Received: 7 September 2015 | Revised: 23 June 2016 | Accepted: 5 July 2016

DOI: 10.1002/smr.1806

WILEY Journal of Software: Evolution and Process

RESEARCH ARTICLE

A model for assessing and re-assessing the value of software reuse

Mikael Svahnberg | Tony Gorschek

Blekinge Institute of Technology, Karlskrona, Sweden

Correspondence
Blekinge Institute of Technology, SE-371 79 Karlskrona, Sweden.
Email: mikael.svahnberg@bth.se

Abstract

Background Software reuse is often seen as a cost avoidance rather than a gained value. This results in a rather one-sided debate where issues such a resource control, release schedule, quality, or reuse in more than one release are neglected.

Aims We propose a reuse value assessment framework, intended to provide a more nuanced view of the value and costs associated with different reuse candidates.

Method This framework is constructed based on findings from an interview study at a large software development company.

Results The framework considers the functionality, compliance to standards, provided quality, and provided support of a reuse candidate, thus enabling an informed comparison between different reuse candidates. Furthermore, the framework provides means for tracking the value of the reused asset throughout subsequent releases.

Conclusions The reuse value assessment framework is a tool to assist in the selection between different reuse candidates. The framework also provides a means to assess the current value of a reusable asset in a product, which can be used to indicate where maintenance efforts would increase the utilized potential of the reusable asset.

KEYWORDS

assessment, software reuse, value

1 | INTRODUCTION

As software systems grow, it becomes more and more costly to develop them from scratch for every new system. Coupled with Lehman's laws on evolution,[1,2] which indicate that the software that defined the cutting edge is prone to become more and more commoditized over time, it becomes impractical to re-create software that may be readily available already. Hence, there is a strong incentive to study software reuse, and accordingly, software reuse has been a long-standing ambition in the software industry and research community (see e.g., Refs [3–8]).

More than several models for assessing the cost and the associated cost savings of reuse (e.g., Refs [8–29]) have been proposed.

However, many of these approaches make rather restricting assumptions about reuse, namely:

- Reuse is about cost savings. Developed code can be reused more cheaply than developing from scratch. Few researchers study the *value* gained by reusing a specific asset.
- Reuse is about one-time reuse of an asset. Few researchers study the recurring value (or even cost-savings) that is incurred when the same asset is reused over multiple releases.
- Reuse reward systems focus primarily on financial reward to business units. However, the obstacles to implementing systematic reuse have less to do with financial reward than they have do with control over resources, contents, quality, and release schedule.
- Reuse reward systems rarely provide any real benefit for the individual developers.

These assumptions limit the usefulness of the proposed models and metrics. For example, to calculate cost savings, one must have access to the original development cost of the reused asset and it is also assumed that the entire asset is of interest to the consumer (in the remainder of this article, we refer to *producer* as an entity that makes an asset available for others to reuse and *consumer* as an entity that is interested in reusing an asset). Moreover, it may not only be the code that is of interest; there may be associated artifacts[17,28,30–33] such as test cases,

 Copyright © 2016 John Wiley & Sons, Ltd.

design documentation, etc. that can further increase the value of the reused asset. Other less tangible values are also of interest such as conforming to standards (internal standards such as a shared look-and-feel in the user interface, as well as external standards and regulations), sharing the long-term burden of maintenance, detecting and avoiding (or utilizing) overlaps in the product portfolio, etc.

By focusing on cost savings alone, a reused asset saves no cost in the next release, unless we also update to a newer version of the reused asset. This is a result of viewing software development as sunk costs without any intrinsic value.[34] A more modern viewpoint (as expressed even in the term "asset") is that there *is* a value in a software asset, and this value remains even if nothing new is added to the asset.

In this article, we also argue that the value of an asset lies in the eye of the beholder; that is, the actual value of an asset is determined by how well it suits a particular need. For example, assume that a reusable asset consisting of 100 KLOC is included in one release of a product. By focusing on cost avoidance, one may easily assume that this constitutes to 100 KLOC that need not be produced; that is, the project avoids the cost of producing 100 KLOC, even if only 10 KLOCs are actually used. In the next release, this cost avoidance is already carried out, and thus the reusable asset has no value but only costs associated with it, for example, to maintain it and to extend it to suit the current needs. Obviously, this reduces the incentives for reusing assets as well as for producing reusable assets. Moreover, it increases the likelihood that a local branch of the reused asset is created.

In this article, we propose a model for assessing the *value* of a reusable asset, reuse value assessment framework (RVAF), and illustrate this with an example. The contribution of this framework is that it not only enables an assessment of the *value* of a reusable asset but also provides a tool to re-assess the *current* value of an asset, based on what is currently provided and what is currently needed.

The remainder of this article is organized as follows: In Section 2, we present background and related work. In Section 3, we briefly outline the empirical support that motivates RVAF and its constituents. RVAF itself is presented in Section 4, followed by an illustration in Section 5. We discuss the framework, and in particular how this framework can be used to re-assess the value of a reusable asset, in Section 6. Finally, the paper is concluded in Section 7.

2 | BACKGROUND AND RELATED WORK

Software reuse started as reuse of source code (or punch cards)[3,35] but has become an integrated and important tool to shorten development lead times, and software reuse now means not only reuse of components, frameworks, and subsystems but also design patterns,[36] software architectures,[37–39] test cases,[40,17] etc.

Moving beyond the basic infrastructure of standard libraries and code documentation frameworks, the common toolbox of a software engineer does not contain many instruments for achieving systematic software reuse. Essentially, it is up to every company to decide and implement for themselves the infrastructure and incentives needed to facilitate reuse. Factors influencing the success of company-specific reuse programmes have been studied in a number of publications (see e.g., Refs [41–43,13,11]).

Fichman and Kemerer and Morisio et al.[41,42] study a number of projects (15 and 24, respectively) in different settings to find causes for reuse failures and successful systematic reuse. Their original stance differs; Fichman and Kemerer study reuse within development projects and Morisio et al. study implementation of reuse programmes, but their findings are similar and confirm previous studies. Failure causes include that reuse is not a priority by management, especially compared with development of new features; reuse is delegated to developers who do not know which reusable assets are available and whose natural state is to distrust assets "not invented here"; the available reusable assets may meet the functional requirements but not the quality requirements on, for example, performance; ownership and control of the reusable assets are unclear when reusing across team borders; and version management is often neglected. Success factors, on the other hand, include a strong reuse philosophy from management that introduces and influences reuse-specific processes as well as non-reuse processes; availability of good quality reusable assets and human factors are also addressed (e.g., through training, incentives programmes, and a committed central architect who champions reuse). One notable aspect discussed by Morisio et al. is the misconception that a repository of reusable assets and object orientation is sufficient for achieving reuse.[42] Not only is this not enough, but clearly, more support is needed in terms of processes for finding and evaluating reusable assets to see whether they sufficiently meet the requirements (including functional and quality requirements but also the need for clear ownership and insight into the development and release schedules of the reusable assets).

Similar to Fichman and Kemerer, Rothenberger et al.[13] study 77 development groups through a large survey constructed through meta-analysis of previously published reuse practices. The answers are analyzed through principal component analysis to explore the influence of 5 dimensions of systematic software reuse strategies on several different reuse settings. The reuse settings range from ad hoc reuse through uncoordinated reuse and systematic reuse with low management support up to systematic reuse with high management support. The dimensions (grouped into organizational dimensions and development environment dimensions) include planning and improvement, formalized process, management support, project similarity, and common architecture. Essentially (although not fully linearly), the results indicate that the more advanced the reuse setting, the higher is the influence of these dimensions on the success of the reuse programme.

While Rothenberger et al. distinguish between different reuse settings,[13] Fichman and Kemerer[41] present a more extensive range of organizational models for reuse starting from a simple library model[44] where the application development teams hold the full responsibility of publishing and finding reusable assets, via a curator model,[41] a product center model,[45] an expert services model,[45] and a team producer model[46] to a reuse factory model[47] where dedicated reuse specialists develop all software components and the application development teams merely communicate requirements and assemble components. It can thus be argued that the range of organizational models focuses on who has control over the assets, where a centralized reuse organization becomes more and more involved and ultimately assumes full control over asset development. Wesselius[43] describes a company's

evolution through some of these models to establish a successful reuse initiative.

Some keywords are of importance when discussing these organizational models: *control* (as discussed above), *quality*,[41,42] and *support*.[41,42,13] These are often listed as enablers of systematic reuse. Interestingly enough, cost and cost savings are not mentioned as 2 of the enablers.

It is also important to notice that cost and cost savings are even more important in subsequent releases than when an asset is first being reused.[19]

Favaro[19] outlines different approaches to reuse investment analysis, including net present value (NPV), payback, average return on book value, internal rate of return, and profitability index. Favaro et al.[16] further introduce real option theory as a means to assess the value of a reuse investment. At the fundamental level, all of these approaches rely on that the value of the reusable asset is expressed as a cost estimation and a cash flow (e.g., a cost saving from avoided work or an added cost to make a component more generalizable). Thus, when the investment analysis is made, any further qualification into separate parameters that have been made for a reusable asset (such as an assessment of the control, quality, and support) has been integrated into a single cost estimate for a particular year. This makes it possible to use more generic methods from, for example, corporate finance. However, this also opens up to an assumption of simplicity; that is, that cost estimations are only about cost savings and expressed in terms of work effort (time). As we argue in this article, this presents a too simplistic view when selecting between different reusable assets, and in particular when looking beyond the first release.

Lim[18] studies 17 models for reuse economics. Common for all of them is that they—like Favaro—express the economics of reuse in terms of costs and cost avoidance. More recent work[48,9–11] indicates a tendency to view the reuse problem in the same way—that reuse is about cost avoidance and is focused on the first inclusion of a reusable asset without considering the remaining value (or cost avoidance) in subsequent releases where the reusable asset is already integrated into the product.

3 | EMPIRICAL SUPPORT

To understand which aspects may influence the value of a reusable asset, we performed a study at one development site within one of the world's leading companies in telecommunication, providing a wide range of products and solutions. The company operates in a market-driven context[49] where the products are sold as generic solutions offered to an open market, although customized versions of the products are also developed and sold to key customers.

3.1 | Study design

The study was conducted as a series of workshops and interviews along with a review of the current reuse system in use in the company. The workshops were mostly held with managers, which caused us to hold more interviews with developers. The results from the interviews were analyzed for factors that may impact the decision on whether to reuse a particular asset or not, and the reasoning behind this. This was then presented and further discussed during the workshops to reach a common understanding of how reuse currently works in the company and where one may improve the current way of working with respect to reuse. To anchor the findings in a real and specific case, we also conducted a series of interviews where we followed a single asset that is reused over several products. Within the scope of this asset, we interviewed developers and managers of the asset itself, developers and managers of the products where the asset is being reused, managers that were responsible for the first time reuse of the asset, and higher level managers of the business units that are reusing the asset.

The interviews and workshops were open ended and semi-structured,[50] where each participant was able to talk freely about their experiences with the reuse of this particular asset and of reuse in general.

A summary of the overall process is presented in Figure 1, and the subjects involved in the overall discussions and the particular asset are presented in Table 1.

3.2 | Main findings

Reuse in the company's range of product occurs at all levels and with many types of asset. For example, user level functions, middleware components, and infrastructure components are all reused, as are, for example, design documents, architectures, source code components, and test frameworks. There is constantly ongoing work to systematically detect and exploit opportunities for reuse. However, some challenges still remain, and the asset value model proposed in this article is an attempt to address or at least pinpoint these challenges.

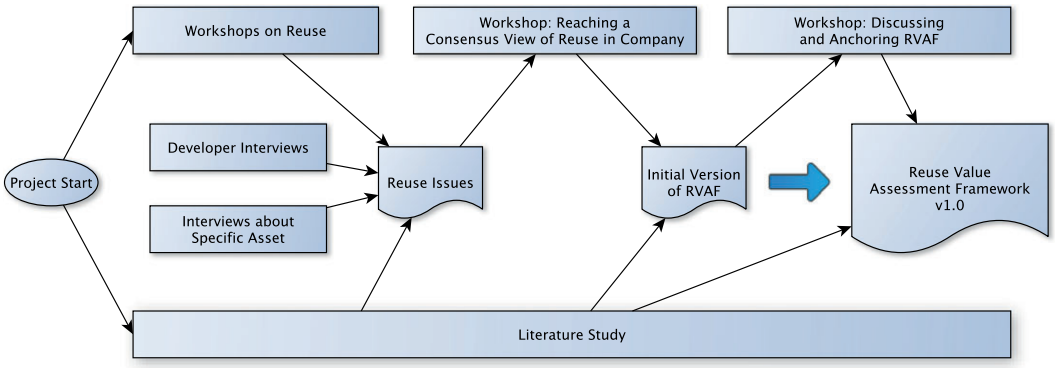

FIGURE 1 Study design

TABLE 1 Study subjects

Quantity	Subject roles	Study involvements
2	Senior managers	Workshops, general discussions, reuse in general
6	Developers	Interview, reuse in general
1	Manager of asset development	Interview, developing for reuse, understanding the reuse case
2	Asset developers	Interview, developing for reuse, understanding the reuse case
3	Developers, users of asset	Interviews, what enables reuse, what hinders reuse
3	Product managers, users of asset	Interviews, what enables reuse, what hinders reuse
2	Business unit managers	Interviews, what is carried out to enable reuse

The main findings from studying the particular asset were that the reasons for it being successfully reused were:

- There is a separate team supporting the asset.
- The basic functionality of the asset is easy to understand, and it is designed to be extended for each use.
- The asset had a couple of strong champions who happened to move between different projects and they brought the asset with them.
- The interface is stable, with only a small amount of gluecode necessary to customize it for each new application.

A number of concerns were also voiced in terms of reuse. During this discussion, the subjects did not limit themselves to the particular asset but held a more generic discussion about challenges they saw in terms of reuse. Issues mentioned were that:

- The quality of assets found in the company's internal reuse repository is unclear; you cannot just download something and start using it because you have to first review and sometimes refine the quality.
- Related to this is that it is unclear how much support is offered for an asset and how frequently it is being updated.
- It was experienced that the assets in the reuse repository were using outdated technologies.
- The available reuse assets are often too product specific and not generalizable enough to be applicable in a wider setting.
- It takes time to understand *how* to reuse a particular asset, and there is no available support for this.
- There is no available support for assessing the costs involved, for example, to determine the cost to develop from scratch and the cost to reuse a particular asset.
- Reuse is currently only measured as saved cost, such that if an asset is reused in one release of a product, the cost saved is the whole cost of developing the asset and the whole cost savings are counted for that particular release of the product without any consideration for the fact that the asset continues to be reused (and may even be more utilized) in subsequent releases.
- In some specific cases, reusable assets have been broken out and formed their own development team. In many other instances, the "reward" for developing a reusable asset is that you are able to maintain it in addition to the regular product development.

These concerns formed the starting point for the RVAF proposed in this article, and in particular the need for support to assess different candidate assets (including the default "develop it ourselves" candidate) and compare them with each other. Each of the value influencing aspects discussed in Section 4 is the outcome of discussions during the interviews and workshops, in that these are aspects that were identified as enabling or hindering reuse.

4 | REUSE VALUE ASSESSMENT FRAMEWORK

Based on the aforementioned empirical studies, we propose a value assessment framework where the current and foreseeable future needs or desires play an instrumental part in deciding the value of those asset candidates that are being considered. In this framework, based on the discussions with the company, there are a number of aspects that influence the value of an asset candidate, denoted *value influencing aspects*. Below, we discuss these value influencing aspects, first in isolation, and then combine them into a model that enables assessment of an asset candidate and a comparison between different asset candidates.

4.1 | Cost

The first value influencing aspect is, paradoxically enough, the cost of an asset candidate. To compare different asset candidates with each other, it is important to know not only the upfront investment cost but also the operating costs involved for each asset candidate.

The cost of a piece of software consists of 2 parts: the cost to actually develop the software and any other costs, such as purchasing or licensing costs. It is sometimes important to consider these separately, so we provide 2 separate definitions. In our future discussions, we also use an overall cost function that includes development costs as well as licensing and, for example, support costs.

- Let $cost_{develop}(k)$ denote the cost of developing k. This can be measured cost or estimated cost.
- Let $cost_{other}(k)$ denote additional costs for k, such as license costs.
- Let $cost(k) = cost_{develop}(k) + cost_{other}(k)$ denote the total cost of developing k.

4.2 | Functionality

The functionality provided by an asset candidate indirectly influences its value because it can be put in relation to the desired functionality. The value here is defined as satisfaction of requirements.[51] In other words, the more of the desired functionality that an asset candidate provides, the less need there is to develop extensions inhouse, and thus the more valuable it is. Thus, we state:

- Let f^d denote the set of desired functionality.

- Let f_k^p denote the set of functionality provided by asset candidate *k*.

In its simplest form, the provided functionality completely matches the desired functionality. However, it is likely that there is desired functionality that is not provided by the asset candidate. To amend this, it is necessary to write *extensions* to the asset. It may also be necessary to write "*glue*" to make the asset candidate fit into the intended operating context. This may involve wrapping a component, wrapping data formats, ensuring a common terminology in a document, etc. Thus:

- Let g_k denote the glue necessary to be able to support asset candidate *k*.
- Let e_k denote the extensions necessary to fully satisfy the desired functionality with the help of asset candidate *k*.
- Thus, $f^d = f_k^p \cup e_k$.

There is also the case where there is additional functionality provided by an asset candidate that is not presently desired. We return to this in Section 4.6.

This provides us with 2 ways of assessing the value of the provided functionality. The first is via an expert assessment of how much of the desired functionality a particular asset candidate provides expressed as a percentage of the desired functionality.

The other way to assess the value of the provided functionality is via an expert assessment of the amount of extensions that is required. This is performed by assessing the amount of extensions required to support *all* desired functionality (call this e_d) and the amount of extensions actually required to complement the functionality already provided by the asset candidate, e_k. One may then express the provided functionality as the inverse of the quotient between these, that is, $Functionality = 1 - (e_k/e_d)$.

While the latter method arguably provides a more exact value because it requires more detailed knowledge of the development effort, it is also (for the same reason) more expensive, especially during the initial comparison of several reuse asset candidates. There are also other approaches, where the provided and desired features are simply enumerated to reach a similar quotient, but this disregards the size of each feature. We do not strongly advocate any of these methods over the others, but instead summarize that:

- The value of an asset candidate with respect to *functionality* can be expressed as a percentage of the desired functionality, that is, 100% if everything that is desired is also provided, and less otherwise.

4.3 | Compliance

Related to the functionality of an asset candidate is its compliance. Compliance is a measure of how well an asset conforms to standards. This may be internal standards such as a common look and feel or a common procedure for accomplishing a certain task. It may also be compliance to external standards such as communication protocols, application programming interfaces, data standards, legal requirements, test frameworks, etc.

Our studies indicate that all of the aforementioned types of standard prove to be challenging when considering the large product portfolio that the studied company supports and the many markets that they cater for. Specific challenges identified during the interviews involve "[*inadequate*] *technologies used in the reused components*," "*old code is too product specific*," "*the component needs to be generic and self contained*. […] *By this I need not too many product specific fixes and logic*."

As with functionality, there are 2 ways of expressing the compliance provided by an asset candidate. The first way depends on that if an asset candidate is not compliant with what is desired, it is necessary to produce extensions or glue to satisfy the desired level of compliance and one may thus assess the amount of extensions and glue necessary to complement the asset candidate to meet the compliance goals. As we discuss in Section 6.1, this is an easy way to continuously track the value over several releases once a particular asset candidate has been selected.

However, for an initial selection between several asset candidates, this becomes impractical because you need to separate the extensions and glue required for compliance from all the other extensions and glue. Instead, we suggest to use an expert assessment of how compliant each asset candidate already is, expressed as a percentage (where 100% means fully compliant).

- The value of an asset candidate with respect to *compliance* can be expressed as a percentage of being fully compliant.

4.4 | Quality

During our interviews, we have seen that one obstacle for reusing an asset is if it is perceived to be of insufficient quality. Examples of statements from the interviews include "*can we ensure the quality of the reused software?*", "*the old code contains bugs*," and problems experienced in specific reuse situations, for example, "*badly written web framework*."

Analogous to compliance, if the desired quality goals are not met by an asset candidate, it must be solved either as extensions or as glue and one may thus use the amount of extensions and glue as a measure of the quality of an asset candidate. For the purpose of continuously tracking the value of a reused asset, this is an easy measure, but it is again impractical when one is conducting the initial assessment of several asset candidates.

Assessing the initial quality can be performed in a number of ways. The simplest approach is to assess the overall quality as perceived by the consumer on a scale from 0 to 1 (this may be further aided by providing different levels, for example, "acceptable quality" = 0.6, "good quality" = 0.8, and "excellent quality" = 0.9).

A more complex approach but with finer granularity is to first break down quality into relevant quality aspects, for example, according to ISO9126,[52] prioritize these, and then assess how well each asset candidate fulfils each quality aspect. An example of a method for how to do this with the help of analytic hierarchy process[53] is presented by Svahnberg et al.[54–56] Eventually, the goal of these finer-grained methods is that the quality assessment should end up as a single value between 0 (none of the quality goals are met) and 1 (all of the quality goals are fully met). Thus:

- The value of an asset candidate with respect to *quality* can be expressed as a percentage of the desired quality goals.

4.5 | Support and control

For reuse to become successful, there must be support offered for the reused assets[41,42,13] and a product owner must be available to enable this support. This is further confirmed in our interviews, for example, "*we got very much code* [...] *that was only partly used and hard to understand*", "*the problem with internal components may be lack of support and updates*", and "*if it is a 3PP, can we guarantee support?*" Analogous with the other value influencing aspects, we may assess the offered support as a percentage:

- The value of the *offered support* for an asset candidate can be expressed as a percentage, where 100% means that support is promptly and accurately provided whenever needed and 0% means that no support is offered at all.

The reaction from a development unit when insufficient support is offered is to require more *control* over the asset. This means not only control over the functional contents and the quality of the asset but also control over, for example, release schedule. As one product manager said, "*If I have to wait for someone else to integrate the updates I need, my product will be late*." Because all consumers have their own release schedule, it is a major obstacle for reuse that the consumer cannot control when a needed update will be carried out to a reused asset.

Control is often associated with added code, that is, e_k and g_k to solve the immediate needs. There may also be additional costs, for example, to maintain a relationship with the providers of the asset and to manage the extensions and glue (this cost of managing the extensions and glue extends to all e_k and g_k, whether they concern functionality, compliance, quality, or support):

- Let $cost(Control_k)$ represent the costs associated with the maintained control over asset candidate k.

4.6 | Additional functionality

One aspect of the functionality provided by an asset candidate is that there may be additional functionality provided that is not immediately desired. This additional functionality can be split into 2 parts: functionality that may be useful to have and functionality that hinders the use of an asset candidate. For example, functionality that enables use of, for example, a common test framework, may be useful, whereas functionality that, for example, re-formats data output in an unconfigurable manner, may be hindering the use of an asset. We refer to these as positive and negative additional functionalities, respectively:

- Let f_k^+ denote the additional functionality provided by asset candidate k, where $f^d \cap f_k^+ = \varnothing$.
- Let f_k^- denote the additional negative functionality provided by asset candidate k.
- Let $cost\left(f_k^{+/-}\right)$ represent the costs associated with the additional functionality in asset candidate k.

What is considered as positive and negative additional functionalities is not constant over time. Positive additional functionality may, though not used at the moment, be used in the future to create additional value. For example, a larger screen on a mobile phone creates options of enhancing web browsing, reading books, or viewing maps on the mobile phone. Other examples include that a design is documented according to a certain standard which creates options of, for example, outsourcing parts of the development, or that by using a specific test framework, the cost of reusing other assets is lower if they too use the same test framework. When such an option is exercised, the functionality moves from f^+ to the provided and desired sets ($f_k^p \cap f^d$). Likewise, functionality may move out of the desired set and become either positive or negative additional functionality.

4.7 | Summary of value influencing aspects

In Figure 2, we combine these definitions of value influencing aspects into a framework. In this figure, an asset candidate k is presented as consisting of a number of value influencing aspects, that is, cost, functionality, compliance, quality, support, and additional functionality. Moreover, Figure 2 also presents technological and organizational reactions needed to completely satisfy the desired asset. Technological reactions include developing extensions and glue to satisfy the desired functionality, compliance, and quality. As previously discussed, the organizational reaction to lack of support is to try and gain more control. This increased control, in turn, is likely to result in more of the desired asset being developed as extensions or glue.

Figure 2 allows us to reason about and compare different asset candidates. For example, if an asset is developed internally by the consumer, then everything is written as extensions e_k or glue g_k, which results in complete control. On the other end of the scale, if an asset candidate fully supports the desired functionality, then e_k and g_k are empty and there is no need for either support or control (for that particular release).

On the x-axis, we have roughly organized the value influencing aspects according to *time horizon*, as illustrated at the bottom of Figure 2. This is not a discrete timescale, but as a broad outline, we may divide the time horizon into immediate, next release, subsequent releases, and future releases. Immediately, we need to, for example, purchase the asset before we can use it (please note that there may also be annual costs associated with an asset, so the placement on the timescale is not an absolute). For the next release, we must ensure that the right functionality, the right quality, and the right level of compliance are met. If this is already satisfied, then there is no need for support for the next release, but we are going to need support to evolve the asset along with our needs in subsequent releases. Additional functionality includes opportunities for the future or things already in our roadmap. Thus, there may be an overlap with what is required in subsequent releases, but there may also be functionality that may have a value in an even longer timescale.

The y-axis expresses *effort* as follows: If the asset candidate supports all the functionality, compliance, and quality that are needed and supports the future needs (either as additional functionality or with prompt and adequate support), then there is no need to develop

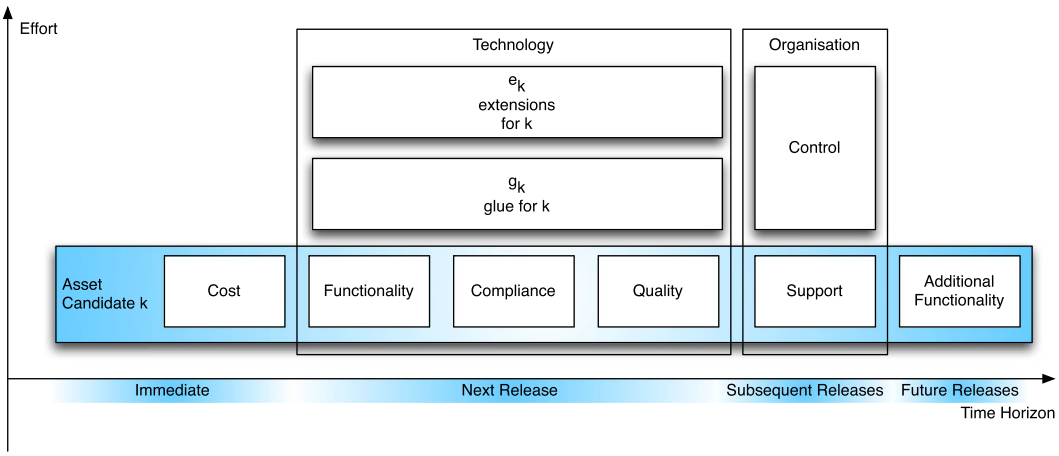

FIGURE 2 Value influencing aspects of an asset candidate

glue, extensions, or to spend effort controlling the asset candidate. Hence, the effort is minimal. On the other hand, if it is necessary to develop glue or extensions to support the desired functionality, compliance, or quality, then this translates into required effort. The moment we have developed glue or extensions inhouse, we also create a need to control these extensions as well as the original asset candidate to minimize future efforts. Thus, we invest effort into control.

This means that when reviewing a particular asset candidate, we have immediate costs and cost in terms of required effort to ensure that the asset candidate meets our expectations in the near term and in the long term. These costs and efforts can be translated into a value assessment of the asset candidate, as discussed in the next section.

4.8 | The value of an asset candidate

Using the RVAF illustrated in Figure 2, we can begin to express the value of different asset candidates in a comparable way. We propose using NPV to this end, as suggested, for example, by Poulin et al., Favaro et al., and Mili et al.[24,25,16,15] (Equation 1).

$$NPV = \sum_{i=0}^{n} \frac{B_i - C_i}{(1+d)^i} \tag{1}$$

In Equation 1, n is the number of years in the product's life cycle, d is the annual discount rate, B_i is the benefit realized in year i, and C_i is the cost incurred in year i.

The *benefits* of using a particular asset candidate k in year i can be expressed as the value of the desired functionality, compliance, quality, and support for that year. We may express these values as percent of what is desired, as follows:

- A baseline asset candidate contains 100% of the desired functionality, compliance, quality, and support in year i.
- Any other asset candidate provides less than 100%.
- The *value* of a baseline asset candidate is equal to the cost of developing the baseline asset candidate yourself, that is, the cost of satisfying 100% of the functionality, compliance, and quality in $e_{baseline}$ and managing 100% of the desired support yourself (i.e., the cost of maintaining 100% control).

For simplicity's sake, we further assume that all 4 value influencing aspects are contributing equally to the overall value. If not, they are easily weighed against each other, for example, using analytic hierarchy process[53] or cumulative voting.[57]

Equation 2 calculates the value of a particular asset candidate k in year i.

$$\begin{aligned} B_{bi} &= cost(e_{baseline})_i + cost(Control_{baseline})_i \;\; // \text{ Value of Baseline candidate} \\ B_{ki} &= B_{bi} * \frac{1}{4*100} (Functionality_{ki} \;\; // \text{ Value of candidate k} \\ &\quad + Compliance_{ki} \\ &\quad + Quality_{ki} \\ &\quad + Support_{ki}) \end{aligned} \tag{2}$$

The *costs* associated with an asset candidate consist of any potential licensing costs, the cost to develop the glue and extensions needed to support the desired functionality, as well as the cost of exercising the needed level of control over the asset candidate and the glue and extensions. There may also be a cost incurred by the need to manage any additional functionality. Equation 3 calculates the costs for asset candidate k in year i.

$$\begin{aligned} C_{ki} = {} & cost(e_k)_i && // \text{ Cost of extensions} \\ & + cost(g_k)_i && // \text{ Cost of glue} \\ & + cost_{other}(k)_i && // \text{ Other costs} \\ & + cost\left(f_k^{+/-}\right)_i && // \text{ Cost to maintain extra functionality} \\ & + cost(Control_k)_i && // \text{ Costs to maintain control} \end{aligned} \tag{3}$$

4.9 | Summary: RVAF

We thus have a model to calculate the initial value of reuse asset candidates using the value influencing aspects introduced in Section 4 and illustrated in Figure 2. This RVAF is introduced in Table 2. The intention of this framework is to gather information about each value influencing aspects for an asset candidate to be able to calculate an NPV for each asset candidate.

In Table 3, we present an overview of a selection of methods that can be used to assess the value influencing aspects sorted under benefits, that is, functionality, compliance, quality, and support. As can be

TABLE 2 Reuse value assessment framework (RVAF)

RVAF—reuse value assessment framework			
Overall			
id	Aspect	Question	
q1	n	Please assess the number of years n that you are expecting the asset to be included in the product.	
q2	d	Please assess the annual expected growth rate d of your desired functionality. Please look up this growth rate in a present value table.	
q3	$cost(e_{baseline})_i$	Please assess, for every year i, the cost of developing the desired functionality by yourself.	
q4	$cost(Control_{baseline})_i$	Please assess, for every year i, the cost of maintaining the desired control by yourself.	
Benefits			
For each year i and asset candidate k, please assess the following:			
id	Aspect	Question	
q5	$Functionality_{ki}$	Please assess the amount (in %) of desired functionality that asset candidate k provides in year i.	
q6	$Compliance_{ki}$	Please assess the amount (in %) of compliance that asset candidate k provides in year i.	
q7	$Quality_{ki}$	Please assess the level (in %) of desired quality that asset candidate k provides in year i.	
q8	$Support_{ki}$	Please assess the amount (in %) of support that is provided for asset candidate k in year i.	
Costs			
For each year i and asset candidate k, please assess the following:			
id	Aspect	Question	
q9	$cost(e_k)_i$	Please assess the cost you expect for developing extensions.	
q10	$cost(g_k)_i$	Please assess the cost you expect for developing glue.	
q11	$cost_{other}(k)_i$	Please assess the non-development related costs for asset candidate k. This includes purchase costs and licensing fees.	
q12	$cost\left(f_k^{+/-}\right)_i$	Please assess the costs you expect for compensating for any additional functionality in asset candidate k.	
q13	$cost(Control_k)_i$	Please assess the costs you expect for maintaining control over asset candidate k.	
Calculate NPV			
q14	$B_{ki} - C_{ki}$	For each year i and asset candidate k, please calculate:	
		$B_{bi} = cost(e_{baseline})_i + cost(Control_{baseline})_i$	// Value of Baseline candidate
		$B_{ki} = B_{bi} * \frac{1}{4*100}(Functionality_{ki} + Compliance_{ki} + Quality_{ki} + Support_{ki})$	// Value of candidate k

		$C_{ki} = cost(e_k)_i + cost(g_k)_i + cost_{other}(k)_i + cost\left(f_k^{+/-}\right)_i + cost(Control_k)_i$	// Cost of extensions
			// Cost of glue
			// Other costs
			// Cost to maintain extra functionality
			// Costs to maintain control

q15	NPV_k	Please calculate: $NPV_k = \sum_{i=0}^{n} \frac{B_{ki}-C_{ki}}{(1+d)^i}$	

This is your assessed net present value of asset candidate k.

TABLE 3 Examples of methods for assessing benefit aspects

Method	Description
Simple	Expert assessment of the extent (in percent) to which the aspect is supported by the asset candidate.
Levels	Expert assessment of the extent (in percent) to which the aspect is supported by the asset candidate using predefined levels, for example, "acceptable" = 60%, "good" = 80%, "excellent" = 90%.
Enumerated	Enumerate all constituents of the aspect that are provided by an asset candidate. Enumerate all constituents that are desired. Take the quotient there between.
Weighted	As enumerated, weigh each desired constituent according to their importance and normalize so that all weights sum up to 1. For each constituent, if it is present, then multiply by its weight. Finally, take the sum of all constituents.
Cost-based	Assess the amount of extensions and glue needed to support all of the aspects at the desired level. Assess the amount of extensions and glue needed to complement the reuse candidate to fully support the desired level. Take the inverse of the quotient of them both: $Aspect = 1 - (e_k/e_d)$

seen, there is a range of methods from simple expert assessments to a cost-based assessment where the costs of supporting the aspect to a desired level with the help of an asset candidate are assessed. Which method to use depends on the level of granularity that is desired for the particular decision, the familiarity with the domain, and the cost and timeframe available for making the decision. Similarly, for the cost-based approach (and for assessing costs in general), we expect software development companies to already have established methods for estimating size, effort, and resource requirements of software systems and software components.

5 | ILLUSTRATION (INDUSTRIAL AGV CASE)

In this section, we illustrate how to use the RVAF to demonstrate its value as a decision support tool. We exemplify with a case from the domain of automated guided vehicles, which consists of a system of systems that together support a warehouse management solution, including inventory, warehouse layout, and automated transportation of goods within the warehouse.

Please note that this is an illustration and not a real case study. This means that the questions in the framework are answered by the authors, according to the reasoning in this section.

In a real case, we would expect a senior developer, perhaps together with a product manager, to answer the questions in the framework. Exactly who would be involved depends on the size of the investment—reuse of some (smaller) components may be taken by a single developer, whereas reuse of larger frameworks (as in this illustration) would require the involvement of persons with more insight into the product roadmap and able to potentially invest in purchasing an external asset candidate. For the remainder of this section, we refer to the person or team of persons involved in taking the reuse decision as "the consumer."

5.1 | Context

In this illustration, we are focusing on a particular product, that is, a warehouse layout product which is a mature product that the company has been developing for some time. This product is used to create a map of a warehouse and to lay out paths for the automated guided vehicles to follow.

For the next release, this product needs to be extended with a new component to provide more flexibility regarding the rules that govern how a warehouse and the paths for the automated guided vehicles are laid out. Previously, the layout options have been rather static and difficult to change, but with this release, they are seeking to implement an extensible system that is based on user configurable rules. Rather than modeling the warehouse as a set of pre-defined paths for the vehicles to follow, they want to model the warehouse as a set of obstacles where the operators can attach rules on how to navigate around them.

There are 3 alternatives for how to include this new functionality in the product:

- Inhouse. Develop a script-based interpreter from scratch.
- External. Buy a third-party rule engine.
- Bizzies. Bizzies is an inhouse developed rule engine that is currently used to model traffic rules for the automated guided vehicles. This can be reused for the current purpose. We call this alternative "Bizzies" to separate it from the other inhouse alternative.

Below, we go through the sets of questions in RVAF and discuss how the company reasons to answer them. This serves as an example of the types of reasoning one may hold to assess an asset candidate.

The result of this discussion is presented in Table 5. This table follows the previously introduced RVAF framework and provides answers to each of the questions posed. The *id* column thus directly corresponds to the *id* column from Table 2. The table then presents each aspect for each of the asset candidates and for each year.

5.2 | Overall (q1-q4)

To assess the value of each of these asset candidates, the consumer first defines a baseline asset candidate and estimates what it would cost to develop this internally. This baseline candidate supports 100% of the desired functionality, compliance, and quality already in the first year and continues to be updated for every year to continue this complete support.

To be able to assess the costs involved in creating this baseline candidate, a fairly detailed list of current requirements and a roadmap that spans the entire investment horizon thus need to be created. The details of this are out of scope for this paper, but the results (i.e., the cost estimates) are presented in Table 4. The investment horizon is set to 5 years. Because this is an "ideal" asset candidate, certain elements are deemed not applicable, denoted "n/a" in the table. The annual values are set to the same as the annual costs for the baseline asset candidate. From this table, we obtain the investment timeframe (5 years), are able to look up the discount rate (0.7935), and obtain the $cost(e_{baseline})$ and $cost(control_{baseline})$ that we use to answer q1 to q4 in Table 5.

5.3 | Benefits (q5-q8)

5.3.1 | Inhouse alternative

When doing the estimations for this alternative, it is decided that the consumer is able to provide all of the desired functionality already in year 1. As much as 90% of the desired compliance can be achieved immediately, and the rest will be achieved in subsequent years. It is decided that for the first year, certain quality attributes need to be down-prioritized, and thus only 50% of the desired quality is reachable in year 1, with a slow incline over the next 4 years. For the inhouse alternative, there is no support available.

5.3.2 | External alternative

The external candidate supports 80% of the desired functionality already, and by studying the roadmap of this asset candidate and talking to their support staff, the consumer expects this to slowly increase to 100% of the desired functionality. Similarly, the values for compliance and quality are estimated with the help of conversations with the support staff from the company that supplies the "external" asset. Because the external component has its own release schedule and a relatively large company sells it, the amount of targeted support

TABLE 4 Illustration: a baseline asset candidate

	Aspect	Year 1	Year 2	Year 3	Year 4	Year 5
Overall	$cost(e_{baseline})_i$ $cost_{other}(baseline)_i$	400 k€ n/a	20 k€ n/a	20 k€ n/a	20 k€ n/a	20 k€ n/a
Costs	$cost(g_{baseline})_i$	n/a	n/a	n/a	n/a	n/a
	$cost\left(f^{+/-}_{baseline}\right)_i$	n/a	n/a	n/a	n/a	n/a
	$cost(control_{baseline})_i$	20 k€	20 k€	20 k€	20 k€	20 k€
	Sum	420 k€	40 k€	40 k€	40 k€	40 k€
Benefits	Functionality	100%	100%	100%	100%	100%
	Compliance	100%	100%	100%	100%	100%
	Quality	100%	100%	100%	100%	100%
	Support	100%	100%	100%	100%	100%
	Overall	100%	100%	100%	100%	100%
	Value	420 k€	40 k€	40 k€	40 k€	40 k€

to assist the consumer company is estimated to be quite low, at a constant 25% of the desired needs.

5.3.3 | Bizzies alternative

It is expected that this alternative starts out with the same amount of desired functionality as the external alternative but that it is easier to influence the development schedule because it is within the same company, and so the amount of desired functionality offered increases slightly faster than the external alternative. Because this asset candidate already exists and is developed within the same company as the consumer product, it is expected to be fully compliant from the start. Although it is likely that the quality is in fact higher, a cautious estimate is that it is not higher than what the consumer may accomplish by themselves, that is, 50% in year 1. Being available within the same company, it is estimated that the amount of support offered is quite high, that is, 75% of what is required.

5.3.4 | All candidates

The total value of the benefits is simply the average of all the value-influencing aspects (q5-q8), multiplied by the value of the baseline candidate.

5.4 | Costs (q9-q13)

5.4.1 | Inhouse alternative

For this alternative, the extensions constitute the bulk of the development effort, so a simple and probably overly cautious estimate for year 1 is to take the average of the offered functionality, compliance, and quality multiplied by the cost of the baseline candidate (Equation 4). For the remaining years, an equally cautious estimate is that the inhouse alternative will cost as much as the baseline candidate, that is, 20 k€ per year.

For the inhouse alternative, there is no need for gluecode and there is no additional functionality created. Certain parts of the inhouse developed solution can be bought, which yields a licensing cost of 10 k€. Because there is no support offered for the inhouse candidate, this is instead a pure cost, that is, $cost(control_{Inhouse})$ 20 k€ per year.

$$cost(e_{inhouse})_1 = cost(e_{baseline})_1 * (1/100) * Average(Functionality_{inhouse,1} + Compliance_{inhouse,1} + Quality_{inhouse,1}) \quad (4)$$

5.4.2 | External alternative

The cost for extensions for the external alternative, $cost(e_{external})_i$, is calculated by summing up what remains for the asset candidate to have 100% of the functionality, compliance, and quality and multiplying this with the cost of developing the entire asset from scratch, that is, $cost(e_{baseline})$. This is presented in Equation 5. Please note that this cost is calculated in the same way for all asset candidates k but is the inverse of how it is calculated for the inhouse alternative.

$$cost(e_k)_i = cost(e_{baseline})_i * (1/100) * (100 - Average(Functionality_{k,i} + Compliance_{k,i} + Quality_{k,i})) \quad (5)$$

Please also note that this is a simplification because it may be costlier to satisfy, for example, the functionality requirements than the compliance and quality requirements. It is possible to weigh the different aspects or to use a more fine-grained estimate of what remains (e.g., how many lines of code are estimated for each of the aspects), but at this stage, this would unnecessarily complicate the illustration.

There is a certain need to develop (and to subsequently maintain) gluecode to fit the "external" candidate with the remaining product. This is estimated to cost 20 k€ in year 1 and 5 k€ per year after that.

The "external" candidate costs 50 k€ to purchase initially and another 5 k€ per year in licensing fees.[1]

There is a certain amount of additional functionality, and it is expected that the consumer is initially going to spend 20 k€ per year to deal with this. In year 3, however, certain pieces of the additional functionality align with the roadmap, such that they become part of the desired functionality. As a consequence, the cost of maintaining the additional functionality decreases.

The cost to complement the offered support to the desired level of control is estimated to cost 15 k€ per year.

[1]Please remember that this is a hypothetical scenario and does not reflect any actual costs of purchasing or licensing any software.

TABLE 5 Illustration: compared asset candidates

	id	Aspect	Asset candidate	Year 1	Year 2	Year 3	Year 4	Year 5
Overall	q1	$n = 5$						
	q2	$d = 0.7935$						
	q3	$cost(e_{baseline})_i$		400 k€	20 k€	20 k€	20 k€	20 k€
	q4	$cost(Control_{baseline})_i$		20 k€	20 k€	20 k€	20 k€	20 k€
Benefits	q5	Functionality	Inhouse	100%	100%	100%	100%	100%
			External	80%	80%	85%	90%	100%
			Bizzies	80%	85%	90%	95%	100%
	q6	Compliance	Inhouse	90%	95%	100%	100%	100%
			External	80%	80%	85%	90%	100%
			Bizzies	100%	100%	100%	100%	100%
	q7	Quality	Inhouse	50%	60%	70%	80%	90%
			External	75%	75%	80%	85%	90%
			Bizzies	50%	60%	70%	80%	90%
	q8	Support	Inhouse	0%	0%	0%	0%	0%
			External	25%	25%	25%	25%	25%
			Bizzies	75%	75%	75%	75%	75%
		Overall	Inhouse	60%	64%	68%	70%	73%
			External	65%	65%	69%	73%	79%
			Bizzies	76%	80%	84%	88%	91%
		Value	Inhouse	252 k€	26 k€	27 k€	28 k€	29 k€
			External	273 k€	26 k€	28 k€	29 k€	32 k€
			Bizzies	320 k€	32 k€	34 k€	35 k€	37 k€
Costs	q9	$cost(e_k)_i$	Inhouse	320 k€	20 k€	20 k€	20 k€	20 k€
			External	87 k€	4 k€	3 k€	2 k€	1 k€
			Bizzies	93 k€	4 k€	3 k€	2 k€	1 k€
	q10	$cost(g_k)_i$	Inhouse	0 k€	0 k€	0 k€	0 k€	0 k€
			External	20 k€	5 k€	5 k€	5 k€	5 k€
			Bizzies	20 k€	5 k€	5 k€	5 k€	5 k€
	q11	$cost_{other}(k)_i$	Inhouse	10 k€	10 k€	10 k€	10 k€	10 k€
			External	50 k€	5 k€	5 k€	5 k€	5 k€
			Bizzies	10 k€	10 k€	10 k€	10 k€	10 k€
	q12	$cost\left(f_k^{+/-}\right)_i$	Inhouse	0 k€	0 k€	0 k€	0 k€	0 k€
			External	20 k€	20 k€	10 k€	10 k€	10 k€
			Bizzies	20 k€	20 k€	10 k€	20 k€	20 k€
	q13	$cost(control_k)_i$	Inhouse	20 k€	20 k€	20 k€	20 k€	20 k€
			External	15 k€	15 k€	15 k€	15 k€	15 k€
			Bizzies	5 k€	5 k€	5 k€	5 k€	5 k€
		Sum	Inhouse	350 k€	50 k€	50 k€	50 k€	50 k€
			External	192 k€	49 k€	38 k€	37 k€	36 k€
			Bizzies	148 k€	44 k€	33 k€	42 k€	41 k€
Benefits-costs	q14		Inhouse	−98 k€	−25 k€	−23 k€	−22 k€	−21 k€
			External	81 k€	−23 k€	−11 k€	−8 k€	−4 k€
			Bizzies	172 k€	−12 k€	1 k€	−7 k€	−4 k€
NPV	q15		Inhouse	−70 k€				
			External	35 k€				
			Bizzies	92 k€				

5.4.3 | Bizzies alternative

The cost for extensions is calculated in the same way as for the "external" alternative, that is, using Equation 5. The amount of required gluecode is cautiously estimated to cost as much as for the "external" alternative. Certain parts of the Bizzies solution can be bought, which yields a licensing cost of 10 k€.

As with "external," there is a certain amount of additional functionality that needs to be managed, and in year 3, this functionality is expected to become part of the desired set. However, in years 4 and 5, new additional functionality is added and the cost of maintaining the old additional functionality increases.

The cost to complement the offered support to the desired level of control is estimated to cost 5 k€ per year.

5.5 | Summary (q14-q15)

Once all the data are estimated for each of the asset candidates over the entire investment horizon, the consumer may calculate the NPV of each asset candidate, which can then be used as decision support when deciding which asset candidate to use. This is presented at the bottom of Table 5. Here, we see that the inhouse alternative has an NPV of −70 k€, "external" has an NPV of 35 k€, and Bizzies has an NPV of 92 k€. Thus, the consumer should go for the Bizzies alternative.

The fact that the inhouse alternative has an NPV of less than 0 is expected because this alternative may at best match the baseline asset candidate and may in fact be less than 100% for each value influencing aspect. Thus, the NPV will be less than the baseline asset candidate.

Bizzies is an internally developed asset, and hence the upfront investment costs are likely to be lower than when purchasing an externally developed asset. Moreover, the chance of obtaining prompt support is higher. These 2 factors are the main reasons for why, in this example, this alternative has a higher NPV than the externally developed "external" alternative.

6 | DISCUSSION

In the previous sections, we introduced RVAF as a model for assessing the value of reusable assets. By focusing on the *value* rather than the costs or cost savings, it is possible to include intangible assets such as the control, quality, and support for the reusable asset into the decision of which, if any, reusable asset to select. In the illustration, we show how this can be used not only in a binary decision as to whether there is a positive return on investment from including a particular reusable asset at a specific point in time but also to compare and weigh several alternatives against each other. One of the asset candidates in this example is to develop the functionality inhouse, but rather than assuming that all functionality, control, quality, and support will be implemented for the first release (and thereby assume that the cost avoidance of reusing an asset only occurs when it is initially being included into the product), we instead use an idealized baseline candidate to compare all other asset candidates with.

The baseline candidate is one step toward a value assessment model that is usable not only for the first release but also for subsequent releases, but more is needed. Below, we discuss and illustrate how to use RVAF to assess the value of a reused asset when the asset changes as well as when the requirements on the asset change. This raises some interesting observations about architecture degradation that we discuss further in Section 6.2.

6.1 | Tracking value through subsequent releases

In Section 4, we describe how to assess the value of an asset candidate so that it may be compared with other alternatives. To be self-critical, this contribution only adds minor clarity over, for example, Poulin's relative cost for reuse[24,25,16,15] by providing additional aspects to weigh in when assessing the value of a reuse asset candidate. The true contribution comes when considering *repeated reuse*. Because 80 to 90% of a product's life cycle is spent in a maintenance phase,[58] we can safely assume that reused assets continue to be reused in subsequent releases. The question is, how does the value of the reused asset change in subsequent releases? The NPV formula discussed in Section 4 forces you to make estimations over the years you expect the asset to be included in the product. Because we have defined and divided the desired and provided functionality as we have (Figure 2), there is a simple way to estimate and track the value of an asset throughout several releases. Implicitly, this has already been presented in the RVAF, but we would like to emphasize it further because this is one of the aspects that set this framework apart from previous value assessment suggestions.

There are 3 scenarios to consider:

- In the first scenario (case a), the product that reuses the asset is updated to a new version but the reused asset itself is kept in the same version.
- In the second scenario (case b), the situation is reversed so that the asset is changed but not the reusing product.
- In the third scenario (case c), both the product that reuses the asset and the asset itself are updated to a new version.

Case (a) occurs when either no new release of the reused asset is available or when it has been decided that nothing relevant to the consumer has been added. Case (b) occurs when there is a new release of the reused asset available and the consumer decides that something relevant has indeed been added. However, because no upgrade of the consumer's product is carried out, the value essentially remains the same until case (c) occurs. We will thus focus on cases (a) and (c).

Assuming that the support for the asset and the control over the asset remains constant, cases (a) and (c) described above require a reassessment of the reused asset. In fact, this can be performed quite simply. Consider Figure 2. If, for a new release of the consumer's product, more glue or extensions need to be written, then the value of the reused asset k has been reduced. If the glue and extensions remain constant, so does the value of the reused asset k. If, on the other hand, either the glue or the extensions becomes smaller, then the value of the reused asset k has increased; that is, it meets more of the needs posed on the system. This can be a result of the needs becoming smaller or (perhaps more typically) the asset itself supporting more functionality. Of course, the actual reduction of the glue and extensions will incur a cost, but this does not affect the value of the reused asset.

A common case worth considering is when functionality that used to be extensions is added to asset k, but where one continues to use and maintain the extensions instead. This may be due to lack of awareness that the functionality is now supported by the asset or lack of resources to actually remove functionality from the extensions and

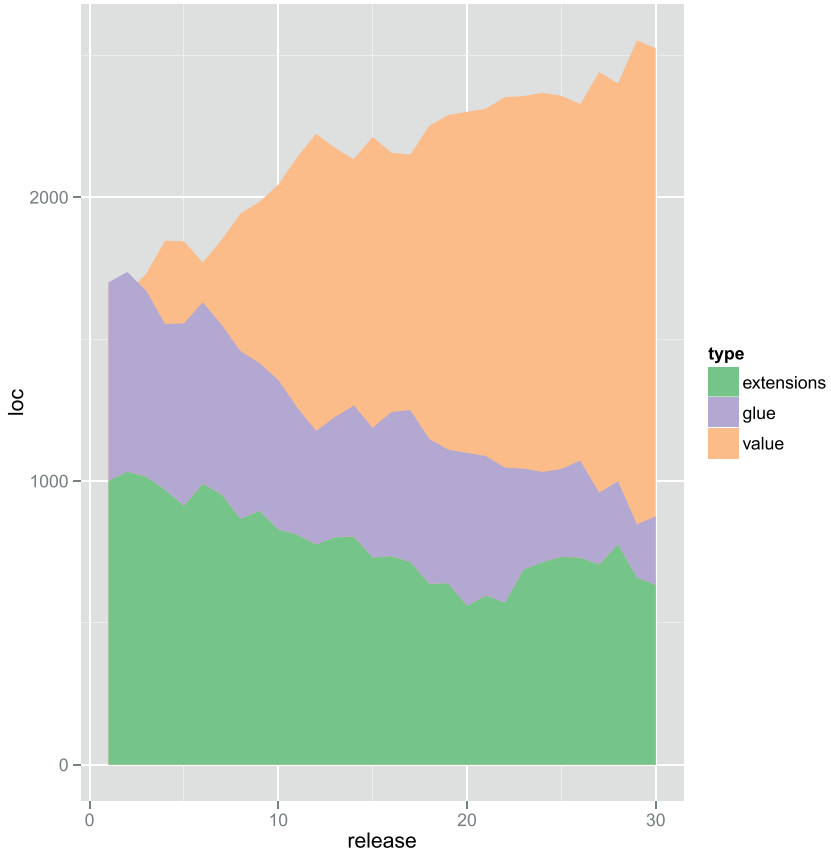

FIGURE 3 An illustration of e_k and g_k being tracked over time

start using the provided functionality. Either way, the value of asset *k* can in this context not be considered to have changed unless this move of functionality is actually performed. In fact, this duplication of functionality may even hinder effective use of the asset and we may consider the same functionality as part of the extensions as well as additional negative functionality.

It should be noted that all the other aspects in the RVAF may also change from one release to another. Quality and conformance are likely to change for a new release of the reused asset. However, the glue and extensions can still be used as primary indicators of the updated value of asset *k*, and any other changed aspects can equally easily be reassessed.

This way of tracking the value of an asset holds many benefits. First, it requires little or no knowledge about *what* has been added to the asset. Second, it requires no access to the actual asset itself. Third, it automatically filters away additions to the asset that were presently of no use to the consumer. Fourth, tracking the amount of code that deals with a particular asset (i.e., e_k and g_k) is a simple and low-key effort that can be set up to be run automatically (e.g., by using design patterns[36] to wrap the asset or by comments or tags in the configuration management system). It does, however, require an active decision to track the extensions and glue related to each reused asset.

6.1.1 | Illustration

To illustrate, suppose that the asset consumer's development organization actively responds to each change in a software asset by adding and removing extensions and glue-code as needed in the next release of their product. Over time, this will create a graph such as in Figure 3.[2] In this figure, we see that e_k (green) slowly decreases over time and that g_k (lilac) is more or less constant, until around release 22 when the amount of extensions grows and the amount of glue shrinks.

Added to Figure 3 is a notion of how the value of asset *k* changes per release (orange). This value is simply the inverse of the amount of e_k and g_k. For the sake of this illustration, we have let the original value of *k* be the same as $e_{k,0} + g_{k,0}$ to clearly see how the value mirrors e_k and g_k.

In the illustration introduced in Section 5, the initial value would be the NPV of the chosen asset candidate. For each new release, the consumer would track the extensions and glue required to support the reused asset, and this would then influence the value upward (the amount of e_k and g_k is reduced) or downward (the amount of e_k and g_k is increased). Note that this requires no additional knowledge of what the reused asset actually supports or how it has changed since the last release, instead focusing solely on how it is being used.

6.2 | Architecture degradation

Returning to Figure 2, we would like to provide a further piece of analysis as to what it would mean to "move" in the space provided.

[2]For the purpose of this illustration, this graph is created by using random numbers such that the increase or decrease in $e_{k,n}$ or $g_{k,n}$ in release *n* is normally distributed around −10 with *SD* = 50 and $e_{k,0} = 1000$, $g_{k,0} = 700$. The measurement unit is lines of code.

The x-axis is fairly straightforward because it expresses time, and we may move the time horizon for our plans closer to present time or further away into the future. We may, for example, only focus on the immediate costs, for example, if we are concerned with a project's budget. Because it is likely that a purchased asset still needs to be fitted into a development context, this would be a short-sighted choice, and so we may extend the planning horizon to the next release and see the cost of adapting an asset (or our software system) to the requirements. This is, in fact, the planning horizon used not only by many companies but also by textbooks on software reuse (e.g., Jacobson et al.[7]). Moving even further into the future, we may concern ourselves with the ease by which we can expedite perfective, corrective, and adaptive maintenance.[59]

Perhaps more interesting and less intuitive are the implications of moving upward or downward on the effort scale. We may see effort in 2 ways here: the initial effort to create (e.g., extensions, glue, and control), but more importantly, the effort needed to *maintain* the extensions, glue, and control.

Assume, for the sake of argument, that an asset *k* is not updated at all, but the product in which it is used continues to be developed and released. We would then expect to see a slow creep upward in effort because of architecture degradation.[60–62] In fact, the architecture degradation may indeed be *caused* by the asset not being updated because any modifications need to be performed outside the asset, that is, in extensions. In turn, these external modifications mean that the system becomes less coherent, and more local control over the asset and the extensions to the assets is needed.

Another example of degradation occurs when newer versions of the asset add functionality that is already solved in the extensions or glue code. Ideally, the next time the product upgrades, these extensions should then be removed to make full use of the new version of the asset. However, this work invariably has to compete for the development resources against customer-value-adding features, and hence it is often postponed to later releases. This has 2 consequences: The architecture becomes degraded and the value of the reused asset remains constant (because e_k and g_k remain unchanged). We may see this as an upward movement of the entire asset, including extensions and glue, because the asset itself now represents a larger effort.

Whenever work is performed to clean up in e_k and g_k, either by culling unused code or by replacing the own extensions with functionality provided by the asset, we see a downward movement in Figure 2. We may see this reduced maintenance effort as also being a return to the original ideas and structural soundness of the software architecture, that is, a decrease in architecture degradation.

Thus, movement upward by extending an asset or adding more local extensions and glue provides an indication of architecture degradation. This movement can also be tracked with relatively easy measures and may thus provide valuable feedback to product managers as to whether there is a need to schedule architecture maintenance work or where to focus efforts (if traceability to source is maintained).

7 | SUMMARY AND CONCLUSIONS

Software reuse is about more than reusing source code for one release of a system. Software reuse also includes design documentation, test cases, standards, and requirements among many other artifact types. What enables reuse is often not only the functional fit of the reused asset but also aspects such as company culture and management support and several other aspects related to compliance to required standards, the quality of the reuse asset, and the support offered for the reused asset. Together, these often contribute more to the decision of whether to include a reusable asset into a product than merely that the required functionality is supported. Moreover, many of these aspects continue to influence the product and the product development long after the original inclusion of the reusable asset, and yet, it is too common to only study the cost saved during the original inclusion without considering the future value that the reused asset contributes to the product.

In this article, we introduce an RVAF where the aforementioned aspects (i.e., compliance, quality, and control/support) are used to assess the value of a reusable asset, given different time horizons ranging from the immediate release, the next release, subsequent planned releases, and future potential releases. We provide an illustration of how this framework can be used in the initial selection of which of a set of asset candidates to invest in.

For subsequent releases, we discuss how, by measuring the extensions and the glue needed to keep the reusable asset up-to-date with current requirements, it is possible to cheaply re-assess the current value that the reusable asset contributes to the developed product.

We also reason about how the framework can be used to track architecture degradation and may be used as indicators to management that architecture maintenance work is required.

The contribution of this is thus that by having a framework that focuses on the value of a reusable asset rather than the cost or cost avoidance, we are able to have a clearer understanding of the *current* contribution of a reusable asset in a product and are able to take more fine-grained decisions, for example, regarding architecture maintenance.

Static and dynamic validation with real industry cases is an important and obvious next step. As future work, we plan on following a set of reused components over several releases and evaluating RVAF's ability to assess the current value of these reused components.

Moreover, the discussion about architecture degradation measured as movement upward and downward in Figure 2 introduces an interesting connection between architecture erosion and software value that can be further investigated. Examples include whether the same reasoning can be applied to the entire product and not only its individual components or whether it can be used as a motivation for increasing the value of a product or component through architecture maintenance.

REFERENCES

1. Lehman MM. Programs, life cycles, and laws of software evolution. *Proc IEEE*. 1980;68(9):1060–1076.
2. Lehman MM. Laws of software evolution revisited. *Proceedings of the European Workshop on Software Process Technology*. Berlin Germany: Springer Verlag; 1996;108–124.
3. McIlroy MD, Buxton J, Naur P, Randell B. Mass-produced software components. In: *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen*. Germany: NATO Science Committee; 1968:88–98.

4. Brooks F. *The Mythical Man-Month: Esays on Software Engineering*. Reading, MA: Addison-Wesley Pub. Co; 1975.
5. Parnas DL. On the design and development of program families. *IEEE Trans Softw Eng*. 1976;2(1):1–9.
6. Frakes W, Isoda S. Success factors of systematic reuse. *IEEE Softw*. 1994;11(5):14–19.
7. Jacobson I, Griss M, Jonsson P. *Software Reuse: Architecture, Process, and Organization for Business Success*. Reading, MA: Addison-Wesley; 1997.
8. Frakes W, Kang K. Software reuse research: status and future. *IEEE Trans Softw Eng*. 2005;31(7):529–536.
9. Jasmine K, Vasantha R. Cost estimation model for reuse based software products. IMECS 2008: International Multiconference of Engineers and Computer Scientists; 2008.
10. Mohagheghi P, Conradi R. An empirical investigation of software reuse benefits in a large telecom product. *ACM Trans Softw Eng Methodol*. 2008;17(3):13
11. Mohagheghi P, Conradi R. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empir Softw Eng*. 2007;12(5):471–516.
12. Nazareth DL, Rothenberger MA. Assessing the cost-effectiveness of software reuse: a model for planned reuse. *J Syst Softw*. 2004;73(2):245–255.
13. Rothenberger M, Dooley K, Kulkarni U, Nada N. Strategies for software reuse: a principal component analysis of reuse practices. *IEEE Trans Softw Eng*. 2003;29(9):825–837.
14. Rothenberger MA, Nazareth D. A cost-benefit model for systematic software reuse. Proceedings of the 10th European Conference on Information Systems (ECIS 2002)2002;371–377.
15. Mili A, Fowler Chmiel S, Gottumukkala R, Zhang L. Managing software reuse economics: an integrated ROI-based model. *Ann Softw Eng*. 2001;11(1):175–218.
16. Favaro J, Favaro K, Favaro P. Value based software reuse investment. *Ann Softw Eng*. 1998;5(1):5–52.
17. Frakes W, Terry C. Software reuse: metrics and models. *ACM Comput Sur*. 1996;28(2):415–435.
18. Lim W. Reuse economics: a comparison of seventeen models and directions for future research. Proceedings Fourth International Conference on Software Reuse, IEEE1996;41–50.
19. Favaro J. A comparison of approaches to reuse investment analysis. Proceedings Fourth International Conference on Software Reuse, IEEE1996;136–145.
20. Lim W. Effects of reuse on quality, productivity, and economics. *IEEE Softw*. 1994;11(5):23–30.
21. Poulin JS, Caruso JM. A reuse metrics and return on investment model. Software Reusability, 1993. Proceedings Advances in Software Reuse., Selected Papers from the Second International Workshop on1993;152–166.
22. Malan R, Wentzel K. Economics of software reuse revisited. Proceedings, 3rd Irvine Software Symposium, UC Irvine1993;109–121.
23. Malan R. Software reuse: a business perspective. Hewlett-Packard Technical Report 1993.
24. Poulin J, Caruso J. A reuse metrics and return on investment model. Proceedings of Advances in Software Reuse., Selected Papers from the Second International Workshop on Software Reusability, IEEE1993;152–166.
25. Poulin J, Caruso J, Hancock D. The business case for software reuse. *IBM Syst J*. 1993;32(4):567–594.
26. Margono J, Rhoads TE. Software reuse economics: cost-benefit analysis on a large-scale Ada project. Proceedings of the 14th international conference on Software engineering1992;338–348.
27. Barns B, Bollinger TB. Making reuse cost-effective. *IEEE Softw*. 1991;8(1):13–24.
28. Bollinger TB, Pfleeger SL. Economics of reuse: issues and alternatives. *Inf Softw Technol*. 1990;32(10):643–652.
29. Barnes B, Durek T, Gaffney J, Pyster A. A framework and economic foundation for software reuse. *Software Reuse: Emerging Technology*. Los Alamitos, CA, USA: IEEE Computer Society Press; 1988;77–88.
30. Jones C. Economics of software reuse. *Computer*. 1994;27(7):106–107.
31. Leach RJ. *Software Reuse: Methods, Models and Costs*. Inc.: McGraw-Hill; 1996.
32. Griss ML, Wentzel KD. Hybrid domain-specific kits. *J Syst Softw*. 1995;30(3):213–230.
33. Reusable asset specification v2.2. OMG Technical Report. http://www.omg.org/spec/RAS/2.2/ Nov 2005. Accessed August 03, 2013.
34. Biffl S, Aurum A, Boehm B, Erdogmus H, Grünbacher P (Eds). *Value-based Software Engineering*. Berlin, Germany: Springer Verlag; 2006.
35. Parnas DL. On the criteria to be used in decomposing systems into modules. *Commun ACM*. 1972;15(12):1053–1058.
36. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-oriented Languages and Systems*. Reading MA: Addison-Wesley; 1994.
37. Buschmann F, Jäkel C, Meunier R, Rohnert H, Stahl M. *Pattern-oriented Software Architecture — A System of Patterns*. Chichester UK: John Wiley & Sons; 1996.
38. Clements P, Northrop L. *Software Product Lines — Practices and Patterns*. Boston, MA: Addison-Wesley; 2002.
39. Bass L, Clements P, Kazman R. *Software Architecture in Practice*. Reading, MA: Addison-Wesley Publishing Co.;1998.
40. Von Mayrhauser A, Mraz R, Walls J, Ocken. Domain based testing: increasing test case reuse. Proceedings of IEEE International Conference on Computer Design1994;484–491.
41. Fichman R, Kemerer C. Incentive compatibility and systematic software reuse. *J Syst Softw*. 2001;57(1):45–60.
42. Morisio M, Ezran M, Tully C. Success and failure factors in software reuse. *IEEE Trans Softw Eng*. 2002;28(4):340–357.
43. Wesselius J. The bazaar inside the cathedral: business models for internal markets. *IEEE Softw*. 2008;25(3):60–66.
44. Griss ML. Software reuse: from library to factory. *IBM Syst J*. 1993;32(4):548–566.
45. Goldberg A, Rubin KS. *Succeeding with Objects: Decision Frameworks for Project Management*. Boston MA, USA: Addison-Wesley; 1995.
46. Fafchamps D. Organizational factors and reuse. *IEEE Softw*. 1994;11(5):31–41.
47. Caldiera G, Basili VR. Identifying and qualifying reusable software components. *IEEE Comput*. 1991;24(2):61–70.
48. Boehm B, Brown AW, Madachy R, Yang Y. A software product line life cycle cost estimation model. Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on2004;156–164.
49. Gorschek T, Wohlin C. Requirements abstraction model. *Requir Eng*. 2006;11:79–101.
50. Robson C. *Real World Research*. 2nd ed. Malden MA: Blackwell Publishing; 2002.
51. Khurum M, Gorschek T, Wilson M. The software value map — an exhaustive collection of value aspects for the development of software intensive products. *J Softw Evol Proc*. 2013;25(7):711–741.
52. Software Qualities ISO/IEC FDIS 9126-1:2000(E) 2000.
53. Saaty TL. *The Analytic Hierarchy Process*. New York NY: McGraw Hill, Inc.; 1980.
54. Svahnberg M, Wohlin C, Lundberg L, Mattsson M. A method for understanding quality attributes in software architecture structures. In: *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*. New York, NY: ACM Press;2002:819–826.

55. Svahnberg M. An industrial study on building consensus around software architectures and quality attributes. *J Inf Softw Technol*. 2004;46(12):805–818.
56. Svahnberg M, Wohlin C. An investigation of a method for identifying a software architecture candidate with respect to quality attributes. *Emp Softw Eng Int J*. 2005;10(2):149–181.
57. Leffingwell D, Widrig D. *Managing Software Requirements — A Unified Approach*. Boston, MA: Addison-Wesley; 2000.
58. Pigotski T. *Practical Software Maintenance — Best Practices for Managing Your Software Investment*. New York, NY: John Wiley & Sons; 1997.
59. Lientz B, Swanson E, Tompkins G. Characteristics of application software maintenance. *Commun the ACM*. 1978;21(6):466–471.
60. Parnas D. Software aging. In: *Proceedings of the 16th International Conference on Software Engineering*. Los Alamitos, CA: IEEE Computer Society Press;1994:279–287.
61. Jaktman C, Leaney J, Liu M. Structural analysis of the software architecture — a maintenance assessment case study. In: *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*. the Netherlands: Kluwer Academic Publisher; 1999.
62. Gurp J, Bosch J. Design erosion: problems & causes. *J Syst Softw*. 2002;61(2):105–119.

How to cite this article: Svahnberg M, Gorschek T. A model for assessing and re-assessing the value of software reuse. *J Softw Evol Proc*. 2017;29:e1806. https://doi.org/10.1002/smr.1806