Process Support for Evolving Active Architectures

R. Mark Greenwood[1], Dharini Balasubramaniam[2], Sorana Cîmpan[3], Graham N.C. Kirby[2], Kath Mickan[2], Ron Morrison[2], Flavio Oquendo[3], Ian Robertson[1], Wykeen Seet[1], Bob Snowdon[1], Brian C. Warboys[1], Evangelos Zirintsis[2]

1: {Department of Computer Science, The University of Manchester, Manchester, M13 9PL, UK.}
{markg,robertsi,seetw,rsnowdon,brian}@cs.man.ac.uk
2: {School of Computer Science, The University of St Andrews, St Andrews, Fife, KY16 9SS, UK.}
{dharini,graham,ron,kath, vangelis}@dcs.st-and.ac.uk
3: {ESIA, Universite de Savoie, 5 Chemin de Bellevue, 74940 – Annecy-le-Vieux, France.}
{Sorana.Cimpan,Flavio.Oquendo}@esia.univ-savoie.fr

Abstract. Long-lived, architecture-based software systems are increasingly important. Effective process support for these systems depends upon recognising their compositional nature and the active role of their architecture in guiding evolutionary development. Current process approaches have difficulty with run-time architecture changes that are not known a priori, and dealing with extant data during system evolution. This paper describes an approach that deals with these issues. It is based on a process-aware architecture description language (ADL), with explicit compose and decompose constructs, and with a hyper-code representation for dealing with extant data and code. An example is given to illustrate the ease-of-use benefits of this approach.

1 Introduction

There is now a substantial proportion of software development that is based on assembling a set of appropriate components into the required system. This typically involves configuring generic components to the particular situation, and writing the necessary glue code to bind the components into a coherent system. This component-oriented approach clearly has impact on the software process used to develop and evolve such systems. The process needs to represent the essential compositional nature of the system. When the system is long-lived, its development process is evolutionary. It is based around the system's decomposition into components, the replacing or modifying of those components, and the recomposition of the evolved system.

A popular way of designing long-lasting compositional systems is to adopt an architecture-based approach. The system architecture describes the assembly of the system in terms of functional components, and the connectors that link them together. This architecture provides a high-level view that is used to understand the system and guide its evolutionary development. In most cases it is essential that this architecture

is not static. As the system evolves the architecture itself must evolve. These are the systems that we consider to have *active architectures*.

Software developers evolving a long-lived system have to deal with both the operational software and the process for evolving (managing) the operational software. This evolutionary process might include software for re-installing the system, or specific components. In addition, it might include utilities that migrate essential state information from the current to the evolved system.

In this paper we present an approach to dealing with this key software process problem. It is based on three features:

- Use of the software architecture to structure the evolutionary development process as well as the software itself.
- The architecture-based software process explicitly represents the composition, decomposition and re-composition that are at the heart of the evolution process.
- The use of a hyper-code representation [CCK+94,ZKM00] so that the current state or context can be effectively managed during the evolution.

The approach is illustrated through an example. This example is deliberately kept small so that it can be explained within the confines of this paper. The aim is to show the ease with which the approach deals with aspects that are often ignored in software processes. In section 2 we place our approach in the context of related work. In particular, we contrast the problem of supporting the evolution of active architectures with the more conventional project-based software processes. In section 3, we examine composition and decomposition in more detail. In section 4, we introduce the concept of hyper-code that is an essential novel feature of our approach. In section 5, we describe the architecture description language (ADL) used in the small example in section 6. Section 6 steps through an example evolution of the example active architecture system, illustrating the three features of the approach. Section 7 describes further work and Section 8 concludes.

2 Related Work

The relationship between architecture and evolution is widely acknowledged. The Unified Process describes software as having both form (architecture) and function, and stresses the relationship between architecture and evolution. “It is this form, the architecture, that must be designed so as to allow the system to evolve.”[JBR99] From a software maintenance background, [RBG00] describes the importance of a flexible architecture in enabling software systems to evolve and continue to meet the changing requirements of their users.

The use of an explicit run-time representation of a system’s architecture to aid evolution of the system at run-time is described in [OT98]. Archstudio [OMT98] 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.

While the relationship between architecture and evolution is recognised, the full potential of combining software process modelling techniques, to explicitly represent system evolution, and the system architecture, to structure that process representation, has not been fully realised. Traditionally, the process modelling research community has adopted a simplistic view of the aims driving business, or software, process modelling [DF94, FH93, FKN94]:

1. Model a process using a textual or diagrammatic representation. (This model is often called a process model definition.)
2. Translate (or compile) the model into a process support system (PSS).
3. Instantiate the library model to give a process model enactment. (This is often called a process model instance.)
4. The PSS runs the model enactment to support the process performance of a specific process instance.

This is based on the context of a software development organisation that undertakes a number of distinct software projects over a period of time. They create a process model enactment for each project. The assumption is that the textual or diagrammatic model represents current best practice, and one of the main benefits of modelling is to disseminate this to all projects. There may be some additional customisation of the model to a specific project when it is instantiated, but the assumption is that the general form of the model is project independent. The focus on process model evolution is on the textual or diagrammatic representation so that future projects can learn from the experience of earlier ones.

The simplistic view of process modelling is closely aligned with a corresponding view of the core of software engineering:

1. Create a program using a textual or diagrammatic representation. (The program source code)
2. Compile the program in a specific run time environment. (This creates the executable program or object code.)
3. Start (or deploy) the executable program. (This creates a process, an instance of the executable program, in the run time environment.)
4. The run time environment runs the process to have the desired effect.

In this view the assumption is that the program is going to be run many times. The emphasis for evolution of the program is at the representation (source code) level. This means that only future program runs will benefit from improvements as the program evolves. The development process is often supported by tools, such as source code control systems, that help maintain control over the evolution of the representation.

One feature that both the simplistic views mentioned above share is the one-off, one-way translation from representation into values. In [GBK+01] we discussed how this is a particular special case of a more general relationship, and that an ongoing, two-way translations between representations and values are needed for evolutionary processes.

Clearly the above software engineering view is not appropriate for long-lived systems that can not be rebuilt from scratch when a change is required. There are two issues: scale, and extant data and code. The simplistic view above is a monolithic

approach making it inappropriate for large problems, and provides no help for the re-use of existing components. The notion of composing a system out of smaller parts, which may themselves be composed from smaller parts and so on, is an essential tool in the management of complexity.

However an architecture-based approach addresses the issue of scale. The architecture provides a structure. This allows the representation (source code) of different parts (components) to be developed independently, often by separate teams. Some parts can be re-used source, or object code, so long as we have a representation of its interaction with the rest of the system. The architecture structure is also used to provide a build structure that manages the dependencies between parts, and is a key influence on deployment. A common deployment approach is to have several parts, or sub-systems, that can be started independently, so that individual sub-systems can be evolved without requiring changes to the other sub-systems.

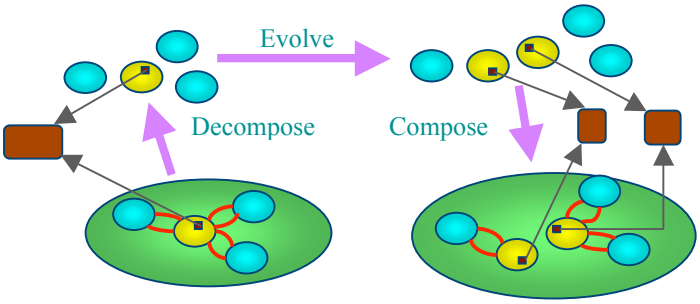
An architecture-based approach does not address the issue of extant data and code. The core issue here is that the current running version of a sub-system, which requires changing, may have valuable data that must be passed on to its evolved version, as well as running code that must be evolved. There is typically no way of referring to current existing data and code values in a source code representation. A typical work around is for the current running version to have a mechanism for writing out its data to a file or database, and the new evolved version includes initialisation code that reads this saved data. This requires some a priori knowledge so that the required data can be saved at the right time, and can be complex for data structures that can include shared or circular references [AM95].

The problems of extant data and code are typically tackled by ad-hoc solutions that are specific to a system. This is another example of how the one-way translation from representations to values places artificial limits on the potential of process support systems. Hyper-code technology [CCK+94, ZKM00] promotes a more general two-way “round trip” translation, which can be performed many times throughout the lifetime of the process.

3 Composition and Decomposition

An essential property of evolutionary systems is the ability to decompose a running system into its constituent components, and recompose evolved or new components to form a new system, while preserving any state or shared data.

This scenario is modelled in the diagram below. The original system can be thought of as a composition of three client components communicating with a server component. The server component refers to some data outwith the four components. This system can then be decomposed into its components with no communication. Note that the server component still maintains its link to the data.



We may then choose to evolve the components so that the clients stay the same while the server is divided into two different components. The new server components still maintain links to the data referred to by the original server. These five components can then be composed together to form a new system with one client communicating with one server and the other two clients communicating with the second server.

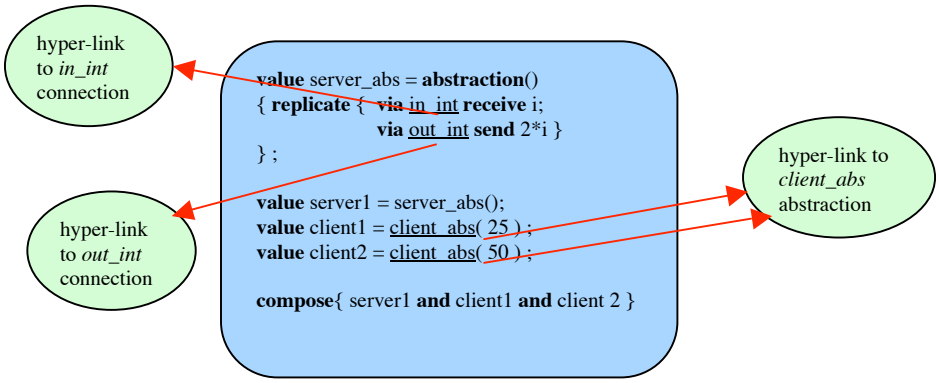
Note that we can interpret this diagram from both an architecture and a process perspective. From the architecture perspective the diagram captures the structure of the current and evolved systems, and the relationships between them in terms of which components are unchanged (the clients), modified (the server) or replaced. From a process perspective the diagram captures how to evolve from the current to the evolved system: decompose into parts, keep the clients, modify the server (splitting it into two), and recombine in the new configuration.

4 Hyper-Code

The *hyper-code* abstraction was introduced in [CCK+94] as a means of unifying the concepts of source code, executable code and data in a programming system. 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* [Zir00], 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 hyper-code system the user composes hyper-code and the system executes it. When evolving the system, for example because an error has occurred, the user only ever sees a hyper-code representation of the program, which may now be partially executed. The hyper-code source representation of the program is structured and contains text and links to extant values.

The figure below shows an example of hyper-code representation of the ArchWare ADL, which is described in section 5 below. The links embedded in it are represented by underlined tokens to allow them to be distinguished from the surrounding text. The first two links are to connection values *in_int* and *out_int* which are used by the *server_abs* abstraction to communicate with other abstractions. The program also has two links to a previously defined abstraction *client_abs*. Hyper-code models sharing by permitting a number of links to the same object. Instances of server and client abstractions are then composed to create a client-server system. Note that code objects

(*client_abs*) are denoted using exactly the same mechanism as data objects (*in_int* and *out_int*). Note also that the object names used in this description have been associated with the objects for clarity only, and are not part of the semantics of the hyper-code.



The ability of hyper-code to capture closures allows us to represent parts of a system after decomposition without losing their context. It provides representations which can be used for both evolving the components and recomposing them into the new system.

The potential benefits of modelling and supporting evolving processes have been well recognised in the software process modelling community. Many process modelling research systems have included facilities for process evolution [BT96,FKN94,War99]. The most common mechanism for providing these facilities is through reflective programming techniques.

A significant problem has been that although such facilities make it possible to write evolving processes, they are frequently hard to write and understand. It is not the basic facility for introducing new code that causes the complication, but the problem of dealing with extant data. This can be particularly severe if the required evolution depends on the current state of the process being evolved. For example, Process*Web* [PWeb01, WKR+99] has distinct reflective facilities for dealing with code and data [GBK+01]. For code the reflective facilities can provide a copy of the current source, edit this to produce new source, and apply this new code. For data the process evolver needs to write a meta-process model which, when it is run, will use the reflective facilities to obtain the current data values, manipulate them as required, and then transfer the required values from the meta-process model to the process model being evolved (for an example see [CGO+00]). In short, while in Process*Web* it is possible to write a universal meta-process for dealing with code changes, the meta-process model for data changes is specific to the model being evolved. For data changes the process evolver has to think at the meta-meta-process level in order to create the specific meta-process required. This is a result of the fact that there is a

source representation for any code value that needs to be evolved, but there is no source representation for every data value.

Hyper-code provides a unification of source and data, and hides the underlying reflective facilities. The benefit here is not that hyper-code enables us to do anything that was not possible with the reflective facilities of Process*Web*; it is that hyper-code offers the ease-of-use that is needed for these facilities to be fully exploited.

5 A Process-aware Architecture Description Language

A software architecture can be seen as a set of typed nodes connected by relations. When describing architectures, the nodes are termed components and the relations termed connectors. These components and connectors and their compositions have specified behaviours, and are annotated with quality attributes. The ArchWare ADL takes the form of a core description language based on the concept of formal composable components and a set of operations for manipulating these components—a component algebra. The key innovation in the ArchWare ADL is the use of mobility to model systems where the topology of components and connectors is dynamic rather than static; new components and connectors can be incorporated and existing ones removed, governed by explicit policies. This focus on architectural evolution at design time and run time distinguishes the ArchWare ADL from others that only deal with static architectures or where the state space of possible changes is known *a priori*. It is this focus on architectural evolution that makes the ArchWare ADL a suitable language for expressing both the architecture of an evolving system and the process of evolving the system. It is for this reason that we refer to it as a process-aware architecture description language (ADL).

The ArchWare ADL is the simplest of a family of languages designed for software architecture modelling. It is based on the concepts of the π-calculus [Mil99], persistent programming and dynamic system composition and decomposition. Indeed, the language design draws heavily on previous work carried out by the Persistent Programming Group at St Andrews on persistent programming languages [Kir92,KRC+92,AM95,MCC+95], by the Informatics Process Group at Manchester on process modelling [GWS96,WKR+99,War99,GRW00] and by the Software Engineering Group at Annecy on formal description languages [CGO+00].

The ArchWare ADL is a strongly typed persistent language. The ADL system consists of the language and its populated persistent environment and uses the persistent store to support itself. To model component algebra, the ADL supports the concepts of behaviours, abstractions of behaviours and connections between behaviours. Communication between components, represented by behaviours, is via channels, represented by connections. The language also supports all the basic π-calculus operations as well as composition and decomposition.

6 Example: A Client-Server System

The concepts discussed earlier are illustrated in this section using an example written in the ArchWare ADL. Consider a server that disseminates data about time, date and the position of a satellite. A number of clients may request data from this server.

The functionality of the server and the client can be modelled as abstractions in the ArchWare ADL. When applied, these abstractions yield executing behaviours. Such behaviours are the components that make up the client-server system. The repetitive nature of both client and server functionalities is captured using recursion.

Components interact by communicating via connections. Each component may specify the connections it uses to communicate with others. At the time of composition, these connections may be renamed to make communication possible.

```
! client
recursive value client_abs = abstraction()
{       value c_put = free connection () ;          ! request connection
        value c_get = free connection( string ) ;   ! reply connection
        via c_put send ;                            ! send request
        via c_get receive s : string ;              ! receive reply
        via c_display send s ;                      ! display reply
        client_abs()                                ! client calls itself
};
```

In the simple client code above, a client sends a signal via connection *c_put*, then receives a reply via connection *c_get*, and then sends the reply value via connection *c_display*.

In the example server below, the connection used determines the nature of the request. For example, a request received via connection *c_put_s_get_time* will be for time. The server will choose to receive a request from one of the three connections and respond to it.

```
! Global data items to keep count of server activities
value time_count, date_count, pos_count = loc( integer ) ;

! server
recursive value server_abs = abstraction(){
        value c_put_s_get_time, c_put_s_get_date,     ! connections to receive requests
                c_put_s_get_pos = free connection() ;
        value s_put_c_get_time, s_put_c_get_date,     ! connections to send data
                s_put_c_get_pos = free connection(string) ;
        choose{                    ! server makes a choice of which request to service
             { via c_put_s_get_time receive ;          ! request for time
               via s_put_c_get_time send time ;        ! send time
               time_count := 'time_count + 1 }         ! increment time count
        or
             { via c_put_s_get_date receive ;          ! request for date
               via s_put_c_get_date send date ;        ! send date
               date_count := 'date_count + 1  }        ! increment date count
        or
             { via c_put_s_get_pos receive ;           ! request for satellite position
               via s_put_c_get_pos send satellite_position ;     ! send position
               pos_count := 'pos_count + 1     }} ;    ! increment position count
        server_abs()                                   ! server calls itself
};
```

Having defined server and client abstractions, we will now create a client-server system by composing one server and three client instances with appropriate renaming. Note that other topologies are also possible, for example two servers and five clients. Renaming ensures that corresponding client and server connections are matched for communication. Defining the composition as a value gives us a handle (*CS_system1*) to the resulting behaviour.

```
! build client-server system
value CS_system1 = {
        compose{                                        ! compose components
            client_abs()                                ! client for time
              where { CS_1_time renames c_put,
                      CS_2_time renames c_get}
        and   client_abs()                              ! client for date
              where{  CS_1_date renames c_put,
                        CS_2_date renames c_get }
        and   client_abs()                              ! client for position
              where{  CS_1_pos renames c_put,
                        CS_2_pos renames c_get }
        and   server_abs()                              ! server
            where{  CS_1_time renames c_put_s_get_time,
                    CS_2_time renames s_put_c_get_time,
                    CS_1_date renames c_put_s_get_date,
                    CS_2_date renames s_put_c_get_date,
                    CS_1_pos renames s_put_c_get_pos,
                    CS_2_pos renames s_put_c_get_pos }
} ;
```

Once the system starts executing, we may wish to change its structure. Feedback from the system, efficiency concerns and changing requirements can contribute to such a decision. We begin this process by decomposing the system into its component parts. The *with* construct gives us handles to the components.

```
! decompose system
decompose CS_system1 with c1, c2, c3, s1 ;
```

Necessary changes can then be made by evolving or redefining some components. In this case we wish to split the functionality of the server into two by creating two new servers, one serving time alone and the other serving date and satellite position. Therefore we create two new abstractions to replace the old *server_abs*.

Using hyper-code representations of the abstractions will enable us to define the new abstractions to use the current values of the count variables without them having to be stored and explicitly reinitialised.

```
! time server
recursive value time_server_abs = abstraction()
{       value s_get_time = free connection() ;
        value s_put_time = free connection( string ) ;
        via s_get_time receive ;
        via s_put_time send time ;
        time_count := 'time_count + 1 ; ! reference to extant data
        time_server_abs()
};

! date and satellite position server
recursive value date_sat_server_abs = abstraction()
{       value s_get_date, s_get_sat_pos = free connection();
        value s_put_date, s_put_sat_pos = free connection( string ) ;
        choose   {
                {   via s_get_date receive ;
                    via s_put_date send date ;
                    date_count := 'date_count + 1 } ! reference to extant data
        or
                {   via s_get_sat_pos receive ;
                    via s_put_sat_pos send satellite_position ;
                    pos_count := 'pos_count + 1 }} ; ! reference to extant data
        date_sat_server_abs() ;
} ;
```

A new client-server system can then be formed by composing the two new servers with the decomposed clients appropriately.

```
! make new client-server system
value CS_system2 = {
        compose{
                c1 where {CS_1_time renames c_put,
                          CS_2_time renames c_get}
        and   c2 where{ CS_1_date renames c_put,
                          CS_2_date renames c_get }
        and   c3 where{ CS_1_sat_pos renames c_put,
                          CS_2_sat_pos renames c_get }
        and   time_server_abs()
            where{   CS_1_time renames c_put_s_get_time,
                     CS_2_time renames s_put_c_get_time }
        and   date_sat_server_abs()
            where{   CS_1_date renames c_put_s_get_date,
                     CS_2_date renames s_put_c_get_date,
                     CS_1_sat_pos renames c_put_s_get_pos,
                     CS_2_sat_pos renames s_put_c_get_pos}
} ;
```

Now client *c1* will communicate with *time_server* and clients *c2* and *c3* will communicate with *date_sat_server*.

7 Further Work

The example described in section 6 illustrates the core idea of using a process-aware ADL to support the evolutionary development of a system. The ADL has been developed as part of the ArchWare project, which is delivering customisable architecture-based environments for evolving systems. To build an effective environment, several additions are required to the core idea described above. The core ADL is relatively low-level and does not have the domain abstractions to provide the higher-level view required by people evolving the system. The ArchWare project is using the notion of styles to allow users to customise the ADL through the development of abstractions that are appropriate to the specific domain and system. The ArchWare project is also tackling evolution consistency, through the notion of the annotation of an architecture with properties, and providing architecture analysis tools. The essential idea is to enable users to set constraints in terms of architecture properties, and prove that the evolution of a system does not violate any of these constraints.

In providing scalable support for evolving systems, a cellular approach is promising. Each component can have its own evolutionary development process that is responsible for local changes to that component [GWS96]. This may have to make requests to the corresponding development processes of sub-components to achieve its desired change, and to signal to its super-component's process when a required change is not within its control. The ArchWare project builds upon previous work in process instance evolution [GRW00], which provides a process for evolving an engineering process through a network of meta-processes that matches the product breakdown structure (the architecture of the engineering product).

The architecture analysis approach is essentially a pre-emptive approach to managing the evolution of a system. An alternative is a healing approach, typified by autonomic systems, where the focus is on automatically recognising what is wrong and initiating the appropriate actions to resolve the situation [AM95,Aut02]. To achieve this ongoing management and tuning the ability to control decomposition, evolution and re-composition explicitly is essential. Monitor components receive meta-data arising from probes and gauges, which dynamically measure various aspects of the running system. When deemed necessary, the monitors initiate the evolution of selected components, by decomposing the component assemblies in which they are embedded (producing hyper-code), updating the decomposed hyper-code representation to use alternative components, and recomposing the result. Various policy rules for when, what and how to evolve may be designed to pursue different goals. Indeed these policies may themselves be components that are subject to evolution.

The approach that we have described is not specific to the software development process. In particular the approach can be applied to evolving the evolutionary development process itself. In an ArchWare environment a generic meta-process, such as the Process for Process Evolution P2E [War99], can be used to specialise the evolutionary development process for a component within the system, for example to use specific software development methods or tools.

Another area where we plan to evaluate this approach is in evolving *in silico* experiments in the bioinformatics domain [GPR03]. Much biological experimentation now takes place *in silico*, combining, comparing and collating biological data and analysis tools that are available on the web. An *in silico* experiment is a process, describing the various data retrieval, transformation and analysis steps that yield the required outputs. As more resources become available, and as experimental best practice changes, these experiments evolve. In addition, scientists want to be able to evolve *in silico* experiments as they are running. If an experiment takes hours, or even days, to complete then the scientists involved want the ability to examine the intermediate results, and if necessary make changes to the later stages of the experiment. The use of process technology to provide an explicit representation of *in silico* experiments provides a way of effectively sharing best current practice within a community. An architecture-based approach offers a promising way of composing and evolving larger experiments from smaller experimental fragments. The potential benefits of a hyper-code representation that can be used through the experimental lifecycle, and enable experiments to directly refer to extant data, are very similar to those in the software engineering domain. The bioinformatics domain is also of interest because the coordination of resources into experiments shares many characteristics with the electronic coordination of businesses into business networks [GWS+02].

8 Conclusion

In this paper we have identified a class of software systems: long-lived architecture-based systems. Such systems are becoming more common and supporting their evolutionary development is a current key challenge for software engineering. This is a challenge where process technology can make a contribution. The compositional nature of these systems means that the architecture plays a dual role. It structures the (operational) software system, and it structures the evolutionary development of that operational software system. However, it is important to recognise that the architecture is active; it evolves as the structure of the system evolves. Furthermore the evolutionary development of these systems involves transferring important state information, extant data, between the current and the evolved system.

The approach that we have described and illustrated addresses these issues through the novel combination of three features:

- Exploiting the software architecture to structure the evolutionary development process as well as the software itself. This configures the process to its context (the software being evolved).

- The architecture-based software process explicitly represents the composition, decomposition and re-composition that are at the heart of the evolution process. The process is expressed in a process-aware Architecture Description Language (ADL) that has explicit compose and decompose constructs.
- The use of a hyper-code representation so that the current state or context can be effectively managed during the evolution. The unification of source code and data in hyper-code eliminates different ways of managing code and data evolution, and the hyper-code representation is used throughout the lifetime of the system.

The benefits of this approach are ease of use and ease of understanding. It provides a simpler abstraction over the basic reflective programming facilities for those evolving the system. They are able to concentrate on the essential problems of describing the required evolution. The approach also directly models the compositional nature of the system through the use of compose and decompose primitives in a process-aware ADL. This gives the benefits of a run-time architecture description that is synchronised with the evolving system. Those evolving the system always have an up to date, high-level view of the system, and state of the system within its evolutionary development process.

There are other potential benefits of this approach. The compositional nature encourages the creation of libraries of re-usable software components, and software process fragments. The hyper-code representation means that such extant values in the environment can be easily referenced and re-used. However, the approach is only an essential foundational building block, and further facilities are needed to give an effective environment that supports the evolutionary development process of long-lived, architecture-based systems.

Acknowledgements

This work is supported by the EC Framework V project ArchWare (IST-2001-32360), and the UK Engineering and Physical Sciences Research Council (EPSRC) under grants GR/R51872 and GR/R67743. It builds on earlier EPSRC-funded work in compliant systems architectures (GR/M88938 & GR/M88945).

References

[AM95]. Atkinson, M.P. and Morrison, R.: Orthogonally Persistent Object Systems. VLDB Journal 4, 3 (1995) 319–401

[Aut02]. Autonomic Computing: IBM's Perspective on the State of Information Technology. IBM, http://www.research.ibm.com/autonomic/ (2002)

[BT96]. Bolcer, G.A. and Taylor, R.N.: Endeavors: A Process System Integration Infrastructure. In Proc. ICSP'4, Brighton, UK, IEEE Comp. Soc. Press, (1996) 76–85

[CGO+00]. Chaudet, C., Greenwood, R.M., Oquendo, F. and Warboys, B.C.: Architecture-driven software engineering: specifying, generating, and evolving component-based software systems. IEE Proc.–Software 147, 6 (2000) 203–214

[CCK+94]. Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Moore, V.S. and Morrison, R.: Unifying Interaction with Persistent Data and Program. In: Sawyer, P. (ed): Interfaces to Database Systems. Springer-Verlag, In Series: Workshops in Computing, van Rijsbergen, C.J. (series ed) (1994) 197–212

[DF94]. Dowson, M., and Fernström B.C.: Towards Requirements for Enactment Mechanisms. In: Proceedings of the Third European Workshop on Software Process Technology, LNCS 775, Springer-Verlag, (1994) 90–106

[FH93]. Feiler, P.H., and Humphrey, W.S.: Software Process Development and Enactment: Concepts and Definitions. In: Proceedings of the 2nd International Conference on Software Process, Berlin, (1993) 28–40

[FKN94]. Finkelstein, A., Kramer, J., and Nuseibeh, B. (eds): Software Process Modelling and Technology. Research Studies Press, (1944)

[GPR03]. Goble, C., Pettifer, S., and Stevens, R.: myGrid: in silico experiments in bioinformatics. In Berman, F., Hey, A.J.G., and Fox, G. (Eds) : Grid Computing: Making the Global Infrastructure a Reality, Wiley, (2003) In press.

[GWS96]. Greenwood, R.M., Warboys, B.C., and Sa, J.: Co-operating Evolving Components – a Formal Approach to Evolve Large Software Systems. In: Proceedings of the 18th International Conference on Software Engineering, Berlin, (1996) 428–437

[GRW00]. Greenwood, M., Robertson, I. and Warboys, B.: A Support Framework for Dynamic Organisations. In the Proceedings of the 7th European Workshop on Software Process Technologies, LNCS 1780, Springer-Verlag, (2000) 6–21

[GBK+01]. Greenwood, R. M., Balasubramaniam, D., Kirby, G.N.C., Mayes, K., Morrison, R., Seet, W., Warboys, B.C., and Zirintsis, E.: Reflection and Reification in Process System Evolution: Experience and Opportunity. In the Proceedings of the 8th European Workshop on Software Process Technologies, LNCS 2077, Springer-Verlag, (2001) 27–38

[GWS+02]. Greenwood, M., Wroe, C., Stevens, R., Goble, C., and Addis, M.: Are bioinformaticians doing e-Business? In Matthews, B., Hopgood, B., and Wilson, M. (Eds) In "The Web and the GRID: from e-science to e-business", proceedings of Euroweb 2002, Oxford, UK, Dec (2002), Electronic Workshops in Computer Science, British Computer Society http://www.bcs.org/ewic

[JBR99]. Jacobson, I., Booch, G., and Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, 1999.

[KRC+92]. Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. and Morrison, R.: Persistent Hyper-Programs. In Albano, A. and Morrison, R. (eds): Persistent Object Systems. Springer-Verlag, In Series: Workshops in Computing, van Rijsbergen, C.J. (series ed) (1992) 86–106.

[Kir92]. Kirby, G.N.C.: Persistent Programming with Strongly Typed Linguistic Reflection. In: Proceedings 25th International Conference on Systems Sciences, Hawaii (1992) 820–831

[MCC+95]. Morrison, R., Connor, R.C.H., Cutts, Q.I., Dustan, V.S., Kirby, G.N.C.: Exploiting Persistent Linkage in Software Engineering Environments. Computer Journal, 38, 1 (1995) 1–16

[Mil99]. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge University Press (1999)

[OMT98]. Oreizy, P., Medvidovic, N. and Taylor, R.N.: Architecture-Based Runtime Software Evolution. Proc. ICSE'20, Kyoto, Japan, IEEE Computer Society Press (1998) 177–186

[OT98]. Oreizy, P. and Taylor, R.N.: On the role of software architectures in runtime system reconfiguration. In Proc. of the International Conference on Configurable Distributed Systems (ICCDS 4), Annapolis, MD. (1998)

[PWeb01]. Process*Web*: service and documentation http://processweb.cs.man.ac.uk/ (accessed on 10 Apr 2003)

[RBG00]. Rank, S., Bennett, K., and Glover, S.: FLEXX: Designing Software for Change through Evolvable Architectures. In Henderson, P. (Ed), Systems Engineering for Business Process Change: collected papers from the ERSRC research programme, Springer (2000) 38–50

[STO95]. Sutton, Jr., S.M., Tarr, P.L., and Osterweil, L.: An Analysis of Process Languages. CMPSCI Technical Report 95-78, University of Massachusetts, (1995)

[WKR+99]. Warboys B.C., Kawalek P., Robertson T., and Greenwood R.M.: Business Information Systems: a Process Approach. McGraw-Hill, Information Systems Series, (1999)

[War99]. Warboys, B. (ed.): Meta-Process. In Derniame, J.-C., Kaba, B.A., and Wastell, D. (eds.): Software Process: Principles, Methodology, and Technology, LNCS 1500, Springer-Verlag (1999) 53–93

[ZKM00]. Zirintsis, E., Kirby, G.N.C., and Morrison, R.: Hyper-Code Revisited: Unifying Program Source, Executable and Data. In Proc. 9th International Workshop on Persistent Object Systems, Lillehammer, Norway, (2000)

[Zir00]. Zirintsis, E.: Towards Simplification of the Software Development Process: The Hyper-Code Abstraction. Ph.D. Thesis, University of St Andrews, (2000)