
rec type list_type ...

let getFromList <- fun(id: string) -> any
  .... !find the value in the list of parameters

! define the types of the hyperlinks
type hl1_type is string
type hl2_type is string

!fetch the values from the list and cast
!them onto their correct types before assignment
let hl1 <- project getFromList("hl1") as X onto
     hl1_type: X
     default: nil(hl1_type)

let hl2 <- project getFromList("hl2") as X onto
     hl2_type: X
     default: nil(hl2_type)

let filter <- fun(in_stream: string) -> string
begin
   hl1 ++ in_stream ++ hl1
end

filter( hl2)
```

Figure 12: Declaring the new identifier

```
rec type list_type ...

let generator<-fun(list:list_type) -> fun()->any{
  let getFromList <- fun(id: string) -> any
        ...! get a value from the list

  type hl1_type is string
  type hl2_type is string
  let hl1 <- project ...
  let hl2 <- project ...

  type return_type is string
  let wrapper <- fun() -> return_type {

      let filter <- fun(in_stream: string) -> string
         hl1 ++ in_stream ++ hl1
      filter( hl2)
  }

  ! generator returns a function which calls wrapper
  fun() -> any
     any(wrapper())
}
generator
```

Figure 13: Adding the *generator* function

generator needs to have a fixed type so it can be invoked from a fixed context in the hypercode system. Its return type is $fun() \rightarrow any$, a function which returns a value of type *any*: which can be explained by looking further down the code to the *wrapper* function. This function wraps around the original hypercode, so that it can later be executed alone without the overhead of projections during the execution of *generator*. The return type of *wrapper* is the same type returned by the original hypercode, *return_type*.

Following the *wrapper* function, a function is defined which calls *wrapper* and returns the result inside an infinite union type. This function is the value returned by *generator*. The *generator* should return the *wrapper* function, so the hypercode can be executed without the projections overhead, but the type of *wrapper* is only discovered during the source code transformation. Therefore the call to *wrapper* is en-

closed in a function which returns a value of type *any*. The final line in Figure 13 is *generator*, hence the result of executing the code is the *generator* function itself.

In summary, when the hypercode system executes the code in Figure 13, it obtains the *generator* function. The execution of *generator* returns a function, and executing this function returns the result of executing *wrapper*, which is a *string*, inside an *any*. Executing the function returned by *generator* is equivalent to executing the hypercode.

Having completed the transformation, the generated code is compiled and executed - this is steps 2 and 3 in Figure 9. Figure 14 shows the section of code in the hypercode system which acts on the transformed hypercode. Step 2 is applying the compiler to the transformed hypercode.

```
!call the compiler with code
!(the transformed hypercode)
let compilation_result <- compile(code)

!project the result of compilation onto the type
!of generator
project compilation_result.result as X onto
   fun() -> any: {
      let Y <- X() !Y is the generator function

      project Y as generator onto
      fun(list_type) -> fun()-> any: {
          let hypercode_function <- generator(list)
          let hypercode_result <- hypercode_function()
          explode(hypercode_result)
      }
      fun(list_type) -> fun(): {
         let hypercode_function <- generator(list)
         hypercode_function()
      default: raise exception
      }
   }
   default: raise exception
```

Figure 14: Compiling and executing the hypercode

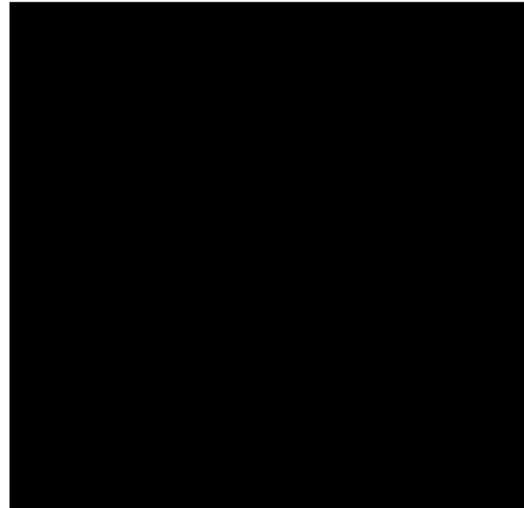
The start of Figure 14 shows the compiler being called with *code*, the transformed hypercode, as its parameter. The compiler returns a structure with the executable function in *result*. In the first *project* statement the result of compilation is projected onto type *fun()* $\rightarrow$ *any* and executed, which is the start of step 3. The execution returns Y, which is the *generator* function, wrapped in an infinite union type. This is in turn projected onto the specific type of *generator*, and then executed. The parameter *list* passed to *generator* is the list of hyperlinks built up during the source transformation. Executing *generator* gives a function, *hypercode_function*, and executing this is equivalent to running the hypercode. This gives the return value of the hypercode inside *hypercode_result* which has type *any*. To extract a value of the correct type from *hypercode_result*, the hypercode operation *explode*, which reifies a value, is used.

The second part of the project statement, where Y is projected onto *fun(list_type)* $\rightarrow$ *fun()*, is used when evaluating a hypercode program which does not return a value. Then *generator* returns a function which returns nothing. The third part of the project statement raises an exception and is chosen when neither of the previous types matches the value, which would only occur if the hypercode system failed.

4.3 Identifier Tracking

During *identifier tracking*, hypercode execution is animated and identifiers in scope from the user's point of view become hyperlinks. The user may click on any identifier and see its value at that point in execution. This paper concentrates on an animation which is only updated at user inserted breakpoints.

Figure 15 shows the user view of identifier tracking on the function *hyphenate*. This function declares



its scope, as well as an ID string, to connect it with a value in the table when it becomes a hyperlink. Figure 16 shows how this information is included in the hypercode as XML. Each identifier is marked up

```
<hl id="divider00" scope="0:0">divider</hl> ++
<hl id="in_pipe00" scope="0:0">in_pipe</hl>
```

Figure 16: XML representation of the user view of identifier tracking

as a hyperlink, the attributes of which indicate its ID and scope. The code in Figure 16 is a sample of what is generated from a syntactic analysis of the hypercode from line 4 of Figure 15(a), where each identifier is replaced with a hyperlink.

The markup included in the hypercode allows the user interface to determine what view to show the user depending on the point of execution. When the code stops at a breakpoint, the user interface receives a message telling it which breakpoint has been reached. From this information the scope can be established, and hence the identifiers which are hyperlinks in that scope can be shown. Hyperlinks which were included as part of the composition of the hypercode will always be hyperlinks.

4.3.2 Producing the executing version

Source transformation to produce the executing version of the hypercode involves adding code for a number of different purposes. Firstly, all the new hyperlinks which have been added to the user view of the code need to be associated with existing values. This is done by including code after each declaration to add the newly created value to a list of values and hyperlink IDs. Therefore, when the user clicks on a hyperlink, the correct value can be discovered from its ID by looking in the table. The value can then be reified to display as hypercode to the user. Secondly, the source code for functions needs to be stored, so that the function values can later be reified. Unlike other data values, where the hypercode view can be generated from the value, function source code must be explicitly saved. Thirdly, code needs to be added at breakpoints to stop the execution and coordinate with the user view.

A step by step example of source code transformation on the *hyphenate* function from Figure 15 is presented. The transformation is performed during the same syntactic analysis which produced the hypercode for the user interface described in the previous section.

The first transformation is inserting code which will store the values of identifiers as part of the hypercode representation, HCR, a data structure containing the source code and links which represent a hypercode program. In Figure 17, each identifier declaration in the *hyphenate* function is followed by a line which does this. The *addHL* function adds a hyperlink to the HCR. The *createHL* function makes a new hyperlink from an ID, e.g. *in_*

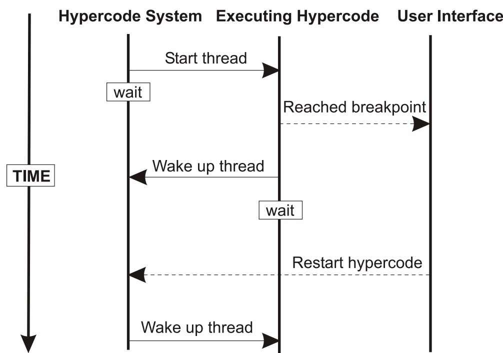


Figure 20: Hypercode system, executing hypercode and user interface interact during identifier tracking

terface are within the hypercode system, they each have their own thread, so they are shown separately in the diagram. Communications between the user interface and the other two threads are in the form of messages, shown as dotted lines. The hypercode system and executing hypercode communicate through semaphores.

Figure 21 shows the portion of code in the hypercode system which manages the interactions between the executing hypercode and the user interface. The

```
...
1. let sem <- newSemaphore(0)
2. let executing_hypercode <-
        start(fun();outer(hcr,sem))

3. sem.wait() ! wait for signal from hypercode
4. getMessage(restart) ! wait here for user message
5. sem.signal() ! restart hypercode
...
```

Figure 21: Hypercode system restarts the hypercode execution after a signal from the user

code in this figure is not part of the transformation, but part of the hypercode system which has generated the transformation. In line 1, a new semaphore object is created. The second line starts a new thread to execute the *outer* function, i.e. the hypercode, which has already been compiled by the time this code executes in the hypercode system. The definition of the *outer* function has also been altered here to take a semaphore as its second parameter. Figure 22 shows the code which is added at this stage of the transformation, and line 1 shows the new definition of *outer*. Looking back to line 3 of Figure 21, it can be seen

```
1. let outer <- fun(hcr: loc[HCR]; sem: semaphore)
2. begin
3.  let hyphenate <- fun(in_pipe: string)->string
4.    begin
     ...
5.     sendMessage(breakpoint_number,scope)
6.     sem.signal() ! wake up main thread
7.     sem.wait() ! suspend self
     ...
8. end
9. outer
```

Figure 22: Adding semaphores to stop the code at breakpoints

that after starting the execution of *outer*, the hypercode system waits. Now the execution passes to the code in Figure 22, which executes down to the breakpoint, at which stage it sends a message to the user interface, in line 5, to inform it which breakpoint it has stopped at and what the current scope is. This information is inserted during the transformation when the syntax analyser can access the current scope. The user has defined where the breakpoints are. In line 6, the hypercode signals the semaphore which wakes up the main thread, and in the next line it suspends its own execution. The hypercode system in Figure 21 has been waiting on line 3 for the signal from the hypercode. Once it receives that, the hypercode execution has been suspended and the user is examining the code halted at the breakpoint. When the user sends a message to restart the code, which is waited on in line 4, the hypercode system signals the semaphore, thereby restarting the execution of the hypercode thread.

4.4 Implementation Status

The current hypercode implementation for ProcessBase includes the hypercode operations and the generators described in this paper. The user interface is still under development.

5 Related Work

5.1 Hypercode

Apart from the work on hyper-programming systems described in the introduction there are a few projects which have implemented programming systems related to the hypercode environment. The Intentional Programming project at Microsoft Research, (Simonyi 1995) built a development environment which operates over *active source*: a graph data structure representing the program. The behaviour of the program source is implemented using methods operating on this graph. Nodes of the graph can be elements from different programming languages, each node is associated with a *declaration of intention* which corresponds to the syntax definition of the programming language. Identifier declarations are associated with their uses in the source graph.

Other projects such as the CodeProcessor, (Vanter & Boshernitsan 2000) built at Sun Microsystems aim to improve programming environments by manipulating source code to exploit the formal structure of the programming language. These tools can use information derived via linguistic analysis to offer services that are impractical for purely text-based tools. However, they fall short of including live data values in the programming process.

5.2 Architectural evolution

There is a growing body of work on the subject of software evolution, where complex and dynamic software systems are of particular interest. These systems have emergent properties which can result in changes to the system becoming necessary after deployment. In addition, the software often operates within a changing business environment. Lehman's first law of software evolution, (Lehman 1996), states that a software system embedded in a real world domain must continually change or becoming increasingly less useful. The changes are driven by the need to repair software faults, cope with new operating environments, and add or modify functionality.

A flexible software architecture aids the process of software maintenance and means that a software system can be configured to meet the needs of users

under various conditions of use. Some researchers argue that software should be able to meet the needs of all users, (Kiczales, Lamping, Maeda, Keppel & McNamee 1993), which leads into work in compliant architectures, (Morrison, Balasubramaniam, Greenwood, Kirby, Mayes, Munro & Warboys 2000*b*). These architectures accommodate, and are thus compliant to, the needs of particular applications and users.

An explicit run-time representation of a system's architecture can aid evolution of the system at runtime. An implementation of such a system is described in (Oreizy & Taylor 1998). ArchStudio is a tool suite that supports architecture-based development. Changes are made to an architectural model and then reified into implementation by a runtime architecture infrastructure. However, this system restricts the user to a particular architectural style.

6 Further Work

In future this work will be extended by incorporating hypercode into the ArchWare system. Hypercode would provide an interface to the Architecture Description Language defined as part of the project. It would also provide the means to implement the *compose* and *decompose* operations which characterise evolution at the architectural level.

7 Acknowledgements

This work is supported by the EC Framework V project ArchWare (IST-2001-32360) and the Overseas Research Students Award Scheme (ORS).

8 Conclusion

This work shows the use of generative technology to implement a hypercode system. Two separate source-to-source transformations have been applied to a piece of hypercode to generate different visualisations of the hypercode evaluation. Compilation of hyperlinks was facilitated by the basic transformation. The transformation for identifier tracking enabled the user to view a representation of the hypercode execution and access the values of identifiers in scope.

Generative technology facilitates an unlimited set of transformations on hypercode. These could include: conversion into an XML Infoset; inclusion of dynamic traces; and customised display of data types. Generators for such purposes assume the input of properly formed hypercode and alter it to produce different visualisations and useful formats. Alternatively, the application of transformations to improperly formed hypercode can be considered, allowing the user to enter programs in any appropriate format. For example, mathematical formulae could be entered in a formula editor, or a graphical tool could assist in designing a program.

Also presented here were the benefits of hypercode when applied to the evolution of software architectures: because hypercode represents closure and offers a unified view of the system it can be used to underpin the *compose* and *decompose* operations.

References

Czarnecki, K. & Eisenecker, U. (2000), *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley.

Farkas, A. & Dearle, A. (1994), 'The Octopus model and its implementation', *Australian Computer Science Comm. — Proc. 17th Annual Computer Science Conf., ACSC* **16**(1), 581–590.

Greenwood, M., Robertson, I. & Warboys, B. (2000), A support framework for dynamic organizations, *in* R. Conradi, ed., 'Proceedings of the 7th European Workshop in Software Process Technology (EWSPT 2000)', Vol. 1780 of *Lecture Notes in Computer Science*, Springer, Kaprun, Austria, pp. 6–20.

Greenwood, R., Balasubramaniam, D., Cimpan, S., Kirby, G., Mickan, K., Morrison, R., Oquendo, F., Robertson, I., Seet, W., Snowdon, B., Warboys, B. & Zirintsis, E. (2003), 'Process support for evolving active architectures', *9th European Workshop on Software Process Technology (EWSPT 2003), Helsinki, Finland* .

Kiczales, G., Lamping, J., Maeda, C., Keppel, D. & McNamee, D. (1993), The need for customizable operating systems, *in* 'Proceedings of the Fourth Workshop on Workstation Operating Systems', IEEE Computer Society Technical Committee on Operating Systems and Applications Environment, IEEE Computer Society Press, pp. 165–169.

Kirby, G., Connor, R., Cutts, Q., Dearle, A., Farkas, A. & Morrison, R. (1992), Persistent hyper-programs, *in* 'Persistent Object Systems', Springer-Verlag, pp. 86–106.

Kirby, G., Morrison, R. & Stemple, D. (1998), 'Linguistic reflection in Java', *Software - Practice and Experience* **28**(10), 1045–1077.

Lehman, M. (1996), Laws of software evolution revisited, *in* '5th European Workshop on Software Process Technology, EWSPT', Nancy, France, pp. 108–124.

Morrison, R., Balasubramaniam, D., Greenwood, M., Kirby, G., Mayes, K., Munro, D. & Warboys, B. (1999), ProcessBase reference manual (version 1.0.6), Technical report, Universities of St Andrews and Manchester.

Morrison, R., Balasubramaniam, D., Greenwood, R., Kirby, G., Mayes, K., Munro, D. & Warboys, B. (2000*a*), 'An approach to compliance in software architectures', *IEE Computing and Control Engineering Journal, Special Issue on Informatics* **11**(4), 195–200.

Morrison, R., Balasubramaniam, D., Greenwood, R., Kirby, G., Mayes, K., Munro, D. & Warboys, B. (2000*b*), 'A compliant persistent architecture', *Software - Practice and Experience, Special Issue on Persistent Object Systems* **30**(4), 363–386.

Morrison, R., Connor, R., Cutts, Q., Dearle, A., Farkas, A., Kirby, G., McGettrick, R. & Zirintsis, E. (1999), Current directions in hyper-programming, *in* 'Lecture Notes in Computer Science 1755', Springer-Verlag, pp. 316–340.

Oreizy, P. & Taylor, R. (1998), On the role of software architectures in runtime system reconfiguration, *in* 'Proceedings of the International Conference on Configurable Distributed Systems (ICCDS 4)', Annapolis MD.

Simonyi, C. (1995), The death of computer languages, the birth of intentional programming, Technical Report MSR-TR-95-52, Microsoft Research Microsoft Corporation.

Vanter, M. L. V. D. & Boshernitsan, M. (2000), Displaying and editing source code in software engineering environments, *in* 'Second International Symposium on Constructing Software Engineering Tools (CoSET2000)'.

Zirintsis, E. (2000), Towards Simplification of the Software Development Process: The Hyper-Code Abstraction, PhD, University of St Andrews.

Zirintsis, E., Dunstan, V. S., Kirby, G. N. C. & Morrison, R. (1999), Hyper-programming in Java, *in* R. Morrison, M. Jordan & M. P. Atkinson, eds, 'Advances in Persistent Object Systems', Morgan Kaufmann, Tiburon, California.