Specifying Software Architectures using a Formal-based Approach

David Hemer, Yulin Ding
School of Computer Science
The University of Adelaide, SA, 5005, Australia
{hemer, yulin}@cs.adelaide.edu.au

Abstract

Architecture description languages (ADLs) describe the abstracted structure of a system. In this paper we describe a new ADL based on extension of the existing CARE *language used in formally specifying and implementing reusable software components. The main elements of this ADL are components and connectors, with functional and non functional behaviours and interfaces defined. The ADL includes a configuration part, describing the connection between components and connectors, defined using a CSP-like notation. The ADL is amenable to the use of theorem proving techniques for establishing correctness of the architecture. The recursive architecture is also specified as a part of the communication. The design for the* CARE *ADL is incorporated with the plan to leverage existing tools for matching and adapting* CARE *components, to develop support for the detection and correction of architecture mismatches (i.e. where components do not interoperate correctly).*

Keywords: *software architectures, formal languages*

1. Introduction

Increasingly complex computer control systems are being built from Commercial-off-the-shelf (COTS) and Government-off-the-shelf (GOTS) components. Examples of complex systems that are built by combining a number of existing off-the-shelf components include: Enterprise Systems (e.g., Human Resource systems, Library Management systems and Student Record systems); SCADA (supervisory control and data acquisition) systems; and Defence applications such as Naval Combat Systems.

On the surface the use of off-the-shelf components for the development of complex systems is an attractive approach, with potential savings in both cost and time. However, there are often incompatibilities between an off-the-shelf component and the system that it is to be integrated into. Also, where multiple off-the-shelf components are to be integrated, incompatibilities often exist between these components. Incompatibilities can include: different data and message formats; time related issues such as latency and throughput; and differences in functionality. If these incompatibilities are not recognised and dealt with early in the development lifecycle, they can lead to huge blowouts in the budget, or failure to meet performance and functional requirements.

1.1. Aims

Our aim is to develop techniques that will help predict and correct incompatibilities, both between pairs of components and across the overall system, early in the design process for computer-based systems. We will develop a formal mathematical based approach to predicting and correcting incompatibilities. This will allow us to describe component interfaces in a more precise and unambiguous manner, and will also allow us to develop automated tool support for detecting incompatibilities. Since many SCADA and Defence applications are safety or mission critical it is not unreasonable to adopt formal methods approach, especially early in the lifecycle. Specifically we aim to develop:

- A machine-checkable language for specifying time-dependent *software architectures*.
- Analysis techniques for detecting functional and timing requirement mismatches.
- Automated techniques for correcting component mismatches.

We will use specification matching techniques as a means of detecting component mismatches. These techniques are used in library component retrieval, and make use of formally specified pre- and post- conditions to increase search recall. We will extend specification matching to handle matching of complex time-dependent properties. More recently, the focus of the specification matching community has been on developing techniques for correcting mismatches between user queries and library components, based on automated adaptation of library components. This work will be a useful starting point for developing automatable methods for correcting architectural mismatches.

1.2. This paper

The focus of this paper is the development of an *architecture description language* that will be amenable to these matching and adaptation techniques. We use the CARE language [4, 5] as a starting point for the ADL, since we have already made good progress in developing matching and adaptation techniques for this language. We can keep the original methodologies, basic notions and functionalities based on CARE. However, as we will illustrate through an example, CARE is not rich enough to capture the essential features of a software architecture. In particular, in the language of Allen and Garlan [1], CARE can capture low level *implementation relationships* between modules, but cannot express implementation independent, or so-called *interaction relationships*.

Implementation relationships express the definition/use dependency relationships between implementation ”modules”. Implementation relationships represent the lower level structure, which contains relationships between parts of a system. It is also called *implementation constraining* in [11], which require a high degree of fidelity of an architecture to its implementation.

Interaction relationships not only contain the functions of implementation relationships, but also reflect directly the abstract interactions that result in the effective composition of independent components. *interaction relationships* describe higher level conceptual architecture of a system. The same concept is called *implementation independent* in [11]. With this type of ADLs, components and connectors are modelled at a high level of abstraction and do not assume or prescribe a particular relationship between an architectural description and an implementation.

In this paper, we extend the CARE language to describe interaction relationships. It is a significant progress that CARE is developed as a sound formal Architecture Description Language (ADL).

This paper is organised as follows. Section 2 reviews the framework of common ADLs. Section 3 extends the existing CARE language into CARE ADL. Section 4 formalises the CARE ADL using the Z specification. Section 5 checks the correctness of the CARE ADL using theorem proving methods. Section 6 describes the recursive architecture and proves the correctness of it. Section 7 compares the advantages, differences and similarities between CARE and other ADLs. Finally, this paper is concluded with section 8.

2. Framework of Architecture Description Languages

Software architectures describe: the elements from which systems are built; interactions among those elements; patterns that guide their composition; and constraints on these patterns [9]. An *Architecture Description Language* (ADL) is a language for defining software architectures.

There are many recently developed Architecture Description Languages (ADLs). The ADLs vary from one to another, each with a different focus and demonstrating their own strength and weakness. However, there are common themes among them. Medvidovic and Taylor [11] classified three essential modeling features for ADLs: components; connectors; and architectural configurations.

Components represent the primary computational elements and data stores of a system [11]. Components may have multiple interfaces (or *ports*), which define a point of interaction between a component and its environment. *Connectors* represent interactions among components, and mediate the communication and coordination activities among components. Connectors also have interfaces that define the roles played by the various participants in the interaction represented by the connector. *Configurations* describe the interactions between components and connectors.

Other features of architecture structures include: types; properties; and constraints. *Types* represent families of related system. An architectural style typically defines a vocabulary of design element types and rules for composing them [9]. For example, the pipe-file style represents dataflow architectures. *Properties* represent semantic information about a system and its components that goes beyond structure. *Constraints* represent claims about an architectural design that should remain true even as it evolves over time. Typical constraints include restrictions on allowable values of properties, topology, and design vocabulary.

In addition architecture description languages typically include a means of defining the behavior of a system. The properties of behavior contain functional and non-functional behaviors. Examples of non-functional behaviors are timing, schedulability, reliability and security analysis. Different ADLs are supported by particular tools to check the compatibilities, refinement of different parts, such as components and connectors, and the correctness of properties of an architecture.

3. Motivating example

3.1. Existing CARE language

To illustrate the use of CARE we look at a simple example for inserting an element into a list. We propose implementing list insertion by sorting the list, then inserting the element in the sorted list. Such a program can be constructed by combining a component that sorts a list with a second component that inserts an element into a sorted list. This is an example of sequential architecture. The architecture is shown in Fig. 1.

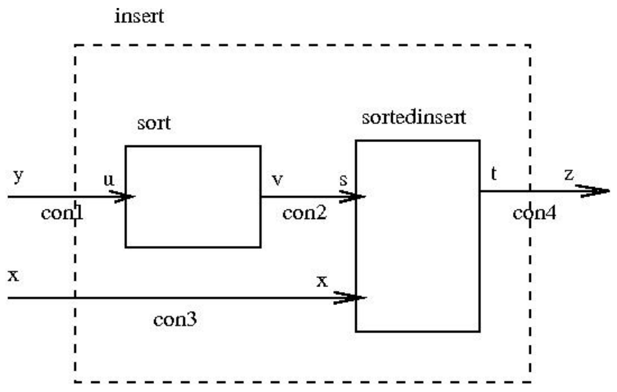

Figure 1. High-level architecture for list insertion

Such a problem can be specified in the existing CARE language. In the following specification the top-level program is represented by the fragment `insert`, which takes an element and a list and returns a list. A precondition is given, stating that the element cannot already be in the list, and the postcondition states that the elements in the return list are the same as those in the original list with the addition of the new element. We implement this by calling two other components. The first, `sort`, takes a (possibly) unsorted list and returns a sorted list. The second, `sortedinsert`, inserts an element in a sorted list, returning a sorted list.

fragment `insert (`**in** `x:E,`**in** `y:List,`**out** `z:List)`
  pre $x \notin elems(y)$
  post $items(z) = [\![x]\!] \uplus items(y)$
`::= sort(y)::u:List;`
  `sortedinsert(x,u).`
fragment `sort (`**in** `u:List,`**out** `v:List)`
  post $items(v) = items(u) \land isSorted(v)$.
fragment `sortedinsert (`**in** `x:E,`**in** `s:List,`**out** `t:List)`
  pre $isSorted(s) \land x \notin elems(s)$
  post $isSorted(t) \land \exists a, b : List \bullet t = a \frown \langle x \rangle \frown b \land s = a \frown b$.

From this specification we can reconstruct the architecture shown in Fig. 1. However the approach is rather clumsy in that we don't explicitly identify the connectors. More importantly it is difficult to see how we might extend the language to include non-sequential connectors, such as those used in other ADLs such as Wright. Our aim is to introduce explicit connector specifications into CARE, whilst at the same time ensuring that the existing tools, in particular those detecting and correcting component mismatches, can still be applied.

3.2. Extending the language

In this section we show how we can extend the CARE language in order to specify component architectures. In particular we add support for explicit component connectors. We follow the lead of other ADLs that allow us to specify "classes" of components, and then create one or more instances of these components. Likewise we specify a class of connectors and then create multiple instances of these.

In Fig. 2 we revisit the list insertion problem. This time we define *insert* as an architecture. We begin by specifying the inputs and outputs of the architecture (also referred to as *ports*). Next we give the *contract* for insert, specifying functional properties that are required by the architecture, as well as properties that are provided by the architecture. Then we give specifications for the two component classes, representing sort and sortedinsert. Each component class has a contract, specifying functional properties. We will create one instance of each of these components, but are by no means limited to creating just a single instance.

The next part defines the interface for a connector class, in this case a simple pipe connector. This connector has two roles, one for writing data to the pipe, the other for reading data from the pipe at the other end.

Next we create instances of the components and connectors. In this case there are two components and four connectors (corresponding to the four arrows in Fig. 1).

Following the interface part of the architecture is the configuration part. In the configuration part we declare the component and connector instances. Next the connections between these instances are defined, giving details of connections between ports and roles.

The final part of the architecture is the communication part, describing how the data is passed between components and connectors. This part models the data flow shown in Fig. 1 using a CSP-like notation. There are two different kinds of events used here: calls to components (e.g. `call(s)`); and flow of data along a connection (e.g. `flow(con1)`). This example joins processes using a sequential join ("`->`") and a parallel join ("`||`").

4. Formalising the CARE ADL

In this section we give a formal definition of the CARE ADL, defining the major elements of an architecture. The CARE ADL is formalised using the Z specification language.

The CARE ADL contains and extends the syntax of the existing CARE language [4]. The extended language gives explicit recognition to component connectors, and the associated machinery of roles and ports as well as configuration information about the architecture. The CARE ADL

```
architecture insert (in x:E, in y:L, out z:L) is
  pre $x \notin elems(y)$
  post $items(z) = [\![x]\!] \uplus items(y)$
  component sort (in u:L, out v:L)
    post $items(v) = items(u) \wedge isSorted(v)$ .
  component sortedinsert (in x:E, in s:L, out t:L)
    pre $isSorted(s) \wedge x \notin elems(s)$
    post $isSorted(t) \wedge \exists a, b : L \bullet t = a \frown \langle x \rangle \frown b \wedge s = a \frown b$ .
  connector pipe (in writer:X, out reader:X)
    params X
    post $reader' = writer$
  instances
    component s::sort
    component si::sortedinsert
    connector con1::pipe(L)
    connector con2::pipe(L)
    connector con3::pipe(E)
    connector con4::pipe(L)
  connections
    con1.writer→ y
    con1.reader → s.u
    con2.writer →s.v
    con2.reader → si.s
    con3.writer →x
    con3.reader →si.x
    con4.writer →si.t
    con4.reader →z
  communication
    ((flow(con1) → call(s)
      → flow(con2)) || flow(con3))
      → call(si) → flow(con4) → stop
end architecture
```

Figure 2. An architecture for list insertion

supports hierarchical architectures. High-level architectures can include sub-architectures that are connected via interface ports. These sub-architectures may be pre-defined, representing commonly used architectural configurations, and may be supplied as part of a library repository.

4.1. Architecture overview

Conceptually an architecture consists of an *interface* and a *configuration*. The interface gives the external view of the architecture, while the configuration describes how the individual parts of the architecture are pieced together.

Architecture

$name : Id$

$interface : ArchInterface$

$configuration : ArchConfiguration$

$\operatorname{ran} configuration.components \subseteq interface.components$

$\operatorname{ran} configuration.connectors \subseteq interface.connectors$

The first condition in the above schema states that every component instance in the configuration part must have a corresponding component type declared in the interface part. The second condition is the corresponding requirement on connectors.

For the list insertion architecture shown in Fig. 2, the interface part of the architecture is everything up to the keyword “instances”. The remainder of the architecture is the configuration part, defining the instances of components and connectors, and how they are pieced together.

4.2. Architecture interface

The interface part of an architecture defines: the external ports of the architecture; the types of any components and connectors used in the architecture; and a contract specifying functional and non-functional properties of the architecture.

ArchInterface

$ports : \mathbb{F}\, Port$

$components : \mathbb{F}\, Component$

$connectors : \mathbb{F}\, Connector$

$contract : Contract$

A *port* is a point at which data flows into or out of an architecture or component. A port is identified by a name. Port names must be locally unique, that is a particular component cannot have multiple ports with the same name. A port also has an associated *type*, identifying the type of data flowing into or out of the data. Finally, a port has an associated *mode*, which indicates whether the data is flowing into the component or out of the component. The mode is represented as an enumerated type with two values, *in* or *out*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Port}} \\ \hline
\textit{name} : \textit{Id} \\
\textit{type} : \textit{Type} \\
\textit{mode} : \textit{Mode} \\ \hline
\end{array}$$

A *contract* defines properties on an architecture, component or connector. These properties may be functional properties, or non-functional properties. In the current version of the CARE ADL, we support functional properties, however in later versions we will extend this to include non-functional properties, in particular timing properties. We split the properties into two broad categories: those properties that are *required* for the architecture, component or connector to operate correctly; and those properties that are *ensured* by the architecture, component or connector.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Contract}} \\ \hline
\textit{requires} : \mathbb{F}\,\textit{Property} \\
\textit{ensures} : \mathbb{F}\,\textit{Property} \\ \hline
\end{array}$$

4.2.1 Components and connectors

Components and connectors are the basic building blocks for architectures. A component type consists of a name, a collection of ports, and a contract.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Component}} \\ \hline
\textit{name} : \textit{Id} \\
\textit{ports} : \mathbb{F}\,\textit{Port} \\
\textit{contract} : \textit{Contract} \\ \hline
\end{array}$$

A *connector* is defined by a name, a set of roles and a contract that states properties about the connector.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Connector}} \\ \hline
\textit{name} : \textit{Id} \\
\textit{roles} : \mathbb{F}\,\textit{Role} \\
\textit{contract} : \textit{Contract} \\ \hline
\end{array}$$

Each connector will have a number of *roles* (interfaces) associated with it. For example a pipe connector will typically have a role that allows data to be written to the pipe, and a role that allows data to be read from the pipe.

4.3. Architecture configuration

The configuration part of an architecture defines the instances of components, connectors and sub architectures used within the architecture. Component instances are modelled as a partial function from instance identifiers to the component type. Similarly connector instances are modelled as a partial function from instance identifiers to the connector type. The topology of the architecture is defined in terms of connections between roles and ports. We differentiate between *internal* connections and *external* connections. External connections are those that connect to the external ports of the main architecture.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ArchConfiguration}} \\ \hline
\textit{components} : \textit{Id} \mathrel{+\!\!\!\!\rightarrow} \textit{Component} \\
\textit{connectors} : \textit{Id} \mathrel{+\!\!\!\!\rightarrow} \textit{Connector} \\
\textit{subarchs} : \textit{Id} \mathrel{+\!\!\!\!\rightarrow} \textit{Architecture} \\
\textit{int\_connections} : \textit{Role} \leftrightarrow \textit{Port} \\
\textit{ext\_connections} : \textit{Role} \leftrightarrow \textit{Port} \\
\textit{communication} : \textit{Process} \\ \hline
\operatorname{dom} \textit{int\_connections} \subseteq \{ r : \textit{Role} \mid \\
\qquad \exists\, c : \operatorname{ran} \textit{connectors} \bullet r \in c.\textit{roles} \} \\
\operatorname{ran} \textit{int\_connections} \subseteq \{ p : \textit{Port} \mid \\
\qquad (\exists\, c : \operatorname{ran} \textit{components} \bullet p \in c.\textit{ports}) \lor \\
\qquad\quad (\exists\, a : \operatorname{ran} \textit{subarchs} \bullet p \in a.\textit{interface}.\textit{ports}) \} \\
\operatorname{dom} \textit{ext\_connections} \subseteq \{ r : \textit{Role} \mid \\
\qquad \exists\, c : \operatorname{ran} \textit{connectors} \bullet r \in c.\textit{roles} \} \\ \hline
\end{array}$$

Roles used in both internal and external connections must be a role from one of the connector instances. Ports used in internal connections must be a port from one of the component instances or an external port of one of the sub architectures.

The communications part of the configuration describes how the order in which data flow between components via connectors. Non-terminal communication *processes* are: sequential processes; parallel processes; or conditional (branching) processes. Terminal processes are: the flow event; the call event; and the skip process.

$$\begin{array}{lcl}
\textit{Process} & ::= & \textit{skip} \\
& \mid & \textit{flow}\langle\!\langle \textit{Id} \rangle\!\rangle \\
& \mid & \textit{call}\langle\!\langle \textit{Id} \rangle\!\rangle \\
& \mid & \textit{then}\langle\!\langle \textit{Process} \times \textit{Process} \rangle\!\rangle \\
& \mid & \textit{parallel}\langle\!\langle \textit{Process} \times \textit{Process} \rangle\!\rangle \\
& \mid & \textit{ifthenelse}\langle\!\langle \textit{Condition} \times \textit{Process} \times \textit{Process} \rangle\!\rangle
\end{array}$$

A flow event corresponds to a flow of data along a connector instance. A call event corresponds to a call to a component instance. The skip process does nothing.

5. Checking the architecture

In this section we describe three correctness checks that can be performed on the architecture: type correctness; partial correctness; and well-formedness.

5.1. Type checking the architecture

The architecture description language includes types for all ports and roles. Ports and roles that are connected should have compatable types. We therefore need to check that for every connection defined within an architecture configuration, the type of the role is equal to the type of the port.

Consider the list insertion architecture, shown in Fig. 2. If we focus on the connection between the *s* port of *sortedinsert* and the *reader* port of *con2*, then it is easy to see that they both have type *L*.

Further type checking can be done for each of the components and connectors defined in the architecture to ensure that produce the correct output type given correct input types.

5.2. Partial correctness

To establish *partial correctness* we show that the postcondition of the architecture holds assuming that the precondition holds. To check the partial correctness of the architecture, we establish the functional behaviour of the architecture from the contracts of the component and connector instances involved, and the communication process. To do this we step through the communications process, to generate a *strongest postcondition*. The strongest postcondition semantics are shown in Fig. 3.

$$
\begin{aligned}
&\mathit{spost}(\mathit{skip}) = \text{true}\\
&\mathit{spost}(\mathit{flow}(c)) = \mathit{InRolePortRel}(c) \wedge\\
&\mathit{OutRolePortRel}(c) \wedge \mathit{Post}(c)\\
&\mathit{spost}(\mathit{call}(c)) = \mathit{Post}(c)\\
&\mathit{spost}(\mathit{then}(p_1,p_2)) = \mathit{spost}(p_1) \Rightarrow \mathit{spost}(p_2)\\
&\mathit{spost}(\mathit{parallel}(p_1,p_2)) = \mathit{spost}(p_1) \wedge \mathit{spost}(p_2)\\
&\mathit{spost}(\mathit{ifthenelse}(c,p_1,p_2)) = c \Rightarrow \mathit{spost}(p_1)\\
&\qquad \wedge \neg\, c \Rightarrow \mathit{spost}(p_2)
\end{aligned}
$$

Figure 3. Strongest postcondition semantics

For flow events, we can assume the postcondition of the connector holds. Moreover we can assume certain relationships between ports and roles. For *in roles*, we can assume that the initial value (before the flow occurs) of the in role is equal to the value at the port that it is connected to. The relation *InRolePort* captures these associations for all in roles. For out roles, we can assume that the value of port that the role is connected to is equal to the final value of the out role (i.e. after the flow occurs).

To establish partial correctness, we need to show that the precondition of the architecture together with the strongest postcondition imply that the postcondition of the architecture holds.

For the `insert` example we need to establish the postcondition:

$$\mathit{items}(z) = [\![x]\!] \uplus \mathit{items}(y) \tag{1}$$

Assuming the precondition, $x \notin \mathit{elems}(y)$ holds, we generate the strongest postcondition by processing the communications part of the architecture. The resulting proof obligation is shown in Fig. 4.

Upon simplifying the proof obligation we get:

$$
\begin{aligned}
&\forall\, a, b \bullet x \notin \mathit{elems}(y) \wedge \mathit{items}(s.u) = \mathit{items}(a \frown b)\\
&\qquad \Rightarrow \mathit{items}(a \frown \langle si.x\rangle \frown b) = [\![si.x]\!] \uplus \mathit{items}\; s.u
\end{aligned}
$$

This can be easily proven using the following properties, together with the fact that the bag union operator, $\uplus$, is associative and commutative:

$$\mathit{items}(a \frown b) = \mathit{items}(a) \uplus \mathit{items}(b) \tag{2}$$

$$\mathit{items}(\langle x\rangle) = [\![x]\!] \tag{3}$$

5.3. Well-formedness

To establish *well-formedness* we need to show that the required properties (pre-condition) of all components and connectors used in the architecture are satisfied.

For example, consider the call to the component *si*. To establish well-formedness for this call, we need to show that the precondition of the instance *si* of `sortedinsert` is satisfied. To do this we can assume that the precondition of the architecture holds, together with the strongest postcondition for the communication process *up to the point of the call*. That is for the process:

```
((flow(con1) → call(s)
  → flow(con2)) || flow(con3))
```

The resulting proof obligation is shown in Fig. 5. The proof of the first conjunct is straightforward. The proof of the second conjunct follows from the following property:

$$\forall\, a, b : \operatorname{seq} X \bullet \mathit{items}(a) = \mathit{items}(b) \Rightarrow \mathit{elems}(a) = \mathit{elems}(b)$$

This proof obligation, together with the partial correctness obligation have been discharged with the Isabelle theorem prover.

6. Recursive architectures

In this section we look at an example of a recursive architecture (as shown in [1]), that is one that repeatedly calls one or more of its component instances. The example converts a list of characters into a new list, where all of the original characters have been capitalised.

$$
\begin{array}{l}
x \notin elems(y) \wedge ((con1.writer = y \wedge con1.reader' = s.u \wedge con1.writer = con1.reader') \Rightarrow \\
\quad (isSorted(s.v) \wedge items(s.u) = items(s.v)) \Rightarrow \\
\quad\quad (con2.writer = s.v \wedge con2.reader' = si.s \wedge con2.writer = con2.reader')) \wedge \\
\quad\quad\quad (con3.writer = x \wedge con3.reader' = si.x \wedge con3.writer = con3.reader') \Rightarrow \\
\quad\quad\quad\quad isSorted(si.t) \wedge (\exists\, a, b \bullet si.t = a \frown \langle si.x \rangle \frown b \wedge si.s = a \frown b) \Rightarrow \\
\quad\quad\quad\quad\quad con4.writer = si.t \wedge con4.reader' = z \wedge con4.reader' = con4.writer \Rightarrow \\
\quad\quad\quad\quad\quad\quad items(z) = [\![x]\!] \uplus items(y)
\end{array}
$$

Figure 4. Proof obligation for partial correctness of insert contract

$$
\begin{array}{l}
x \notin elems(y) \wedge ((con1.writer = y \wedge con1.reader' = s.u \wedge con1.writer = con1.reader') \Rightarrow \\
\quad (isSorted(s.v) \wedge items(s.u) = items(s.v)) \Rightarrow \\
\quad\quad (con2.writer = s.v \wedge con2.reader' = si.s \wedge con2.writer = con2.reader')) \wedge \\
\quad\quad\quad (con3.writer = x \wedge con3.reader' = si.x \wedge con3.writer = con3.reader') \Rightarrow \\
\quad\quad\quad\quad isSorted(si.s) \wedge si.x \notin elems(si.s)
\end{array}
$$

Figure 5. Well-formedness proof obligation for insert architecture

Previously we would have represented this using a single component (or fragment), with a list as input and another list as output. Recursion was handled within the component implementation. For this example, as shown in Fig. 6, we have a component (*upper*) that capitalises a single character at a time. The recursion is handled within the communication process.

The architecture includes a single component instance and two connectors. The first connector connects the input list to the *up* component. The connection is made by getting the first element of the list and sending this to the *up* component. The second connector connects the result from the *up* component to a buffered list which is accumulated to give the final result. Initially this buffered list is empty.

The communication process is defined using a recursion block. A local recursive process, *f*, is defined. We define the ports used in the recursion block, and the external connectors involved in the recursion block. Within the recursion block is a conditional process. If the input port *s* is non-empty, then a recursive call to *f* is made after the component *up* is called. If the input list is empty then the process is complete.

```
architecture Capital(in s:List,out t:List) is
  post $t = map\ \ toupper\ \ s$
  component upper(in x:Char, out y:Char)
    post $y = toupper(x)$.
  connector queuew(in wr:List,out rdr:Char)
    post $rdr' = head\,wr \wedge wr' = tail\,wr$
  connector queuer(in wr:Char out rdr:List)
    init $rdr = \langle\,\rangle$
    post $rdr' = \langle wr \rangle \frown rdr$
  instances
    component up::upper
    connector con1::queuew
    connector con2::queuer
  connections
    con1.wr→s
    con1.rdr→up.x
    con2.wr→up.y
    con2.rdr→t
  communication
    rec f(s,t; con1, con2) .
      if $s \neq \langle\,\rangle$ then
        flow(con1) -> call(up) ->
        f(s',t') -> flow(con2)
      else
        skip
    end rec
end architecture
```

Figure 6. An architecture for capitalising a list of characters

6.1. Partial correctness

To prove partial correctness for the *Capital* architecture we will consider the two paths in the communication process separately. We shall not give a formal treatment of this here, instead we sketch the main details.

Firstly consider the else branch, where $s = \langle\,\rangle$. To prove partial correctness for this branch we are required to show:

$$t = map\ toupper\ \langle\,\rangle \tag{4}$$

Simplifying the right-hand side, it remains to show that $t = \langle\,\rangle$. From the initialisation statement for $con2$, we know that

$$con2.rdr = \langle\,\rangle \tag{5}$$

Furthermore, because there is no flow along $con2$ in this branch of the process, we can assume

$$con2.rdr' = con2.rdr \tag{6}$$

Finally we can assume that the ports s and t used in the call to the recursion block are connected to the writer and reader roles respectively of the connectors $con1$ and $con2$. Therefore we can assume:

$$con1.wr = s \tag{7}$$
$$con2.rdr' = t \tag{8}$$

Combining (8), (5) and (6) we can infer that $t = \langle\,\rangle$ thus completing the proof for the else case.

To generate the proof obligation for the if branch, we need to introduce some additional notation and assumptions concerning the recursive call `f(s',t')`. For a connector con and a role r, we shall use $f.con.r$ to refer to role r of the connector con as it is used in the recursive call. We make the following assumptions regarding recursive roles:

1. For any role r belonging to a connector con, which comes *before* the recursive call $f\,f.con.r = con.r'$

2. For any role r belonging to a connector con, which comes *after* the recursive call $f\,f.con.r' = con.r$

Furthermore we can assume that the architecture post-condition holds between s' and t' for the recursive call, i.e.

$$t' = map\ toupper\ s' \tag{9}$$

Finally we can assume that the ports s' and t' used in the recursive call $f(s', t')$ are connected to the writer and reader roles respectively of the connectors $f.con1$ and $f.con2$, i.e.

$$f.con1.wr = s' \tag{10}$$
$$f.con2.rdr' = t' \tag{11}$$

Combining these assumptions with the strongest post-condition semantics for the other constructs in the if branch, we can derive the proof obligation as shown in Fig. 7. The conjecture of this proof obligation, represented in the last line, can be discharged using an inductive proof.

7. Comparison between Care and Other ADLs

7.1. Other ADLs

Darwin [8] is a declarative binding language used to define hierarchical compositions of interconnected components for parallel and distributed systems. It supports both static and dynamic structures, where the latter may evolve during execution. The central abstraction managed by Darwin are components and services, where services are the means by which components interact. Primitive components contain the local names of required and provided services. Composite components are constructed other components (primitive or composite), by declaring both the instances of other components they contain and the bindings between those components. Darwin provides support for reducing complex configurations to a system of primitive component instances.

The semantics of Darwin are expressed in $\pi - calculus$. The configuration description is a precise specification of the potential structure at execution time. Darwin also supports hierarchical architecture mechanisms and reuse. However, Darwin does not have strong support for architecture abstraction or connectors.

UniCon [10] emphasises the structural aspects of executable software systems. It is actually a compiler and provides matching from implementation code to the architecture. The executable code is extracted as the architecture via a wrapper, then represented in the textual or graphical notations. The major elements of this ADL are primitive components, connectors and composite components. Composite components contain primitive components, connectors, instantiation of primitive components and connectors, an implementation topology and communications of the architecture. Unicon supports hierarchical decomposition by allowing components to be defined in terms of either a sub-configuration or a concrete implementation. The primitive component contains a built-in type, players and the implementation. The implementation in the component is further specified with attributes to emphasise the features of a particular part of a software system to achieve a tight match in between the architecture and the system. UniCon creates an initialisation routine that starts up a particular composite component and handles arbitrary topologies correctly. Unicon supports non-functional scheduleability properties and incorporates a tool for analysing the scheduleability.

Rapide [6] is an executable specification language. The result of executing a Rapide architecture (a set of interfaces and connections) is a partially ordered set of events, describing dependencies and independencies between events [6]. The main elements of Rapide are the interface, the connect and the constraint. The interface defines the type of components and provides an abstract definition of externally vis-

$$\begin{array}{l}
s \neq \langle\,\rangle \wedge con2.rdr = \langle\,\rangle \\
\quad s = con1.wr \wedge con1.rdr' = head\, con1.wr \wedge con1.wr' = tail\, con1.wr \wedge con1.rdr' = up.x \Rightarrow \\
\quad\quad up.y = toupper(up.x) \Rightarrow \\
\quad\quad\quad f.con1.wr = con1.wr' \wedge f.con1.rdr = con1.rdr' \wedge f.con2.wr' = con2.wr \wedge f.con2.rdr' = con2.rdr \wedge \\
\quad\quad\quad t' = map\ toupper\ s' \wedge f.con1.wr = s' \wedge f.con2.rdr' = t' \Rightarrow \\
\quad\quad\quad\quad con2.wr = up.y \wedge con2.rdr' = \langle con2.wr \rangle \frown con2.rdr \wedge con2.rdr' = t \Rightarrow \\
\quad\quad\quad\quad\quad t = map\ toupper\ s
\end{array}$$

Figure 7. Partial correctness proof obligation for if-branch of Capital architecture

ible behaviour. The behaviour (action) in the interface is expressed as functions, classified as "in" and "out", where "in" is for received message and "out" is for outgoing message (trigger event). The interface is instantiated before it is used by the connect. The connect calls functions in the interface via each instantiated interface in a similar way of calling methods from a class in Object-Oriented languages. The connect shows dependencies, indicating whether a system is sequential or concurrent. Rapide executes the processes in the connect to run an architecture. Rapide also provides the use of event pattern mappings to define the relationship between two architectures at different levels of abstraction. It accommodates hierarchical refinement. Connections in Rapide are refinable into architectures.

Wright [1] is a conceptual architecture description language and uses CSP as its major mathematical tool for specifying sequential and concurrency processes. The major structure of Wright contains components, connectors, instances, binding and attachments (topology). The interface is called port in a component and role in a connector. The behaviour is called computation in a component and Glue in a connector. The topology of an architecture is drawn in attachments. Wright specifies connectors explicitly, providing clear abstraction for the architecture. Properties are checked in Wright using the FDR model checker, applying deadlock and refinement checks among different parts of an architecture to ensure the architecture's correctness. The specification is predefined for all possibilities of a finite system, thus Wright can do exhaustive checking. However, Wright cannot simulate an on-the-fly system run.

There are other ADLs which are not reviewed in detail. For example, ACME [3], which is for interchange between difference architectures; MetaH AADL [2], which is mainly for dynamic real time system; and C3 [7].

7.2. CARE Compared with Other ADLs

Compatibilities

The CARE ADL is designed to accommodate the best features of the ADLs described above. As described in the earlier sections, the structure of CARE ADL consists of: components; connectors; configurations; communications and constraints. Components and connectors are compatible with those in the above ADLs. The interfaces of components in CARE are compatible with players in Unicon, and ports in Wright; the interfaces of connectors in CARE are compatible with roles in Unicon and Wright. Configuration and communication together describe the overall structure and topology of the ADL, and are compatible with "Bind" and "Connect" in Unicon, "attachment" in Wright and "connect" in Rapide. Behaviour in CARE is compatible with "implementation" in UniCon, "glue" for the connector and "computation" for the component in Wright and "interface" in Rapide. The architecture style in CARE is implicitly represented within the components and connectors, as done in Wright, this differs to Unicon which represents the style explicitly. Similar to Darwin and Unicon, where composite components contain primitive components, and "mapping" in Rapide, CARE supports hierarchical architectures and sub-architectures. CARE is amenable to using theorem proving methods to check the architecture, with the correctness conditions derived in this paper checked using the Isabelle theorem prover. This differs to other ADLs, such as Wright, which use model checking methods.

Advantages

The CARE ADL specifies configuration and communication separately. This means that the CARE ADL is more expressible than other ADLs for connections between instances of connectors and components, and processes of an architecture. The CARE ADL separates functional and relational behaviours in components and connectors making it possible to check correctness using different tools according to the features of behaviours. Architecture checking is done using theorem prover in this paper. However in the future we anticipate using model checking to check certain non functional properties. This makes CARE more flexible than other ADLs like Wright, although some ADLs such as Unicon do already support different kinds of non-functional properties. Architecture properties (contracts) in the CARE ADL are more expressible than the similar functions in other ADLs. For the above reasons, we decided to extend CARE as a ADL instead of using other existing ADLs.

A major weakness of Rapide that is addressed in the CARE ADL is that it does not specify connectors and topology separately, and gives no clear interface between different parts of the system. UniCon functions as a compiler to convert executive code into the architecture, which is sound for a software system but not so applicable for systems that include hardware. In contrast, the focus of the CARE ADL is at a more abstract level, applicable during the design phases and suitable to both software and hardware.

Finally, we intend to utilise one of the main features of the existing CARE system, that is the notion of reusable library components. In this case we propose a library of reusable architectures or sub-architectures which can be applied to solve a variety of design problems. Such a feature is not supported by other ADLs described here.

8. Conclusions and Future Work

This paper introduced an architecture description language which extends the existing CARE language. The structure and major elements of this ADL have been formalised using the Z specification. The major elements of the ADL are components, connectors, connections and communications, where components and connectors are specified using the existing CARE style and the communication is specified using CSP-like notations.

Architectures specified using the CARE ADL can be checked for functional correctness using theorem proving technologies. Recursive architectures can be specified using recursion blocks inside the communications part, thus enabling quite a sophisticated connection of components within an architecture, beyond the simple sequential and parallel connections.

In future work we will focus on extending the ADL to support the definition of adaptable and reusable architecture templates. Furthermore we will extend the language to support the specification of non-functional properties, specifically timing properties, and develop techniques for checking whether or not these timing properties are satisfiable. We will also look at incorporating model checking based techniques to check features such as deadlock freedom and refinement checking. Eventually, we will implement a comprehensive case study for a real time system.

Acknowledgements

This work was funded by Australian Research Council Grant DP0664479, Early Detection of Component Incompatibility in Time-dependent Computer Architectures.

References

[1] Robert Allen and David Garlan. (1997). A formal basis for architectural connection. In *ACM Transactions on Software Engineering and Methodology* (TOSEM), Pp. 213-249, Vol.6, Issue 3 (July 1997).

[2] Pam Binns and Steve Vestal. (2001). Formalizing software architectures for embedded systems. in T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, Pp. 451-468.

[3] D. Garlan, R. Monroe, and D. Wile. (1997). ACME: An architecture description interchange language. Proc. CASCON'97.

[4] David Hemer and Peter Lindsay. (2005). Template-based construction of verified software. IEE Proceedings-softw., Vol. 152, No.1, Feb.,2005.

[5] David Hemer, (2005). A formal approach to component adaptation and composition. In *Proc. of Australia Computer Science Conference*(ACSC2005).

[6] David C. Luckham and James Vera. (1995). An event-based architecture definition language. *IEEE Transactions on Software Engineering*, archive Vol. 21(9),(Sep. 1995) Pp.717 - 734.

[7] Jorge Enrique Perez-Martinez. (2003). Heavyweight extensions to the UML metamodel to describe the C3 architectural. In proc. of ACM SIGSOFT Software Engineering Notes. Vol.28. issue 3. pp.1-6.

[8] Jeff Magee, Naranker Dulay, Susan Eisenbach and Jeff Kramer. (1995). Specifying distributed software architectures. In *Proc. of the* 5^{th} *European software engineering conference*(ESES'95).

[9] Mary Shaw and David Garlan. (1996). Software Architecutre: Perspectives on an Emerging Discipline. Prentice-Hall, Inc. ISBN 0-13-182957-2. Pp. 242.

[10] Mary Shaw et al. (1995). Abstractions for software architecture and tools to support them. In IEEE Transactions on Software Engineering. Vol.21, No.4, Pp.314-335, 1995.

[11] Nenad Medvidovic and Richard N. Taylor. (2000). A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering*. Vol.26, No.1, pp.70-93. Jan. 2000.