Constructing Active Architectures in the ArchWare ADL

[1]Ron Morrison, [1]Graham Kirby, [1]Dharini Balasubramaniam, [1]Kath Mickan, [2]Flavio Oquendo,
[2]Sorana Cîmpan, [3]Brian Warboys, [3]Bob Snowdon, [3]R Mark Greenwood

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

Abstract

Software that cannot change is condemned to atrophy: it cannot accommodate the constant revision and re-negotiation of its business goals nor intercept the potential of new technology. To accommodate change in such systems we have defined an active software architecture to be: dynamic in that the structure and cardinality of the components and interactions are not statically known; updatable in that components can be replaced dynamically; and evolvable in that it permits its executing specification to be changed.

Here we describe the facilities of the ArchWare architecture description language (ADL) for specifying active architectures. The contribution of the work is the unique combination of concepts including: a π-calculus based communication and expression language for specifying executable architectures; hyper-code as an underlying representation of system execution; a decomposition operator to break up and introspect on executing systems; and structural reflection for creating new components and binding them into running systems.

1 Introduction

Software architectures [1, 2] describe systems in terms of their components and interactions between components. We define an active software architecture to be: dynamic in that the structure and cardinality of the components and interactions are not statically known; updatable in that components can be replaced dynamically; and evolvable in that it permits its executing specification to be changed.

Active architectures address problems of co-evolution in dynamically changing commercial environments where business changes create pressures on the software to evolve, and at the same time technology changes create

pressures on the business to evolve. The business effects of introducing, or changing, such software systems are often emergent and require their software architecture models to accommodate their demands by being dynamic, updatable and evolvable.

Figure 1 shows an evolving system. At the initial stage (a), the system is composed of three components of one kind (say dynamic clients) interacting with one component of another kind (say server) that has access to some data. At stage (b), this system has been decomposed to yield the individual components with the server still maintaining its access to the data. The next stage (c) sees the components evolved so that we have three clients and two servers both of which maintain the access to the shared data. Finally at stage (d) a new evolved system is formed by composing the five components so that one client interacts one server and the other two clients interact with the other server.
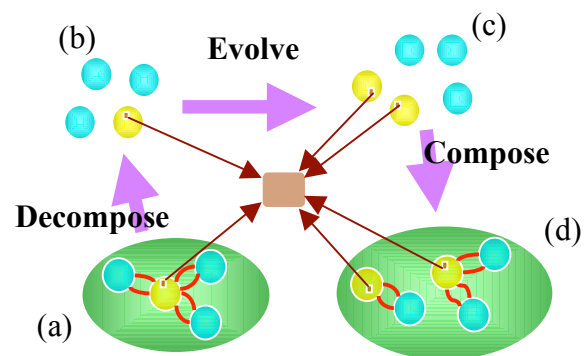

Figure 1. An evolving system

This paper describes the ArchWare Architecture Description Language [3] which is sufficiently rich to provide executable specifications of active systems. We define a core language on which architectural styles can be layered and on which a construction methodology can be applied. Our focus is on the base technologies required

to support dynamic and evolvable systems and we present examples of how the ADL may be used to model active architectures.

2   Related work

Hitherto researchers have proposed many formal ADLs for representing and analysing architectural designs. Adage [4] supports the use of architectural frameworks in the avionics industry; Aesop [5] has architectural styles; Meta-H [6] has specific guidance for real-time avionics control software; SADL [7] provides a formal basis for architectural refinement; C2 [8] supports the description of user interface systems; Wright [9] supports the specification and analysis of interactions and UniCon [10] supports a mixture of heterogeneous component and connector types.

Containment Units [11] provide a mechanism for dealing with anticipated change. ArchStudio [12] is a tool suite that supports architecture-based development. Changes are made to an architectural model and then reified into implementation by a runtime architecture infrastructure. In [13], specific component managers identify external architecture changes by listening to events, and then react in order to preserve architecture constraints. The constraints themselves cannot be evolved. Each ADL has its own focus according to needs and taste with little integration of the overlapping ADL concepts. ACME [14] is an attempt at such integration but does it at the level of a lowest common denominator.

All of the above ADLs work on the specification and enactment model where the formal properties of the system are specified, analysed and then executed. The focus of the ArchWare project [15] is in evolving systems, with emergent properties, where the system is executing continuously. The model specification and the model enactment are both regarded as part of the executing state. At any point in time the model specification will be an accurate description of the model execution. Our claim is that such systems have active architectures and a paradigm shift in ADLs is required to accommodate them.

3   Change in active architectures

We have identified three architectural kinds of change in active architectures. These are:

- Dynamic change: allows the topology of the components and interactions to be determined dynamically. New components and interactions may be created during execution.
- Update change: allows components to be replaced dynamically. Whereas dynamic change is additive, update change may be regarded as subtractive and then additive (atomically).

- Evolutionary change: allows the specification of the components and interactions to be changed during execution.

There are two main stages to changing active architectures: first deciding which changes are required and when, and second making the changes. The first recognises that all change is made in response to some stimulus. Taken from control systems, Figure 2 shows how application knowledge, obtained by measurement perhaps, is used in predicting a goal and then causing a reaction. The reaction may be to continue execution or to initiate some change mechanism.
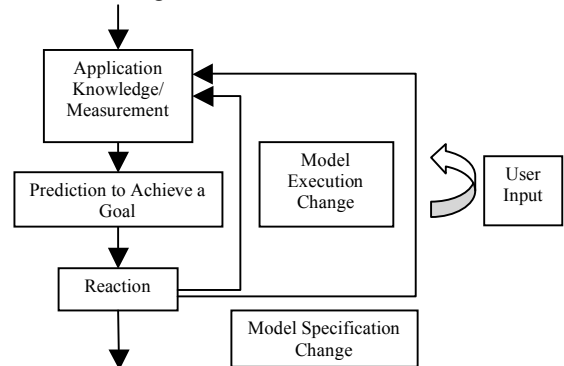

Figure 2. Change mechanisms

In an active architecture the specification of the architectural model changes in lock-step with the model execution. Thus changes during execution will change the specification and changes to the specification will affect the execution. However at any time in the execution of the model the specification is dynamically up-to-date.

In the ArchWare ADL we have provided a number of change mechanisms reflecting our estimate of the frequency of the expected type of change. Dynamic change and update change are made using mechanisms built into the specification language. For example, this may be adding more components to execute in parallel or passing components to other components through connections (dynamic change) or replacing a component from a library of parts by assignment (updatable change).

Evolutionary change is characterised by changing specifications and requires a reflective system. In this that part of the system to be changed is stopped, its specification altered and the new specification enacted in the executing model. Evolutionary change may be implemented using introspection on the component to be changed to yield its specification, and reflection to rebind the new specification. The reflection itself can use existing specifications to make the alterations. In the limit user input may be necessary to provide for change that is unexpected.

4   ArchWare overview

This work is undertaken within the EC funded ArchWare software architecture framework. The ArchWare project takes a holistic view of software development. Its aims are to advance and integrate research on software architecture and reflective systems to develop languages, frameworks and tools for architecting and engineering dynamic and evolvable software systems. The important aspects of the ArchWare approach can be described as follows:

- A formal, style-based, executable architecture description language to describe architectural structure, behaviour, qualities and evolution of systems
- A suite of tools based on the ADL for architecture design and analysis
- Run-time and environment framework to support the development and deployment of software systems and coordination of design and analysis tools
- Generic and customisable process models for evolutionary, architecture-centric development of software systems

Here we concentrate on the required environment and base technologies within the ArchWare ADL for dynamic expression and evolution. These include the following:

- A formal foundation based on higher-order π-calculus [16] for specifying components of architectures with dynamic structure and cardinality
- Integration of a π-calculus based language for communication and an expression based language to yield executable specifications
- Hyper-code as a representation for system execution to support reification
- A *decompose* operator essential to break up active systems into their components prior to evolution and recomposition
- Structural reflection to support evolution

The theoretical foundation enables formal analysis of the architecture and proof of its desired properties. Executable specifications reduce the cost and complexity of separately implementing the corresponding software systems. The separation of concerns of co-ordination, communication and computation of components make the system easier to understand and evolve. Dynamic expression is built into the constructs of the language. The facilities for composition and reification together with support for decomposition and reflection enable evolving systems.

5   The ArchWare ADL

The ArchWare ADL is the simplest of a family of languages designed for modelling active software architectures based on the concepts of π-calculus, persistent programming and dynamic system composition and decomposition.

The ArchWare ADL is a strongly, and mostly statically, typed persistent language. The ADL system consists of the language and its populated persistent environment and uses the persistent store to support itself. To model the component and communication algebra, the ADL supports the concepts of behaviours, abstractions of behaviours and connections between behaviours. Communication between components, represented by behaviours, is via channels, represented by connections. For expressing data the language also supports a number of data types: integer, boolean, real, string, locations, views, sequences and higher order functions. These can be regarded as syntactic sugar since they can all be encoded in the π-calculus. The language also supports all the basic π-calculus and expression based operations as well as composition and decomposition.

The ArchWare ADL is designed using the three principles of abstraction, correspondence and type completeness [17, 18, 19].

- The principle of *abstraction* allows abstractions over every semantically meaningful syntactic category in the language. Thus functions are abstractions over expressions.
- The principle of *correspondence* states that the rules for introducing and using names should be the same throughout. In particular there should be a one-to-one correspondence between introducing names in declarations and as parameters.
- The principle of *type completeness* states that the rules for using data types must be complete with no gaps. For example, general rules for type constructors should have no exceptions.

The application of these design rules yields languages that are both small in the number of concepts and powerful. They are small in that there are no exceptions to the rules and powerful since every combination is valid. These properties are important in the design of hyper-code and its programmable interface.

5.1   The ArchWare ADL layers

The ArchWare ADL is designed using a layered approach with the above-mentioned principles of programming language design guiding the process. Layering the language helps to separate different concerns. There are currently three layers to the ArchWare ADL.

The base layer (Base) defines a coordination language without any data values. This corresponds to non-higher-order monadic π-calculus. Connections are provided for communication but since there are no values to be communicated, communication merely provides

coordination. Thus there is no mobility at the base layer. Principles of abstraction and correspondence trivially apply at this level; without any data types, the principle of type completeness is not applicable. This layer is already dynamic though the provision of the *replicate* operator.

The first-order layer (FO) builds on the base layer to provide data values, behaviours, abstractions over values and behaviours (called functions and abstractions respectively) and mobility for data values and connections. This layer corresponds to non-higher-order polyadic π-calculus. The principles of abstraction and correspondence apply at this level. The principle of type completeness applies to all values except abstractions and behaviours since these are not permitted to be communicated via connections.

The higher-order layer (HO) develops the first-order layer to add mobility for abstractions and behaviours. Thus this layer corresponds to higher-order polyadic π-calculus. All three principles of programming language design apply at this level without exception. It is this layer that we present in this paper.

5.2   The ArchWare ADL type system

The ArchWare ADL type system is based on the notion of types as a set structure imposed over the value space. Membership of the type sets is defined in terms of common attributes possessed by values, such as the operations defined over them. These sets or types partition the value space. They may be predefined, like *integer*, or they may be formed by using one of the predefined type constructors, like *view*.

The constructors obey the principle of type completeness. That is, where a type may be used in a constructor, any type is legal without exception. This has two benefits. Firstly, since all the rules are very general and without exceptions, a very rich type system may be described using a small number of defining rules. This reduces the complexity of the defining rules. Secondly the type constructors are as powerful as is possible since there are no restrictions on their domain.

The universe of discourse of the ArchWare ADL can be described as follows. The following base types are defined:
1.   The scalar data types are *integer*, *real*, and *boolean*.
2.   Type *string* is the type of a character string; this type embraces the empty string and single characters.
3.   Type *any* is an infinite union type; values of this type consist of a value of any type together with a representation of that type.
4.   Type *behaviour* is the type of an executing process in the ADL.
  The following type constructors are defined:
5.   For any type T, *location [T]* is the type of a location that contains a value of type T.

6.   For any type T, *sequence[T]* is the type of a sequence with elements of type T.
7.   For identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, *view[I_1: $t_1,...,I_n$: t_n]* is the type of a view with fields I_i and corresponding types t_i, for i = 1..n and $n \geq 0$.
8.   For any types t and $t_1,...,t_n$, *function[$t_1,...,t_n$]* $\rightarrow t$ is the type of a function with parameter types t_i, for i = 1..n, where $n \geq 0$ and result type t. Functions abstract over expressions.
9.   For types $T_1, ..., T_n$, *connection[$T_1, ..., T_n$]* is the type of a connection (channel in π-calculus) which can send or receive values of types $T_1, ..., T_n$.
10. For any types $t_1,...,t_n$, *abstraction[$t_1,...,t_n$]* is the type of an abstraction with parameter types t_i, for i = 1..n, where $n \geq 0$. Abstractions abstract over behaviours.
  The world of data values is defined by the closure of rules 1 to 4 under the recursive application of rules 5 to 10.

Communication between components, represented by behaviours, is via channels, represented by connections. Abstractions over behaviours are called abstractions and may be parameterised by any data type. For example functions can be passed to abstractions and abstractions can be communicated over connections. Passing behaviours over connections yields mobility. All types can be regarded as syntactic sugar since they can all be coded within the π-calculus.

5.3   Control constructs in the ADL

The ArchWare ADL provides all of the usual control structures associated with expression based languages, namely sequence, choice, iteration, and function call including recursion. To model update change the ADL uses locations and assignment. Any data type may be stored in a location and be updated by a value of the same type.

Since the ArchWare ADL is formally based on the higher-order π-calculus it provides constructs analogous to those provided by the π-calculus for specifying control flow, communication and dynamic topology. The default execution pattern for behaviours in the ADL is parallel. In addition the ADL provides a rich set of control constructs.

Replication of a behaviour, indicated by ! in the π-calculus, is equivalent to a potentially infinite number of copies of that behaviour executing in parallel. This allows the specification of dynamic structure since replication generates copies as they are required. In Figure 3, the shown behaviour is replicated each time a value is received on connection *in_channel*. The behaviour waits at its reduction limit for input. Upon receiving input it creates a clone of itself waiting at the reduction limit, and

sends twice the received value on the *out_channel*. Many clones of the behaviour may be executing in parallel thus capturing dynamic topologies in the architecture, and supporting dynamic change.

```
replicate{
            via in_channel receive num ;
            via out_channel send 2 * num
        } ;
```

Figure 3. Replication

The *choose* clause, denoted by + in the π-calculus, allows the non-deterministic selection of one behaviour from two or more behaviours. In Figure 4, one of behaviours *client1*, *client2* or *client3* will be chosen at random by the run-time system.

```
value client1 = … ;
value client2 = … ;
value client3 = … ;
choose{        client1
    or        client2
    or        client3
} ;
```

Figure 4. Choice

Sequence, indicated by "." or *then* in the π-calculus, can be modelled by ";" in the ADL. Therefore in Figure 3, *num* will be received on *in_channel* by the behaviour before the output value is sent on *out_channel*.

The π-calculus also provides the facility to restrict names to processes. In the ArchWare ADL this restriction is enabled partly by block structured programming scope rules and partly by an explicit *free* construct that specifies the values to be available further.

5.4   Components

Software architectures describe systems in terms of their components and their interactions. Components are units of structure and functionality. In the ArchWare ADL components can be modelled by behaviours that are analogous to processes in the π-calculus. The code in Figure 3 specifies a server component that receives a number and sends back twice its value.

In order to facilitate design and reuse, the ADL allows the definition of abstractions that abstract over behaviours. Applying an abstraction results in a behaviour as illustrated in Figure 5.

```
value server = abstraction( )
{      via in_channel receive num ;
       via out_channel send 2 * num } ;
server( ) ; ! applies the abstraction to yield a behaviour
```

Figure 5. Abstraction

There are two aspects to the interaction between components: coordination and communication in the ADL. The former is concerned with synchronisation of components and the latter with exchange of data between components. Connections, analogous to channels in the π-calculus, are used by the ArchWare ADL for both aspects.

Behaviours can communicate, i.e. send and receive values, via connections if they share connections or if their connections have been explicitly unified. Empty messages via connections are used for coordination alone.

So far we have seen how to implement predictive change through *replicate* for dynamic change, and locations and assignment for update change. We now turn our attention to the facilities for unexpected evolutionary change – hyper-code, decomposition, reflection and reification.

6   Hyper-code

The *hyper-code* abstraction was introduced in [20] as a means of unifying the concepts of source code, executable code and data in a programming system. The motivation is that this may ease the task of the programmer, who is presented with a simpler environment in which the conceptually unnecessary distinction between these forms is removed. In terms of Brooks' *essences* and *accidents*, this distinction is an accident resulting from inadequacies in existing programming tools; it is not essential to the construction and understanding of software systems [21]. In a hyper-code system the user composes hyper-code and the system executes it. When evolving the system, for example because an error has occurred, the user only ever sees a hyper-code representation of the program, which may now be partially executed. The hyper-code source representation of the program is structured and contains text and links to extant values. Figure 6 shows an example of a hyper-code representation in the ArchWare ADL. The links embedded in it are represented by underlined tokens to allow them to be distinguished from the surrounding text. The first link is to an integer location value *count* that is used as a parameter in the application of the *server_abs* abstraction. The program also has two links to a previously defined abstraction *client_abs*. Hyper-code models sharing by permitting a number of links to the same value. Note that code values (*client_abs*) are denoted using exactly the same mechanism as data values (*count*). Note also that the value names used in this description have been associated with the values for clarity only, and are not part of the semantics of the hyper-code.

The importance of hyper-code in active architectures is that it is rich enough to represent executing code. Thus as the program executes, the hyper-code changes in line with the semantics of the language. Since hyper-code can represent closure, through sharing links, it may be used as a representation for introspection of the executing system.
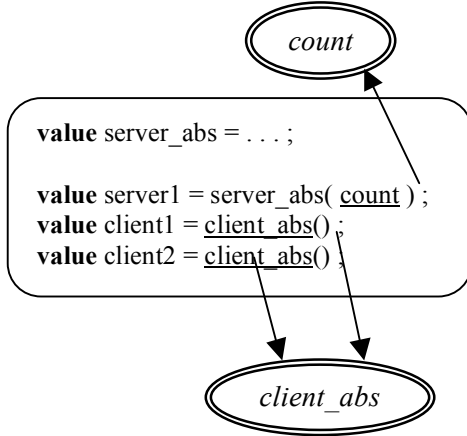

Figure 6. ADL hyper-code

7   Composition and decomposition

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

The ArchWare ADL provides a *compose* operator which operates over a number of behaviours (components) and returns a single handle to these behaviours executing in parallel. The result of this composition is also a behaviour. Thus hierarchical systems may also be modelled by components that are made up of other components. Figure 7 illustrates composition.

Behaviours *client* and *server* are composed to give *system*. The *as* construct permits meaningful labels to be associated with behaviours. As the details of *client* and *server* are not of interest in this example, the *position* sent by *server* is shown as a hyper-link which has previously been defined.

```
value channel_1 = connection() ;
value channel_2 = connection( string ) ;

value client = replicate{
            via channel_1 send ;
            via channel_2 receive pos : string } ;
```

```
value server = replicate{
            via channel_1 receive ;
            via channel_2 send position } ;

value system = compose{   pos_client as client
            and           pos_server as server }
```

Figure 7. Composition

The ADL also provides a *decompose* operator which breaks up a behaviour into its constituent behaviours and returns them in a suspended state. Figure 8 illustrates the use of the decomposition operator on the composition from Figure 7.

```
value pos_seq = decompose system ;

value client_val = pos_seq::1.bhvr ;
value server_val = pos_seq::2.bhvr ;
value comp1_label = pos_seq::1.label
```

Figure 8. Decomposition

Decomposition returns a sequence of views consisting of behaviours and their labels (if any) in the order that they were composed. All the behaviours are at their reduction limit for that composition. Figure 8 shows how these behaviours and their labels may be accessed from the sequence. These behaviours can be returned to the user as hyper-code, modified and recomposed.

Explicit composition is not normally required since two behaviours will communicate if they share the same connection and communication is ready. Composition gives a handle to the new composed behaviours. The higher order nature of the language means that two behaviours that have been sent to a third may wish to communicate but do not share the same connection value. The *compose* operation has a variant that allows the unification of connection values during composition to facilitate communication in these circumstances. Figure 9 shows an example of unification.

```
value client = abstraction()
{   value out_request = connection() ;
    value in_reply = connection( string ) ;
    replicate{
                via out_request send ;
                via in_reply receive pos : string }
} ;

value server = abstraction()
{   value in_request = connection() ;
    value out_reply = connection(string) ;
    replicate{
                via in_request receive ;
                via out_reply send position }
} ;

value system =
        compose{  pos_client as client() and
                  pos_server as server()
          where {  pos_client::out_request unifies
                        pos_server::in_request,
                    pos_client::in_reply unifies
                        pos_server::out_reply } }
```

Figure 9. Unification

8   Reflection and reification

A hyper-code system may be thought of as operating within two abstract domains: entities (E) and representations (R). E contains all the first class values defined by the language while R contains the concrete representations of the values in E. Given these domains, four domain operations over E and R may be defined.

- *reflect* maps a representation to its corresponding entity (R $\Rightarrow$ E)
- *reify* maps an entity to a corresponding representation (E $\Rightarrow$ R)
- *execute* executes an entity, possibly generating a result (E $\Rightarrow$ E)
- *transform* maps one representation onto another (R $\Rightarrow$ R)

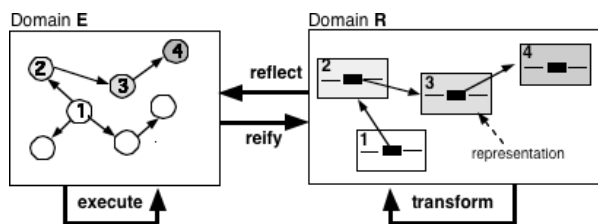Figure 10 illustrates the domain operations.

Figure 10. Domain operations

Reflection and reification are of particular interest for this paper. In the context of Figure 1, it may be seen how these operations play a vital role in evolving systems. Once a system is decomposed, reification allows us to view the representations of its components. These representations may be evolved to capture new requirements. Reflection allows the evolved or new components to be bound back into the system.

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

9   A change example

The concepts discussed in earlier sections are now illustrated using an example written in the ArchWare ADL. This example is based on a long running *in silico* experiment [22]. The user is a scientist who wants to run the experiment from a desktop-based client. The experiment itself runs on a server machine with access to a range of resources, e.g. corporate databases. The scientist's client will connect to a server that disseminates data about the status of the experiment. The client also allows the scientist to control (start and end) the experiment. Initially there will be a single client and a single server, but later the system will be evolved so that the functionality of the server is split into two.

The functionality of the server and the client can be modelled as abstractions in the ArchWare ADL. When applied, these abstractions yield executing behaviours. Such behaviours are the components that make up the client-server system. The repetitive nature of both the client and the server is captured using replication. Thus the dynamic nature of the system is already present in that the server may replicate itself to deal with data from the experiment and the client many replicate itself in parallel to react to the data sent by the server.

Since exact details of the experiment are not of interest to this example, we will assume that values of data type *exp_view* provide all necessary information about the experiment. The client abstraction can then be defined as shown in Figure 11.

```
! client
value client_abs = abstraction()
{   value c_start = connection() ;
    value c_stop = connection() ;
    value c_get = connection( exp_view ) ;
    via c_start send ;
    replicate
        choose{
            {   via c_get receive ev : exp_view ;
                via c_display send ev }
         or
```

```
        {   via user_input receive ;
            via c_stop send } }
}
```

Figure 11. The client abstraction

The client defines the connections it needs to communicate on, sends a message to start the experiment and then on demand replicates itself to choose either to receive details of the experiment and display it to the user or to receive a command from the user and send a message to end the experiment. The former provides the scientist with an ongoing view of the experiment and the latter with the means to stop the experiment if its progress is not satisfactory. *c_display* and *user_input* connections are shown as hyper-links in the code as they have previously been defined elsewhere.

Figure 12 shows the definition of the server abstraction. The body of the server mirrors that of the client. It defines its connections, receives the start message, begins the experiment and then on demand replicates itself to choose either to receive the stop message and end the experiment or to receive the current values of the experiment and send them on. As before connection *exp_input* and function *stop_experiment* are shown as hyper-links.

```
! server
value server_abs = abstraction()
{   value s_start = connection() ;
    value s_stop = connection() ;
    value s_put = connection( exp_view ) ;
    via s_start receive ;
    start_experiment() ;
    replicate
        choose{
            {   via s_stop receive ;
                stop_experiment() }
            or
            {   via exp_input receive current_view ;
                via s_put send current_view } }
}
```

Figure 12. The server abstraction

Having defined server and client abstractions, we can now create a client-server system by composing instances of the server and the client abstractions with appropriate unification. Unification ensures that corresponding client and server connections are matched for communication. Defining the composition as a value gives us a handle (*CS_system1*) to the resulting behaviour. Figure 13 shows the composition of one client and one server.

The *as* construct allows users to associate meaningful labels with behaviours being composed. In addition to aiding the identification of behaviours after decomposition, this facility also connections to be uniquely identified for unification.

```
! client-server system
value CS_system1 =
compose{
        client as client_abs() and server as
server_abs()
        where{  client::c_start unifies server::s_start,
                client::c_stop unifies server::s_stop,
                client::c_get unifies server::s_put }
    }
```

Figure 13. The client-server system

Once the system starts executing, we may wish to change its structure. The scientist may want to share a view of the *in silico* experiment with colleagues, or the experiment may take longer than expected and the scientist may wish to get advice before deciding whether the server should be stopped aborting the experiment.

We begin this process by decomposing the system into its component parts as shown in Figure 14. The result of this decomposition is a sequence of views containing the following information about each behaviour of the system: label, behaviour value and list of connections. In long-running systems, labels associated with behaviours may help identify their purpose and identity.

```
! decompose system
value cs_seq = decompose CS_system1
```

Figure 14. Decomposition

Necessary changes can then be made by evolving or redefining some components. In this case we wish to split the functionality of the server into two by creating two new servers, one serving status alone, which can be shared among multiple clients, and the other serving the command messages, of which the scientist who started the experiment wants to retain control. Therefore we create two new abstractions to replace the old *server_abs*.

Using hyper-code representations of the abstractions will enable us to define the new abstractions to use the current values of variables without having to explicitly store and reinitialise them as shown in Figure 15. Abstraction *view_server_abs* disseminates status information about the experiment while *command_server_abs* allows the experiment to be controlled. Note that start messages are ignored as the experiment has already been running.

```
! view server
value view_server_abs = abstraction()
replicate
{   via exp_input receive current_view ;
    via s_put send current_view

}


! command server
 value command_server_abs = abstraction()
replicate
    choose{
        {   via s_start receive }
        or
        {   via s_stop receive ;
            stop_experiment() }
```

Figure 15. The new server abstractions

A new client-server system can then be formed by composing the two new servers with the decomposed client appropriately as shown in Figure 16.

```
! make new client-server system
value CS_system2 =
compose{ client as cs_seq::1.bhvr
    and    view_server as view_server_abs()
    and    command_server as command_server_abs()
    where{
        client::c_start unifies command_server::s_start,
        client::c_stop unifies command_server::s_stop,
        client::c_get unifies view_server::s_put }
} ;
```

Figure 16. The changed system

Now the client will communicate with both servers. If experiment information is required the client will talk to *view_server* and if the scientist wishes to control the experiment then the client will talk to *command_server*.

10   Conclusions

We have identified the need for dynamic evolution of software architecture definitions. For reliability it is important that the architectural definition of a system is automatically kept consistent with the state of the system at all times during its execution and evolution. Our approach to meeting these requirements involves the combination of a number of technologies: reflection, reification and representation for closure (hyper-code).

We have combined these with a variant of the π-calculus that yields dynamic expressions and communication while providing the basis for formal

analysis tools in the form of theorem provers, type checkers and model checkers.

These elements provide a core evolutionary support system. We are currently investigating its usability in practice.

11   References

[1]   D. Garlan, M. Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, Volume 1, World Scientific Publishing Co., 1993.

[2]   D. Perry, A. Wolf, "Foundations for the Study of Software Architectures", *ACM SIGSOFT Software Engineering Notes*, October 1992, pp. 40-52.

[3]   F. Oquendo, S. Cîmpan, D. Balasubramaniam, G. Kirby, R. Morrison, "The ArchWare ADL: Definition of the Textual Concrete Syntax", European RTD Project IST-2001-32360, Deliverable D1.2b, December 2002.

[4]   L. Coglianese, R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proceedings of AGARD'93*, May 1993.

[5]   D. Garlan, R. Allen, J. Ockerbloom, "Exploiting Style in Architectural Design Environments", *Proceedings of the ACM SIGSOFT'94 Symposium on Foundations of Software Engineering*, New Orleans, 1994.

[6]   P. Binns, M. Engelhart, M. Jackson, S. Vestal, "Domain-Specific Software Architectures for Guidance, Navigation, and Control", *International Journal of Software Engineering and Knowledge Engineering*, 1996.

[7]   M. Moriconi, R.A. Riemenschneider, "Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies", Computer Science Laboratory, SRI International, Technical Report SRI-CSL-97-01, March 1997.

[8]   R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr, J.E. Robbins, K.A. Nies, P. Oreizy, D.L. Dubrow, "A Component-and Message-Based Architectural Style for GUI Software", *IEEE Transactions of Software Engineering*, June 1996.

[9]   R. Allen, "A Formal Approach to Software Architectures", PhD thesis, Carnegie-Mellon University, 1997.

[10]   R. DeLine, "Toward User-Defined Element Types and Architectural Styles", *Proceedings of the Second International Software Architecture Workshop*, San Francisco, 1996, pp.47-49.

[11]   J.M. Cobleigh, L.J. Osterweil, A. Wise, B.S. Lerner, "Containment Units: A Hierarchically Composable Architecture for Adaptive Systems", *Proceedings of the 10th International Symposium on the Foundations of Software Engineering*, Charleston, South Carolina, November 2002.

[12] P. Oreizy, N. Medvidovic, R.N. Taylor, "Architecture-Based Runtime Software Evolution", *Proceedings of ICSE'20*, Kyoto, Japan, IEEE Computer Society Press, 1998, pp. 177-186.

[13] I. Georgiadis, J. Magee, J. Kramer, "Self-Organising Software Architectures for Distributed Systems", *Proceedings of the ACM SIGSOFT Workshop on Self-healing Systems*, Charleston, South Carolina, 2002.

[14] D. Garlan, R. Monroe, D. Wile, "ACME: An Architecture Description Interchange Language", *Proceedings of CASCON'97*, Toronto, November 1997, pp.169-183.

[15] http://www.arch-ware.org

[16] R. Milner, *Communicating and Mobile Systems: The Pi-Calculus*, Cambridge University Press, 1999.

[17] C. Strachey, "Fundamental Concepts in Programming Languages", Oxford University Press, Oxford , 1967.

[18] R.D. Tennent, "Language Design Methods based on Semantic Principles", *Acta Informatica* 8, 1977, pp. 97-112.

[19] R. Morrison, "On the Development of S-algol", PhD thesis, University of St Andrews, 1979.

[20] G. Kirby, R. Connor, Q. Cutts, A. Dearle, A. Farkas, R. Morrison, "Persistent Hyper-programs", Persistent Object Systems, Springer-Verlag, 1992, pp. 86-106.

[21] E. Zirintsis, "Towards Simplification of the Software Development Process: The Hyper-code Abstraction", PhD thesis, University of St Andrews, 2000.

[22] C. Goble, S. Pettifer, R. Stevens, "Knowledge Integration: In silico Experiments in Bioinformatics", *The Grid: Blueprint for a New Computing Infrastructure*, Second Edition, eds. I Foster and C Kesselman, 2003.