SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **30**:363–386

A compliant persistent architecture

Ron Morrison[*,1,†], Dharini Balasubramaniam[1,‡], Mark Greenwood[2,§], Graham Kirby[1,¶], Ken Mayes[2,‖], Dave Munro[3,**] and Brian Warboys[2,††]

[1]*School of Computer Science, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, U.K.*
[2]*Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, U.K.*
[3]*Department of Computer Science, University of Adelaide, South Australia 5005, Australia*

SUMMARY

The changing needs of modern application systems demand new and radical software architectures to support them. The attraction of persistent systems is that they define precisely the extent to which they are open, thereby allowing the dynamically changing resource requirements of applications to be tracked accurately within the persistent environment. Thus, an ever-growing body of work is being established to study the nature of running applications, and to use the information gleaned, to improve the run-time execution of these applications. Here we propose a new architectural approach to constructing persistent systems that accommodates, and thus is compliant to, the needs of particular applications. By separating policy from mechanism in all components, the architecture may be tailored to the policy needs of the application.[‡‡] We first propose a generic architecture for compliance, and then show how it may be instantiated. Finally, we describe an example of how the architecture operates in a manner that is compliant to a target application. We postulate, since we have not yet measured, that the benefits of compliant architectures will be a reduction in complexity, with corresponding gains in flexibility, portability, understandability in terms of failure semantics, and performance. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: persistence; nano-kernels; process modelling; programming languages; software architectures

[*]Correspondence to: Ron Morrison, School of Computer Science, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, U.K.
[†]E-mail: ron@dcs.st-and.ac.uk
[‡]E-mail: dharini@dcs.st-and.ac.uk
[§]E-mail: markg@cs.man.ac.uk
[¶]E-mail: graham@dcs.st-and.ac.uk
[‖]E-mail: ken@cs.man.ac.uk
[**]E-mail: dave@cs.adelaide.edu.au
[††]E-mail: brian@cs.man.ac.uk
[‡‡]Policy may be regarded as strategy for achieving a goal, such as a cache eviction algorithm, whereas mechanism is the method by which the objective is achieved, such as the physical movement of the cache lines. As we see later, policy and mechanism are composable to form new mechanism.

Received 23 April 1999
Revised 4 October 1999
Accepted 1 December 1999

INTRODUCTION

The growing requirements of persistent application systems challenge software engineers to provide the appropriate architectural infrastructure. Where the structure of the data is simple, file systems and operating systems are sufficient to supply the infrastructure for the storage and use of data and the execution of programs. When the structure of the data becomes more sophisticated, data models, and in particular object-oriented models, implemented in databases may be used. The contribution of orthogonal persistence [1] is to integrate the notions of long- and short-term data, and to allow programs to be treated as first class data objects [2]. As the application system evolves, more data, meta-data, programs and users are accumulated and must be accommodated by a further step in architectural design. While process modelling systems [3] provide the conceptual support for this evolutionary style, there is no agreed software architecture to supply these needs [4].

The challenge in supporting process modelling systems is in efficiently providing the radically varying facilities required by different process models [5–7]. When implemented on a conventional software architecture, there is often a repetition and duplication of both mechanisms and policy in different architectural components which conflict with each other and add complexity to the overall system. The repetition occurs when the same policies/mechanisms are implemented in different components, and duplication occurs when the policies/mechanisms in different components are radically different and may interact in an undesirable manner. This complexity makes it difficult to predict how the system will perform, particularly in terms of failure semantics and run-time efficiency.

Process modelling systems may be seen as archetypal data intensive applications systems, sometimes referred to as Persistent Application Systems (PASs) [8]. They are potentially long-lived, concurrently accessed and consist of large bodies of programs and data. In particular, process models evolve frequently, and therefore must accommodate dynamic change within the application domain. Thus, the architectural problems encountered in implementing process modelling systems are similar to, if not the same as, those investigated by the persistent programming community over many years now [8–22].

This paper describes a new architectural approach to the construction of persistent systems that accommodates the needs of particular applications. We describe this as being *compliant* to these needs. The architecture takes advantage of the fact that persistent systems are well structured to discover the dynamically changing demands of applications; since they define precisely the extent to which they are open, the computation which takes place within the persistent environment may be relatively easily tracked.

The novelty of a compliant architecture is that the components are designed, top down, with the philosophy of fitting the architecture to the needs of the particular application. This contrasts with the more traditional, bottom up, approach to providing static abstract components or layers designed to meet the predicted needs of the majority of applications. The key scientific advance, in our approach, is to separate mechanism and policy, a technique that has been used before but not all at once, and consistently, on every component of the architecture.

We report here on our first attempt at building a compliant architecture. We start by proposing a generic architecture for compliance and then show how it may be instantiated with components. We also show how the architecture operates in a manner that is compliant to a target application.

The key components of our initial compliant architecture are: a nano-kernel, HWO, and its associated library-based operating system which together constitute the Arena system [23,24]; a new persistent programming language, ProcessBase [25]; and the mechanisms for defining and separating policy and mechanism. We postulate, since we have not yet measured, that the benefits of compliant architectures will be a reduction in complexity, with corresponding gains in flexibility, portability, understandability in terms of failure semantics, and performance.

STRUCTURING SOFTWARE SYSTEMS

There are two common generic approaches to structuring complex software, which roughly correspond to the notions of in-process and out-of-process computation. With in-process system structuring, the architectural components are layered upon one another, communicating via the interface provided by the supporting layer. All computation takes place within one process although it may be multi-threaded. With out-of-process system structuring, the architectural components communicate with one another through their published interfaces using a mechanism equivalent to Inter-Process Communication (IPC).

In practice, both in-process and out-of-process system structuring are required for the most complex systems. Furthermore, there are many variations of these extremes that complicate the structuring domain. For example, in-process and out-of-process structuring may be mixed, where some of the components are layered and some communicate using IPC. This may even occur at different layers within the architecture.

Our goal is to achieve compliance within the system architecture and for simplicity immediately focus on layered architectures. In effect, we are concentrating on a single application executing in a single address space. Since we can regard all processes running on a single machine as running on a single address space, we postulate that we will be able to adapt our work to out-of-process systems at a future date.

Both in-process and out-of-process computation use self-defining components that present an interface to the outside world. Each component is potentially a separate Virtual Machine (VM) that may be scheduled accordingly. Understanding the overall architecture requires preserving the components, since they define their own context in which the user may wish to consider the component (layer).

Within a layered architecture, however, it is often tempting to assume that, for efficiency, the layers may be flattened, by some sophisticated compilation or reflective technique, to a single layer for execution.[*] However, the flattening of layers is time-constrained in that the single layer is only of use until the next time the system evolves. The evolution is best understood in the context of the evolving layers, and that is precisely why it is necessary to preserve that context. There is, therefore, an efficiency trade-off between the cost of execution in the layered form, and the costs of flattening and execution in the flattened form. Such trade-offs are application-dependent and can only be evaluated dynamically; thus, they are unlikely to yield a single winner.

Since our application domain is that of systems that evolve frequently we will forego the temptation to flatten layers for the present. Our focus is therefore on constructing layered architectures that are compliant to the needs of the application.

[*]This is the situation found in real-time applications. The term *operating software* has been used to characterise such real-time systems to emphasise the intimate connection between the real-time application and the real-time operating system [26].

DISCOVERING AND USING APPLICATION KNOWLEDGE

The key to the run-time efficiency of any software or hardware architecture is to ensure that the correct components are available and ready to use at the time they are required. For example, within a storage hierarchy, it is imperative to run-time efficiency to ensure that the data and programs are in the correct place in the storage hierarchy when they are required for use. Thus, register allocation mechanisms, caching techniques and page replacement algorithms should be optimised for such an effect.

To achieve optimum application efficiency, which requires application-specific knowledge, a software architect must address two major questions. They are:

- How does the system discover what the application is doing?
- How should the architecture be structured to utilise the above knowledge?

Conventional software architectures, which we would term non-compliant, provide static abstract layers to meet the average predicted needs of the majority of cases. The application knowledge discovery is performed statically by simulating and benchmarking the applications that are intended to be run on the system. The architecture is then structured to perform well under the benchmark conditions. Optimisations are often based on strategies such as overall throughput and may use average or common application execution profiles. The Unix operating system interface is a good example of such an interface between kernel and application layers.

The advantage of the static interface approach is that it provides a high degree of code reuse and therefore savings, in terms of code re-writing and portability. Many applications may reuse the interface without regard to its implementation. However, they do so at the possible expense of individual applications since the optimisations necessary for individual cases cannot be accommodated in the general case. Thus, in such an architecture individual applications only occasionally run optimally, and even then it is by accident.

In this paper, we will assume that applications have some information, such as working set size, that may be used to improve their performance if used by the supporting architecture. We recognise an ever growing body of work is being established to study the nature of running applications in persistent systems [27–31], databases [32–34] and operating systems [35–37]. We presume that we may intercept the results of any of this work, and concentrate here on how the software is structured to make use of such information.

Throughout this paper we consider a typical PAS: a process modelling application which is implemented in a Process Modelling Language (PML). The PML is in turn implemented in a Persistent Programming Language (PPL) which is itself hosted by some Operating System (OS). This is illustrated in Figure 1. As described above, we will maintain the layers of abstraction for ease of evolution and maintenance of the system. Our goal is to achieve this in conjunction with efficient implementation of the application.

In the non-compliant approach of Figure 1, the interfaces between the OS and the PPL, and the PPL and PML are fixed and all management policy and mechanism is encapsulated behind the interface. Thus even if an application has some knowledge of its particular execution profile it may be unable to use it to expedite its own execution, since it is not able to communicate the information to the lower layers.

Non-compliance occurs in many persistent programming systems in the interactions amongst the application, the persistent object store manager and the operating system storage manager. Knowledge

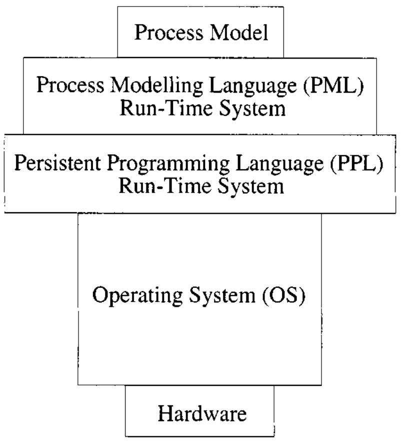
Process Model
Process Modelling Language (PML) Run-Time System
Persistent Programming Language (PPL) Run-Time System
Operating System (OS)
Hardware

Figure 1. A non-compliant architecture.

of the pattern of access to data at the application level, such as object clustering, is usually not useable since management of the persistent storage is encapsulated within the persistent run-time system, and ignores other software components, including the application. Furthermore, there are often competing and harmful interactions between the duplicated storage management policies of the persistent object store and the operating system secondary storage manager. For example, the after-image, shadow-page mechanism [13] of the Napier88 standard release [38] implements a page replacement policy in the persistent storage manager. This policy has to compete with the page replacement algorithm of the operating system, sometimes with disastrous effects on performance. Non-compliant architectures have the potential for many such impedance mismatches.

The crucial step in designing a compliant architecture is to recognise the power of structuring complex software in layers while freeing it from the dangers of encapsulation [39]. Our desired effect of high reuse of architectural components while eliminating repetition and duplication can be achieved by the separation of policy from mechanism. In this the mechanisms presented by an architectural layer may be controlled, in terms of policy, by the layers above. Thus, the layers provide the mechanism through their interfaces, and the higher layers provide the policy in the manner in which they use the interface calls.

SEPARATING POLICY AND MECHANISM

The desirability of separating mechanism from policy has been recognised for some time. It was a motivating force in the design of the Hydra operating system [40], and more recently the trend in operating systems has been to move policy out of the kernel and into the realm of the user.

Micro-kernels such as Mach [41] and Chorus [42] support user-provision of store, file and network management. The Psyche system [43] and work on scheduler activations [44] take thread scheduling out of the kernel by enabling the kernel to make up-calls to the user level. User-level library-based application-specific resource management appears in PANDA [45] and Exokernel [46].

Application-oriented systems have been produced to support real-time systems [47], customisable thread packages [48], communications protocols [49] generalised kernels [50–52] and languages [53]. These approaches allow flexibility in terms of resource management policy, to the extent of dynamic, run-time policy change [52]. In all of these systems it is the function of the application's run-time system to specialise the operating system.

In general, the above systems use two techniques to accommodate user-level policy: by providing a user-level server process together with an Inter-Process Communication (IPC) mechanism, or by providing a library of routines. Both Mach and Chorus use IPC, whereas PANDA and Exokernel are termed library operating systems.

In the persistent world, the Grasshopper operating system [22,54] uses both libraries and an IPC mechanism to provide a persistent micro-kernel. More recently work by the same group has produced a library-based nanokernel called Charm [55,56]. This has similar design goals to Arena, to avoid imposing inappropriate abstractions on higher layers, but goes further in exposing hardware-specific details.

Figure 2 illustrates how our example architecture may be structured to provide the policy/mechanism separation. The interfaces between the levels of the architecture define a set of up-calls and down-calls. The down-calls constitute the mechanisms that the higher layers may call upon. These include calls for passing policy information. Up-calls constitute entry points to the higher layers that may be used to request policy information.

At any particular layer, the mechanism provided from below, together with the policies implemented at that level, constitute mechanism for the higher layers. Thus:

$$\mathrm{policy}_{\mathrm{n}} + \mathrm{mechanism}_{\mathrm{n}-1} = \mathrm{mechanism}_{\mathrm{n}}$$

Figure 2 is illustrative of a particular application system. As we will see later, it may be implemented in a number of ways depending on the components and the interface mechanisms.

The work reported here builds significantly on our experience in designing, constructing and using operating systems, persistent object systems and process support systems–with particular reference to our previous work on the Arena operating system and the Flask persistent object store [57]. Before we can describe our approach to a generic compliant architecture, we will describe the techniques used in Arena and Flask for compliance. We will then describe our first attempt at constructing a compliant persistent architecture.

Arena

The Arena customisable operating system [24] provides a toolkit of user-level library resource managers (ARMs–Arena Resource Managers), which act as a framework for operating system policy instantiations. These resource managers operate on the Arena abstract hardware interface, provided by a nano-kernel, the hardware object (HWO), which gives access to the mechanisms provided by the hardware. Only the HWO need be re-implemented when the system is ported to other target hardware platforms. The basic structure of the Arena system is illustrated in Figure 3.

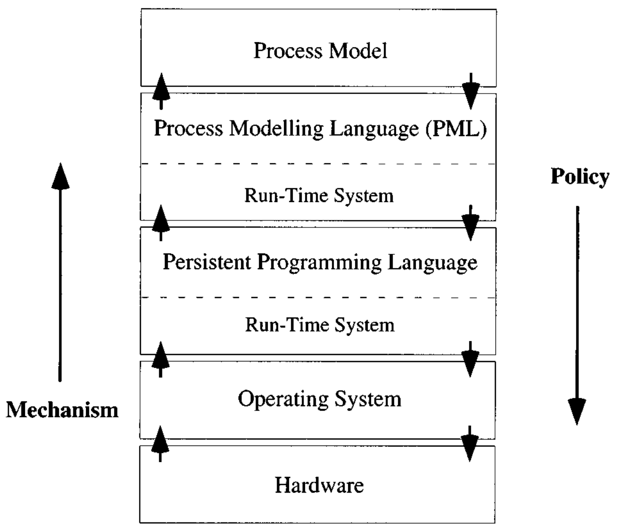

Figure 2. A compliant version of Figure 1.

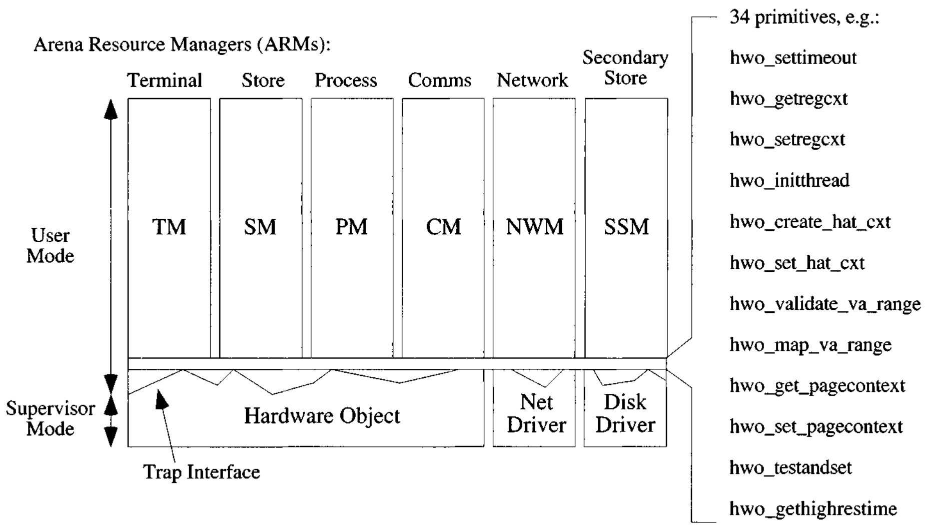

Figure 3. The Arena operating system.

The Arena system allows the application implementation architecture to correspond exactly to the application logical architecture. This is made possible by coalescing the operating system policy with the application run-time system policy. That is, the run-time implementation of the application is linked to the ARMs.

The HWO nano-kernel is a thin layer of largely policy-free code that provides access to low-level mechanisms via a set of primitives. This interface provides a set of down-calls that is used by the hardware-independent ARMs. At present there are 34 primitives in the HWO interface, which is accessed through a library.

The HWO is implemented at user-level wherever possible. However, for protection, the HWO may operate in supervisor mode by traversing the trap interface. The HWO deals mainly with low-level concerns such as register contexts and address translation, with the policy being provided by the ARMs in the user-level libraries. For example, although the HWO has no notion of threads, it does provide context-switching primitives to get and to set the register context (*hwo_getregcxt* and *hwo_setregcxt*). Through these primitives, the ARMs may implement context switching and thus provide the policy for scheduling threads.

The ARMs may interact with each other via published interfaces. So, for example, the Process Manager (PM) may call the Store Manager (SM) with a request for a region of virtual address space to use as a thread stack. Resource management policy may be changed by linking to a different resource manager or by defining a new one. However, the interfaces remain the same (or within a sub-class hierarchy) so that different versions of a manager may use a common third party. Thus, a PM may pre-allocate thread stacks by obtaining a single large SM region at start-up, but both this pre-allocating and another non-pre-allocating PM can use the same SM in the same manner.

Significantly, the interfaces provided by the user-level ARMs have three components:

(i) the operations available to the application code (down-calls from the application),
(ii) the operations used by other managers (horizontal calls), and
(iii) the operations used by the HWO (up-calls from the HWO).

A description of the up-call mechanism in Arena will be given later, but for the present it is sufficient to note that in any separation of policy and mechanism (and not just in Arena), all three sets of operations are required to completely describe the interface.

The Flask architecture

The Napier88 persistent object store [17] provides a generic architecture that has been used to support Napier88 and, by others, to provide persistence to a variety of programming paradigms. These include functional programming (Staple) [58], typeful programming (P-Quest) [59], object-oriented database programming (Galileo) [9], LISP [60], and the PIOS database [61]. The most recent design of the persistent object store, Flask [57], provides a flexible system in which concurrency control schemes and recovery mechanisms may be specified according to the needs of the application. This allows the same data to be used in conjunction with different concurrency control schemes and recovery mechanisms.

The framework of the Flask architecture is shown in Figure 4 as a ‘V-shaped’ layered architecture to signify the minimal functionality built-in at the lower layers. At the top layer the specifications of the model are independent of the algorithms used to enforce them and can take advantage of the semantics of these algorithms to exploit potential concurrency. For example, a particular specification

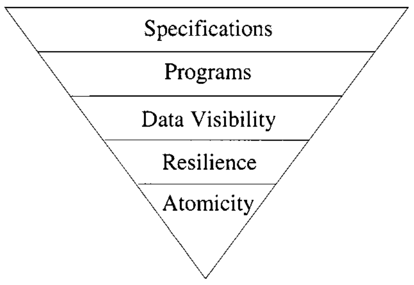

Figure 4. The Flask V-shaped architecture.

may translate into an optimistic algorithm or alternatively a pessimistic one, while the information they operate over remains the same. More importantly, such an approach can accommodate different models of concurrency control.

The focus of interest in the Flask architecture centres on the visibility of data from different actions (e.g. threads, processes). This visibility is expressed in terms of the control of movement between a globally visible database and conceptual stores called access sets. Each action is associated with a local access set and may use other shared access sets. The specification of the movement of data may be determined implicitly or explicitly but requires an interface language, such as CACS [62] to specify the movements. As a result, the CACS specification determines the policy of data movement.

At the lowest level the atomicity layers ensures consistent update to the access sets and the global database. The failure resilience layer utilises this atomicity to effect an action making its changes permanent.

The Flask architecture has been implemented by marrying the notion of data visibility as expressed by access sets to a concurrent shadow paging mechanism. The layers in Flask constitute a clear instantiation of the policy/mechanism separation.

GENERIC COMPLIANCE

Our approach to defining and implementing a compliant architecture builds on the work of both the Arena system and the Flask architecture. However, the Arena system collapses the architecture into a single level by combining the application run-time system with the operating system. Furthermore, Arena uses a common library binding mechanism, which often restricts it to be single language. By contrast, the Flask V-shape only addresses concurrency control for policy/mechanism separation, but it is multi-level and multi-language, a feature we wish to retain.

Our first attempt at a generic compliant architecture is illustrated in Figure 5.

The compliant architecture retains layers for the reasons described earlier. The system functions are the entities over which the application wishes to exert control through the policy/mechanism separation.

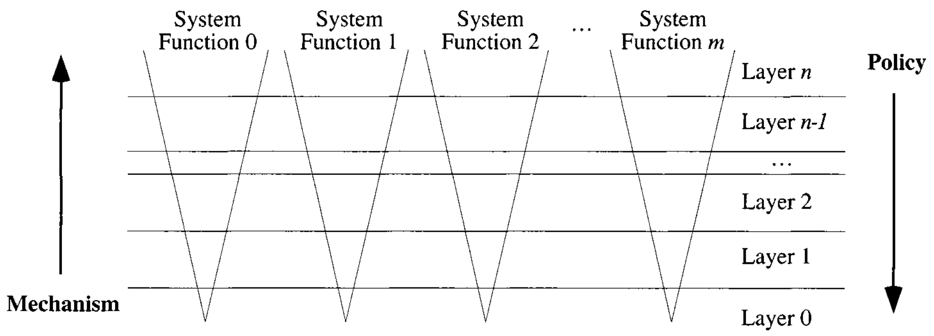

Figure 5. A generic compliant architecture.

We expect the system functions to be entities such as concurrency control, scheduling, address space and recovery, but there may be others, such as distribution, depending on the application.

The key feature of the compliant architecture is that mechanisms provided by layers are controlled by policies implemented at higher layers, thereby providing increased functionality moving up the levels of abstraction. This provides the necessary flexibility but also permits efficiency gains, since many optimisations are achievable at a higher-level in the architecture.

Of course, layers may choose to pass mechanism directly to higher layers for policy to be implemented at a more appropriate level, thereby effectively collapsing the layers within any particular V (system function). Furthermore, layers may vary the mechanism used to implement their policy. Thus, an application may run under an atomic transaction regime supported by a particular stable storage mechanism on one occasion, and under a SAGA [63] regime supported by the same or a different stable storage mechanism on another, depending on the requirements of the application.

The system functions are themselves replaceable and extendable, and may provide different interfaces at each layer on different occasions. For example, should it be discovered that a certain process model requires a particular scheduling abstraction, this can be provided at the correct layer by replacing the scheduling system function.

A compliant architecture may therefore be structured by determining the following:

(i) the number of layers in the architecture;
(ii) the system functions that the architecture allows applications to control (e.g. recovery, scheduling, clock ticks, etc.);
(iii) the method used for specifying policy information;
(iv) the method used for passing system information between layers and system functions (up-calls, down-calls and horizontal calls).

Once the number of layers in an architecture has been fixed, the other three issues must be addressed at the interfaces between all the layers. Policy specification may be different between different layers, and the calling mechanisms do not rely on one particular technique but may be specific to the interfaces between particular layers. Furthermore, system functions may be hidden at some level and only

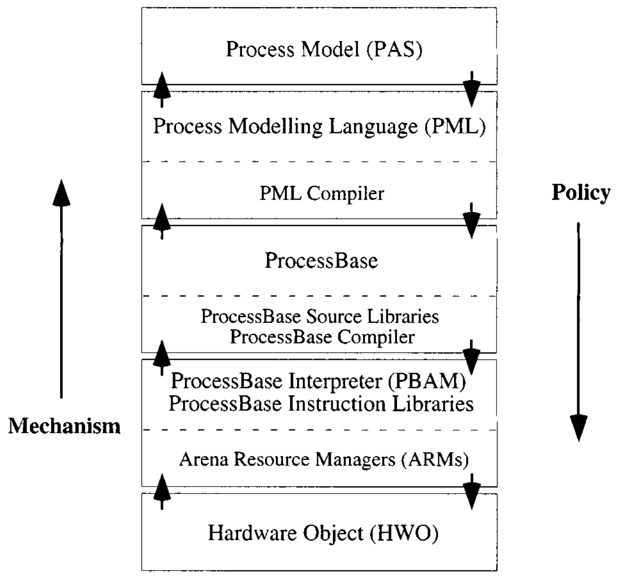
Process Model (PAS)
Process Modelling Language (PML)
PML Compiler
ProcessBase
ProcessBase Source Libraries
ProcessBase Compiler
ProcessBase Interpreter (PBAM)
ProcessBase Instruction Libraries
Arena Resource Managers (ARMs)
Hardware Object (HWO)
Mechanism
Policy

Figure 6. A compliant persistent architecture.

accessible to the layers below. This, in effect, cuts off a particular V and makes the system function opaque to the upper layers–presumably for implicit mapping.

The alternative to the above approach for constructing the architectural layers is to use a single technique for up-calls and down-calls at the interfaces between all layers. However this may not always be the most compliant approach, and we wish to preserve flexibility.

A COMPLIANT PERSISTENT ARCHITECTURE

Figure 6 provides a more detailed and compliant version of the architecture given in Figure 1. There are five layers in the architecture, discounting the hardware itself, remembering that the layers may collapse within any particular system function. The layers are: the process model (PAS); the process modelling language (PML); the persistent programming language (ProcessBase); the Arena ARMs, which are linked with the ProcessBase run-time system; and the nano-kernel (HWO).

To illustrate how the architecture is constructed, we will describe the policy/mechanism divide between the ARMs and the HWO, and between the ProcessBase language and Arena. Furthermore we will use parameter passing to simplify the specification of policy information, and concentrate on how information is transmitted between the layers. A major strength of our compliant architecture is that it does not rely on any particular binding technique nor common interface structure. Thus, as we

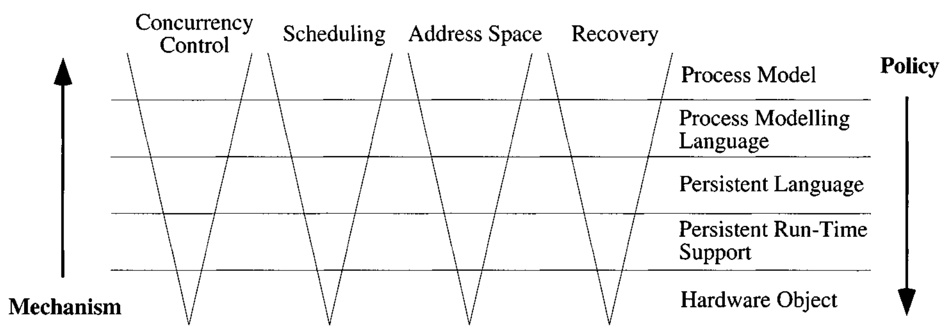

Figure 7. System functions in the compliant architecture.

will see, the interface between the ARMs and the HWO is quite different from the interface between ProcessBase and Arena.

Another view of the compliant persistent architecture is given in Figure 7, which shows the system functions of concurrency control, scheduling, address space and recovery.

Before describing a particular example of a compliant architecture, we must first finish our description of the ARM/HWO interface.

Arena policy and HWO mechanism

At the bottom level of the compliant architecture, the HWO interface is a set of C++ functions in a C++ library. These provide the mechanism that Arena uses to access the hardware services. The ARMs of Arena are also written in C++ and are also kept in C++ libraries for the upper layers of the architecture to access. Thus, the overall binding mechanism is that of C++, and the interface may be extended using the inheritance hierarchy, for extensions that conform to the sub-class late binding rules, or by adding new libraries.

Compliance in Arena

We have already described how the Arena ARM code calls the HWO code (down-call) and other Arena code (horizontal call). The missing component for compliance is the up-call, which signifies an event in the HWO that is handled by ARMs.

Event handling comprises a major component of resource management policy. Event handling at user-level in Arena has two components: first, to register a handler for an event; and secondly, to make an up-call to execute that handler.

1. *Registering an event handler*. In Arena, the HWO defines a fixed set of event types. Event handling at user-level is achieved by registering distinguished event handler threads with the HWO and the PM. Each is specific to a particular event type, and will be made runnable in response to an event of that type.

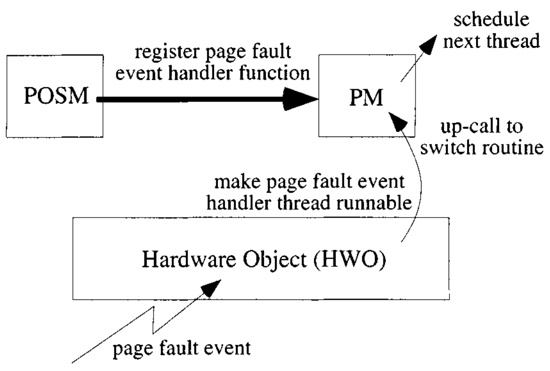
schedule next thread
register page fault event handler function
POSM
PM
up-call to switch routine
make page fault event handler thread runnable
Hardware Object (HWO)
page fault event

Figure 8. An Arena up-call.

A number of uninitialised event threads are pre-allocated in the PM. Registering specific event handlers is a two-stage process. The first stage occurs at initialisation, when the PM obtains a pointer to the event state data structure in the HWO, using a call to *hwo_geteventstate()*, and fills in the uninitialised event thread context block pointers. The second stage occurs when an ARM, interested in handling a particular event type, calls the PM through *m_registerEventHandler(func, evnum)*, passing it a function to be run by the specified event handler thread. The PM allocates a new thread stack and calls *hwo_initeventthread()* to initialise the designated event thread block.

2. *Executing an event handler–the up-call*. The Arena up-call consists of running the PM *switch()* routine so that a runnable event thread can be scheduled. On an up-call, arising from an event, the processor state has already been saved by the HWO, when the trap occurred. The user-level context has been saved to a logical *Event Context Buffer* (ECB) whose implementation is processor-dependent [67]. The HWO makes the thread corresponding to the event runnable, using its pointer to the appropriate event thread context block residing within the user-level PM. This pointer is set up as part of the registration process.
   Having made the event thread runnable, the up-call mechanism proceeds by loading the PM *switch()* execution context, and returning the processor to user-mode, control thus passing to the PM. When *switch()* is invoked by the up-call, the saved context is copied from the ECB to the context block of the application thread, whose execution had been interrupted by the event. This is achieved via a call to *hwo_getregcxt()*. The *switch()* routine then selects the next thread to run. This may, according to the application-specific scheduling policy, be the runnable event thread. The selected thread begins execution via a call to *hwo_setregcxt()* made at the end of the *switch()* routine.

Figure 8 shows an example in which the persistent object store manager (POSM) handles page fault events via up-calls from the HWO.

Thus, in Arena, the application or language system can determine event handling policy in two ways. The policy implemented by the event thread can be language-specific, and also the policy implemented by the thread scheduler can be language-specific. This is all achieved by binding in the appropriate library functions.

Pending events are queued by the HWO so that the PM has opportunities to schedule the corresponding threads and therefore handle events of the same kind in the correct order.

ProcessBase policy and Arena mechanism

The ProcessBase language and support system is designed as a core persistent implementation platform. It consists of the language and its persistent environment. The persistent store is populated and, indeed, the system uses objects within the persistent store to support itself. The ProcessBase programming system provides the following facilities:

(i) orthogonal persistence–models of data independent of longevity,
(ii) type completeness–no restrictions on constructing types,
(iii) higher-order procedures–procedures are first class data objects,
(iv) information hiding without encapsulation–views of data that hide detail [39],
(v) a strongly typed stable store–a populated environment of typed data objects that may be updated atomically,
(vi) hyper-code–a single representation of a value throughout its lifetime [65,66],
(vii) linguistic reflection–to allow evolution [67],
(viii) exceptions–for recovering from exceptional conditions.

The ProcessBase language is in the algol tradition, as were its predecessors S-algol [68], PS-algol [69] and Napier88 [38]. The type system contains: the base types integer, real, boolean and string; higher-order procedures that allow code to exist in the value space [2]; aggregates formed using the vector and view types; and finally, an explicit constructor to provide locations. The type system is mostly statically checkable, a property we wish to retain wherever possible. However, dynamic projection out of unions for type any allows the dynamic binding required for orthogonal persistence [1] and system evolution [70].

Compliance in ProcessBase

ProcessBase programs are compiled into ProcessBase Abstract Machine (PBAM) code that may then be executed by the PBAM interpreter, which is written in C++. Compliance is accommodated at the language level by a set of libraries, written in ProcessBase, which extend the functionality to comply with any particular architecture. At present I/O, threads, semaphores, persistence, and string and arithmetic functions are considered to be platform-dependent, compliant parts of the ProcessBase language. Other compliant extensions occur as applications require them.

The PBAM is also split into a core part and an extensible, platform-dependent, compliant part. The core part must be provided by all implementations whereas the compliant part may vary and may be implemented in different ways depending on the host platform. The compliant part takes the form of extra PBAM instructions. Figure 9 illustrates the mapping of a combination of application code and compliant libraries onto the PBAM. The normal mechanism is for programs to be compiled into PBAM core instructions. However this may be extended in both core and compliant ProcessBase code, by down-calls to both core and compliant PBAM instructions.

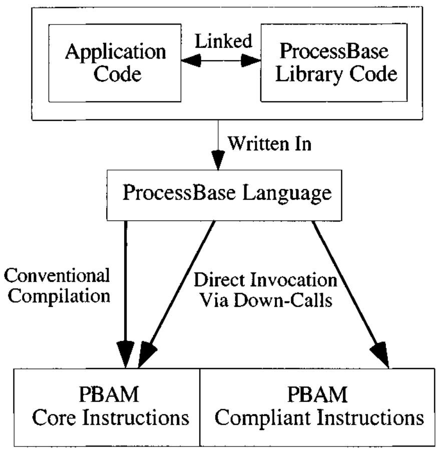

Figure 9. Mapping ProcessBase to the PBAM.

Down-calls, from ProcessBase to the PBAM interpreter, are performed by using a special syntax to invoke abstract machine instructions directly. The syntax is:

```
downcall opcode (...)
```

where the *downcall* construct is followed by the op-code for the PBAM instruction and any parameters it requires. The construct may also return a computed value. For example,

```
let sin ← fun (x : real) → real; downcall sinOp (x)
```

is a procedure that implements the *sine* function by invoking the PBAM instruction denoted by the *sinOp* opcode. The sine function could, of course, be written directly in ProcessBase, in which case it would be compiled to the core PBAM instructions.

In summary, a ProcessBase program may use a down-call to execute any PBAM instruction, core or compliant. In reality any particular interpreter may partially disallow such freedom for safety, and only permit a subset of the down-calls to be effective. A ProcessBase program written in the core language is compiled to core PBAM instructions, but may also contain down-calls to core PBAM instructions. A program written in compliant ProcessBase will also be compiled into core ProcessBase, but may also contain down-calls to core and compliant PBAM instructions. The reason for the difference is that the libraries that implement the compliant part also define the compliant op-codes available to the application programmer.

The down-call mechanism provides a method of extending PBAM to be compliant to the needs of the application. Where new PBAM instructions are required by the application, they may be provided by the compliant part of the PBAM interpreter and called by the special syntax above. The down-call mechanism can be used to pass policy information to a compliant PBAM interpreter that has provided an instruction implementing mechanism that can be specialised by the given policy.

From Figure 9, we can see that policy can now be transmitted from the ProcessBase language to the PBAM interpreter. The PBAM interpreter is written in C++ and is linked with the ARMs. Thus, the policy information from the application may be transmitted to the resource managers of Arena via the down-call mechanism of ProcessBase, and C++ parameter passing. The correct selection of resource managers to interpret the application data is imperative to the success of any particular instance of the compliant architecture.

ProcessBase libraries

The library mechanism, as part of the persistent store, is used to store all the standard functions and data for an instantiation of the architecture. The standard functions may be implemented in the core or the extension part of the language, and map appropriately to the PBAM.

Horizontal calls between separately compiled units can be made by binding in ProcessBase library values and then calling them. Down-calls are usually placed in the library at the behest of the interpreter and called in the same manner by the ProcessBase programs. Thus, the interpreter establishes which op-codes are available in the extension defined by the library. Separate versions of the compliant libraries may be implemented for each new architecture.

Although libraries may contain both compliant PBAM instructions and ProcessBase values, these elements are composed in different ways. ProcessBase values are specified as required for each compilation unit, whereas compliant PBAM instructions are specified for a particular instantiation of the PBAM.

Up-calls in ProcessBase

Up-calls can be made to PBAM from Arena through the Arena event mechanism. Correspondingly, an up-call from PBAM can be made to a ProcessBase program through the ProcessBase interrupt mechanism. Thus, low-level events may be transmitted as a mechanism to the ProcessBase program, which can handle the interrupt and apply the appropriate policy. This is achieved without collapsing the architectural layers.

A choice of mechanisms is available for handling interrupts. We could have chosen the Arena mechanism of using a separate thread to handle an interrupt, but since threads are not part of the ProcessBase core, we have chosen to use higher-order functions to achieve our goal.

To illustrate the ProcessBase interrupt mechanism, consider an application that wishes to know about clock ticks (every second say). To establish that the interrupt is available to the compliant application code in ProcessBase, the interpreter arranges for the definition:

```
let clock ← interrupt 17 (int)
```

to be placed in the ProcessBase *interrupt* library. This indicates that such an interrupt, supplying a single integer parameter, is available for use by compliant ProcessBase programs in this instantiation of the architecture. An integer constant, 17 in this example, is specified as an implementation-level identifier for the interrupt. The *interruptDowncall* procedure in the library passes policy information to particular interrupts using the PBAM instruction *interruptOp*. For example, the policy information could define when the interrupts should be active or inactive. The *interruptDowncall* procedure is defined by:

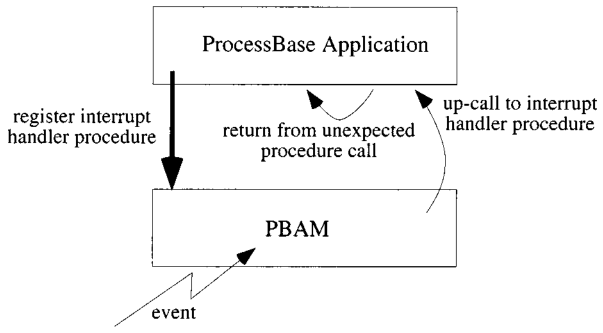
ProcessBase Application
register interrupt handler procedure
return from unexpected procedure call
up-call to interrupt handler procedure
PBAM
event

Figure 10. A ProcessBase up-call.

```
let interruptDowncall ← fun (name, status: string)
begin
  ! Map name to corresponding identifier id.
  downcall interruptOp (id, status)
end
```

and may be called, for example, by:

```
interruptDowncall ("Clock", "on")
```

to indicate that clock interrupts may be taken. Interrupts are handled by a procedure that is associated with the interrupt using the **handle interrupt** clause. For example

```
handle interrupt clock using fun (time: int) ; {}    ! Do something with time
```

The **handle interrupt** clause is valid for a particular scope and results in an unexpected procedure call if the interrupt is ‘on’ and the event occurs. That is, control resumes from the point of interruption after the procedure is called. Note that the parameter and result types of the procedure must be the same as those of the interrupt. Figure 10 illustrates the ProcessBase up-call mechanism.

We have now completed all the techniques for separating policy and mechanism in our compliant architecture. We will now proceed with an example of how the architecture is designed, constructed and executes.

AN EXAMPLE OF COMPLIANCE

Most persistent application systems have an object store garbage collector. In our example architecture we wish to inform the garbage collector, which is incremental, that if it encounters certain objects (roots) then the application may wish to nominate other objects to be clustered with those roots. The roots and nominated objects may change from time to time and in relation to each other. For example, a root object may have different nominated objects at any particular point in the execution of the application.

The garbage collector runs as an asynchronous thread and is activated, to execute one increment, by a timer interrupt. The whole activity of associating objects with others may be turned on and off by the application as it requires.

The process of defining an architecture compliant to the above needs involves four stages as described earlier:

(i) The first is to specify the number of layers in the architecture. Given space restrictions and the fact that we have only described the bottom layers, we will again restrict the architecture to having the HWO, the Arena run-time system and the ProcessBase language. We therefore assume that the application is written in or translated into ProcessBase for this example. To extend the definition to higher layers would involve defining further up-call and down-call mechanisms.
(ii) We have also restricted the number of system functions controlled by the architecture to that of storage placement, and even then to particular root values. We assume that all the other functions are pre-defined and not under explicit control of the programmer, but of course these may still be specific to the architecture and used by the language processor.
(iii) Policy information will be passed as ProcessBase parameters.
(iv) The final activity in defining the compliant architecture is to specify the up-calls and down-calls. We start from the top defining the ProcessBase level. The compliant ProcessBase library for this application will contain a down-call to define the roots from which we wish to cluster. It will also include the definition of an interrupt (up-call) which will ask the application to specify the objects to be clustered around the given root. Finally the library contains a down-call to control switching the interrupt on and off. The following code specifies all of this:

```
let cluster ← interrupt 23 (any) → *any
! The cluster interrupt, when handled, takes a root and returns a vector of objects.
! The type any is used to make the interrupt dynamically polymorphic.

let clusterOp ← opcode 251 (*any)
! This compliant instruction, op-code 251, specifies the roots of clustering.
```

These definitions constitute mechanism and are defined by the ProcessBase compliant library and provided by the PBAM compliant interpreter.

Two data structures are maintained in the ProcessBase application to facilitate the clustering of particular roots. They are an array of roots, and for each root an associated array of cluster objects. Type *any* is again used to make the typing polymorphic. The data structure and the application fragment to control the up-call and down-call are defined by the following code:

```
let init ← any (0)
let initObject ← vector @1 of [init, init, init]
! A vector (array) of dummy objects.

let clusterRoots ← vector @1 of [init, init, init, init, init]
! The empty vector of cluster roots.

let clusterObjects ← vector @1 of [initObject, initObject, initObject, initObject]
! The empty vector of vectors of associated objects.
! The root index is used to identify the vector of associated objects within this.
```

```
handle interrupt cluster using fun (root: any) → *any
begin
  let i ← loc (lwb (clusterRoots))

  while 'i <= upb (clusterRoots) and root ~= clusterRoots (i) do i := 'i+1
  ! The ' is a location dereference operation.

  if 'i <= upb (clusterRoots) then clusterObjects (i) else initObjects
end

! The handle interrupt clause handles the interrupt if it is switched on.
! It returns the vector of associated objects for the root to the PBAM interpreter.
! The program then resumes from where it left off.

...
! Set up the cluster roots.
! Set up the associated cluster objects.
! Make a down-call to clusterOp to establish the roots.

! Turn the up-call on.
ionterruptDowncall ("cluster", "on")

...

! Turn the up-call off again.
interruptDowncall ("cluster", "off")
```

The policy for clustering is now set by initialising the cluster roots and their associated objects. At any time in the application execution, the roots may be altered by assigning to the roots vector and making a down-call to the *clusterOp* instruction. The associated objects may also be altered dynamically by merely assigning into the *clusterObjects* vector. Thus we have established the mechanism provided by the PBAM interpreter and indicated how the dynamic policy may be passed from the ProcessBase application to the PBAM interpreter.

At some level in the compliant architecture the policy/mechanism interface is fixed. For compliance, the closer that interface is made to the hardware the better. In our case it is the HWO/ARM interface, which is a little above the bare hardware since the HWO was also designed for portability. Our remaining task is to use this interface, and to integrate the PBAM interpreter with the appropriate ARMs, re-implementing them if necessary. Again we will concentrate only on object clustering. Figure 11 illustrates how the PBAM interpreter is placed in relation to the other Arena code.

The incremental garbage collector (GC) is a PBAM function that shares state with the PBAM. The function is registered with the PM and associated with the event timer. The event timer is set using the *hwo_settimeout(n)* hardware object call. Thus, every *n* ticks the garbage collector will be made runnable to collect one increment. When the garbage collector has completed its increment it suspends itself on the *eventTimer* queue.

When activated the garbage collector first waits until the interpreter comes to the end of its current instruction, to ensure that the execution of the PBAM instruction is atomic. It then proceeds to garbage collect, and if it encounters a root object, which it can identify from the shared state with the PBAM interpreter, it makes an upcall to ascertain the associated cluster objects. The interpreter is now restarted to handle the interrupt and return the cluster objects which the garbage collector can then use.

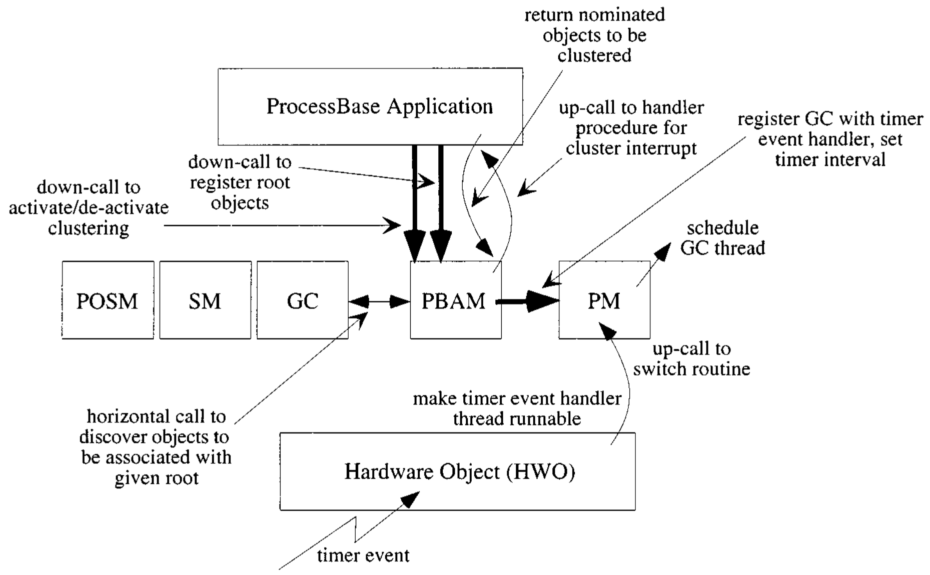
return nominated objects to be clustered
ProcessBase Application
up-call to handler procedure for cluster interrupt
register GC with timer event handler, set timer interval
down-call to register root objects
down-call to activate/de-activate clustering
schedule GC thread
POSM
SM
GC
PBAM
PM
up-call to switch routine
make timer event handler thread runnable
horizontal call to discover objects to be associated with given root
Hardware Object (HWO)
timer event

Figure 11. Example compliant architecture.

The dynamic sequence of activity may look something like:

(i) The PBAM interpreter registers the GC with the timer event in the PM and sets the timer interval.
(ii) The application registers some root objects (down-call from ProcessBase (PB) to PBAM).
(iii) The application activates the clustering (down-call from PB to PBAM, horizontal call from PBAM to GC).
(iv) A timer interrupt activates the GC (up-call from HWO to PM, horizontal call from PM to GC).
(v) The GC encounters a root and requests a list of nominated objects from the application (horizontal call from GC to PBAM, up-call from PBAM to PB, return from PB, return to GC with object list).
(vi) The application deactivates the cluster facility (down-call from PB to PBAM, horizontal call from PBAM to GC).

The degree to which an application is made compliant depends upon the amount of individual programming that the implementor will deem cost effective. There is a trade-off between fixing the interfaces for reuse, and thereby denying compliance, and the cost of re-coding for compliance. Such trade-offs can only be evaluated in the context of the particular application system.

CONCLUSIONS

Our experience in designing, constructing and using operating, persistent object and process support systems has led us to the conclusion that the next breakthrough in the architectural support for potentially large, long lived and concurrently accessed, user-centred systems is in customisable software support. We have coined the term *compliant* to describe architectures that are customisable and accommodate the needs of particular applications.

This paper recognises the power of structuring complex software in layers and has shown how to achieve compliance within such a system. Central to this is the separation of policy from mechanism across the interface layers. Down-calls may be used to define the mechanism that a lower layer provides, while up-calls can supply the necessary policy details to drive the mechanisms.

Many techniques may be used for compliance. In this paper, we have suggested that the following decisions have to be made to instantiate such an architecture:

(i) the number of layers in the architecture,
(ii) the system functions that the architecture allows applications to control (e.g. recovery, scheduling, clock ticks, etc.),
(iii) the method used for specifying policy information, and
(iv) the method used for passing system information between layers and system functions (up-calls, down-call and horizontal calls).

We postulate that any architecture implementing the above can be made compliant without the need for any common or regular interface mechanisms. There are clearly a number of questions that merit further investigation, including:

(i) What are the limits to the degree of compliance that can be achieved using this architecture?
(ii) In which situations should mechanism be made available to higher layers, and in which should it be hidden?
(iii) Can undesirable interactions arise between up-calls, down-calls and horizontal-calls?
(iv) What are the required features of systems programming languages used in this style?

While it is clear that compliant architectures may be built, it is not obvious that they will meet their design goals. Here we have described a proof of concept experiment where a compliant architecture is built out of a nano-kernel, HWO, and its associated library-based operating system which together constitute the Arena system; a new persistent programming language, ProcessBase; together with the mechanisms for defining and separating policy and mechanism by up-calls and down-calls. At the time of writing, the instantiation of the architecture includes the following:

(i) Arena running on the EDS multi-processor and on Pentium;
(ii) a distributed object store running on Arena;
(iii) a Java graphical interface to Arena;
(iv) an after-image shadow paging recovery manager running on Arena;
(v) a compiler for ProcessBase;
(vi) a ProcessBase interpreter running on Arena and on Solaris.

This is certainly not the only manner in which compliance can be achieved, and we suspect it is only the first in many such explorations by ourselves and others.

ACKNOWLEDGEMENTS

This work was supported by EPSRC grants GR/L32699 and GR/L34433 ‘Compliant Systems Architecture’, and is continuing with support by EPSRC grants GR/M88938 and GR/M88945 'Compliant Systems Architecture Phase 2'. The ideas contained within, and presentation of, this paper were enhanced by a long discussion with Al Dearle on the return journey from a Pastel Workshop in Lillehammer, and by the comments of the anonymous referees.

REFERENCES

1. Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott WP, Morrison R. An approach to persistent programming. *Computer Journal* 1983; **26**(4): 360–365.
2. Atkinson MP, Morrison R. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems* 1985; **7**(4): 539–559.
3. Warboys B. The IPSE 2.5 Project: Process modelling as the basis for a support environment. *Proceedings of 1st International Conference on System Development Environments and Factories*; Berlin: Germany, 1989.
4. Finkelstein A, Kramer J, Nuseibeh B (eds.), *Software Process Modelling and Technology*; Research Studies Press, 1994.
5. Bruynooghe RF, Parker JM, Rowles JS. PSS: A system for process enactment. *Proceedings of 1st International Conference on the Software Process: Manufacturing Complex Systems*, Redondo Beach, CA, 1991; 142–158.
6. Reisig W, Rozenberg G. Informal introduction to petri nets. *Proceedings of the Advanced Course on Petri Nets*; *Lecture Notes in Computer Science*, **1491**, Reisig W, Rozenberg G (eds.), Dagstuhl, 1996; 1–11.
7. Wise A, Lerner, BS, McCall, EK, Osterweil LJ, Sutton SM. Specifying coordination in processes using Little-JIL. Department of Computer Science, University of Massachusetts at Amherst *Technical Report 98–38*, 1998.
8. Atkinson MP, Morrison R. Orthogonally persistent object systems. *VLDB Journal* 1995; **4**(3):319–401.
9. Albano A, Cardelli L, Orsini R. Galileo: a strongly typed, interactive conceptual language. *ACM Transactions on Database Systems* 1985; **10**(2):230–260.
10. Morrison R, Dearle A, Bailey PJ, Brown AL, Atkinson MP. The persistent store as an enabling technology for integrated project support environments. *Proceedings of 8th IEEE International Conference on Software Engineering*; London, 1985; 166–172.
11. Atkinson MP, Morrison R, Pratten GD. A persistent information space architecture. *Proceedings of 9th Australian Computing Science Conference*, Australia, 1986.
12. Dearle A. On the construction of persistent programming environments. *PhD Thesis*, University of St Andrews, 1988.
13. Brown AL. Persistent object stores. *PhD Thesis*, University of St Andrews, 1989.
14. Connor RCH, Brown AL, Carrick R, Dearle A, Morrison R. The persistent abstract machine. *Persistent Object Systems*; Rosenberg J, Koch DM (eds.), *Proceedings of the 3rd International Workshop on Persistent Object Systems*, Newcastle, Australia, 1990; 353–366.
15. Rosenberg J. The MONADS architecture–a layered view. *Implementing Persistent Object Bases*; Dearle A, Shaw GM, Zdonik SB (eds.), *Proceedings of the 4th International Workshop on Persistent Object Systems*, Martha's Vineyard, USA, 1990; 215–225.
16. Atkinson MP, Birnie A, Jackson N, Philbrow PC. Measuring Persistent Object Systems. *Persistent Object Systems*; Albano A, Morrison R (eds.), *Proceedings of the 5th International Workshop on Persistent Object Systems (POS5)*, San Miniato, Italy, 1992; 63–85.
17. Brown AL, Mainetto G, Matthes F, Müller R, McNally DJ. An open system architecture for a persistent object store. *Proceedings of 25th International Conference on Systems Sciences*, Hawaii, 1992; 766–776.
18. Brown AL, Morrison R. A generic persistent object store. *Software Engineering Journal* 1992; **7**(2):161–168.
19. Dearle A, Rosenberg J, Henskens FA, Vaughan F, Maciunas KJ. An examination of operating system support for persistent object systems. *Proceedings of 25th International Conference on Systems Sciences*, Hawaii, 1992; 779–789.
20. Matthes F, Schmidt JW. System construction in the tycoon environment: architectures, interfaces and gateways. *Proceedings of Euro-ARCH '93*, Hamburg, 1993; 301–317.
21. Munro DS. On the integration of concurrency, distribution and persistence. *PhD Thesis*, University of St Andrews. *Technical Report CS/94/1*, 1993.
22. Dearle A, di Bona R, Farrow J, Henskens F, Lindström A, Rosenberg J, Vaughan F. Grasshopper: An orthogonally persistent operating system. *Computer Systems* 1994; **7**(3):289–312.
23. Mayes KR. Trends in Operating Systems Towards Dynamic User-Level Policy Provision. University of Manchester *Technical Report UMCS-93-9-1*, 1993.
24. Mayes KR, Bridgland J. Arena–a Run-Time Operating System for Parallel Applications. *Proceedings of 5th EuroMicro Workshop on Parallel and Distributed Processing (PDP'97)*, 1997; 253–258.
25. Morrison R, Balasubramaniam D, Greenwood M, Kirby GNC, Mayes K, Munro DS, Warboys BC. ProcessBase reference manual (Version 1.0.6). Universities of St Andrews and Manchester, 1999.

26. Mukherjee B, Schwan K, Gopinath P. A survey of multiprocessor operating systems kernels. Georgia Institute of Technology, *Technical Report GIT-CC-92-05*, 1993.
27. Cutts QI, Connor RCH, Kirby GNC, Morrison R. An execution driven approach to code optimisation. *Proceedings of 17th Australasian Computer Science Conference (ACSC'94)*; Christchurch, New Zealand, 1994; 83–92.
28. Sjøberg DIK, Cutts QI, Welland R, Atkinson MP. Analysing persistent language applications. *Persistent Object Systems*; Atkinson MP, Maier D, Benzaken V (eds.), *Proceedings of the 6th International Workshop on Persistent Object Systems*, Tarascon, France, 1994; 235–255.
29. Hosking A. Residency check elimination for object-oriented persistent languages. *Persistent Object Systems: Principles and Practice*; Connor RCH, Nettles S (eds.), *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, NJ, 1996; 174–183.
30. Cutts QI, Lennon S, Hosking A. Reconciling buffer management with persistence optimisations. *Advances in Persistent Object Systems*; Morrison R, Jordan M, Atkinson MP (eds.), *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and 3rd International Workshop on Persistence and Java (PJW3)*, Tiburon, CA, 1999; 51–63.
31. Hosking A, Nystrom N, Cutts QI, Brahnmath, K. Optimizing the read and write barriers for orthogonal persistence. *Advances in Persistent Object Systems*; Morrison R, Jordan M, Atkinson MP (eds.), *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and 3rd International Workshop on Persistence and Java (PJW3)*, Tiburon, CA, 1999; 149–159.
32. Cattell RGG, Skeen J. Object operations benchmark. *ACM Transactions on Database Systems* 1992; **17**(1):1–31.
33. Dietrich SW, Brown M, Cortes-Rello E, Wunderlin S. A practitioner's introduction to database performance benchmarks and measurements. *Computer Journal* 1992; **35**(4):322–331.
34. Maynard AMG, Donnelly CM, Olszewski BR. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *ACM SIGPLAN Notices* 1994; **29**(11):145–156.
35. Gallivan K, Gannon D, Jalby W, Malony A, Wijshoff H. Experimentally characterizing the behaviour of multiprocessor memory systems: A case study. *IEEE Transactions on Software Engineering* 1990; **16**(2):216–222.
36. Bowen NS, Pradhan DK. Program fault tolerance based on memory access behavior. *Proceedings of 21st International Symposium on Fault-Tolerant Computing*, Montreal, Canada, 1991; 426–435.
37. Wall DW. Predicting program behavior using real or estimated profiles. *AGM SIGPLAN Notices* 1991; **26**(6):59–70.
38. Morrison R, Brown AL, Connor RCH, Cutts QI, Dearle A, Kirby GNC, Munro DS. Napier88 reference manual (Release 2.2.1). University of St Andrews, 1996.
39. Kirby GNC, Morrison R. A persistent view of encapsulation. *Computer Science '98*; McDonald C (eds.), *Proceedings of the 21st Australasian Computer Science Conference (ACSC'98)*, Perth, Australia, 1998; 231–244.
40. Wulf WA, Cohen E, Corwin WM, Jones AK, Levin R, Pierson C, Pollack FJ. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM* 1974; **17**(6):337–345.
41. Acceta M, Baron R, Bolosky W, Golub D, Rashid R, Tevanian A, Young M. Mach: A new kernel foundation for Unix development. *Proceedings of Summer USENIX Conference*, 1986; 93–112.
42. Rozier M *et al.*, CHORUS distributed operating systems. *Computing Systems* 1988; **1**(4):305–367.
43. Marsh BD, Scott ML, LeBlanc TJ, Markatos EP. First-class user-level threads. *ACM Operating Systems Review* 1991; **25**(5):110–121.
44. Anderson TE, Bershad BN, Lazowska ED, Levy HM. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computing Systems* 1992; **10**(1):53–79.
45. Assenmacher H, Breitbach P, Buhler P, Hübsch V, Peine H, Schwarz R. Meeting the application in user space. *Proceedings of 6th ACM SIGOPS European Workshop*, 1994; 82–87.
46. Engler DR, Kaashoek MF, O'Toole J. Exokernel: An operating system architecture for application-level resource management. *ACM Operating Systems Review* 1995; **29**(5):251–266.
47. Gheith A, Schwan K. CHAOSarc: Kernel support of multiweight objects, invocations and atomicity in real-time multiprocessor applications. *ACM Transactions on Computer Systems* 1993; **11**(1):33–72.
48. Bershad BN, Lazowska ED, Levy HM. Presto: A system for object-oriented parallel programming. *Software–Practice and Experience* 1988; **18**(8):713–732.
49. Hutchinson NC, Mishra S, Peterson LL, Thomas VT. Tools for implementing network protocols. *Software–Practice and Experience* 1989; **19**(9):895–916.
50. Campbell RH, Johnston GM, Russo VF. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review* 1987; **21**(3):9–17.
51. Shrivastava SK, Dixon GN, Parrington GD. An overview of the Arjuna distributed programming system. *IEEE Software* 1991; **8**(1):66–73.
52. Mukherjee B, Schwan K. Experimentation with a reconfigurable microkernel. *Proceedings of USENIX Symposium on Microkernels and other Kernel Architectures*, San Diego, CA, 1993; 45–60.

53. Philbin J. Customizable policy management in the Sting operating system. *Lecture Notes in Computer Science* 748; *Proceedings of the US/Japan Workshop on Parallel Symbolic Computing: Languages, Systems and Applications*, 1992; 380–401.
54. Rosenberg J, Dearle A, Hulse D, Linström A, Norris S. Operating system support for persistent and recoverable computations. *Communications of the ACM* 1996; **39**(9):62–69.
55. Hulse D, Dearle A. Trends in operating system design: towards a customisable persistent micro-kernel. University of Stirling, *Technical Report Pastel RT1R4*, 1998.
56. Dearle A, Hulse D. Operating system support for persistent systems: past, present and future. *Software–Practice and Experience* 2000; **30**(4):295–324.
57. Munro DS, Connor RCH, Morrison R, Scheuerl, S, Stemple D. Concurrent shadow paging in the Flask architecture. *Persistent Object Systems*; Atkinson MP, Maier D, Benzaken V (eds.), *Proceedings of the 6th International Workshop on Persistent Object Systems (POS6)*, Tarascon, France, 1994; 160–42.
58. Davie AJT, McNally DJ. Statically typed applicative persistent language environment (STAPLE) reference manual. University of St Andrews, *Technical Report CS/90/14*, 1990.
59. Cardelli L. Typeful programming. DEC Systems Research Center, *Technical Report 45*, 1989.
60. Kirschke H. *Persistenz in Objekt-Orientierten Programmiersprachen*; Logos Verlag: Berlin, 1997.
61. Rabitti F *et al*, Design and implementation of PIOS: a physically independent object server. ESPRIT BRA Project 6309 FIDE$_2$ *Technical Report FIDE/93/70*, 1993.
62. Stemple D, Morrison R. Specifying flexible concurrency control schemes: an abstract operational approach. *Proceedings of 15th Australian Computer Science Conference*, Hobart, Tasmania, 1992; 873–891.
63. Garcia-Molina H, Salem K. Sagas. *ACM SIGMOD Record* 1987; **16**(3):249–259.
64. Mayes KR, Quick S, Warboys BC. User-level threads on a general hardware interface. *Operating Systems Review* 1995; **29**(4):57–62.
65. Kirby GNC, Connor RCH, Cutts QI, Dearle A, Farkas AM, Morrison R. Persistent hyper-programs. *Persistent Object Systems*; Albano A, Morrison R (eds.), *Proceedings of the 5th International Workshop on Persistent Object Systems (POS5)*, San Miniato, Italy, 1992; 86–106.
66. Connor RCH, Cutts QI, Kirby GNC, Moore VS, Morrison R. Unifying interaction with persistent data and program. *Interfaces to Database Systems*; Sawyer P (ed.). *Proceedings of the 2nd International Workshop on User Interfaces to Databases*, Ambleside, Cumbria, 1994; 197–212.
67. Kirby GNC. Persistent programming with strongly typed linguistic reflection. *Proceedings of 25th International Conference on Systems Sciences*, Hawaii, 1992; 820–831.
68. Morrison R. On the development of Algol. *PhD Thesis*, University of St Andrews, 1979.
69. PS-algol Reference Manual, 4th edition; Universities of Glasgow and St Andrews, *Technical Report PPRR-12–88*, 1988.
70. Morrison R, Connor RCH, Cutts QI, Kirby GNC, Stemple D. Mechanisms for controlling evolution in persistent object systems. *Journal of Microprocessors and Microprogramming* 1993; **17**(3):173–181.