Consistency Checking for Component Reuse in Open Systems

Peter Henderson[1] and Matthew J. Henderson[2]

[1] Electronics and Computer Science, University of Southampton, Southampton, UK
p.henderson@ecs.soton.ac.uk
[2] Mathematics and Computer Science, Berea College, Berea, KY 40404, USA
matthew_henderson@berea.edu

Abstract. Large scale Open Systems are built from reusable components in such a way that enhanced system functionality can be deployed, quickly and effectively, simply by plugging in a few new or revised components. At the architectural level, when new variations of a system are being planned by (re)configuring reusable components, the architecture description can itself become very large and complex. Consequently, the opportunities for inconsistency abound. This paper describes a method of architecture description that allows a significant amount of consistency checking to be done throughout the process of developing a system architecture description. An architectural design tool is described that supports consistency checking. This tool is designed to support component reuse, incremental development and collaborative working, essential for developing the architecture description of large systems.

1 Introduction

Systems Architecture is that branch of Information System design that determines the overall structure and behaviour of a system to be built. Typically, an architecture is captured as an evolving set of diagrams and specifications, put together by a team of System Architects and iteratively refined over a period of consultation with the customers for the solution. The extent to which the architecture represents a buildable or procurable solution depends a great deal on the consistency and completeness of the architecture description and the extent to which it can be validated prior to commitment to procure.

Validation of the architecture as early as possible in the process of development is important. This aspect of System Engineering is not well supported by tools. In this paper we advocate an approach to architecture description that lends itself to validation throughout architecture development.

Open Systems have modular, or component-based, architectures based on a catalogue of reusable components with publicly maintained interfaces. Devising a new configuration involves selecting reusable components from the catalogue, devising new ones or variations of existing ones, and plugging them together according to the architecture description. Opportunities for inconsistent reuse of existing components are particular pitfalls which need to be avoided eventually, but which need, for pragmatic reasons, to be tolerated during design and development of a complex architecture description.

The key idea is that the architects define a metamodel that enumerates the types of entities they are going to use in their description, along with the relationships between these entities. For example, as in this paper, they might choose to describe their architecture in terms of components and interfaces.

As part of the metamodel, the architects will also specify constraints which a *valid* description must satisfy. Validation of the architecture description comprises checking the extent to which these constraints are satisfied. We have developed a tool, WAVE, to support this approach.

2 Background

In Systems Engineering, in particular for software intensive systems, the design of a solution is normally developed around a system architecture.The description of this architecture is a shared model around which the architects work to understand the customer requirements and how these can be mapped on to programs and databases to realise a solution.

The field of architecture description is now relatively mature [1–7]. Specific approaches to architecture description have, in many respects, found their way into standard representations such as UML [8] and SysML [9], so it is now quite common to find these notations in use in industrial projects. There are other approaches to architecture description [10–21] but these are generally ideas that are readily adopted as specialisations of the standard notations. Indeed, the Model Driven Architecture approach to system development [22] effectively assumes that a notation needs to be open to extension and specialisation.

The need for architects to share a model, and for this model to evolve, immediately introduces the realisation that for most of its life the architecture description will be incomplete and probably inconsistent. Looking for inconsistencies in the architecture description is the principal means of validating it early in its life [23–25, 11, 26]. The research reported here builds on those ideas.

In particular, like others, we take an approach to architecture description based on relational models [10, 23]. We capture the details of an architectural description in UML or SysML, but support this with a precise *specialised* metamodel that has been expressed relationally. This means that we can capture the consistency constraints very precisely in relational algebra [27, 28] and formally validate the metamodel as we develop the architecture description.

The consequences for the research reported here are that we have a method of capturing an architectural metamodel, of capturing an architecture according to this metamodel in UML and a means of presenting that architecture for consistency checking throughout its life. We claim that this method (and our tool) supports component reuse, incremental development and collaborative working, for which we will give evidence in a later section. In order to describe the method, we begin with an example of architecture description based on UML Component diagrams.

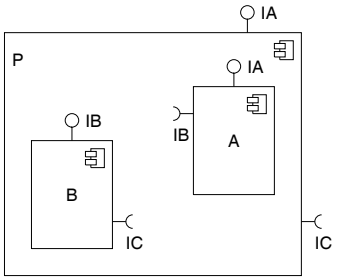

Fig. 1. A consistent structure

3 Components and Interfaces

Large scale Open Systems benefit from having their architecture described using Component diagrams in UML. These diagrams use Components to denote units of functionality and they use Interfaces to show how these units are plugged together to form assemblies with greater functionality. Components often denote quite substantial units of functionality (such as web servers, database servers etc.). Moreover, in large Open Systems there will often be dozens or hundreds of these replaceable components of various sizes and in many versions.

Figure 1 shows a (simplified) Component diagram in which we see that Component *P* has nested within it two further components *A* and *B*. Strictly speaking, because it shows nesting, this is actually an example of a UML 2.0 Composite Structure diagram [8], where that diagram has been specialised to show nesting of Components.

The interfaces in Figure 1 are shown by the ball-and-socket notation. For example Component *A* shows that it *requires* an Interface *IB* by its use of a socket with that label. Fortunately, also nested within Component *P* is a Component *B* which *supplies* such an interface, shown by the ball with that label. Normally, in a UML 2.0 diagram, this association between *requires* and *supplies* would be shown by an arrow from one to the other. Rather than do that here, since our diagrams are so simple, we have relied upon the reader's ability to associate the two Components through their reference to an Interface by name.

We see that all the components in Figure 1 supply or require interfaces. Normally, a component will both require and supply many interfaces, not just one of each as shown in this simple example. We say that the example in Figure 1 is *consistent* because all of the interface requirements are satisfied. The fact that Component *B* requires Interface *IC* is satisfied by this being brought to the outside and shown as a required interface on the parent *P*. Similarly, that Component *P* supplies Interface *IA* is satisfied by the fact that it contains a nested Component *A* which is the source of this interface.

In contrast Figure 2 shows an inconsistent Component diagram. This time Component *Q* contains Components *A* and *D* which leads to some mismatches. Most obviously, we see that Component *A*, in this context, does not have its requirement for Interface *IB* satisfied, because there is no sibling that supplies that interface, nor has it been brought to the outside and made a required interface of the parent *Q*. We refer to this missing connection as *dangling requires*. We say that, within Component *Q* there is a dangling-requirement for Interface *IB*.

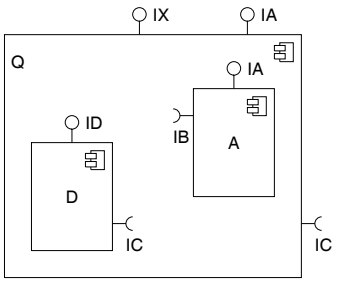

Fig. 2. An inconsistent structure

Moreover Figure 2 shows an example of what we will call a *dangling supplies*. This is because Component Q supplies Interface IX but that is not one of the available interfaces supplied by one of its nested members. Again, note that this is a consistency constraint which is specialised from the metamodel that we are adopting and this will be explained later. Further, while we say that Component Q has a dangling-supplies of Interface IX, we do not consider the unused Interface ID of Component D to be a problem (again, a decision of the specialised metamodel)

So far, what we have presented is an example of the type of Architecture Description that we advocate. It is based on a metamodel that we will introduce in a little while. The metamodel determines what types of entities will be described (here Components and Interfaces) and the consistency constraints that they must satisfy. In general, the metamodel will be defined by the Systems Architects for the specific system being designed and may use quite different entities and/or consistency constraints. We will discuss other metamodels later but first we will show how this one is formalised.

4 Specialised Metamodels

Consider the way in which components and interfaces are conventionally described in a design notation such as UML or in a programming language such as Java or Python. A component will normally supply a number of interfaces and also make use of a number of interfaces supplied by other components.

Figure 3 shows the entities and relations introduced by this description. It is the beginning of the metamodel against which we will check our system descriptions. The rest of the metamodel comprises the consistency constraints among these entities. Of course, this simple metamodel also needs to be extended with additional entities to be sufficiently useful in practice and these entities in turn will require further constraints.

We will formalise the constraints that a correct design must obey in terms of the basic relations shown on the metamodel diagram. We denote the (natural) join operator of two relations by a dot (as for example in Alloy [29]). It forms the relational composition of its operands. Thus, for example

```
contains.requires
```

denotes a binary relation formed from the composition (join) of two existing binary relations. That is, `contains.requires` denotes the relationship between Components and

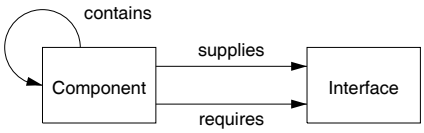

Fig. 3. A metamodel

Interfaces that we would describe as "Component `c` contains an unnamed Component that requires Interface `i`" .

It is worth noting at this point that this focus on the whole-relation, leads to a holistic approach to the analysis of Systems Architectures, which is something we will return to in the section on Pragmatics. It is almost always the case that our relations are many-to-many. The relational algebraic approach affords a way of "reading-off" the diagram the derived relations that will be constructed as consistency rules in our metamodel.

The way in which one uses relational algebra, as we will illustrate in the next section, is to construct predicates and challenge the consistency checker to construct the set of entities that fail to pass the test.

5 Consistency and Completeness

The Architecture Description technique that we advocate assumes that the System Engineer will specify a metamodel and record the design against that metamodel. The metamodel will comprise entities, relationships and constraints. This section describes two such metamodels, shown respectively in Figure 3 and Figure 4. We develop consistency constraints that, according to the System Engineer who designed these metamodels, are among those that need to be satisfied if the Architecture being described is to be consistent. We also address a notion of completeness.

We will assume that during development of the Architecture Description, interim models will not be consistent. The consequence for us here is that the constraints will be specified as sets of inconsistencies. The designer's eventual objective is to achieve a design in which these sets of inconsistencies are eliminated. This approach to design supports both incremental and cooperative working.

5.1 Dangling Requires

As a first example of a consistency rule, let us define the example we discussed in an earlier section, dangling-requires. Using the relations and entities illustrated in Figure 3 we can construct the relation `dr` as follows

```
dr = contains.requires - contains.supplies - requires
```

Here, in addition to using relation join (composition) denoted by dot, we have used set difference, denoted by minus. This expression defines a relation `dr` which relates Components to Interfaces. The relation `contains.requires` contains all pairs `(c,i)` with the property that `c` contains a Component that requires `i`. Similarly `contains.supplies`

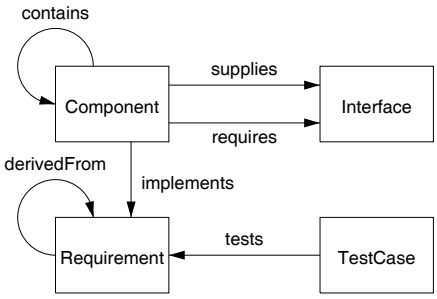

Fig. 4. An extended metamodel

contains all pairs `(c,i)` with the property that `c` contains a Component that supplies `i`. Thus the difference of these two relations contains all pairs where `c`'s requirement for `i` is not satisfied internally.

Finally, by then constructing the difference between this set and the relation `requires`, we have that `dr` is the relation between Components `c` and Interfaces `i`, where `c` contains a nested Component that requires `i` but where that Interface is neither supplied internally nor required by the parent. This is exactly what we meant by dangling-requires.

Constructing the relation `dr` has two benefits. First we have accepted that during development a design will be inconsistent and have decided to derive at any point in time a set of inconsistencies that the designer will eventually wish to remove. Second, by constructing a relation, we have taken a holistic approach, addressing the whole architecture description with our analysis rather than just looking at local inconsistencies.

5.2 Dangling Supplies

We described informally, earlier, what we mean by dangling supplies. Formally, in terms of our metamodel we can specify this as follows

```
ds = dom(contains)<:supplies - contains.supplies
```

The operator `<:` is domain-restrict. The first term in the definition of `ds` is just the relation `supplies` restricted to the domain of `contains`, which is just the relationship between *composite* Components and the Interfaces they supply. By constructing the difference between this relation and `contains.supplies` we get `ds` which relates composite Components to Interfaces that they supply but which are not supplied by any of their children. Precisely what we meant by dangling-supplies when we introduced it informally, earlier.

5.3 Replacements

As a final example of a consistency constraint imposed by a metamodel on an Architecture, consider the situation when our Architecture is for an Open System, where we have potentially alternative suppliers of interchangeable Components. A system is Open if

its interfaces are fully defined and available for exploitation, in that alternative suppliers can produce replacement or enhanced Components that plug into slots vacated by other components. How can we determine which Components are potential replacements for others?

Consider

```
canReplace =
   { (c1, c2) | supplies[c2] <= supplies[c1] and
                requires[c1] <= requires[c2]}
```

As before, this is a binary relation (`<=` denotes subset and `[]` denotes relational image). It is the relationship between Components with the property that if `(c1, c2)` is in `canReplace` then `c1` can replace `c2`, wherever it might occur, simply because it supplies all the Interfaces that `c2` must supply and requires only Intefaces supplied in the location that `c1` would occupy. The way that this computed relation is used in practice is when (as in Figure 2) there is a mismatch, we can use `canReplace` to determine possible candidates to replace `D`.

This means that we have, in this metamodel, taken a particular view of what we mean by an Interface. An entity which represents an Interface by name, effectively encodes in that name both the syntax and semantics of the Interface. This is not unusual in practice but does leave undeveloped here how unequal, but related Interfaces, are to be handled in our metamodel. This is beyond the scope of this paper.

5.4 Completeness

In addition to rules for checking consistency of an architecture as it is developing, there will be many rules that specify completeness of the description. An example of such a rule for the metamodels used in this paper might be that every Component should have at least one Interface that it either requires or supplies.

When constructing a constraint for completeness we will work in the same way as we have for consistency and report incompletenesses during development. For example, we might report the set of Components for which there are, as yet, no Interfaces either required or supplied.

In other systems for which we have developed metamodels, the kind of completeness rules we have developed include constraints such as every Component/Requirement pair should have at least one TestCase (see, for example Figure 4) or the constraint that every entity should have at least one Documentation fragment attached to it.

Reports generated of architectures in development would then include sections listing incompletenesses alongside those listing inconsistencies. The Architect's objective would be, eventually, to eliminate these sections.

6 Pragmatic Issues

The example we have developed in the paper is rather simple. In practice, Architecture Descriptions of this sort can be very large. They will normally be developed incrementally and collaboratively by a team. They will also be constrained by the fact that they

are building from legacy components and/or devising a modular architecture that comprises reusable components. Tools to support this type of development process must be able to deal with the consequences of these observations.

6.1 Tools

The WAVE tool allows its users to describe Architectures. It will produce documentation based on these descriptions, including reports on the inconsistencies.

A prototype implementation of WAVE is available at `http://ecs.soton.ac.uk/~ph/WAVE`. It comprises a collection of scripts which transform among representations of the architecture descriptions and which support the merging of independently developed models.

WAVE also supports the inclusion of specialised metamodels. The prototype assumes that descriptions made according to these models will have an XML representation. In practice, we have used WAVE as an adjunct to a UML tool (Sparx Systems Enterprise Architect) from which the architecture description can be dumped as an XMI file.

This XMI description is first turned into a realtional model, capturing data from the XML according to the chosen metamodel. The consistency checking is then done by a script that computes the "inconsistency" relations such as `dr`, `ds` and `canReplace` as described here. A further script then constructs the documentation of the architecture including the reports on inconsistencies in that report.

We plan to replace this XML based implementation with one based on a conventional relational database in the near future, in order to inherit the robustness and transactional properties of those systems. The architect will then be able to publish their incremental description and share it with others via that shared database.

6.2 Collaborative Working, Incremental Development and Component Reuse

The architects work by capturing new parts of the description and then using the tool to generate interim documentation. The inconsistencies are highlighted. So they continue to develop, all the while trying to remove inconsistencies they have added.

Collaborative working is supported because the architects can work independently as follows. Each publishes their models to the others. When running consistency checking, each architect can merge their work with the work of others. The generated interim documentation contains inconsistencies that are now across the *whole architecture*. It will be apparent to each architect which inconsistencies are their responsibilities. They continue to work on their independent parts, while following this process of incremental development.

Where the architecture is something being developed around a catalogue of reusable components (which is the norm these days) then the component descriptions can be held with the component in such a way that integrating them into a new "build" (i.e. build of the architecture) is straightforward. Thus plug-and-play at the eventual implementation stage, that is afforded by reusability, is mirrored at the design stage by having reusable component descriptions.

7 Conclusions

We contend that architecture description is an important problem for Information Systems development, especially for large Open Systems with many architects, many engineers and many legacy components.

We have observed that such an architecture description is likely to be incomplete and inconsistent for much of its life. We have argued that capturing the architecture description according to a precise specialised metamodel introduces the kind of redundancy in description that allows inconsistencies to be detected early in the life of an architecture description.

We have given examples of the kinds of metamodels that can be developed. We have described tools based on UML that allow a pragmatic approach to developing both metamodels and architecture descriptions that are compliant to those metamodels. The particular example we have chosen to illustrate the method used Components and Interfaces, which are particularly appropriate in the context of component reuse. We have shown how the method lends itself to reuse of component descriptions, as well as to reuse of components.

In the future, we will refine our methods and tools in particular ways, not least in integrating the architecture description with its documentation based on narrative structure [26]. We plan to publish detailed metamodels that we have developed along with collaborators. We recognise that many of the more complex consistency constraints that we specify have analogues in graph algorithms and wish to pursue this potentially rich theme. In particular, it might allow us to investigate the kinds of architectural complexity that Alexander described in the 60s and then deprecated in the 70s [30] or put more flesh on the bones of interesting new developments such as the Algebra of Systems [25]

The method we have presented is now quite mature and is being applied in practice by our industrial collaborators. The tool is usable and integrates well with established tools. We believe that the method and tools together constitute a sound and practical method of enhancing architecture description.

References

1. Kruchten, P.: Architectural Blueprints - the 4+1 view model of software architecture. IEEE Software **12**(6) (1995) 42–50
2. Shaw, M., Garlan, D.: Software Architecture - Perspectives on an emerging discipline. Addison-Wesley, Upper Saddle River, NJ (1996)
3. Maier, M.W., Rechtin, E.: The Art of System Architecting, 2nd Ed. CRC Press LLC, Boca Raton, FL (2002)
4. Henderson, P.: Laws for dynamic systems. In: International Conference on Software Re-Use (ICSR 98), IEEE (1998)
5. Henderson, P., Yang, J.: Reusable web services. In: 8th International Conference on Software Reuse (ICSR 2004), IEEE (2004)
6. Rozanski, N., Woods, E.: Software Systems Architecture. Addison-Wesley, Upper Saddle River, NJ (2005)
7. Shaw, M., Clements, P.: The golden age of software architecture. Software **March/April** (2006) 31–39
8. OMG: Unified Modeling Language, superstructure. http://www.uml.org (2007)

9. OMG: OMG Systems Modeling Language. http://www.uml.org (2006)
10. Holt, R.C.: Binary relational algebra applied to software architecture. In: CSRI Technical Report 345, University of Toronto (1996)
11. Hadar, E., Hadar, I.: Effective preparation for design review - using UML arrow checklist leveraged on the guru's knowledge. In: OOPSLA, ACM (2007)
12. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: A platform independent component modeling language for distributed real-time and embedded systems. J. Comput. Syst. Sci. **73**(2) (2007) 171–185
13. Dekel, U., Herbsleb, J.D.: Notation and representation in collaborative object-oriented design: an observational study. In: SIGPLAN Notices, Volume 42 , Issue 10, ACM (2007)
14. Egyed, A.: UML/analyzer: A tool for the instant consistency checking of UML models. In: ICSE '07: Proceedings of the 29th international conference on Software engineering, New York, NY, USA, ACM (2007)
15. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of statecharts specifications. In: In 29th International Conference on Software Engineering (ICSE07). (2007) 54–64
16. Balzer, B.: Tolerating inconsistency. In: ICSE '91: Proceedings of the 13th international conference on Software engineering, New York, NY, USA, ACM (1991)
17. Egyed, A.: Instant consistency checking for the UML. In: ICSE '06: Proceedings of the 28th international conference on Software engineering, New York, NY, USA, ACM (2006) 381–390
18. Egyed, A.: Fixing inconsistencies in UML design models. In: ICSE '07: Proceedings of the 29th international conference on Software engineering, New York, NY, USA, ACM (2007)
19. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., Chechik, M.: Consistency checking of conceptual models via model merging. In: In RE. (2007) 221–230
20. Sabetzadeh, M., Nejati, S., Easterbrook, S., Chechik, M.: Global consistency checking of distributed models with tremer. In: In 30th International Conference on Software Engineering (ICSE08), 2008. Formal Research Demonstration (To Appear
21. Nuseibeh, B., Easterbrook, S., Russo, A.: Making inconsistency respectable in software development. Journal of Systems and Software **58** (2001) 171–180
22. OMG: MDA guide. http://www.uml.org (2003)
23. Beyer, D., Noack, A., Lewerenz, C.: Efficient relational calculation for software analysis. Transactions on Software Engineering **31**(2) (2005) 137–149
24. Chang, K.N.: Consistency checks on UML diagrams. In: International Conference on Software Engineering Research and Practice, SERP07, IEEE (2007)
25. Koo, B.H.Y., Simmons, W.L., Crawley, E.F.: Algebra of systems: An executable framework for model synthesis and evaluation. In: Proceedings of the 2007 International Conference on Systems Engineering and Modeling, IEEE (2007)
26. Henderson, P., de Silva, N.: System architecture induces document architecture. In: 20th International Conference on Software Engineering and Knowledge Engineering (SEKE 2008), IEEE (2008)
27. Date, C.: Database in Depth - Relational Theory for Practitioners. O'Reilly Media Inc., Sebastopol, CA (2006)
28. Beyer, D.: Relational programming with CrocoPat. In: ICSE, IEEE (2006)
29. Jackson, D.: Software Abstraction. MIT Press, Cambridge, MA (2006)
30. Alexander, C.: Notes on the Synthesis of Form (with 1971 preface). Harvard University Press, Cambridge, MA (1964)