

Addressing Non-Functional Properties in Software Architecture using ADL

Christopher Van Eenoo Osama Hylooz Khaled M. Khan
School of Computing and Information Technology
University of Western Sydney
Locked bag 1797 Penrith DC
NSW 1797 Australia
k.khan@uws.edu.au

Abstract

Architecture definition languages (ADLs) are used to specify high-level structural details of software systems. Many ADLs have emerged recently and whilst all address structural and functional elements such as components and connectors, few can be used to specify non-functional requirements such as security or performance. This paper proposes some constructs to one such ADL, WRIGHT, for specifying non-functional requirements and assurances. The proposed ADL constructs are used in an example to demonstrate their applicability. Although the constructs are informal at this stage of the research, this will enhance the user's ability to understand the non-functional requirements and assurances of the system.

1. Introduction

System architecture is commonly described from multiple views. Each view addresses a specific set of concerns, and captures some particular aspects of the system. In the current practice, system architecture only reveals the systems functionality, business objects and their collaborations. In the current state of practice and theory, systems level non-functional properties such as maintainability, usability, security, portability, reusability are not integrated with the design artifacts due to lack of proper supporting methodology or language [8]. Getting the non-functional properties correct with the system functionality and operations would definitely set the stage for everything to come in the system's life. Ignoring these issues during the architecting phase means that the quality of the entire system may go dangerously wrong, which may cause the entire fabric to unravel [10].

Software systems are usually characterized both by their functionality and by their non-functional properties. Both aspects are relevant to software development but, traditionally much work has been done for the former than for the latter, especially concerning specification languages and, lately architecture definition languages (ADL). The primary purpose of an ADL is to specify the structural composition of a software system in terms of

system's components and connectors through the means of a formal representational language. Reasons for the lack of support for specifying non-functional properties in existing ADLs could be: (1) non-functional properties are usually too abstract, and stated informally; (2) in most cases the separation of the non-functional requirements from the functional requirements is not simple; and (3) non-functional properties are often considered as an 'after thought' of the development process.

Without incorporating the non-functional properties of software systems along with the systems functionality at the architectural level, systems architects consequently may end up with wrong architecture by unknowingly contradicting the security features of the system [9]. Even a modification or alternation of a design feature may unknowingly undermine the already employed non-functional properties of the system. We need a fresh approach to address this pressing issue.

It is becoming increasingly important for software architects to specify systems non-functional properties at the architectural level [11,12]. ADLs must therefore be broadened and extended to incorporate methods which allow for the unambiguous specification of non-functional properties. This point serves as the impetus of this paper. By allowing for the specification of non-functional requirements and assurances in software architectures, the field of component based software engineering can take another step towards becoming a more matured discipline. This paper proposes some ADL constructs which use a combination of WRIGHT [2], CSP notation [6] and structured English. This combination compels the proposed constructs to be formal yet still flexible and readable, promoting adoption and wide-spread use.

The rest of the paper is organized as follows. A literature review cited in section 2 discusses some of the related work. Section 3 briefly describes WRIGHT notations, and then goes on to introduce our proposed extensions to WRIGHT. The proposed constructs are then illustrated with an example in section 4. Following the example, the strengths and weaknesses of the proposed extensions are discussed in section 5. Finally, the paper concludes in section 6.

2. Related Work

A quick review of related literature on ADL suggests that is a notable lack of support for specifying non-functional properties in existing ADLs, despite the need for and many benefits of doing so [1]. Of the few ADLs that do provide functionality for specifying non-functional requirements, for example, MetaH provides only limited notational functionality for modeling schedulability, reliability and security whereas UniCon is limited to modeling schedulability only [1]. Although not entirely architecturally focused, in their paper on composing security-aware software with components, Khan and Han [3] propose a security characterisation framework which addresses the specification of components' security properties, the runtime analysis of a multi-component system's compositional security properties, the analysis of an entire system's security properties and how to make the characterised properties available at runtime. The paper focuses on the publishable security properties of atomic components in terms of *required* or *ensured* security properties. Originally intended for component based systems, the paper has broader implications for software architecture.

Another work [4] addressing solely on quality characteristics in ADLs has proposed a generic method which allows the incorporation of non-functional properties into software architecture. This approach, called 'NoFun', does not use any particular ADL notation. Therefore, the work has developed an ad-hoc notation for describing system units (components and connectors), which are considered the functional parts of the system specification. Focusing on the non-functional issue, it distinguishes between three concepts. First, *non-functional attributes* which describe and/or evaluate the software system (such as efficiency, reliability, and usability). Second, *non-functional behavior* which refers to the values assigned to the non-functional attributes and finally, *non-functional requirements* which are sub-factors of the non-functional attributes. For each component and connector two separated parts are attached --the *non-functional specification* containing *non-functional attributes*, and the *non-functional implementation* containing *non-functional behaviors*. NoFun is a powerful language because non-functional attributes can be defined in various ways and bounded to a variety of software units.

The Process^{NFL} [7] has been designed to consider specific characteristics of non-functional requirements: such as the correlations and conflicts between different non-functional properties. This language can be used to express the non-functional properties at three abstract levels such as how to define the non-functional properties (NF-attributes), how to express constraints of the properties (NF-Properties), how to make decisions (NF-

Actions) in order to achieve the constraints (NF-Properties) imposed over NF-Attributes.

Some recent work in aspect oriented software development (AOSD) trying to capture the non-functional constraints in architectural description have been reported in [13, 14, 15]. AOSD focuses on specific crosscutting issues referred to as aspects such as persistence, security, dynamic behaviour, and performance of the system [13]. To specify the non-functional aspects, [14] proposes a method called creating aspectual scenarios which represent crosscutting scenarios using interaction pattern specification. The similar approach reported in [15] also uses pattern to describe crosscutting concerns. However, these approaches do not address the issue of automatic reasoning of the non-functional requirements and assurances in the architecture.

The design philosophy of QuA proposed in [16] is to identify desirable features of current components architectures and to experiment to support QoS management of the system. It provides a potential viable solution for mapping between application level QoS and resource level management decisions. The approach is appropriate at the very high level of the system design level. In another approach, a lexical modeling language for QoS called Component Quality Modelling Language (CQML) [17] could be used in addition to an interface definition language (IDL/CIDL) to specify QoS of the software components. The approach is very promising, but it could be further extended to enable reasoning about the security properties offered by different components at run-time.

3. Our approach

This section begins with a description of the existing WRIGHT notation and highlights its limitations in terms of non-functional requirements.¹ The section concludes by introducing our proposed extensions, which enable the specification of non-functional properties, to the WRIGHT notation.

3.1 Description of WRIGHT

WRIGHT ADL is designed around the three basic software architectural elements: *components*, *connectors* and *configurations* [2]. A component description consists of two sub-elements, the *interface* and the *computation*. The *interface* sub-element may consist of any number of *ports*. Each port is described in terms of a number of events. An event is the basic unit that represents an action of the behavioural specification. A port represents an

¹ This paper uses the WRIGHT notation as an example of a typical ADL. The proposed extensions, however, have implications to most other ADLs.

interaction in which the component may participate with its environment. The computation describes what the component actually does in terms of events specified in the ports. The computation realizes the interactions specified by the ports. A connector description also consists of two sub-elements, the *role* of the connector and the *glue* of the connector. A role defines the behaviour of the participant in an interaction. In other words, a role of a connector specifies what a participant component is expected to do. The glue of a connector describes how the participants work together to create a composition. The glue represents the behavioural specification of the connector. Components and connectors are combined into a configuration description which consists of *instances* and *attachments* [2]. A configuration is a collection of component *instances attached* via connectors. The *attachment* declarations put together each of the instances of an architectural description. The *attachment* specification in WRIGHT ADL merges the descriptions of these elements and sub-elements to construct an architectural definition of a system. Figure 1 shows a simple example of the WRIGHT notation.

WRIGHT syntax is heavily based on communication sequential process (CSP) [6]. The bold texts are reserved words in WRIGHT. The right arrow shows the sequence of the events. The overbar denotes the initiated event by a process. The § symbol represents a stop sign of a process. More on the WRIGHT notations can be found from [2] and [6].

WRIGHT is a robust ADL but still has its shortcomings. WRIGHT does not provide the ability for refinement maps or traces and has constrained dynamism. WRIGHT is also constrained by the rule that ports may only be attached to roles and vice versa [1]. Another major shortcoming, and the focus of this paper, is its inability to specify non-functional requirements. This concern is directly addressed by the following proposed constructs to the existing WRIGHT notation.

3.2 Addressing non-functionality in WRIGHT

The new construct to the existing ADL must support the followings: (1) to specify the non-functional property; (2) to differentiate the required properties from the ensured properties by an entity in a composition; and (3) to handle the multiple number of properties in one interface element. This construct contains two classes of properties: required and ensured. We argue that a software system could have a non-functional property which might be ensured or required by an entity. This construct also contains such information as from which entity non-functional properties are required and which entities ensured properties are provided to. The construct

could also handle multiple requirements and assurances. The proposed construct to the WRIGHT ADL is discussed as follows.

Non-functional Prop = (REQUIRES \supseteq {*provider.rrqmt*},
ENSURES \supseteq {*port/role.erqmt*})

This construct can be read as “non-functional property *Prop* requires *rrqmt* from *provider* and ensures *erqmt* on *port/role*.” In the proposed expression, ‘**Non-functional**’ is a reserved word which serves as a header indicating that the following descriptions pertain to non-functional properties of the component or connector in question. The requirements or properties are not listed under *constraint* descriptions of WRIGHT because *constraints* pertain neither to structural issues, nor under the *glue* or *computation* descriptions because they pertain to behavioral or functional issues. ‘Non-functional’ is expressed in bold font and is indented similar to ‘Port’ or ‘Role’ under the ‘Component’ or ‘Connector’ headings respectively.

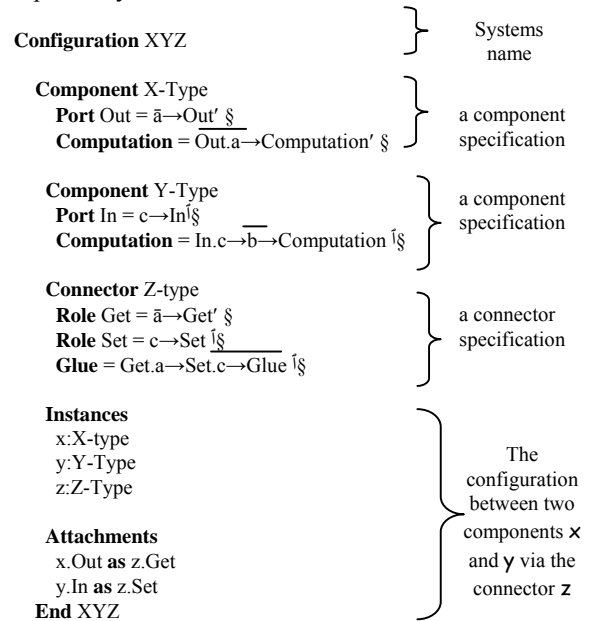


Figure 1: Example of WRIGHT notation

The term ‘Prop’ is a variable to be replaced with a non-functional property such as security, performance or efficiency. It is recommended, however, to restrict the property names to those identified by ISO in their software product quality model [5].² As many properties as necessary can be listed under one ‘Non-functional’ heading. Properties are written in title case followed by

² Care should be taken, however, on the use of the quality term ‘Functionality’ because there is a strong argument that these requirements are intrinsically *not* non-functional and therefore the description of these types of requirements should be modelled using WRIGHT’s existing methods for *behavioural* notation.

an '='. All descriptions pertaining to a particular property are enclosed in parentheses and are separated internally by commas.

The term 'REQUIRES' is a reserved word which indicates the functional requirements that the component or connector *must* receive from its environment. The term 'ENSURES' is also a reserved word which indicates the functional properties that the component or connector *always* provides to its environment. Both reserved words are written in uppercase. If a non-functional property has needs for only one of these reserved phrases, they must still both be listed. Requiring that 'REQUIRES' and 'ENSURES' be listed for each non-functional property listed, along with the concept of an empty requirement set, allows the architect to explicitly state the *absence of functional requirements* and avoids any potential ambiguities. Empty requirement sets are noted as '{}'.
 Immediately following the 'REQUIRES' and 'ENSURES' reserved words but separated by a '⊇', the CSP notation for 'contains', is a listing of the properties and a description as to where required properties are required from and to where ensured properties are provided. These requirement phrases are written in structured English. The term '*rrqmt*' indicates the property that is required whereas the term '*provider*' indicates the component or connector from which '*rrqmt*' requires the property. Similarly, the term '*erqmt*' indicates the property that is provided and '*port/role*' indicates to where it is provided, usually it might be a 'Port' or a 'Role'. The concatenations of each '*provider*' to '*rrqmt*' and '*port/role*' to '*erqmt*' are enclosed in curly brackets, italicized and are joined with a period. Because both 'REQUIRES' and 'ENSURES' must always be listed but may not always have requirements, the provision has been made for noting an empty requirement set and has been described above. If the required properties are not met by the provider, the event is assumed to end in an unsuccessful termination and therefore no new notation has been identified for this purpose.

The above rules form the basis of the proposed extensions. The rules can, however, be extended to include multiple requirements and mathematical notation where necessary.

In order to express that a single non-functional property requires multiple requirements, the following notation may be used:

$$\begin{aligned} \text{Non-functional Prop} = & (\text{REQUIRES} \supseteq \{ \textit{provider}_a.\textit{rrqmt}_a \}, \\ & \text{REQUIRES} \supseteq \{ \textit{provider}_b.\textit{rrqmt}_b \}, \\ & \text{ENSURES} \supseteq \{ \textit{port/role}.\textit{erqmt} \}). \end{aligned}$$

The above notation is applicable to the 'ENSURES' phrase as well. This notation is to be used when the multiple 'REQUIRES' or 'ENSURES' properties are relatively unrelated. Alternatively, to express requirements that are closely related, they are expressed

together within the curly brackets and are joined by a semi colon ';', the WRIGHT sequencing operator [2]:

$$\begin{aligned} \text{Non-functional Prop} = & (\text{REQUIRES} \supseteq \{ \textit{provider}_a.\textit{rrqmt}_a, \\ & \textit{provider}_b.\textit{rrqmt}_b, \\ & \textit{provider}_c.\textit{rrqmt}_c \}, \\ & \text{ENSURES} \supseteq \{ \textit{port/role}.\textit{erqmt} \}) \end{aligned}$$

The above notation is used to express requirements originating from a single source or properties supplied to a single sink. To further extend the notation when the form '*provider.rrqmt*' is insufficient to express all requirements, mathematical notation in conjunction with structured English may be used:

$$\begin{aligned} \text{Non-functional Prop} = & (\text{REQUIRES} \supseteq \{ \}, \\ & \text{ENSURES} \supseteq \{ \textit{port/role}.\textit{erqmt} > x \}) \end{aligned}$$

where x is some numeric value or attribute. To indicate that a requirement is required from all components or connectors in its environment, the phrase '*all*' is used in place of '*provider*' or '*port/role*':

$$\begin{aligned} \text{Non-functional Prop} = & (\text{REQUIRES} \supseteq \{ \textit{all}.\textit{rrqmt} \}, \\ & \text{ENSURES} \supseteq \{ \textit{all}.\textit{erqmt} \}) \end{aligned}$$

The above additions to the proposed extension allow for a wide range of non-functional properties to be expressed unambiguously yet still in an easy to read manner. To see the proposed extensions in practice, an example is shown in the next section.

4. Example

To illustrate how the proposed construct works, we use an example similar to that used in [3]. Consider an e-healthcare system that regards all clinical information passed between stakeholders as confidential. Stakeholders include general practitioners (GP), specialists (SP) and pharmacists (PH).

Several GP-type components provide patients' data to their environment. The data is consumed by a 'shared variable' connector. The connector does not accept any patients' data from such a component unless it is authenticated. In addition, the 'shared variable' does not allow the simultaneous action of reading and writing. A GP component first authenticates itself to its environment before it starts writing test results data on its outbound port.

An SP-type component gets the patients' data from the 'shared variable' connector and begins computing the diagnosis of the patient based on the available test results data. The diagnosis report generated by the SP is then written to its outbound port. However, before the SP component may begin writing the diagnosis report, it ensures that the receiving environment, a pipe connector,

has a minimum buffer size of 50kb and that the data reading speed is at least 1Mb/second.

A PH-type component reads a diagnosis report from a pipe from which it then produces (computes) a prescription. The prescription is then forwarded back to the originating GP-type component through a remote procedure call connector. The prescription, however, is only accepted by the GP if it is encrypted with a public key. This forces the PH component to send all prescriptions encrypted. If the GP component determines that a required security property has not been provided by its environment, the process is terminated. If the required security property is ensured, the GP component will receive all data before terminating successfully.

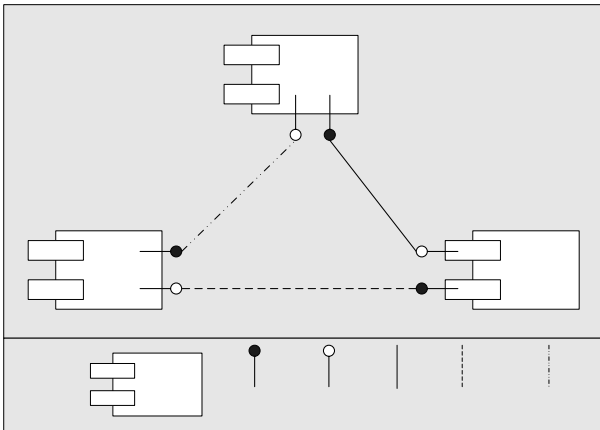


Figure 2: Conceptual diagram of the example

A conceptual diagram of the components, connectors and data involved in the example is shown in Figure 2. The example specification translated into the WRIGHT notation is presented in Figure 3. Because they do not pertain to the discussion at hand and in the interest of readability, the WRIGHT choice operators, success processes and overbars representing initiation have been omitted from the description shown in Figure 3.

Note that in the Configuration description presented in Figure 3 there is no reference to the *authentication*, *read/write*, *buffer size*, *read speed* or *encryption* requirements originally mentioned in the systems specification. These requirements, which imply certain non-functional properties, when using the existing WRIGHT notation, would be lost and may not surface again until later during system testing.

```

Configuration e-health
Component GP [description of the component GP]
  Port authenticate = authenticate?x → authenticate
  Port write = start → accept!x → write
  Computation authenticate.authenticate?x → write.start →
    computation
Component SP [description of the component SP]

```

```

Port patient_test_data = data?x → patient_test_data
Port compute_diagnosis = start → data!x → compute_diagnosis
Computation = patient_test_data.data?x → compute_diagnosis.start
  → computation
Component PH [description of the component PH]
  Port diagnosis = diagnosis?x → diagnosis
  Port prescription = start → diagnosis!x → prescription
  Computation = diagnosis.diagnosis?x → prescription.start
    → computation

Connector Shared_Variable [description of the connector]
  Role GetTestData = accept?x → GetTestData
  Role GiveTestData = data!x → GiveTestData
  Glue = GetTestData.accept?x → GiveTestData.data!x → Glue
Connector Pipe [description of the connector pipe]
  Role GetDiagnosis = data?x → GetDiagnosis
  Role GiveDiagnosis = diagnosis!x → GiveDiagnosis
  Glue = GetDiagnosis.data?x → GiveDiagnosis.diagnosis!x → Glue
Connector Remote_Procedure_Call [description of the connector]
  Role GetDiagnosis = diagnosis?x → GetDiagnosis
  Role GivePrescription = prescription!x → GivePrescription
  Glue = GetDiagnosis.diagnosis?x → GivePrescription.prescription!x
    → Glue

Instances [Instances of components and connectors creation]
GP:GP
PH:PH
SP:SP
Shared_Variable:Shared_Variable
Pipe: Pipe
Remote_Procedure_Call: Remote_Procedure_Call
Attachments [system compositions]
GP.write as Shared_Variable.GetTestData
SP.Patient_Test_Data as Shared_Variable.GiveTestData
SP.Compute_Diagnosis as Pipe.GetDiagnosis
PH.diagnosis as Pipe.GiveDiagnosis
GP.authenticate as Remote_Procedure_call.GivePrescription
PH.prescription as Remote_Procedure_call.GetPrescription
End e-health

```

Figure 3: A system expressed in WRIGHT

Notice that if a process receives data, it is an input, and specified with a question mark in WRIGHT such as *authenticate ? x*. If the output produced by a process is specified with an exclamation point such as *data ! x*.

Now, using our proposed constructs, the requirements described in the example, which were not visible in the WRIGHT description shown in Figure 3, can now be incorporated into a new Configuration description. The WRIGHT description with our proposed construct is shown in Figure 4. Again in the interest of readability, the choice operators, success processes and overbars representing initiation have been omitted from Figure 4. The system has altogether six non-functional properties required and/or ensured by the participating components

```

Configuration e-health
Component GP
  Port auth = authenticate?x → auth
  Port write = start → accept!x → write
  Computation auth.authenticate?x → write.start → computation
(1) Non-functional Security =
  (REQUIRES  $\supseteq$  {Remote_Procedure_Call.encryption},

```

```

    ENSURES  $\supseteq$  {auth.authentication}
Component SP
  Port patient_test_data = data?x  $\rightarrow$  patient_test_data
  Port compute_diagnosis = start  $\rightarrow$  data!x  $\rightarrow$  compute_diagnosis
  Computation = patient_test_data.data?x  $\rightarrow$ 
    compute_diagnosis.start  $\rightarrow$  computation
(2) Non-functional Performance =
  (REQUIRES  $\supseteq$  {Pipe.BufferSize  $\geq$  50KB;
    Pipe.ReadSpeed  $\geq$  1MB/sec},
  ENSURES  $\supseteq$  {})
Component PH
  Port diagnosis = diagnosis?x  $\rightarrow$  diagnosis
  Port prescription = start  $\rightarrow$  prescription!x  $\rightarrow$  prescription
  Computation = diagnosis.diagnosis?x  $\rightarrow$  prescription.start
     $\rightarrow$  computation
(3) Non-functional Security = (REQUIRES  $\supseteq$  {},
  ENSURES  $\supseteq$  {Prescription.encryption})
Connector Shared_Variable
  Role GetTestData = accept?x  $\rightarrow$  GetTestData
  Role GiveTestData = data!x  $\rightarrow$  GiveTestData
  Glue = GetTestData.accept?x  $\rightarrow$  GiveTestData.data!x  $\rightarrow$  Glue
(4) Non-functional Security = (REQUIRES  $\supseteq$  {GP.authentication},
  ENSURES  $\supseteq$  {})
  Availability = (REQUIRES  $\supseteq$  {},
  ENSURES  $\supseteq$  {GetTestData.WhenRead  $\neq$  GiveTestData.Write;
    GiveTestData.WhenWrite  $\neq$  GetTestData.Read})
Connector Pipe
  Role GetDiagnosis = data?x  $\rightarrow$  GetDiagnosis
  Role GiveDiagnosis = diagnosis!x  $\rightarrow$  GiveDiagnosis
  Glue = GetDiagnosis.data?x  $\rightarrow$  GiveDiagnosis.diagnosis!x  $\rightarrow$  Glue
(5) Non-functional Performance = (REQUIRES  $\supseteq$  {},
  ENSURES  $\supseteq$  {GiveDiagnosis.BufferSize  $\geq$  50KB;
    GiveDiagnosis.ReadSpeed  $\geq$  1MB/sec})
Connector Remote_Procedure_Call
  Role GetDiagnosis = diagnosis?x  $\rightarrow$  GetDiagnosis
  Role GivePrescription = prescription!x  $\rightarrow$  GivePrescription
  Glue = GetDiagnosis.diagnosis?x  $\rightarrow$  GivePrescription.prescription!x
 $\rightarrow$  Glue
(6) Non-functional Security =
  (REQUIRES  $\supseteq$  {},
  ENSURES  $\supseteq$  {encryption })

Instances
...
Attachments
...
End e-health

```

Figure 4: Example with the proposed construct

and connectors. Each of these non-functional properties is attached with a number to make reference to the specific properties in our discussion. The (1) **Non-functional** is related to security and specified in the component GP. It states that the component requires the connector `Remote_Procedure_Call` to encrypt the object it sends to the component. The component ensures to the connector that it would provide its authenticated data to the connector. In the (2) **Non-functional** description specified in the component SP states that its performance attribute requires connector `Pipe` to have a

buffer size of 50 KB. The component ensures the connector nothing in this case. The (3) **Non-functional** is security and specified in the component PH. The component does not require any property from the participating components, but it ensures that the prescription data will always be encrypted. The non-functional specifications (4), (5), and (6) described in the connectors are self explanatory.

The example clearly illustrates that all of the functional requirements missing from the original specification can be expressed in such a manner that non-functional properties become apparent in the revised, extended architectural description cited in Figure 4.

To ensure the compliance of the required properties of one element with the ensured properties of another, the existing WRIGHT ‘Attachments’ descriptions are used. This compliance checking can be carried out to verify that all functional, and therefore non-functional, requirements have been met by each component and connector.

5. Discussion

The example cited in the previous section draws attention to several potential uses for the proposed construct in WRIGHT as well as their strengths, including their natural usability and readability.

The proposed construct is based on the existing structure and form of WRIGHT. The notational structure of the new construct is similar to the existing structure of various elements of WRIGHT. This, along with its inherent intuitiveness, causes the extensions to be easily deployed in new architectural descriptions or incorporated into existing descriptions.

The fact that the proposed construct allows for the use of structured English in the requirements specification, along with the unambiguous specification of the *Prop* phrase in plain English, promotes readability of the requirements, which in turn promotes the use and adoption of the extensions.

By introducing the new element and header, ‘Non-functional’, the existing behavioural elements, such as ‘Computation’ or ‘Glue’ are unaffected. The new header then is minimally intrusive to the existing notation. A choice can be made to include or not include the non-functional properties without any affect on the existing WRIGHT descriptions.

Even though an entirely new construct has been introduced, existing WRIGHT and CSP notations and concepts are kept intact. This significantly reduces the amount of time required to learn the new construct. Architects can quickly grasp the concepts introduced in the extensions and immediately begin to use them in new architectural descriptions.

We acknowledge that the construct lacks a formal approach that the proposed construct may require. This is a conscious trade-off for readability and usability. We do believe that there is a good balance between formal approach and plain English allowing the architect to accurately, yet flexibly, describe the non-functional requirements in an unambiguous, repeatable, ‘templated’ manner.

The construct does suffer from the fact that if a single requirement applies to more than one non-functional property, the requirement has the potential to be listed several times. Although this is not syntactically incorrect, it is recommended to list the requirement under the non-functional property to which it most pertains.

The two major challenges related to our proposed approach remain unresolved: (1) how to determine and specify the actual value of a non-functional attribute; (2) how to formalize the proposed construct so that it could be machine readable and reasoned about.

Referring back to the example, the construct was shown to be able to handle an array of non-functional requirements, albeit a small array based on the extent of the example. Further study into the wider application of the proposed construct in larger case studies and with different ADLs is therefore necessary to prove the broader application of the extensions presented in this paper.

6. Conclusion

The paper has presented a construct to and existing ADL notation which specifically allows for the inclusion of non-functional properties in an architectural description. The construct is built around the concepts of ‘requires’ and ‘ensures’ and is modeled after existing WRIGHT ADL and CSP notation, although it could be applied to most other ADLs. As demands for better quality software steadily increase, software architects must meet these demands by including quality as a design component in the architecture. In order to include quality in the architecture, the design tool that the architects use must support the inclusion of non-functional properties. The proposed construct in this paper is the tool that allows architects to build quality into architectures from the onset of the process. As the area of software architecture grows and matures so must the supporting ADLs grow and mature as well.

References

- [1] Medvidovic, N. and Taylor, R.N., ‘*A Classification and Comparison Framework for Software Architecture Description Languages*’, IEEE Transaction on Software Engineering, vol. 26, no. 1, January 2000.
- [2] Allen, R.: *A Formal Approach to Software Architecture*, Ph. D. thesis, Carnegie Mellon University, 1997.
- [3] Khan, K. and Han, J., ‘*Composing Security Aware Software*’, IEEE Software, January/February 2002, pp. 34-41.
- [4] Franch, X. and Botella, P., ‘*Putting non-functional requirements into software architecture*’, Proceedings of the 9th International Workshop on Software Specification and Design, 16-18 April 1998, pp. 60 – 67.
- [5] ISO/IEC FCD 9126-1.2, ‘*Information Technology—Software Product Quality, Part 1: Quality Model*’, 1998, cited in Losavio, F., Chirinos, L., Matteo, A., Levy, N. & Ramande-Cherif, A., ‘*ISO Quality Standards for Measuring Architectures*’, The Journal of Systems and Software 72, 2004.
- [6] Hoare, C.: *Communicating Sequential Processes*, Prentice Hall International, 2003
- [7] Rosa, et al., “Process NFL: A language for describing non-functional properties”, Proceedings of the 35th Annual Hawaii International Conference (HICSS), pp. 3676-3685.
- [8] Abowd, G., Allen, R., Garlan, D., “Formalizing Style to Understand Descriptions of Software Architecture”, ACM Trans. on Software Engineering and Methodology, 4(4), 1995, pp. 319-365
- [9] Viega, J., McGraw, G.: *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, Reading, Mass., 2001.
- [10] Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*. (Addison-Wesley, 1998).
- [11] Voas, J., “Software’s Secret Sauce: The -ilities” , IEEE Software, Nov/Dec. 2004, pp. 14-15.
- [12] Bollinger, T., Voas, J., Boasson, M., “Persistent Software Attributes”, IEEE Software, November/December 2004, pp. 16- 18.
- [13] Rashid, A., Moreira, A., Tekinerdogan, B., “Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design”, Editorial, IEE Proceedings Software, Vol. 151 (4), 2004, pp. 153-154.
- [14] Whittle, J., Araujo, J., “Scenario Modelling with Aspects”, IEE Proceedings Software, Vol. 151 (4), 2004, pp. 157-171.
- [15] France, R., Ray, I, Georg, G., Ghosh, S., “Aspect-oriented Approach to early Design Modelling”, IEE Proceedings Software, Vol. 151 (4), 2004, pp. 153-185.
- [16] Staehil, R., Eliassen, F., “QuA: A QoS-aware component architecture”, Research report, No. 2002-12, Simula Research lab, Norway, 2002.
- [17] Aagedal, J.: *Quality of Service Support in Development of Distributed Systems*, Thesis, doctor Scientiarum, Department of Informatics, University of Oslo, 2001.