Unifying Static and Dynamic Approaches to Evolution through the Compliant Systems Architecture

Katrina Falkner, Henry Detmold, Diana Howard, David S. Munro
School of Computer Science
The University of Adelaide
S.A. 5005, Australia
{katrina, henry, diana, dave}@cs.adelaide.edu.au

Ron Morrison and Stuart Norcross
School of Computer Science
University of St Andrews
North Haugh, St. Andrews
Fife KY16 9SS, Scotland
{ron, stuart}@dcs.st-and.ac.uk

Abstract

Support for evolution can be classified as static or dynamic. Static evolvability is principally concerned with structuring systems as separated abstractions. Dynamic evolvability is concerned with the means by which change is effected. Dynamic evolution provides the requisite flexibility for application evolution, however, the dynamic approach is not scalable in the absence of static measures to achieve separation of abstractions. This separation comes at a price in that issues of concern become trapped within static abstraction boundaries, thereby inhibiting dynamic evolution.

The need for a unified approach has long been recognised but existing systems that attempt to address this need do so in an ad-hoc manner. The principal reason for this is that these approaches fail to resolve the incongruence in the underlying models. Our contention is that this disparity is incidental rather than fundamental to the problem. To this end we propose an alternative model based on the Compliant Systems Architecture (CSA), a structuring methodology for constructing software systems.

The overriding benefit of this work is increased flexibility. Specifically our contribution is an instantiation of the CSA that supports unified static and dynamic evolution techniques. Our model is explored through a worked example in which we evolve an application's concurrency model.

1 Introduction

Support for evolution can be classified as static or dynamic. Static evolvability is principally concerned with structuring systems as separated abstractions, and then evolving these abstractions offline. By identifying all dependencies, and isolating those abstractions that are independent, static evolvability enables individual application requirements to be changed in isolation. Dynamic evolvability is concerned with the means by which online change is effected. Techniques such as type-safe linguistic reflection are used to perform run-time modifications to code, dynamically changing the way in which the code executes in response to system changes. In this paper, we focus on the use of dynamic evolution techniques under application control. Specifically, we consider the use of dynamic evolution to fulfil the changing needs of non-functional requirements.

A standard practice in software development is to partition systems into modules implementing separate abstractions [29]. Dynamic or static evolution can then be applied to individual abstractions, reducing the potential for programmer error and decreasing application complexity. This separation comes at a price, however, in that issues of concern become trapped within static abstraction boundaries.

A complete model of evolution must support:

- Changes to the implementation of an abstraction.
- The reallocation of responsibilities for an abstraction.
- The merging of abstractions through the construction

Function	Static	Dynamic
Change Implementation	Yes	Yes
Merge Abstractions	Yes	Yes
Reallocate Responsibilities	**Yes**	**No**
Continuous Operation	**No**	**Yes**

Table 1. Support for evolution as characterised by static and dynamic evolution techniques.

of a higher-level abstraction. This is an important special case of the previous point.

- Continuous application operation throughout the evolution process.

Static and dynamic evolution techniques each support only a subset of these requirements, as depicted in Table 1. The conflict between support for continuous operation and reallocation of responsibilities between abstractions represents an incongruence between the models of static and dynamic evolution. Run-time information used by dynamic evolution techniques can not in extant models be used to impact dynamic reallocation of responsibilities across abstraction boundaries. Correspondingly, reallocation of responsibilities can not be easily performed dynamically in response to changing application requirements.

An additional consideration is whether the changes are anticipated or unanticipated. Anticipated dynamic evolution requirements may be partially accommodated through the definition of appropriate abstractions; however, unanticipated changes, the majority of cases [28], can not.

Applications are commonly divided into their functional requirements (fundamental requirements of the application) and their non-functional requirements (requirements designed to support the application such as concurrency or distribution) [12] [19]. For example, an application might perform sorting of data provided by multiple clients (threads). The functional requirements of this application are the provision of a sorting algorithm and an interface for client access. That is, an abstraction is made available to the client. The non-functional requirements include supporting consistency in the face of concurrent access through synchronising access to the functional interface. Systems that support static evolution encourage the consideration of non-functional requirements as separated abstractions. In this example, concurrency abstractions are partitioned from functional requirements.

Non-functional requirements that are identified as separated abstractions are constrained by the same conflict between dynamic and static evolution models as functional requirements. In this paper, we propose a model for applying both static and dynamic evolution techniques to non-functional requirements that supports both continuous application operation and the reallocation of responsibilities between abstractions.

Further, non-functional requirements are divided into those requirements that are identified as separated abstractions and those that are tangled within existing abstractions. The inability to reallocate abstraction responsibilities dynamically constrains this latter group from recognition as a separate abstraction. Moreover, the selection of the point of division is frequently system-dependent rather than application-dependent.

The need for a unified approach that combines static and dynamic evolution techniques has long been recognised [7] [9] [33] but existing systems that attempt the unification do so in an ad-hoc manner. The principal reason for this is that these approaches fail to resolve the incongruence in their underlying models.

We propose an alternative model based on the Compliant Systems Architecture (CSA) [23] [24], a structuring methodology for constructing software systems. Architectural compliance allows the support architecture to be varied dynamically to meet the changing demands of the application. Supporting compliance at the language level facilitates the development of applications that are capable of exploiting and adapting the run-time system to support their needs.

The fundamental result of Reynolds [31] tells us that any layered software architecture necessarily has a fixed point. Hence, it is no surprise that the CSA cannot deliver completely evolvable systems. Any instantiation of the CSA statically defines a fixed point in the operational abstractions which accordingly defines a limit on the dynamic evolution that can be applied.

The property that distinguishes the CSA from other dynamic evolution systems is a separation of *policy* (the strategy defining an objective) and *mechanism* (the implementation of an objective). This separation of policy and mechanism, well established as an important property of system design [5] [35], enables the development of new policy from run-time information, independently of the mechanisms used. Mechanism and policy typically exist within a spectrum (see Figure 1) of design choices. Support systems generally select a fixed point further along the spectrum, integrating core mechanism with part or all of the available policy. *Compliant policy* is defined as policy that has been completely removed from mechanism and is, hence, unconstrained by the assumptions and implementation of mechanism.

A separation of mechanism and policy decomposes applications into groups defined by policy interdependence. Hence, where a subject-based categorisation may identify two non-functional requirements that must be defined as separate abstractions, a categorisation based on mechanism

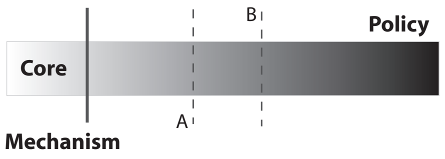

Figure 1. The spectrum of mechanism and policy.

and policy particular to an application may identify one combined non-functional requirement, removing static barriers to specifying requirement interdependencies. Further, the strict methodology of separating mechanism and policy across *all* system components (removing the division within non-functional requirements) delivers support for unified evolution of all application requirements.

The overriding benefit of our model is increased flexibility. The identification of which requirements to separate and the extent to which these abstractions are interdependent is dynamically under application-control. Specifically, we introduce an instantiation of the CSA that unifies support for static and dynamic evolution.

We have constructed a compliant architecture [6] through which we explore evolution of policy. This architecture supports experimentation into compliant policies for distributed systems, such as memory management [25] and cache coherency [10]. Here, the efficacy of the compliant evolution model is demonstrated through worked examples, detailing policy specification for concurrency and distribution models.

2 Developing a Compliant Evolution Model

Static evolution involves the separation of code defining functional requirements from code defining non-functional requirements, thereby enabling independent evolution of both requirement groups [19]. Early static evolution systems enforce a strict separation of application requirements, and introduce distinct programming languages or modules for defining specific concerns. Subsequent to their independent specification, each non-functional requirement is unified with the functional requirements via a language-specific integration process.

More recent static evolution systems relax the separation by annotating application requirements with information that enables a generic unification process. Although unification may be invoked online, thereby supporting dynamic evolution of each separated requirement, the identification of which requirements may be separated and the integration process remain statically defined.

The information barrier present in existing unified static and dynamic evolution systems constrains evolution excepting the case where requirements have been defined independently. A well-recognised problem is how to perform effective run-time evolution where information is required from multiple separated modules [33]. The support for scalable evolution via separation of concerns is in conflict with the need to evolve as required by the application.

2.1 Support for Static Evolution within the CSA

The CSA acknowledges that a strict separation of requirements is not always beneficial to the application or its dynamic evolution. By allowing the form of separation to be evolved in a uniform and unrestrictive manner, the application can flexibly define its evolution needs.

A formal treatment of the CSA is presented in [24]. The CSA defines systems in terms of operational abstractions. Mechanism and policy at any level can be replaced dynamically with the combination of mechanism and policy that better suits the application. Within this view, operational abstractions are themselves evolvable, enabling the construction of new architectures.

System services (non-functional requirements) are defined via layers of mechanism and policy; at any one layer, x, the combination of mechanism and policy at any lower layer, $y : y < x$, is viewed by x as composed mechanism of y. Generic compliance enables mechanism to be placed (and evolved) at the layer at which it is of most benefit. A compliant system supports bi-directional information sharing between system layers through the use of *upcalls* and *downcalls*. Downcalls allow policy to dictate the operation of lower layers. Upcalls trigger the execution of policy code written at a higher level, enabling lower levels to influence policy choices. An instantiation of the CSA is defined by the selection of operational abstractions, where one of these abstractions, typically the lowest architectural layer, represents the necessary fixed point.

2.2 Application-controlled Separation of Requirements

All application requirements (be they functional or non-functional) are modelled through a separation of mechanism and policy, with interactions between these captured using upcalls and downcalls. In a compliant system, applications are free to choose whether they use the abstractions provided or interact with lower level mechanism directly through the instantiation of new policy.

The CSA layers an architectural model over a system (which need not be a layered architecture) based on the ordering of mechanism and policy composition. These layers and the contents of the each layer are dynamically evolvable. Flattening of the architecture, as performed by the uni-

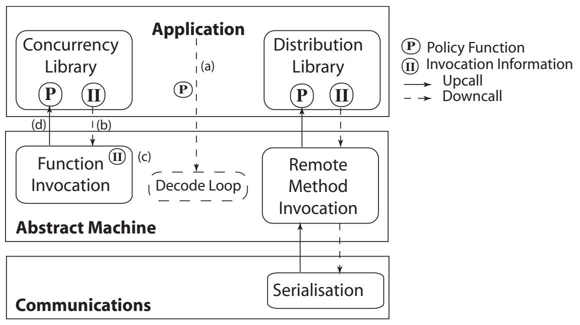

Figure 2. Static evolution in the CSA supported by upcalls and downcalls.

fication process in static evolution systems, limits the ability to evolve the structure. In contrast, the CSA allows the application to specify and modify its interactions dynamically, with no pre-determination of required interactions. Unification is therefore an ongoing and dynamic process.

Figure 2 depicts a CSA instantiation designed to support the integration of separated requirements. In this model, the CSA supports dynamic evolution of mechanism and policy between a system whose operational abstractions, for simplicity of explanation, consist of an application layer, an abstract machine layer and a communications layer. The abstract machine groups its instructions into core instructions, which represent the fixed point of the system, and compliant instructions (such as function invocation), which are evolvable and consist of layered mechanism and policy.

Requirements that the application wishes to separate are defined in distinct modules; code libraries that may be reused by later applications. These modules define the policy that represents the requirements and data sets (such as invocation information) that are required by lower levels to indicate when the module should be applied during application execution. The application, via a downcall, registers both a module's policy and its data sets with the abstract machine. Applications (dynamically or statically) construct their libraries with the level of interdependence that they require for evolution. The definition of the libraries, compliant instructions and their composition can be a dynamic process with the degree of structuring dictated by the application, rather than the underlying system. This defines a tradeoff rather than a strict separation: with greater interdependence the process of evolving each module using a run-time evolution mechanism such as reflection [17] increases in complexity. The benefit of exposing this tradeoff to the application is increased application control.

The process of unification of support for static evolution and dynamic evolution measures throughout the execution of the application is as follows:

(a) The application registers policy handlers for all requirements for which it has defined separated libraries. In this example, the application has defined requirements affecting concurrency and distribution.

(b) Each module provides, via a downcall, any *invocation information* that is required by the lower level. In this case, the concurrency module is used to specify synchronisation constraints between two functions defined by the application. The policy that defines the synchronisation must be invoked (via an upcall) before either of the two functions is invoked. The invocation information passed to the abstract machine in this example consists of a list containing references to the functions in question.

(c) The abstract machine determines whether the function to be invoked requires synchronisation by examining the invocation information provided by the application.

(d) If synchronisation is required (requiring policy direction) an upcall is performed to the concurrency module, causing the execution of the concurrency policy code registered in step (a). In this example, the synchronisation constraints are examined by the policy handler function and it is then determined whether the invocation of the synchronised function may proceed.

It is important to note that this process is not necessarily sequential. An application chooses when it registers an upcall handler, and may change and evolve these handlers or data sets at any time.

3 An Instantiation of the CSA

ProcessBase [22] is a compliant language that utilises the modern practice of an abstract machine model. ProcessBase has the properties of strong typing, first class procedures, hyper-programming [17] and persistence [4].

Figure 3 illustrates the ProcessBase architecture: an instantiation of a distributed CSA that consists of four distinct architectural layers.

The application layer views the remaining layers as a unified system through a single logical address space (with transparency of the address space controlled by policy). The second operational abstraction comprises a set of abstract machines, one per node in the system. Each abstract machine is further decomposed into a run-time system and a local cache. The communications layer facilitates communication between the abstract machines. A collection of object stores constitutes the fourth layer and provides orthogonally persistent storage. In this instantiation, upcalls are defined as a form of interrupt and downcalls as the direct execution of an abstract machine instruction.

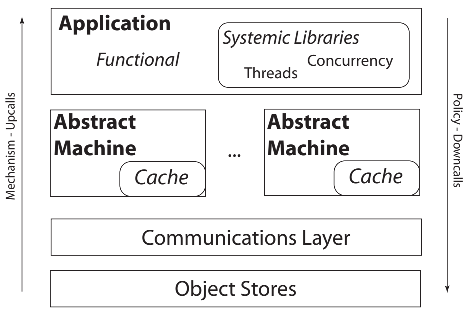

Figure 3. The Distributed ProcessBase architecture.

3.1 The Constraints of our Instantiation

In order to explore compliance in a practical setting, we have chosen a pragmatic limit on the available mechanisms. Here, the definition of mechanism below the application layer is static; the addition of compliant instructions requires the identification of any further mechanism and the static integration of this mechanism into the appropriate layer. This constraint affects the identification and placement of mechanism and policy, reducing the potential benefit of the operational abstractions. All policy that requires run-time evolution must be identified as such and placed at the application level.

Policy which represents separated requirements are defined through application-level ProcessBase code. Applications can access commonly used policy through the use of ProcessBase libraries. In addition, applications may choose to define their own libraries or define functions for specific requirements within individual applications.

Within the limitations of a static abstract machine layer, instructions that require upcalls or integration of invocation information must be statically identified and integrated with the abstract machine before application execution (representing the fixed point). This defines a static interface between the application and the abstract machine, and hence places a strict limit on evolution. An implementation of a separated requirement consists of the following steps:

- Identification of compliant mechanism to be extended with policy.
- Identification of the points of policy invocation.
- Identification of the information interface.
- Identification of policy for the requirement.
- Evolution of policy and data sets.

The work represents a proof of concept of the unification of support for static evolution and dynamic evolution principles. We are designing an alternative language system where compliance is extended to the abstract machine and communications layers. This alternative system follows an abstract machine design (based on the Jikes Research Virtual machine [1]) and hence operational abstractions identified in this work are applicable to this future research. Our research is strongly influenced by similar research into the Arena library-based operating system [21], a nano-kernel which is layered underneath the ProcessBase run-time system.

4 Policy Evolution in the CSA

The principal motivations behind concurrent and distributed systems design are the promises of resource sharing, computational speed-up and increased reliability. However, delivering these benefits typically involves a trade-off between application programming complexity and underlying system complexity. It is because of this tradeoff that concurrency and distribution are commonly selected as areas in which to explore evolution.

4.1 Concurrency

Concurrency requirements are often chosen as an exemplar of non-functional requirements [3] [18] [19] [20]. Their separation allows object and function synchronisation to be defined independently of an application's objects and functions. However, it is also common for synchronisation constraints to be dependent on data expressed within a functional requirement. It is therefore often impossible to complete the separation and maintain application functionality.

Figure 4 shows a non-separated implementation of the bounded buffer problem in ProcessBase, using semaphores to implement the synchronisation constraints[1]. In this example, these constraints apply to the `put` and `get` methods and are defined through transitions between states where specific methods are enabled. The transition definitions interact with the application methods, in that data must be shared between them - namely the size of the buffer. This sharing leads to the transitions being interspersed throughout the methods.

In contrast, Figure 5 shows a separated implementation of the bounded buffer problem in ProcessBase, where a concurrency library with policy at the abstract machine level is used to implement the sychronisation constraints. Similar models can be found throughout the literature [20] [18].

Synchronisation conditions for the `get` and `put` functions are provided through their respective synchronisation functions: `syncGetFun` and `syncPutFun`. Calls to a

[1]For simplicity of explanation, the implementation of the buffer has been elided

```
type boundedBuffer is
    view[put: fun(int); get: fun()->int]
let createBoundedBuffer <-
    fun() -> boundedBuffer;
begin
   let slots <- loc(buffer.length)
   let count <- loc(0)
   let empty <- new semapore(buffer.length)
   let full <- new semaphore(0)

   let putFun <- fun(x : int);
   begin
      empty.wait()
      count := 'count + 1;
      full.signal()
   end

   let getFun <- fun() -> int;
   begin
       full.wait()
       slots := 'slots + 1;
       empty.signal()
   end
   view(put <- putFun, get <- getFun)
end
```

Figure 4. The Bounded Buffer problem.

```
type boundedBuffer is
   view[put: fun(int); get: fun()->int]
let createBoundedBuffer <-
   fun() -> boundedBuffer;
begin
    let count <- loc(0)
    let putFun <- fun(x : int);
       count := 'count + 1;
    let getFun <- fun() -> int;
       count := 'count - 1;

    let syncGetFun <- fun () -> bool;
       if 'count > 0 then true;
       else false
    let syncPutFun <- fun () -> bool;
       if 'count < buffer.size then true;
       else false

    // synchronisation library
    syncLib.activate()
    syncLib.add(syncGetFun,any(getFun))
    syncLib.add(syncPutFun,any(putFun))

    view(put <- putFun, get <- getFun)
end
```

Figure 5. The compliant Bounded Buffer problem.

synchronised function cause the invocation of the respective synchronisation function, which is able to determine by inspecting application data whether the invocation may proceed.

The mutual exclusion of the synchronisation functions is guaranteed by the underlying abstract machine. In this model, abstract machine level semaphores are used to ensure exclusive access; different implementations are possible, for example application-level semaphores or monitors could be used to perform the synchronisation. This implementation also abstracts over the use of downcalls and upcalls, which are concealed in the synchronisation library, `syncLib`.

The `syncLib` module is responsible for managing the data required by the abstract machine and for performing any downcalls needed to register policy handlers or invocation information. In this case the synchronisation library creates a handler function which is able to invoke the synchronisation functions created by the application. The application programmer could choose to install their own handler if they require more complex evaluation. The library function `activate` invokes downcalls to install the handler and the invocation information consisting of tuples of synchronised functions and their synchronisation functions.

4.2 Distribution

Distribution transparency reduces application complexity but precludes the application from controlling its execution in a distributed context and, therefore, its performance. We are using our CSA model to explore systems where transparency of distribution is also compliant: the application is able to define what aspects of distribution it wants to control, and can ignore all others. Two areas in which we are examining compliance and policy evolution are remote method accessibility and migration.

4.2.1 Accessibility

Many distribution systems define remote interfaces for modules or objects; these interfaces define the signatures of those functions or methods that are available for remote access, defining the parameter transfer modes and, dependent on the system, the call mode. Typically, the underlying distribution support system will make decisions regarding serialisation models, synchronisation of remote call invocation, object transfer, communications protocol and any migration opportunities.

In our compliant system, we are interested in exposing the following areas to compliant application control:

- Defining multiple remote interfaces that may expose different functions to remote access and with different policies regarding the call mode, communications protocol or migration policy.
- Selecting which interface is provided on a per-client basis.
- Evolving their set of interfaces, and the interface elements.

4.2.2 Migration

A fundamental question is *what is permitted to migrate?* It is possible, and often desirable, to let applications determine the answer to this question [13]. The Emerald [14] approach supports a language-level concept that *fixes* objects to particular sites. There are primitives in the language to flag that a particular object is fixed, or unfixed, but that marks the end of application control. The current CSA instantiation provides a more rigorous method of controlling whether application-level entities are permitted to migrate. The CSA framework's support for evolution extends the basic concept in three ways. Applications can evolve:

- The set of fixed objects (simple application of evolution) but also:
- How the notion of being fixed is determined, and
- What being fixed means.

The notion of restricting migration to a (dynamically determined) subset of objects means that the implementation of a migration request must determine if the given object is allowed to relocate. That is, migration involves an operation (*isfixed*) to test whether the object in question is fixed. This property could be based on the type or mutability of the object, assigned by the application, or determined by the object or application's current access capability. The *isfixed* operation is implemented as a handler; any policy code may be registered to this handler.

When the set of fixed objects is determined by application-level assignment, the current set is easily maintained in an application-level data structure. This is an example of where rigid separation of concerns is not always appropriate. For instance, an integral part of a mobile agent's functionality may be to move between several sites in a distributed system and then become a permanent resident at one, perhaps based on some run-time information. The ability to place such a directive in the agent implementation is a more accurate model of the problem domain than trying to implement location or migration information in a separate component.

An application can dynamically evolve this model into one based on type. Then, instead of searching through a list of fixed objects, the migrate operation inspects an object's type, comparing it against the set of permitted migration types. This model entails a greater degree of migration transparency in that applications no longer explicitly identify which objects are permitted to move. The evolution process consists of defining the type-comparison code and installing it with the *isfixed* handler.

The set of relocatable types may be defined entirely within this policy code, or it may be augmented by the application via a similar fix/unfix notation. This illustrates the potential for different levels of policy weaving. At one level, there is the registration of policy code for the *isfixed* operation. At a higher level, the application may continue to participate in the definition of a particular policy, extending the weave process. Application specification of fixed objects or relocatable types are examples of this continued policy involvement.

When the *isfixed* policy determines that an object is tied to a particular site, the migration concern must interpret this decision. The migration abstraction's response represents another avenue for evolution - and one that may involve interaction with other distributed concerns. It may be appropriate to simply abort the migration, but another response is to replicate the item instead. This approach requires cooperation between migration, replication and coherency policies. Applications may also determine their own, individual, response, such as moving code to data when the latter may not be transferred.

5 Related Work

Static support for evolution is traditionally viewed as a software engineering technique [19] [33]. Aspect-oriented programming [16] introduces the separation of concerns as a technique designed to encompass both static and dynamic support for evolution. Aspect-oriented programming systems use a weaving process to perform static unification of the functional and non-functional requirements. AspectJ [15] defines single aspects that are not composable. Hyper/J [26] [27] defines an extended model of aspect-oriented programming where multiple aspects may be defined and integrated according to a static application controlled composition definition.

In addition to aspect-oriented programming, there exist several similar models that define separation of application requirements according to some subject-based categorisation [11] [3] [18].

Duclos *et al* [9] recognise the need for the dynamic definition of new aspects that can be integrated with the functional requirements of the application. The authors define an *aspect definition* language which can be used to define independent aspects that are introduced into the system. In order to extend this system to aspects with interdependencies, the authors recognise the need to introduce a constraint definition language, used to broach the barrier between the functional requirements and each non-functional requirement. This model has a similar goal to our work, but we believe it is made artificially complex by the adoption of separate structural models for the development of functional requirements, non-functional requirements and constraint identification. Wand *et al* [34] propose a model of aspect-oriented programming where weaving is applied dynamically at points of execution. However, Wand *et al* focus on the denotational semantics and correctness of the weaving process itself, rather than the potential for dynamic evolution.

Recent research into dynamic aspect-oriented systems has shown that existing models are monolithic and exhibit performance problems [30]. In addition, dynamic weaving may be insecure in that some weave points may be updated with evolved implementations while others are not. Popovici *et al* [30] introduce a novel model of dynamic aspect-oriented programming based on just-in-time compilation that addresses the performance issue. The CSA supports a broader model of support for evolution than that supported by dynamic aspect-oriented programming systems in that it encompasses the identification of aspects, the integration or weaving process, and provides evolution capabilities limited only to the fixed points within the architectural instance.

Adaptive systems [2] [8] support a form of dynamic evolution, but are inherently constrained by their nature to respond to past application behaviour. In order for an application to obtain the most benefit out of any evolution process, it must be an active as well as a reactive process.

Computational reflection [20] and metaobject protocols [7] represent unified evolution systems. These systems provide a reflective architecture and meta-languages that provide alternate views of the underlying system. It is through these meta-languages that applications are able to dynamically evolve their requirements. Metaobject protocol systems are also typically constrained by the static definition of abstraction barriers.

Sullivan [32] describes a model of aspect-oriented programming controlled by metaobject protocols (MOPs). In this model, MOPs control how the weaving process of aspects is performed, introducing the potential for dynamic control of this process through mutable MOPs. This work has similar aims to that described in this paper, but is still in its initial phase of development.

6 Conclusions and Future Work

Support for static evolution provides a separation of abstractions which, when combined with support for dynamic evolution, produces a scalable and flexible evolution system with reduced potential of programmer error. Existing approaches to the unification of support for static and dynamic evolution introduce static abstraction barriers which artificially constrain the resulting evolution system.

We introduce a model based on the Compliant Systems Architecture. Our model provides the generic framework needed to support scalable dynamic evolution without any abstraction barriers. The CSA can describe a separation of policy and mechanism between components of any system and can be used to drive evolution in a controlled manner. The fixed point of evolution depends on the instance of the CSA and once that is defined the amount of dynamic evolution can be ascertained and implemented in a structured manner.

This work represents a proof of concept of the unification of support for static and dynamic evolution principles. We are designing a more complex system where compliance is extended beyond the application level to the abstract machine and communications layers. In this extended system we plan to pursue research into evolution of mechanism and policy at multiple levels, specifically evolution of the operational abstractions within an instantiation of the CSA, and evolution of the interfaces between separated requirements.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] C. Amza, A. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive Protocols for Distributed Shared Memory. In *Proc. of the IEEE*, volume 87, pages 467–475, March 1999.

[3] C. Atkinson, S. Goldsack, A. D. Maio, and R. Bayan. Object oriented Concurrency and Distribution in DRAGOON. *Journal of Object-Oriented Programming (JOOP)*, 4(1):11–19, March 1991.

[4] M. Atkinson, P. Bailey, K. Chisholm, W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, 1983.

[5] D. Black. Scheduling support for concurrency and parallelism in the Mach OS. *IEEE Computer*, 23(5):35–43, Apr. 1990.

[6] W. Brodie-Tyrrell, H. Detmold, K. Falkner, M. Lowry, R. Morrison, D. Munro, S. Norcross, T. Olds, Z. Tian, and F. Vaughan. Design of the Distributed ProcessBase Architecture. Technical Report TR2001-01, Department of Computer Science, University of Adelaide, February 2001.

[7] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[8] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Trans. on Computer Systems*, 13(3):205–243, August 1995.

[9] F. Duclos, J. Estublier, and P. Morat. Describing and Using Non Functional Aspects in Component Based Applications. In *Proceedings of AOSD 2002*, pages 65–75, 2002.

[10] K. Falkner, H. Detmold, D. Munro, and T. Olds. Towards Compliant Distributed Shared Memory. In *Proceedings of the Second International Conference on Cluster Computing and the Grid, (WSDSM'02)*, pages 305–310, May 2002.

[11] W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of OOPSLA'93*, pages 411–428, 1993.

[12] W. H*ü*rsch and C. Lopes. Separation of Concerns. Technical report, College of Computer Science, Northeastern University, February 1995.

[13] D. Howard. Towards Flexible Distribution Support in a Database Programming Language. In *Proceedings of the Twentieth British National Conference on Databases (BNCOD20), PhD Forum, to appear*, July 2003.

[14] N. C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, University of Washington, January 1987.

[15] G. Kiczales, E. H. ad J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327–353, 2001.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Meada, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland*, June 1997.

[17] G. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems*. PhD thesis, University of St Andrews, 1992.

[18] K. Löhr. Concurrent Annotations for Reusable Software. *Communications of the ACM*, 36(9):81–89, September 1993.

[19] C. Lopes. *D: A Language Framework For Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, November 1997.

[20] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) '91*, pages 231–250, 1991.

[21] K. R. Mayes. Trends in operating systems towards dynamic user-level policy provision. Technical Report UMCS-93-9-1, University of Manchester, Computer Science Department, Sept. 1993.

[22] R. Morrison, D. Balasubramaniam, M. Greenwood, G. Kirby, K. Mayes, D. Munro, and B. Warboys. ProcessBase Reference Manual (version 1.0.6). Technical report, Universities of St Andrews and Manchester, 1999.

[23] R. Morrison, D. Balasubramaniam, R. M. Greenwood, G. N. C. Kirby, K. Mayes, D. S. Munro, and B. Warboys. An Approach to Compliance in Software Architectures. *IEE Computing & Control Engineering Journal, Special Issue on Informatics*, 11(4):195–200, 2000.

[24] R. Morrison, D. Balasubramaniam, R. M. Greenwood, G. N. C. Kirby, K. Mayes, D. S. Munro, and B. C. Warboys. A Compliant Persistent Architecture. *Software, Practice & Experience Special Issue on Persistent Object Systems*, 30(4):363–386, 2000.

[25] D. S. Munro, K. Falkner, M. Lowry, and F. Vaughan. Mosaic: A Non-intrusive Complete Garbage Collector for DSM Systems. In *Proceedings of the First International Symposium on Cluster Computing and the Grid (WSDSM'01)*, pages 539–546, May 2001.

[26] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 734–737, June 2000.

[27] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2001.

[28] H. Ossher and P. Tarr. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. *Communications of the ACM*, 44(10):43–50, October 2001.

[29] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[30] A. Popovici, G. Alonso, and T. Gross. Just-In-Time aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 100–109, March 2003.

[31] J. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, Aug. 1972.

[32] G. Sullivan. Aspect-Oriented Programming Using Reflection and Metaobject Protocols. *Communications of the ACM*, 44(10):95–97, October 2001.

[33] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of ICSE'99*, pages 107–119, 1999.

[34] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages, AOSD 2002*, pages 1–8, April 2002.

[35] W. Wulf, E. Cohen, W. Corwin, A. Jones, L. R., C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor System. *Communications of the ACM*, 17(6):337–245, June 1974.