18th World IMACS / MODSIM Congress, Cairns, Australia 13-17 July 2009
http://mssanz.org.au/modsim09

Modelling Software Architectures Using CRADLE

David Hemer
School of Computer Science
The University of Adelaide
SA, 5005, Australia
david.hemer@adelaide.edu.au

Yulin Ding
Airborne Mission Systems, Air Operations Division
Defence Science and Technology Organization
PO Box 1500, Edinburgh, SA, 5111, Australia
yulin.ding@dsto.defence.gov.au

Abstract Architecture description languages (ADLs) and architecture analysis and design languages (AADLs) are important modelling and analysing languages for the software design phase. Around a decade ago, a significant number of ADLs, such as Unicon (9), Rapide (5), Wright (1) and C3 (7), were developed to describe a variety of system features. For example, Rapide describes event triggered dynamic architectures; while Wright describes architectures at a predefined conceptual level. However, these ADLs do not provide strong support for analysis. More recently, the MetaH AADL (2) has been developed, with improved support for analysis and the expression of real-time behaviour.

Developed from MetaH AADL, SAE AADL (4) provides further advances on previous ADLs and AADLs, with support for describing dynamic features in real-time systems, and stronger support for analysis than earlier languages. However, this AADL still lacks certain flexibility to express generic and conceptual features of software architectures. Furthermore, it does not provide strong support for error correction and non-functional behaviour [1] is not expressed formally.

We propose a new ADL, named CRADLE, with our main aims to provide strong support for: architecture analysis (detecting inconsistencies or violations of functional and/or non-functional constraints); architecture adaptation (correcting these inconsistencies and constraint violations); and architecture expressiveness (being able to describe a variety of architectural features including function properties and non-functional timing constraints using a formal notation).

A key feature of CRADLE is the support for defining generic architectures that can be adapted by instantiating parameters. This allows us to model and analyse architectures with more genericity, allowing a single architecture model to capture, for example, differing control styles (e.g. concurrent versus sequential), varying numbers of components, and configurable timing constraints. Furthermore, by providing a library of generic architectures, both high level architectures and lower level sub-architectures, we can provide support for architecture adaptation, by extending existing approaches to component adaptation.

Tool support for checking and adapting CRADLE architectures will be based on model checkers and theorem provers. In general, we plan to use model checking to check non-functional properties whereas theorem proving is used to check the functional properties.

This paper describes a case study to model an air traffic control scenario using CRADLE. The previous generation ADL Rapide modelled the architecture of the air traffic control system. For simplicity, the architecture consists of aircraft radio components and control tower transmitter/receiver components and the radio components are connected to the transmitter/receiver components. In reality, CRADLE is able to specify bi-directional connection between the aircraft and the tower. There are two possible communication styles: pipeline style (sequential); or broadcast style (concurrent). Both of these architectures can be expressed separately using non-generic CRADLE constructs as Rapide. However using CRADLE we are able to express both possibilities in a single specification. The air traffic control model is described in two parts: interface and configuration. The interface contains types of component and connector. The configuration contains "instance", "connection" and "communication". The structure of the architecture is adjusted by parametrisation. Thus, one architecture describes all possibilities of the system. We demonstrate how the generic and other advance features of CRADLE are properly used in this system to support its particular features and requirements.

[1] In this paper, functional behaviour is refered to properties such as "$a + b = c$"; non-functional behaviour is refered to properties such as timing, scheduling and safety.

1 Introduction

The focus of this paper is on providing support for specifying generic architectures, i.e. software architectures that are adaptable and can be applied to a variety of problems. One benefit of this is that we can provide common architecture patterns, which can be used by software engineers to describe their particular architecture. Moreover, by providing a library of generic architectures, we plan to develop support for semi-automated correction of component incompatibilities. To do this, specification matching techniques will be used to plug components into generic architectures to find an architecture that removes any incompatibilities. Finally, the generic language will allow us to specify some aspects of dynamic architectures, in that with one generic architecture we can capture a wide range of possible dynamic configurations.

In this paper we describe extensions to a formal-based software architecture description language, called CRADLE (3). The extensions provide support for specifying software architectures in a generic manner. Genericity is supported through the use of parameters, which can represent not only types, but other information, such as: the number of components to create; boolean valued properties for the overall architecture providing alternative configurations; and even boolean valued properties for individual components and connectors.

An air traffic control system is commonly used to illustrate the use of architecture description languages. This system has been specified in Rapide (5), in a non-generic form. It is specified in two separate forms to represent its concurrent and sequential architectures respectively. In this paper, we specify the air traffic control system in a generic manner: fully flexible and conceptual. The architecture can be programmed and adjusted according to the varying number of the aircraft. The style and topology of the architecture can also be adjusted according to the specific requirements. All forms are concisely described in a single generic architecture specification.

In Section 2 we give an example of generic architecture using the CRADLE language. In the example we generalise the air traffic control system as described in the previous paragraph. In Section 3 we give an example of the use of the generic architecture from the previous section. In the example we illustrate the instantiation of formal parameters, and describe how the resultant concrete architecture is derived from the generic architecture. In Section 4 we discuss future work, in particular further extensions to CRADLE to incorporate timing properties (including deadline, period, latency and computation time) into the architecture descriptions, and the development of tools and techniques for detecting and correcting architecture mismatches.

2 Modelling the Air Traffic Control Architecture Using CRADLE

In this section we use CRADLE to specify a generic architecture for an air traffic control system, based on an earlier example specified in Rapide ADL (5). The architecture consists of aircraft radio components and control tower transmitter(Tx)/receiver(Rx) components. The radio components are connected to the Tx/Rx components. There are two possible communication styles: pipeline style (sequential); or broadcast style (concurrent).

The pipeline style is depicted in Fig. 1. In this architecture each aircraft radio is connected to a control tower Tx/Rx. The control tower Tx/Rxs are in turn connected in sequence, with the Tx/Rx sending an acknowledgement to the next Tx/Rx when it has finished receiving the message from its connected radio. The depicted architecture is static, with a fixed number of radio and Tx/Rx pairs, however we will see later that a predefined dynamic architecture can be supported, with an arbitrary number of pairs, together with predefined dynamic configuration where connections are only made for aircraft in range. The example shows the flexible, adaptable, and highly conceptual features of CRADLE.

The broadcast style architecture is shown in Fig. 2. In this case each radio in the in-range aircraft communicates with the tower concurrently and there is no need for any connection between the individual control tower Tx/Rxs.

Both of these architectures can be expressed separately using non-generic CRADLE constructs as Rapide (5), however using generic constructs we are able to express both possibilities in a single specification. Fig. 3 contains a generic architecture specification for the air traffic control system, which can be adapted to support the two configurations described above.

The architecture, named "AirTrafficControl", is parametrised over: a natural number *n*; a boolean valued proposition *Pipeline*; and an array *inrange*. All these parameters will be delivered into the architecture body to instantiate the architecture. This is one of the major features of CRADLE. The parameter *n* represents the number of radio and Tx/Rx pair instances that need to be created. The proposition *Pipeline*, determines whether a sequential or concurrent communication style will be used. If *Pipeline* is instantiated to *true*, then a sequential style will be adopted, otherwise a concurrent style is adopted. The array *inrange* is an array of boolean values, with one value for each aircraft created. A value of *true* indicates that the aircraft is in range, and as such should be connected to the corresponding tower Tx/Rx. A value of *false* means that the aircraft is out of range and does not need to be connected yet.

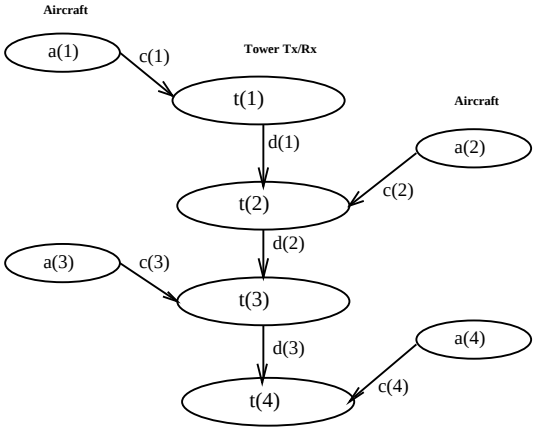

Figure 1: *Air Traffic Control* architecture with sequential communication

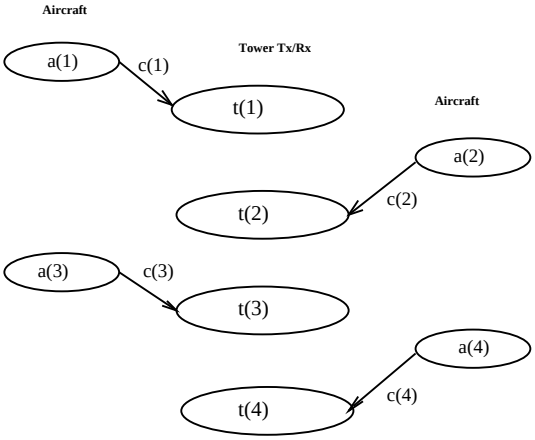

Figure 2: An Architectural Description for *Air Traffic Control* where the control tower receives signals from the aircraft concurrently

Applicability conditions are given to ensure that the array is of the appropriate length and each value in the array is a boolean value. This part contains the delivered parameters after the name of the architecture.

The *interface* part specifies the types of components and connectors used by the architecture. In this case, components for representing the tower Tx/Rxs and the aircraft; and connections between the aircraft and tower Tx/Rxs, and between pairs of tower Tx/Rxs, are declared. For components, the ports, representing points at which the component can be joined via a connector, are defined. For example, the component *TowerTx/Rx* [2] contains two incoming and a single outgoing port. A data type is given for each port representing the type of data that is transmitted in or out of the port. In the case of the *TowerTx/Rx* component it has incoming ports that accept a message (*Msg*) and boolean value (*Bool*) respectively.

The architecture configuration is divided into *instances*, *connection* and *communication* parts. The *instances* part defines the concrete instances of components and connectors that are used in the architecture. In the generic architecture the exact number of components and connectors varies depending on how the architecture parameters are instantiated. This varying number of components and connectors is supported in the generic architecture description through the use of an array construct. For the air traffic control example the parameter n, which we recall takes a natural number value, determines the number of components and connectors.

The *connection* part describes how the individual instances of components and connectors are joined. In generic architectures we utilise *forall* statements to create joins over a variable number of components and connectors, as well as condition statements to establish joins under certain circumstances. In the example we create joins between the aircraft and the tower Tx/Rxs, conditional on the fact that the aircraft is in range (governed by the *inrange* parameter described earlier). Similarly joins are made between tower Tx/Rxs, provided that we wish to create a pipeline architecture (governed by the boolean valued parameter *Pipeline*). Joins are only made between tower Tx/Rxs with those aircraft in range.

The *communication* part describes how data flows between components and connectors using a simple communication protocol language. The communication protocol language has been extended for generic architectures to include indexed sequential and parallel processes, together with conditional processes. In the example, an indexed sequential process is used for pipeline architectures (using the keyword "SEQ"). Notice that this is indexed over a variable i with range from 1 up to n. For pipeline architectures, if an aircraft is in range, then the resulting communication process depends on whether or not there is a subsequent aircraft in range. For each style architecture, if the aircraft is not in range, the particular sub topology related to this aircraft is skipped.

For the sake of simplicity of explanation we assume that aircraft components and corresponding tower Tx/Rxs are always created regardless of whether or not the aircraft is in range. That said, it would be possible to capture more complex behaviour using more conditionals and extra parameters. Furthermore, CRADLE can also describe systems with strategic and bidirectional communications such as aircraft-to-aircraft and aircraft-to-tower, or vice versa.

[2] when we talk about *TowerTx/Rx* or creating instances of *TowerTx/Rx*, what we really mean is the component of tower Tx/Rx or creating an instance of a Tx/Rx within a single tower.

```
generic architecture AirTrafficControl [n::nat, Pipeline::prop,
inrange::array] is

applicability conditions
   len inrange = n;
   FORALL i In 1..n @ inrange(i) = true or inrange(i) = false.

begin interface
   component TowerTx/Rx(in msg:Msg, in ready: Bool, out finished:Bool).
   component Aircraft(out msg:Msg)
   connector AircraftToTower(in sender:Msg, out receiver:Msg)
   connector TowerToTower(in finished:Bool, out start:Bool)
end interface

begin configuration
   instances
      component a::array[n] of Aircraft.
      component t::array[n] of TowerTx/Rx.
      connector c::array[n] of AircraftToTower.
      connector d::array[n-1] of TowerToTower.
   connection
      FORALL i In 1..n @ IF inrange(i) THEN
         c(i).sender -> a(i).msg
         c(i).receiver -> t(i).msg
      IF Pipeline THEN
         FORALL i IN 1..n-1 @
            IF (inrange(i)and EXISTS j IN i+1..n @ (inrange(j) and
               FORALL k IN i+1..j-1 @ not inrange(k)))
            THEN
               d(i).finished -> t(i).finished
               d(i).start -> t(j).ready
   communication
      IF Pipeline THEN
         (SEQ i IN 1..n @
            IF inrange(i) THEN
              IF (EXISTS j IN i+1..n @ inrange(j))
              THEN
                 call(a(i)) -> flow(c(i))
                 -> call(t(i)) -> flow(d(i))
              ELSE
                 call(a(i)) -> flow(c(i)) -> call(t(i))
            ELSE
              skip)
      ELSE
         (PAR i IN 1..n @
            IF inrange(i) THEN
               call(a(i)) -> flow(c(i)) -> call(t(i))
            ELSE
               skip)
 end configuration
 end architecture
```

Figure 3: A generic air traffic control architecture

3 Instantiating the Generic Architecture

In this section we explain how generic architectures are used. To map a generic architecture to a concrete architecture we need to instantiate the architecture parameters with actual values. Different architectures can be generated by instantiating the values in different ways.

To begin with we show how the Air Traffic Control architecture shown in Fig. 3 can be instantiated to give a sequential communication architecture with four aircraft and the Tx/Rxs in the tower, with one aircraft (the second) out of range. To achieve such an architecture, we instantiate the parameters as follows:

$n \rightsquigarrow 4$
Pipeline $\rightsquigarrow$ *true*
inrange $\rightsquigarrow [true, false, true, true]$

The resulting architecture is shown in Fig. 4.

```
architecture AirTrafficControlPipeline is

begin interface
   component TowerTx/Rx(in msg:Msg, in ready: Bool, out finished:Bool).
   component Aircraft(out msg:Msg)
   connector AircraftToTower(in sender:Msg, out receiver:Msg)
   connector TowerToTower(in finished:Bool, out start:Bool)
end interface

begin configuration
   instances
      component a(1), a(2), a(3), a(4) of Aircraft.
      component t(1), t(2), t(3), t(4) of TowerTx/Rx.
      connector c(1), c(2), c(3), c(4) of AircraftToTower.
      connector d(1), d(2), d(3) of TowerToTower.
   connections
      c(1).sender -> a(1).msg
      c(1).receiver -> t(1).msg
      c(3).sender -> a(3).msg
      c(3).receiver -> t(3).msg
      c(4).sender -> a(4).msg
      c(4).receiver -> t(4).msg
      d(1).finished -> t(1).finished
      d(1).start -> t(3).ready
      d(3).finished -> t(3).finished
      d(3).start -> t(4).ready
   communication
      call(a(1)) -> flow(c(1)) -> call(t(1)) -> flow(d(1)) ->
      call(a(3)) -> flow(c(3)) -> call(t(3)) -> flow(d(3)) ->
      call(a(4)) -> flow(c(4)) -> call(t(4))
end configuration end architecture
```

Figure 4: A four aircraft sequential ATC architecture

Having instantiated the parameters we must check that the *applicability conditions* are satisfied. In this case the conditions can be trivially discharged, so we can proceed with configuring the architecture. By instantiating n to 4, four instances of the *Aircraft* component are created. These are named $a(1)$, $a(2)$, $a(3)$ and $a(4)$. Similarly, four *Tower* components, four *AircraftToTower* connectors and three *TowerToTower* connectors are also created.

Next, aircraft to tower Tx/Rx and tower Tx/Rx to tower Tx/Rx connections are created. Aircraft to tower Tx/Rx connections are only created for the in-range aircraft. For *i* between 1 and 4 we test whether *inrange*(i) is true; in this case it is true for all but the second aircraft, so connections are created only for these in-range aircraft and the corresponding tower Tx/Rx. The tower Tx/Rx to tower Tx/Rx connections are more complicated. In the case of the first tower Tx/Rx we get following conditional statement, derived by substituting *i* with 1 and *n* with 4.

```
IF (inrange(1)and EXISTS j IN 2..4 @ (inrange(j) and
   FORALL k IN 2..j-1 @ not inrange(k)))
THEN
   d(1).finished -> t(1).finished
   d(1).start -> t(j).ready
```

This states that we connect the first tower Tx/Rx to the next tower Tx/Rx that has an aircraft in range (if one exists). The second tower Tx/Rx does not have an aircraft in range, so we cannot connect to that. The third and fourth tower Tx/Rxs do have an aircraft in range, but the third is the most immediate, therefore $j = 3$, i.e. the first tower Tx/Rx connects to the third tower Tx/Rx.

Finally, the communication for sequential architectures is again dependent on whether the aircraft are in range. For the first aircraft we get the simplified condition:

```
IF inrange(1) THEN
  IF (EXISTS j IN 2..4 @ inrange(j))
  THEN
     call(a(1)) -> flow(c(1))
     -> call(t(1)) -> flow(d(1))
  ELSE
     call(a(1)) -> flow(c(1)) -> call(t(1))
ELSE
  skip)
```

The first line is true, since we know the first aircraft is in range. The second line is also true since we know that both the third and fourth aircraft are in range (in this case it is enough to know that there is at least one subsequent aircraft in range, it does not matter which one). Therefore the communication following the first "then" clause is used. In the case of the fourth aircraft, we note that it is in range, but there are no other subsequent aircraft in range, so there is no flow along a tower Tx/Rx to tower Tx/Rx connector.

To generate a parallel architecture we instantiate the parameter *Pipeline* to *false*. Supposing the other parameters are instantiated as before. The component and connector instances are the same as before. Connections are established between the aircraft in range and the corresponding tower Tx/Rx instance (i.e. for the first, third and fourth aircraft), but no tower Tx/Rx to tower Tx/Rx connections are established since the conditional in the if-then statement is false. The communication part is given by

```
(PAR i IN 1..4 @
        IF inrange(i) THEN
           call(a(i)) -> flow(c(i)) -> call(t(i))
        ELSE
           skip)
```

which after expansion and simplification becomes:

```
   (call(a(1)) -> flow(c(i)) -> call(t(1)))
|| skip
|| (call(a(3)) -> flow(c(3)) -> call(t(3)))
|| (call(a(4)) -> flow(c(4)) -> call(t(4)))
```

4 Conclusion and Future Work

In this paper, we have modelled an air traffic control system using the CRADLE language. CRADLE's support for parametrised architecture specifications allows us to specify a highly adaptable architecture within a single model. The architecture can be adapted by changing the number of components and connectors, as well as the topology of these connections

and the nature of the communications protocol. We have described the generic aspects of the CRADLE language by giving an example air traffic control architecture. We then illustrated how this architecture can be adapted to provide various concrete architectures. Next, we will attempt to model a real time system using CRADLE.

In the example given in this paper the CRADLE language conveys the structure of the architecture, but beside the communications protocol, little is said about the properties of individual components and connectors or the overall desired behaviour of the architecture. Currently we are looking at extending the language to allow functional and non-functional properties to be specified for components, connectors and the overall architecture. In particular we are focusing on timing properties. For example, we may specify the computation time for component, or the latency for a connection. For an overall architecture we might specify a deadline for the communication process. Having specified such timing properties, we will then check that the properties are satisfiable; for example we may check that a deadline can be met for a communication process, based on the computation times and latencies of the components and connectors. We are investigating the use of model checking and theorem proving to automate (or at least semi-automate) property checking.

In the case where functional or non-functional properties are violated, we will attempt to *correct* the architecture. For example, consider the case where a deadline property is violated. We could potentially satisfy this property by either reducing the number of components and connectors from the architecture (hence reducing computation times and latencies), or switch from a sequential to a parallel architecture. We will support this kind of architecture adaptation by matching our non-compliant architecture against a generic architecture (8), and then generating alternative instantiations of the generic architecture parameters until one is found that satisfies the properties (or until we have exhausted all possible alternatives). For example, given a concrete air traffic control architecture, with a sequential communication process, which violates a non-functional property. We would then match this against the generic air traffic control architecture, generating an instantiation of the parameters, then we would look at possible mutations of this instantiation. One possible mutation would be to change the instantiation $Pipeline \rightsquigarrow true$ to $Pipeline \rightsquigarrow false$, thus generating a parallel communication process, which could then result in an architecture satisfying the non-functional properties.

Acknowledgements

The support of the Australian Research Council with Grant No. DP0664479, and the Long Range Research team in AMS, AOD, the DSTO is acknowledged.

References

[1] Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. In *ACM Transactions on Software Engineering and Methodology* (TOSEM), 6(3):213–249, 1997.

[2] Binns, P. and Vestal, S. (2001). Formalizing software architectures for embedded systems. In T.A. Henzinger and C.M. Kirsch (Eds.): EMSOFT 2001, LNCS 2211, pages 451–468, Springer.

[3] Hemer, D. and Ding, Y. (2008). Specifying software architectures using a formal-based approach. In *Proceedings of the 19th Australian software engineering conference*(ASWEC08), pages 279–288, IEEE Computer Society.

[4] Feiler, P., Gluch, D. and Hudak, J. (2006). The Architecture Analysis and Design Language(AADL): An Introduction, Technical Note, CMU/SEI-2006-TN-011.

[5] Luckham, D. C. and Vera, J. (1995). An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[6] Medvidovic, N. and Taylor, R. (2000). A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering*. 26(1):70–93, 2000.

[7] Perez-Martinez, J. E. (2003). Heavyweight extensions to the UML metamodel to describe the C3 architectural. In proc. of ACM SIGSOFT Software Engineering Notes. 28(3):1–6, 2003.

[8] Zaremski, A.M. and Wing, J. M. (1997). Specification matching of software components. *ACM Transactions on Software Engineering*, 6(4):333–369, Oct. 1997.

[9] Shaw. M. et al. (1995). Abstractions for software architecture and tools to support them. In IEEE Transactions on Software Engineering. 21(4):314–335, 1995.