An Approach to Compliance in Software Architectures

Ron Morrison†, Dharini Balasubramaniam†, Mark Greenwood¥, Graham Kirby†,
Ken Mayes¥, Dave Munro* & Brian Warboys¥

†School of Computer Science, University of St Andrews
¥Department of Computer Science, University of Manchester
*Department of Computer Science, University of Adelaide

Abstract

Conventional software architectures are designed to meet the average predicted needs of the majority of applications that use them. By contrast, a compliant software architecture can accommodate the needs of a particular application. As the application evolves, its requirements change and the supporting software components, if compliant, change accordingly to meet these new requirements. The degree of compliance can be measured by the goodness of fit of the supporting architecture to the evolved application.

In this paper we introduce the concept of compliance and show how it may be useful in software systems. We illustrate some component technologies that form the basis for implementing compliance and finish with an example of compliance in operation.

1 Introduction

Software that cannot evolve is condemned to atrophy. Despite this, the most successful approach to constructing software systems has been to use components that are irrevocably glued together by some binding mechanism. Once bound, the constructed system cannot easily evolve unless the user has specifically programmed in facilities for evolution and even then these facilities cannot accommodate circumstances unanticipated at the time of construction.

Of course, evolution may always be achieved by retaining the source code and evolving it by editing, re-compiling and re-binding. Thus components may be re-defined, compiled and linked into a new version of the software. However, this may not always be acceptable especially in large or long-running systems nor indeed where the source code is no longer available or perhaps not compatible with the running system. Perhaps more importantly local encapsulated data generated during the computation may be lost by this approach of evolution. Examples of this style of software construction are systems made up of CORBA [1] or DCOM [2] components, and applications that use a fixed database interface which in turn uses a fixed programming language interface built upon a fixed operating system interface. Systems constructed from MS Access/Visual Basic/Windows NT or Oracle/C/Unix fall into this latter category.

The success in this conventional approach to constructing software is in its re-use capabilities. CORBA and DCOM components may be used multiple times in different applications and the Unix and Windows NT interfaces have been the basis of stable application platforms. By a strange coincidence these stable interfaces, which provide the levels of abstraction, also hide the very information that may be required for evolution. For example, a user of a database system may be able to construct new indexes over the data but is unlikely to be able to change the indexing algorithm used by the database system, should that be a requirement of the application evolution. Thus there is a tension between the manner in which we use components for reuse and the way in which we use them for evolution.

Another view of our traditional approach to software construction is gained by considering the separation of policy and mechanism. The interfaces that components provide for users define their mechanisms. Thus the mechanisms in Unix are the system calls and the mechanisms for a programming language are the programs that may be written. The components also have policy that is hidden behind the interface, such as a page replacement algorithm in an operating system or the scheduling mechanism for threads in a programming language.

Each of the components in a conventional architecture is designed to meet the average predicted needs of the majority of uses. Thus the policy algorithms of a component will be tuned according to the benchmarks designed to predict the profile of programs commonly using the component. As a consequence, different architectural components will contain repetition and duplication of both mechanism and policy. The repetition occurs when the same policies/mechanisms are implemented in different components; duplication occurs when the policies/mechanisms in different components are different and may interact in an undesirable manner. For example, the persistent store manager policies of an operating system and a database that runs on the operating system may move pages in conflict with one another. The complexity arising from repetition and duplication makes it difficult to predict how the system will perform, particularly in terms of failure semantics and run-time efficiency.

We define a compliant architecture as one that accommodates the needs of a particular application [3]. The degree of compliance can be measured by the goodness of fit of the application to the architecture, under any specific criteria. The question is: can we do this without losing the powerful abstraction mechanism of defining components with interfaces which is so useful for re-use?

Clearly an architecture that contains repetition and duplication of policy and mechanism will be not be very compliant. However, the compliance of an architecture may be improved by evolving it to the changing requirements of the application. Thus the underlying strategy for implementing compliance is similar to, if not the same as, that required for system evolution.

Our goal is to take the next step in the construction of software architectures. This entails techniques and methodologies that will allow software to evolve to its own and externally changing requirements – co-evolution in terms of business systems. As such we will be able to construct software architectures that are compliant to the needs of individual applications.

2 Composition and Decomposition

An essential property of evolution is the ability to decompose a system and recompose it in an evolved state. We define the composition of a system S from components P and Q under rule ® as[1]

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

The $\Rightarrow$ symbol represents the binding mechanism, such as: a #include mechanism, a symbolic linker or a dynamic binding mechanism. The binding mechanism operates under the rules ®. For example, a compiler may merge a number of source files with the program to be compiled. The merging facility is the binder and the rules under which it operates determines the order and scoping of the included files. Other examples of this form of composition include the linking of .o files in the C linker and the dynamic loading of classes in Java [4]. In most current computer systems $\Rightarrow$ is hard wired and ® implicit.

---

[1] This can be generalised in a straight forward way to an arbitrary number of components.

Composition becomes more interesting when the components being combined are active. In a transaction system we can think of P and Q as being the transactions themselves, the binding mechanism being the part of the system that allows transactions to run together and the rules under which they are combined being the ACID properties.

While composition allows for the construction of software architecture it does not provide the necessary infrastructure for system evolution. For that the composition process has to be reversible. This we define as

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

The $\Leftrightarrow$ symbol represents a reversible compose/decompose mechanism under the rules ®. The fact that such compose/decompose operators are not common in our programming systems illustrates the lack of general facilities for evolution. Again compose/decompose is most interesting when the components are active. For example, we can think of two long running threads, such as occur in workflow systems, that are bound together under some synchronisation rules. The threads decide that they wish to change the synchronisation rules without losing any of their internal or shared data. With the above compose/decompose mechanism they may separate themselves into the components and the synchronisation rules, and rebind under a new rules set. We will see an example similar to this later in the paper.

The need for compose/decompose to support evolution has some unexpected consequences. It means that in order to evolve an object it may be necessary to break encapsulation to view its inner components and that active systems must be able to separate into components without losing local or shared state. The challenge is therefore to find the form of components, binding rules and compose/decompose operators that will provide these facilities.

3 Component Technologies for Compliance

From the above arguments we can see that for evolution to be completely flexible the components, the composition rules ®, and the compose/decompose operation $\Leftrightarrow$ must all have the ability to evolve. While we concentrate here on components, composition rules and operators we recognise that facilities for introspection are also required. That is, as well as being able to decompose a system the user needs a concrete representation for the results of the decomposition. When introspection is combined with the specific binding mechanism of compilation the technique known as linguistic or structural reflection results [5].

Since composition and decomposition depend on the form of the components and the nature of the rules ® we will concentrate on what look like promising technologies for the latter two. We will assume that introspection in the system is available and that structural reflection can be used to generate evolved systems. It will become obvious from our description how decomposition is achieved. We introduce the following technologies to meet the above requirements:

- Hyper-code for describing components
- The Communicating Action Control System (CACS) for capturing the rules

3.1 Hyper-code

The first requirement of a system description for evolution is the ability to represent active objects with shared values. Traditional source code cannot do this since it has no facility to denote shared values or structures. Hyper-programs, which were developed in the context of persistent systems, do have this facility and therefore show promise in a solution to the evolution problem.

Traditionally, a program which accesses another potentially shared object during its execution contains a name for the object. The name may be a file name (or some object in an object store) to be resolved at run-time by the file (or object) manager, or it may be as simple as a variable name to be resolved using the scope rules of the language during computation.

In a persistent programming system, programs may be constructed and stored in the same environment as that in which they will subsequently be executed. This means that objects accessed by a program may already exist when the program is composed. Direct links to the objects can be included in the program rather than textual descriptions of the access paths by which they can be located at execution time. A program containing both text and links to objects is a hyper-program [6].

The example of persistence may be extended to any run-time system. As such, at any point in the execution of the system, presumably the point at which evolution is required, some names (access paths) will have been resolved into values, and can be represented by links, and other will not. The point is that since the links are pointers different ones may point to the same values and thus represent sharing. An example of a hyper-program is given in Figure 1 where there are links to the method *Person.marry* and the objects *vangelis* and *mary*.

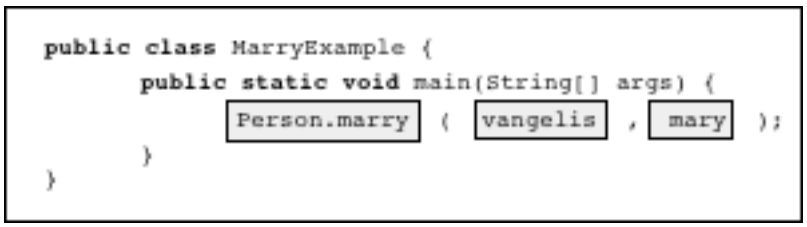

Figure 1: An example of a hyper-program

The hyper-program idea has been further developed to that of hyper-code, in which the distinction between source programs and executable code is completely hidden by a user interface that presents a single uniform view of all software entities throughout their lifetimes [7, 8]. That is, the user composes hyper-code and the system executes it. If a error occurs during this process the user only ever sees a hyper-code representation of the program, which may be partially executed. There are a number of benefits of hyper-programming and hyper-code. The most significant here are:

- that it is possible to provide full source representations of all programs, including those that may, due to the context in which they were defined, contain references to other existing data;
- and that because of this, it is possible to maintain automatically associations between all executable programs and the source programs from which they were derived.

Both of these points are important in relation to decomposition. The automatic associations mean that the original source code that defined a composition will never be lost, while the ability of the source code to contain embedded references to extant data and code makes it possible to fully represent the state of the composed unit at the time of the decomposition, even if it is during execution.

We will see shortly how hyper-code may be used in defining compose/decompose operations.

3.2 CACS

The Communicating Actions Control System (CACS) is an abstract operational model designed to allow the specification of coherency protocols for accessing shared data [9]. The CACS model consists of actions (computations) that access objects (shared data).

A particular coherency protocol, for example atomic (ACID) transactions, is defined by a set of *significant events* and a set of *rules*. The significant events specify the operations on shared data that need to be coordinated by the protocol. The rules specify the details of this coordination. For example, consider the atomic transaction protocol:

- The significant events are *begin*, *commit*, *abort*, *read* and *write*.
- The rules give an operational specification of how the ACID transaction properties are to be enforced.

As it runs, each action generates a sequence of significant events, which are handled by the CACS *controller*, according to the rules. Each rule specifies what the controller should do in response to a particular significant event. In addition to performing arbitrary computation, the controller may suspend and resume actions, and generate additional, synthetic events that are added to the incoming event stream.

The CACS architecture gives considerable flexibility both in defining new coherency protocols, and in varying the policy implementing a particular protocol. For example, both optimistic and pessimistic flavours of atomic transactions could be defined in a similar way [10]. The significant events would be the same in each case, but the bodies of the rules would vary. For an optimistic scheme the controller would allow actions to read and write shared data without restriction, recording which data objects had been accessed. On a *commit* event, the controller would test for conflict with other transactions, and generate an *abort* event for each conflicting transaction. For pessimistic transactions the controller would suspend an action on the first attempted conflicting access to shared data.

In general it is not possible for the CACS system to deduce automatically the points in an action at which significant events are generated. The source program must thus be annotated to indicate these. In some special cases, however, this may be done automatically. For example with atomic transactions the system may deduce where *read* and *write* events occur, but not *begin*, *commit* or *abort*.

CACS specifications, in terms of events and rules, can be written for a wide variety of coherency protocols. These include traditional schemes such as atomic transactions, nested transactions [11], monitors [12] and Sagas [13], and more complex application-specific schemes. The programming involved in the correct implementation of these schemes can be defined and placed in a library for reuse. The writer of a CACS action thus does not have to write CACS specifications in cases where standard coherency protocols are sufficient, but has the flexibility to define new schemes if required.

The CACS specifications are sufficient to define the rules under which components interact ". As such we can use them in the composition and decomposition of systems.

4 Composition and Decomposition

Using hyper-code to represent system components and the CACS specifications to determine the binding rules, evolution can be supported by providing the operations *compose* and *decompose* [14, 15]. Software components are initially defined as *scripts*. A script is a piece of hyper-code together with a suspended thread, initially suspended at the start of the code.

The *compose* operation is used to spark a collection of scripts, returning an *activity*. As the script threads execute, the underlying system automatically modifies the corresponding hyper-code so that it always reflects the current thread states (this involves replacing textual identifiers by direct references to their values as they come into scope, updating those values on assignment, and restoring the textual identifiers as they leave scope and cease to be defined). Compose is defined by

```
compose : fun (script[], rule[] → activity)
```

It should be noticed that the activity is composed from both scripts and a CACS rule set.

The *decompose* operation takes an activity and returns the scripts with their threads suspended together with the rule set. As outlined above, each script will fully reflect its current state at the time of decomposition. A *compose* directly following a *decompose* restarts the scripts from their suspension point as if the *decompose* had never occurred. It is also possible however to evolve the activity by editing the scripts before reactivating them. Decompose is defined by

```
decompose : fun (activity → script[], rule[])
```

In addition to being able to design coherency protocols suitable for a particular application from the outset, it is possible to evolve a protocol dynamically. The CACS specification is supplied, as a first-class entity, to the *compose* operation when an activity is initially created. It then forms part of the state of the component scripts: if the activity is subsequently decomposed, the scripts may be edited in order to replace the initial protocol specification with a new one. The new protocol will then control the interaction of the scripts when they are reactivated.

Combining hyper-code, the CACS rules and the compose/decompose functions above yields the facilities for evolution and thus compliance that we seeking. An active system may be decomposed into its components. The components, the rule set and even the compose function may then be evolved to meet the new requirements of the software. All of this may be achieved without losing local or shared contexts.

5 An Example of Compliance

Our example of compliance concerns two engineers working on a shared document, a drawing perhaps, but wishing to keep separate all their other calculations and data. In order to keep their designs consistent the engineers have agreed a protocol that determines when the shared diagram is in a consistent state and may be committed. The protocol has been specified as a set of CACS rules[2].

As the design is completed the engineers wish to disentangle themselves and operate in isolation from one another within a system of atomic transactions. The evolution of the protocol is non-trivial since it entails altering all the read and writes from and to the persistent data. It also has to redefine the meaning of *commit*. For our purposes it is sufficient to know that these protocols can be written and we will assume that they have been predefined.

The initial state of the system shows two engineers *eng1* and *eng2* (the components in hyper-code), a set of CACS rules *R* for shared commits and the shared document. The composition illustrates that the activation uses the shared state and the set of rules. Figure 2 depicts the system.

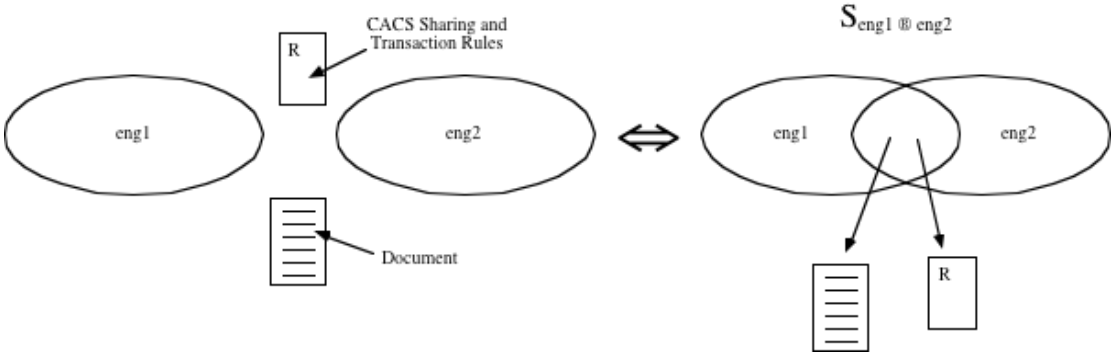

Figure 2: System composition

[2] The rules themselves have been omitted here since the protocol is non-trivial and the details of the CACS system would have to be described to understand the rules.

After some computation and the completion of the design, Figure 3 depicts the decomposition of the system into its components: the engineers, the shared state and the CACS rules.

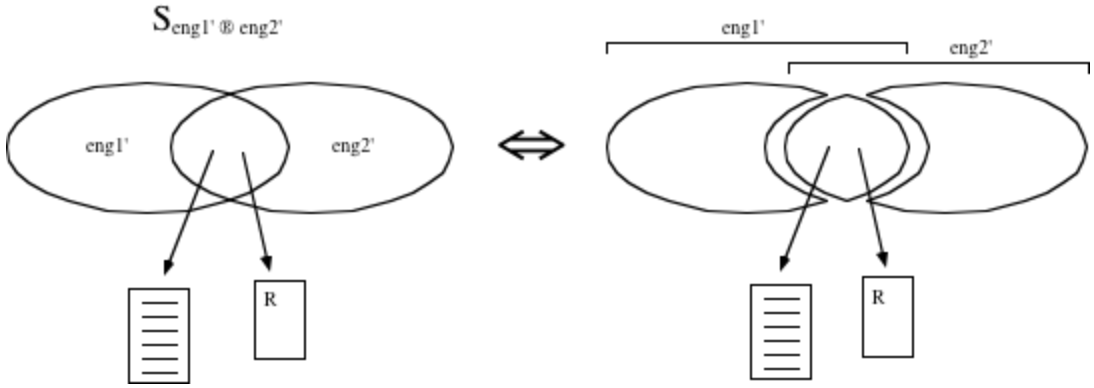

Figure 3: System decomposition

When decomposed the system may be evolved with a different set of rules. For this to be consistent the hyper-code that represents the engineers will also have to be changed and the system composed again to complete the evolution. This is shown in Figure 4.

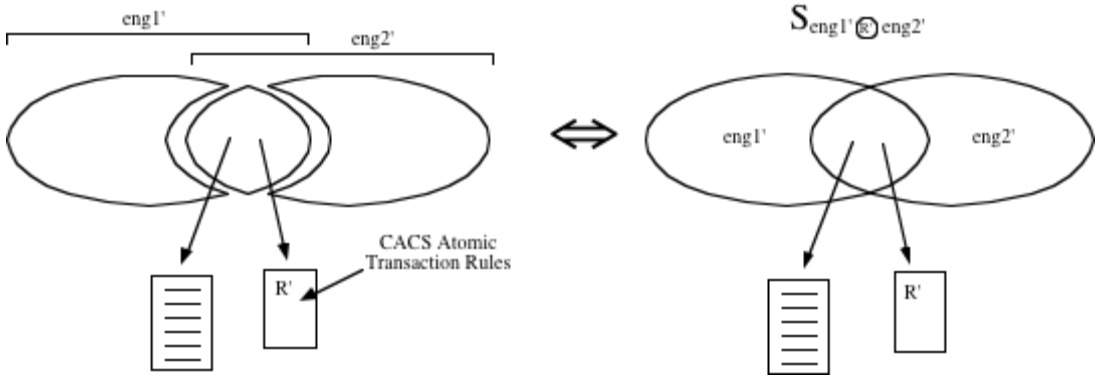

Figure 4: System evolution

The final system now only allows access to the design diagram under the ACID transaction rules. The example shows how the system can evolve using the hyper-code and CACS technologies thereby making the architecture more compliant to the changing needs of the application.

6 Conclusions

Our goal is to develop software architectures that are compliant to the changing needs of application systems. We have demonstrated that compliance requires the same technologies for evolution in software systems. For that, the composition process must be reversible into the base components and the rules under which the binding took place. The problem is most interesting when all of the components, the binding rules and the compose/decompose operations can evolve in the presence of active executions with shared resources such as data.

To illustrate our ideas we have shown how two technologies: hyper-code and the CACS system may be used in conjunction to be the basis of dynamic evolution.

7 Acknowledgements

This work is supported by EPSRC Grants GR/M88938 and GR/M88945 both entitled "Compliant Systems Architecture".

8 References

[1] "The Common Object Request Broker: Architecture and Specification, Revision 2.3.1". Object Management Group (OMG) (1999).

[2] Microsoft Corporation "DCOM Technical Overview". (1996).

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

[4] Gosling, J., Joy, B. & Steele, G. "The Java™ Language Specification". Addison-Wesley (1996).

[5] Kirby, G.N.C., Morrison, R. & Stemple, D.W. "Linguistic Reflection in Java". Software - Practice & Experience 28, 10 (1998) pp 1045-1077.

[6] Morrison, R., Connor, R.C.H., Cutts, Q.I., Dunstan, V.S. & Kirby, G.N.C. "Exploiting Persistent Linkage in Software Engineering Environments". Computer Journal 38, 1 (1995) pp 1-16.

[7] Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Moore, V.S. & Morrison, R. "Unifying Interaction with Persistent Data and Program". In **Interfaces to Database Systems**, Sawyer, P. (ed), Springer-Verlag (1994) pp 197-212.

[8] Zirintsis, E., Kirby, G.N.C. & Morrison, R. "Demonstration of Hyper-Programming in Java". In Proc. 25th International Conference on Very Large Databases (VLDB'99), Edinburgh, Scotland (1999) pp 734-737.

[9] Stemple, D. & Morrison, R. "Specifying Flexible Concurrency Control Schemes: An Abstract Operational Approach". In Proc. 15th Australian Computer Science Conference, Hobart, Tasmania (1992) pp 873-891, Technical Report ESPRIT BRA Project 3070 FIDE FIDE/92/35.

[10] Eswaran, K.P., Gray, J.N., Lorie, R.A. & Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System". Communications of the ACM 19, 11 (1976) pp 624-633.

[11] Moss, J.E.B. "Nested Transaction: An Approach to Distributed Computing". MIT Press, Cambridge, Massachusetts (1985).

[12] Hoare, C.A.R. "Monitors: An Operating System Structuring Concept". Communications of the ACM 17, 10 (1974) pp 549-557.

[13] Garcia-Molina, H. & Salem, K. "Sagas". ACM SIGMOD Record 16, 3. Proc. ACM SIGMOD Annual Conference, San Francisco, California (1987) pp 249-259.

[14] Warboys, B.C., Balasubramaniam, D., Greenwood, R.M., Kirby, G.N.C., Mayes, K., Morrison, R. & Munro, D.S. "Instances and Connectors: Issues for a Second Generation Process Language". In **Lecture Notes in Computer Science 1487**, Gruhn, V. (ed), Springer-Verlag (1998) pp 137-142.

[15] Warboys, B.C., Balasubramaniam, D., Greenwood, R.M., Kirby, G.N.C., Mayes, K., Morrison, R. & Munro, D.S. "Collaboration and Composition: Issues for a Second Generation Process Language". In **Lecture Notes in Computer Science 1687**, Nierstrasz, O. & Lemoine, M. (ed), Springer-Verlag (1999) pp 75-91.