Applying Lessons Learnt from Software Reuse to Other Domains

T P Kelly, B R Whittle

Rolls-Royce University Technology Centre
Systems and Software Engineering
Department of Computer Science
University of York
York, UK, YO1 5DD
Tel: +44 1904 432773
Fax: +44 1940 432708
Email: tpk/ben@minster.york.ac.uk

28th April 1995

Abstract

The position this paper promotes relates to the applicability of software reuse techniques to other non-software domains: What lessons learnt from techniques developed primarily for software reuse, can be applied to reuse of artifacts in other disciplines? The prime motivation for this research comes from a project attempting to apply reuse and evolution to the arguments of a safety justification. A discussion of the issues from this domain, as well as from more traditional domains within software development are used as illustration.

Our exploration of the issues relating to safety arguments has led us to recognise anew that the essential problem of reuse is one of matching contexts. Whatever the domain we must be able to recognise and abstract the essential technical and non-technical characteristics of the domain. The success of reuse depends upon our ability to support, communicate and exploit the context in which artifacts are built, used and hopefully reused. The paper uses the examples of domain analysis and component description to show how this problem has been tackled with software artifacts and sets forward the motivation for domain modelling that provides a unifying basis for component identification, storage and retrieval for any domain.

Keywords: Software Reuse, Non-Software Domain, Context, Domain Analysis, Component Description, Reuse Model, Domain Model

Workshop Goals: Discuss domain modelling, assess suitability of software reuse principles to non-software domains

Working Groups: Domain Analysis / Engineering, Reuse process models, Reuse handbook
1 Background

Both authors work within the Rolls-Royce University Technology Centre for Systems and Software Engineering: Rolls-Royce established this research group within the University of York, England to examine systems and software development issues for the high integrity control systems, both in the aerospace and power generation industries, produced by Rolls-Royce plc.

One of the major activities of the group has been in establishing a successful reuse strategy within the company. This work has been principally undertaken by Ben Whittle and has examined artifact reuse across all phases of system development, from requirements specifications to test plans. The work of Tim Kelly has been in extending the concept of reuse to another costly aspect of high integrity system production, namely safety case development. It is in this work that Tim has been examining the applicability of ideas and techniques from software reuse to wider domains.

2 Position

The software industry has proclaimed the benefits of reuse for some time, for example: shorter development time, increases in reliability, economies of scale. This has led to an increasing number of domains being examined for their reuse potential. Candidate domains for this scrutiny are typified by the complexity of the artifacts produced being sufficiently great to outweigh the pickup effort associated with a 'reusable' artifact [?] and where some generality of type or application of the artifacts is suggested. Artifacts in these domains may already be reused on an ad-hoc basis. However, it is systematic reuse, as exemplified in the field of software reuse, that is being seen as having potential long-term benefit. The majority of existing reuse research has concentrated on software and software-related artifacts. Unlike physical artifacts, software artifacts are intangible and their concept, content and context highly variable. Because of this, software reuse research has been forced to deal in generalities and to address issues concerning reusable component identification, support and management. Although no panacea exists, the reuse problem has similarities across all domains and fundamental lessons can be learnt from software reuse research. The position this paper explores relates to the applicability of software reuse techniques to other, non-software, domains: What lessons learnt from techniques developed primarily for software reuse, to reuse of artifacts in other disciplines?

Software is often generalised as a single domain. Of course this is not strictly true, it should be considered as a large number of domains. Authors active in software research come from a large number of organisations in different parts of the world. The strategies and models they promote have been developed for their particular domain. Therefore, these models are suitable for their domain but probably not wholly appropriate for other domains. The exploration of reuse within non-software domains shows the need for an approach that recognises, supports and exploits the characteristics of the domain whether for software or not.

2.1 Issues in Promoting Systematic Reuse

Regardless of domain, there are three issues associated with reuse:

- Development for Reuse
How do we identify and develop potentially reusable assets within the domain?

- **Development with Reuse**
  How do we abstract these assets so that they can be stored, retrieved and reapplied in new situations?

- **Managing reuse**
  How do we measure the costs and benefits, and reason about the tradeoffs involved in reusing assets?

These three activities depend on sound modeling of the domain in which artifacts are initially produced and applied. We focus primarily on the first two activities given here. Management of reuse, being a vast subject in its own right, cannot be adequately covered in a paper of this length.

### 2.1.1 Developing for Reuse

When placed in a new environment and told to instigate reuse, what do you do? Software reuse theory and practical experience tells us that the first thing should be to learn something about that environment or 'domain'. In the software domain our first step is to perform a domain analysis [7] - "The activity of identifying the objects and operations of a class of similar systems in a particular problem domain". The same is true for almost any domain. The resulting output from this activity should be an appreciation of the context in which domain products are developed and used, including:

- **Product aspects**
  - Taxonomies and architecture of the products
  - Functional models of the products
  - Domain languages
  - Properties of the component produced

- **Process Aspects**
  - Life-cycle, methods used in the development of the products
  - Standards used in the development of the products
  - Resource used in the development of the products

With domain information such as this we can start to determine those artifacts with reuse potential, e.g. by spotting common functions or a regular structure in the architecture of the product. It is only by assessment of the generality of application or content of an object, based upon their defining characteristics determined by the domain analysis, that we can estimate future reuse potential.

Emphasis in domain analysis applied to software products has been primarily on appreciating and modeling the technical architecture and characteristics of the domain. To a certain extent, this has not mattered, the reusability of basic block software components can be crudely judged on
technical content. However, inspection of the component content cannot reveal attributes such as the development integrity level. With other domains such as safety arguments we require a more complete understanding of the models, experience, assumptions and evidence on which the argument was based in order to appreciate the extent to which it could be usefully reapplied in a new context. In these cases, the context defines the product to a greater extent and without it the products purpose, generality and applicability is eroded. For example, without knowing the assumptions on which a safety argument depends it is impossible to determine the strength of the argument and whether the argument can be appropriately reused in a new context.

It has been accepted that domain analysis is crucial to making the first steps in software reuse. Further, we argue that domain analysis is essential in any domain, provided it considers and identifies the most pertinent of all, technical and non-technical, influences over the potentially reusable assets in question. The domain analysis phases is crucial to 'pinning-down' the context, and therefore generality, of the initial development and application of components.

2.1.2 Developing with Reuse

Storage and retrieval of components is only an issue to the extent that it influences successful application of 'reusable' component. When we wish to reapply a component developed in one context in a new context it is crucial that we fully understand the component domain and the component's relationship to that domain, in order that the component is appropriately used. For example, with a safety-critical software component we need to determine the extent to which it has been developed and tested in order to justify its re-application it in a new safety-critical context. It is therefore essential that, whatever storage and retrieval mechanism is used, we capture or communicate a component's relationship to its development context by abstraction and relationship to a domain model that captures all aspects of that context.

The central concern in storage and retrieval of components is that it is fast and intuitive to the reuse engineer. In the field of software reuse, faceted classification schemes [?] have been suggested as an approach. Using this method, classes of components are described by defining faceted lists of keywords. Components are stored with, or according to, their faceted classification by their particular grouping of keywords selected from the 'facets'. The principal benefit of this method is the extent which it synthesises a context in which the reusable components can be placed; by defining facets for classification we are modeling a context not too dissimilar to either the original or intended context. If the principle of faceted classification schemes were extended to wider domains, for storage and retrieval of components we would need to re-create a context in which the component can be 'understood'. This context should be derived from the key, technical and non-technical, characteristics defined in the model produced from the initial domain analysis. Therefore, faceted scheme applied to new domains should also capture these attributes, as exemplified by the REBOOT [?] approach.

2.2 Examples

We have argued that all three of the fundamental concerns of reuse, namely: starting; supporting and managing, should, for any domain, be based upon a sound model, defining the key characteristics of the domain. In the field of software reuse, their has been much effort in defining the 'best' domain analysis technique or the 'best' faceted classification scheme. Generalising, software reuse principles to other domains, we can see that for any domain starting, supporting and manag-
ing reuse should be rooted from a model derived from analysis of the domain which abstracts the original technical and non-technical context of the components in question.

In software reuse, we often assume a common understanding of a component's context or rely heavily upon the content of the component to communicate that context. For example, we have a common understanding of a particular programming language or development method and believe that, given time, we can work out the original purpose of a component by examining the code contained within. As the degrees of freedom in a domain increase and we can rely upon less of a common understanding, modeling the domain context becomes increasingly important. To illustrate this point, let us imagine 3 scenarios:

- **Mathematical routine**
  With a description that says it adds two number of type integer and returns an integer result.

- **Signal validation component**
  for a real-time systems that takes in a signal from a heat probe, validates and range checks the signal. If the signal is outside the acceptable range a model value is used. The software module that generates the model value is a separate subroutine, adhering to a coding standard, that provides a value when invoked, the value the model provides must also be subject to the range check.

- **Safety case fragment** which argues the reliability of a component from evidence given in a number of safety analyses, testing and operational evidence from similar components within the framework provided by company, government and industry codes and standards.

In each case, when the component is reused it will form part of a system which is to be used in a safety critical application. We need to examine how the component meets the requirements placed upon it within the domain and the properties of those components that are pertinent to the domain. The software components will be used as part of the systems itself, the safety case fragment a part of the justification that the system is fit for its purpose, and has been developed to acceptable standards.

There are a couple of questions we need to ask:

- What information do we need to know about the component?
- What would existing models like the 3C model [?] or the REBOOT model tell us about the component?
- Is there any way of knowing, from a domain model or domain modeling, the information that we will need to provide with a component in order to reuse it?

If we are to reuse the mathematical routine in a safety critical application it would typically be the case that the component would have to be developed to an equivalent standard to the other components that we were using. Thus we need to know something about the development context of the component in order to infer the quality of the component. In searching for a component we may wish to prescribe standards for the components we search for; they must conform to a given programming standard. We require that reused components must have been developed to that standard.
In order to reuse the signal validation component we must know about the associated hardware, the heat probe. We can use the knowledge about the heat probe to work out what kind of model we need if we are going to use a model value of the temperature instead. These can be regarded as the extended technical context of the component.

If we are going to reuse the safety case fragment we need to consider on what bases the justification is founded; the model, strategy, justifications and assumptions used need to be made explicit, and therefore, verifiable. For the new context we need to determine whether the model of the component, the strategy used, the justification for that strategy and the assumptions made are still valid. For example, it is common to justify a components claimed safety by reference to a good operational record. If however, between the time of creating the safety case fragment and wishing to reuse it, this operational record has been damned by numerous accidents the original justification is invalidated. Safety case fragments depend heavily upon their original application context. For there to be any case for successful reuse of safety case fragments, this context must be made explicit by means of an accurate domain model.

The REBOOT component model, faceted classes and ‘reusability’ measures have been developed specifically for software components. The REBOOT abstractions would therefore be suitable for reasoning about the software components in our example, though possibly considered overkill for the simple component, but would fail to capture the contextual properties of the safety case fragment. The suitability of the 3C model depends upon the extent to which we further subdivide the broad categories of context, concept and content. The 3C model provides a suitable starting point for developing domain-specific models that could be realistically used for identification, storage and retrieval purposes. The strength of the REBOOT model is that it has defined a ‘generic’ model for software component which captures most of the non-domain specific properties of software components. The strength of the 3C model is that is suitably generic to be used as a basis for development of any domain model. However, it’s failure is that without further definition we are left with vague notions of the domain with inscrutable implicit attributes.

This brief example has illustrated that different properties are required to reason about components from different domains. While we could attempt to reapply models unchanged from the software reuse world or develop entirely new models, these would not be suitable for all domains.

3 Summary of Position

The problem of reuse is one of matching contexts. Components are developed in one context and, hopefully, reused in another. The decisions we take in reusing a component depend on our appreciation of both of these contexts. We usually have a fair understanding of the context in which we work. However, we have difficulty in appreciating a context separated from our own. If the context of the component isn’t recorded then we are required to assume a context, as is commonly the case with vanilla software reuse. We have highlighted that for some domains, such as safety justification, the assumption is potentially more damaging, and the information communicated by the component content becomes of less use.

Mathematical routines have been successfully reused for some time. Their strength lies principally in well defined interfaces supported by the constructs of the implementation language. The functional dependencies and context of the component can be successfully bundled up as a neat set of parameters and return variables. The difficulty in reusing components such as the safety case fragment described lies in the many dependencies and contextuality that the component possesses.
We have no well understood boundary or interface to the component. The underlying content and structure of the component lends no support to defining this context. We therefore have to find a way of defining the boundary of the component and specifying those 'parameters' of dependency within the component's context. Between these two extremes lie a great many domains, each with varying levels of contextuality and support from their underlying content or technology. What is common to these domains, is that the context of the component has to be considered. The type and degree of context that is recorded should depend upon the domain in which the component is developed, applied and reapplied.

We have used the techniques of domain analysis and faceted classification as an example of how software reuse has tackled a problem that has to be faced in any domain, that of comprehending and abstracting contexts. To capture the essence of these techniques developed for software reuse and apply them to other non-software domains we need to recognise and record the process and product characteristics most pertinent to the initial development and application context. The information that we require for successful component identification, storage, retrieval and management all derives from our comprehension of this context. By developing a single model of this context, on a per-domain basis we will maximise understanding of where components can be reused and be able to reason about how reuse can 'carry value' from one context to the other.

4 Biography

Ben Whittle graduated in Agricultural Economics from UW Aberystwyth in 1989. He subsequently completed a masters in Computer Science. Foremost among Mr Whittle's research interests are component reuse and reuse education. Mr. Whittle is currently with the University of York, in the working within the Rolls-Royce sponsored Systems and Software Engineering University Technology Centre (UTC). His main task within the UTC is the introduction of advanced reuse techniques to the development of real-time systems within Rolls-Royce. Mr Whittle has recently been elected chairman of the British Computer Society Reuse Special Interest Group committee and was formerly the editor of the group newsletter.

Tim Kelly graduated in Computer Science from the University of Cambridge, England in 1994. He is a research student working within the Rolls-Royce Systems and Software Engineering University Technology Centre (UTC). He is investigating the development and assessment of safety cases. In particular, he is looking at the reuse and maintenance of safety arguments as a means of supporting evolvable safety cases. Before joining the UTC, he was involved for a number of years with high-integrity systems and software research within the Rolls-Royce group, working particularly in the area of Integrated Project Software Environments.