A Knowledge Level Analysis Approach for Assessing Reuse

Abstract. The process of assessing the suitability of reuse of a software component is complex. Indeed, software systems are typically developed as an assembly of existing components. The complexity of the assessment process is due to lack of clarity on how to compare the cost of adaptation of an existing component versus the cost of developing it from scratch. Indeed, often pursuit of reuse can lead to excessive rework and adaptation, or developing suites of components that often get neglected. This paper is an important step towards modelling the complex reuse assessment process. In order to assess the success factors that can underpin reuse, we analyze the cognitive factors that belie developers behavior during their decision making when attempting to reuse. This analysis is the first building block of a broader aim to synthesize a framework to institute activities during the software development lifecycle to support reuse.

Keywords
Reuse, Reuse Process Modelling, Software Development Ontology, Components

1. INTRODUCTION

It is well accepted that successful software components reuse can reduce the development cost and time of the software. Successful reuse in software development can lead to shorter coding time and more reliable code. These expected benefits were strong motives in the reuse research that has been ongoing since the late 1960s. It continued with the take up of OO technology in 80’s and 90’s (Henderson-Sellers 1996), the interest in Service Oriented Architecture (SOA) (Shashwar 2017), and current interest in Open Source platforms such as GitHub (Oliveira et al. 2017). Reuse often seems an appealing economical path to software development, but it is fraught with disappointing pitfalls that hinder economic reuse from
taking place. For example, a web developer may attempt to adapt a component which seems initially suitable, only to find after substantial time adapting it, that starting from scratch was easier and quicker. Whilst much work has been accomplished, there is still much to do before the software development via reuse is completely achieved (Frakes and Kang 2005; Beydoun et al 2005b).

Reuse may cause unforeseen maintenance problems, for example, a database developer may be tempted to reuse a data retrieval component which later turns out to be very memory hungry and constantly causes his database to crash. Little effort has gone into understanding the actual mental processes involved in reuse and the factors that render it a success or a failure. Towards this, in this paper, we propose an abstract model of the process of software component reuse and seek to identify the reuse factors that underlie the decisions in this process. We apply a knowledge level analysis to the process of reuse. This analysis yields an abstract description of the reuse process. This description also identifies features of reusable components and the factors at play in the process. Towards operationalisation of the result of this analysis, we sketch an ontology-based approach to index and represent the knowledge required. In particular, to enable the use of these features and development of an appropriate repository, details are elaborated on using an ontology that can be used to index reusable components.

The rest of this paper is organised as follows: Section 2 presents the overarching perspective that applies to the reuse assessment framework. Section 3 presents an example illustrating the processes identified in Section 2 to enable further elaboration of the reuse factors. Section 4 presents the reuse factors highlighting how they can be described at the knowledge level. Section 5 presents how the overall approach can be operationalised with appropriate ontologies to interface to reusable components. Section 6 concludes and summarises the contributions of the paper.
2. MODELLING ASSESSMENT OF REUSE PROCESS

Majority of software development projects rely heavily on reuse. The systems themselves are often developed from an assembly of existing reusable components (Beydoun et al 2005a). Whether reuse is one of reusing components by developers or project management strategies by a senior manager, the reuse process as we will see involves iterative decomposition and adaptation of knowledge artefacts. In order to assess the reuse of components effectively, it is necessary to measure the reusability of these components (Sharma 2009). Although, the length and the complexity of reuse process steps may depend on the domains and the reusable artefact. For example, a project manager may spend longer time assessing the reuse potential of a development methodology than what a software developer would spend in evaluating a software component.

In formulating our framework, we first separate the knowledge involved from the processes involved in reuse. We model the actual process of reuse as an iterative process cycle interleaving analysis/retrieval/reuse/decomposition (see Figure 1). Some examples of knowledge involved in this process are component retrieval knowledge, component adaptation knowledge, component reuse knowledge and component construction knowledge. The availability of these and other kinds of knowledge impact the transition between the various activities involving reuse. We identify the various types of knowledge involved in reuse and present guidelines to assess the viability of reuse using knowledge complexity measures applied to the various kinds of knowledge involved. For example, reuse involves locating a library component and then adapting it. If either the knowledge and/or effort involved in locating and/or adapting the component is too much to afford, then reuse may not be viable. As we develop this framework at the knowledge level, we model the actual reuse process which seems to us to be a domain independent process. This is in contrast with the
domain dependence of the complexity of the actual components to be reused, as results in (Frakes et al 2005; Hendershon-Sellers 1996) suggest.

![Diagram of the iterative reuse process]

**Figure 1**: The iterative reuse process- Initial problem statement is examined. Ideally, a reusable solution is adapted and problem solved. More likely, the problem is decomposed into smaller problems and a partial solution via reuse is possible. Reuse may fail in two circumstances: if adaptation fails or if merging with the rest of the emerging solution fails.

Our separation between reuse process and knowledge being reused is also useful in identifying success factors in reuse. The process factors can also be construed as developers related factors whilst the knowledge being reused in practice reflect the nature of the components reused and how they are organised (i.e. the reuse libraries). We delve into both sets of factors and delineate negative factors that hinder a successful and economical reuse from taking place, and positive factors facilitating a successful and economical reuse. We also identify various relations between these factors.
Factors that emanate from the nature of knowledge reused include:

- Factors related to the software components (required) e.g. how complex the component is and has this component been already reused?

- Factors that are related to the domain of the component. E.g. How application area specific is the component?

- Factors related to the libraries used e.g. how well documented they are. E.g. If the access to the components is obscured due to bad documentation then reuse may suffer.

Factors that emanate from the reuse process:

- Factors that are related to the software development lifecycle (SDLC) e.g. is reuse encouraged or frowned upon. For example, a scrum master of a safety critical application development team may dictate the level of reuse depending on certain security requirements.

- Factors that are related to the developers involved e.g. do they wear programming (rather than reuse) as a badge of honour? In an open source environment, some developers may wish to leave their mark by avoiding reuse.

All those factors will be further analysed and operationalised in our model. Not all lend themselves to a knowledge level analysis with the same ease. E.g. Component complexity can be seen from a knowledge complexity perspective only if we see a piece of software as an embodiment of some knowledge and in itself as a knowledge artefact.

3. **A case study: A web based application to sell health insurance policies**

In this section, we present a reuse episode in a software development example to highlight the features of our framework and the reuse process modelled highlighted in the previous section. We first describe the requirements of the software pursued.
A developer is tasked with providing a web-based interface to sell health insurance policies. The selling process involves the web customer entering the various options associated with a policy as well as entering their personal details and payment options. The developer is reasonably familiar with an e-commerce shopping trolley web component which instantly comes to their mind as they think about the requirements of the interface. The shopping trolley component is typically used by websites like Amazon.com where more than one product is bought in one e-shopping session. In what follows, we illustrate the pitfalls that reusing the shopping trolley may present when developing a web interface to sell insurance policy with various option. We also analyse the knowledge involved/required/missing in the various stages of this reuse episode.

The developer decomposes the functionalities of the shopping trolley into two: collect products and collect payments (register functionality). He maps these two to collecting insurance options and collect insurance payments and surmises that since the shopping trolley is an available component which seems comprehensible, it seems worth reusing. In other words, the process of assessing to reuse or not is hierarchical and involves decomposing the functionalities of the artefact to be reused. In support of the process described in Figure 1 there are sub-processes assessing the artefact against the decomposition of the problem (Figure 2). We overview the adaptation of each of these two functionalities to the requirements of the web based insurance policies which leads the developer reassessing the suitability of the shopping trolley.

A health insurance policy is unlike a book or a flower bouquet, it is pointless for a person to have more than one of them. Government regulations may also well imply that an insurance company can only sell one to any one person. This creates additional requirements for the interface: (1) authenticating the identity of the customer to ensure that they do not already have an existing policy, (2) if they have one and they are pursuing additional options only then
in this case the interface requires identity authentication too (3) a connection to a database of existing customers is required.

An insurance policy is often a hierarchical product. A kind of insurance is chosen for example, a hospital cover or a dental cover. When the broad category is chosen various options are made available. In other words, the ideal representation is a hierarchy of windows presenting the various options at various stages during the sale. This is not possible with a shopping trolley. A trolley simply collects the products in one set, so strictly speaking the initial view of the developer that an option equates to a product is a compromise which will restrict the functionalities provided by their interface.

From this simple episode of reuse attempt, these kinds of knowledge were evident and have impact on the final outcome of the development:

- Knowledge of the shopping trolley may have been short in not seeing immediately that it does not provide hierarchical representation or authentication of buyers.
• Knowledge of alternative choices may have motivated the initial desire to use the shopping trolley.

• Initial knowledge of the domain may have thwarted the desire to reuse the shopping trolley.

• Technical knowledge interfacing/development may ultimately decide which way the developer goes with the shopping trolley.

And finally, perhaps the knowledge embedded in the shopping trolley itself and how complicated it is to reuse will have an impact on the final outcome.

4. Characterising Factors Surrounding Reuse

We know that certain components, such as databases, are regularly and frequently reused. We know also that not all attempts to reuse turn out to be successful. So, the right balance need to be found. Too many components in the library make the task of selecting a suitable component difficult. Too few components render the library also relatively useless. However, it appears that not only that the general fact that a component could be reused makes it fit to be placed in the component library but also the degree to which it is easy to determine whether a given component can be effectively reused for a given project. In other words, a key feature of a component to be reused is whether it can be readily identified as suitable for a project or otherwise dismissed without any undue effort. This entails on the one hand that the capabilities and limitations of a library component should be easy and quick to grasp for a system designer/developer. On the other hand, even if the functionality is easy to grasp, there should only be a rather minimalistic set of components that need to be considered for a given project. For example, numerous slight variations of a given base function are probably not worth having in a library as the variations might be considered many times and end up being dismissed anyway. So, they would consume considerable time of the system
designers/developers without contributing accordingly. In the odd situation where a given function requires some slight modification, it might be much easier to then use the base version of the function and develop some wrapper function around it from scratch in order to meet the requirements.

4.1. Developer related factors

The developer’s context and requirements are clearly a primary determinant. This leads to their search of a reusable component in the first place. There is evidence that some application domain knowledge on the developer’s side may be helpful (Miller et al 2014; Beydoun et al 2011) to the ultimate outcome to reuse or not to reuse. Indeed, this is not too surprising as this may lower the cognitive costs involved in reuse. The application domain knowledge determines the nature of their initial plan idea, initial code ideas and so forth. In other words, identifying the most suitable component to be reused seems to be a knowledge driven task. This knowledge comes from experience and components require proper description to be identified properly (Lorca et al 2016). On the other hand, in some cases programmers eagerness to prove themselves can be an obstacle to reuse. What seems clear though is that there are issues specific to code reuse (e.g. code easy to reproduce and replicate) and many general reuse issues applicable to code reuse as well (e.g. requiring experience).

4.2. SDLC related factors

Reuse can also be seen from the perspective of the development project managers. The project manager’s knowledge of the team, the repository availability and any later maintenance are all determinants to the uptake or success of reuse. Some of these reuse issues are common across many domains (not only in software engineering) and in relations to many artefacts, e.g. car designs, building components. For instance, some process actions are impacted across team processes. Reuse may require additional communication within the team to ensure the
interfaces between components are feasible or economical to develop. Reuse may impact software developers who may be tempted (or avoiding) to reuse components are (or not) familiar with. But specific to software perhaps, reuse can also influence early phases of the development e.g. the requirements gathering (for example, a client may be tempted to request extensions to their system).

4.3. **Component related factors**

The success of reuse is also based on the features of the components available themselves. Components that are easy to locate or to adjust to fit the current context are more likely to be reused. The component reuse cost depends on the complexity of entity reused. The more complex the entity is to re-write from scratch, the more likely it is that it will be successfully reused; the less complex it is the less justifiable the above costs are. Furthermore, the more frequently needed a component is, the more developing from scratch is justified as the overhead cost per component is reduced.

4.4. **Library related factors**

Successfully locating a component which may be reusable also depends on the repository itself. This can be quite hard in some instances e.g. trying to locate a reusable function in a large file of code. Hence, availability and interfaces of the repository of components are key factors. The maintenance of the repository is also a longer term factor. This depends on the cohesion of the repository, the representation of components and as well as the knowledge about components. More storage can render people untidy and lead to increase in the complexity of libraries and their reuse. However, broadening access can also enhance reuse about will require additional protocols for keeping libraries tidy and user friendly. More web-based interfaces such as GitHub aim towards this (Brown et al 2016).
One has to wonder what kind of component library would really be useful. To characterise when reuse is economical, we first consider the theoretical upper bound on the cost of reuse. Suppose that there is a library containing all possible functions of some limited complexity. The size of this library requires that a complex search is undertaken to find a function, where the search complexity outweighs the complexity of the function to be retrieved. In this case, reuse can never be justified. However, in reality reuse does occur, and libraries of code are used. This indicates that the complexity of reused components does indeed exceed the complexity of finding the component within the library. Considering a theoretical case where a library for reuse is available that contains all possible source code programs up to a certain length. Then selecting the correct function from that library would pose a serious problem. One simple access key would actually be to specify the code itself. This would obviously not help at all. The other end of the spectrum though, would contain no function in the library at all, which would also be useless.

5. Operationalisation of Reuse Framework with Ontologies

As discussed above, one of the most important issues in software reuse is storage and retrieval from the libraries of components that will be reused. In fact, searching for information is one of the most dominant daily activity of the participating developers and the retrieval of reusable assets requires an appropriate description (Happel et al 2008; Oliveira et al 2017). The libraries used typically have some descriptions of their functionality in natural language and do not incorporate formalized knowledge about application domains or the functionality of the artefacts (Happel et al 2006; Beydoun et al 1998). Several traditional keyword-based search methods have been proposed in last years but they have not been very successful and found their shortfalls in finding software assets for reuse (Hadji and Choi 2009). The use of semantic web technologies provides better representation mechanisms for components that can be described by means of meta-data. Some different approaches of providing meta-data to
software artefacts have been proposed. For example, in the work presented in (Li and Strasunskas 2005) a General Process Ontology to annotate common and general software process is proposed. Business process modelling has also been represented by means of ontologies and ontology based tools to support the graphical modelling of business processes with information derived from domain ontologies (Born et al 2007). In (Happel et al 2006) ontologies are used to describe software artefacts independently from a particular format.

Assessing quickly to what degree a given reusable component will match the requirements of a project is critically important for enabling effective reuse on a larger scale. While proper documentation of the capabilities and limitations of components would be expected, current practice is certainly falling behind those expectations. More importantly, though, it is generally difficult to describe capabilities and limitations if those future potential uses have not been considered so far. E.g. in the example above the designers of the shopping trolley modules would not have considered the specific needs for selling health insurance policies online. We propose to represent the components by means of ontologies. An ontology can be defined as “a formal and explicit specification of a shared conceptualization” (Studer et al 1998; Xu et al 2011). Ontologies provide a formal, structured knowledge representation, with the advantage of being reusable and shareable. Ontologies provide a common vocabulary for a domain and define -with different levels of formality- the meaning of the terms and the relations between them. Knowledge in ontologies is mainly formalized using five kinds of components: classes, relations, attributes, axioms and instances. Classes in the ontology are usually organized into taxonomies. Sometimes, the definition of ontologies has been diluted, in the sense that taxonomies are considered to be full ontologies (Studer et al 1998).

The ontology that describes the components have to take into account different types of knowledge (Figure 3) such as knowledge about the complexity, types and reuse of software components, knowledge about the functionality of the component and knowledge related to
the domain of the component. The software artefact description ontology models the different types of software artefacts (Specification, Diagrams, Databases schemas, software components, libraries, web services, etc.) and the complexity of each component and some reuse measures.

I.e. rather than hoping that documentation of developed software will eventually improve and allow more effective reuse, we are advocating the need for interactive tools that assist in quickly assessing the suitability of existing software components for a given project.

The functionality of the software component and the domain where this artefact has been used (i.e. finances, e-commerce, etc.) would be represented with annotations to other ontologies such as a software ontology or a general ontology respectively.

![Software ontology](image)

**Figure 3:** The knowledge represented in the description of a software artefact.

The development of an ontology for a particular domain is a time-consuming task while a shared vocabulary is necessary for obtaining an agreed ontology. The software ontology would represent a vocabulary of Software engineering representing the functionality of the software component. Some ontologies and vocabularies for Software engineering have been developed in the last few years. These ontologies have been applied in different steps of Software Engineering lifecycle and these approaches can be classified in four areas (Happel et al 2006):
• Ontology-driven development subsumes the usage of ontologies at development time that describe the problem domain itself.

• Ontology-enabled development also uses ontologies at development time, but for supporting developers with their tasks.

• Ontology-based architectures use an ontology as a primary artefact at runtime. The ontology makes up a central part of the application logic.

• Ontology-enabled architectures (OEA) finally, leverage ontologies to provide infrastructure support at the run-time of a software system.

For the purposes of this work, the second category of ontologies is required. That is, ontologies that semantically describe the functional properties of software artefacts that can be reused. A representative example within this area is shown in (Lasheras et al 2009), in which an ontology for requirements specification documents has been developed and used for modelling reusable security requirements. In (Hartig et al 2008), the ontology-based DESWAP system is presented. In the context of the DESWAP project, a knowledge base with comprehensive semantic descriptions of software and their functionalities were developed. Thus, by taking into account the shortcomings of developing a new ontology from scratch, the ontologies developed under the scope of the DESWAP project for representing the features and functional properties of the software projects have been adapted and reused. An excerpt of the software ontology representing a classification of software is shown in Figure 4.
6. DISCUSSION AND CONCLUSION

Re-users often cannot articulate a proper description of the component they are seeking. This research also aims at looking how to take an incomplete articulation of what is sought and develop it through a sequence of interactions to zoom in on a reusable component, or in a worst case scenario ascertain that the possibility of effective reuse is not available. This work will help developers to select the best component in terms of its reusability, which will improve the maintainability of the overall system (Sharma 2009).

The proposed framework is an explanatory tool. It can help in explaining failure and success stories or reuse. It identifies critical thresholds of success and failure, e.g. domain knowledge in some cases is essential for effective reuse, a minimal components complexity is required to justify reuse. The model is also a predictive tool that can point to the potential pitfalls and
caveats in pursuing or broadening the scope of reuse e.g. how to best represent libraries, how to best avail interfaces in reuse of the form web services or applying a P2P community based searching for components (Tran et al 2007; Tran et al 2006), how to best interact with the user to ensure that a component that can be reused effectively once it is found.

Knowledge Management in Software Engineering has attracted significant interest in the research community as well as in industrial practice (Aurum et al 2008; Beydoun et al 2014). However, the work we are aware of has essentially focussed on specific knowledge relevant to the organisation and concerning the general software design process as opposed to issues discussed in this paper.

REFERENCES


Beydoun, G., Hoffmann, A., 1998, Simultaneous modelling and knowledge acquisition using NRDR, PRICAI’98: Topics in Artificial Intelligence, 83-95


Beydoun, G., Low, G., Tran, N., Bogg, P., 2011, Development of a peer-to-peer information sharing system using ontologies, Expert Systems with Applications 38 (8), 9352-9364


Shashwar, A., Kumar, D., 2017, A service identification model for service oriented architecture, 2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT), IEEE.


Tran, N., Beydoun, G., Low, G., 2007, Design of a peer-to-peer information sharing MAS using MOBMAS (ontology-centric agent oriented methodology), Advances in Information Systems Development, 63-76

Xu, D., Wijesooriya, C., et al., 2011, Outbound logistics exception monitoring: A multi-perspective ontologies’ approach with intelligent agents, Expert Systems with Applications 38 (11), 13604-13611