A case study on the cost and benefits of bus-oriented architectures

David Walschots

August 25, 2010

Master Software Engineering
Universiteit van Amsterdam

Thesis supervisor: dr. J.J. Vinju

Research institute: Centrum Wiskunde & Informatica

Abstract

The use of software buses has increased due to the popularity of service-oriented architectures, even though it is unknown whether software buses and the architectures within which they reside achieve their proposed benefits, and at what cost. If benefits are not achieved or are difficult to achieve, then evolution of systems containing a software bus will be more difficult and costly than expected. This thesis observes the achievement of the proposed benefit of time decoupling, which enables two components to participate in an interaction without both being available at the same time. Time decoupling is found to be achievable, with its successful employment depending on the granularity of operations found within components connected to the software bus. The complexity cost of coordination scripting shows a close to linear growth relative to the growth of components which are coordinated.

Software buses can achieve time decoupling. Relative cost does not grow as systems using a software bus evolve.

1 Introduction

The idea of bus-oriented software architectures originated from the hardware bus found in computer architectures. A hardware bus acts as a central component for inter-hardware communication and allows for a wide variety of hardware components to be connected with each other. In the 1990s various academic software buses were proposed. Similar to the hardware bus these software buses connect a wide variety of software components with each other.

Service-oriented architectures commonly contain a software bus for communication between services. Due to the increased popularity of service-oriented architectures, usage of software buses is on the rise; even though it is unknown whether software buses and the architectures within which they reside achieve their proposed benefits, and at what cost.

The following benefits of bus-oriented architectures are mentioned in literature:

- Unconstrained connectivity between components implemented in different languages and developed for different platforms. Thus, a bus-oriented architecture should be heterogeneous by putting no restrictions on the languages and platforms used for implementing its components [32, 3, 23].
- Easy adaptation of the execution location of a component. There should be no need to modify a component when its location changes [32].
- Component independence, enabling the easy replacement of components and easier reuse of their implementation [3].

Unconstrained connectivity between different languages and platforms, and the easy adaptation of execution location are simply *features* offered by a software bus. Component independence on the other hand is not a feature. The software bus provides features which can make it easier to implement a system which achieves component independence, but it is up to the development team to successfully apply these features.

Independence of components is achieved by decoupling them. Eugster et al. define three types of decoupling which can be provided by a communication paradigm [14], such as a software bus. Later work by Aldred et al. formalised these types of decoupling [1]. For use in this thesis both works are combined into the following definitions:

Space decoupling is provided when interacting components are unaware of each others location.

Synchronisation decoupling is provided when the main thread of control of both the sending and receiving component can continue their execution whilst an interaction takes place between them.

Time decoupling is provided when components do not need to participate in an interaction at the same time. This is achieved by communicating through an intermediary component which stores messages, such as a message queue.

The achievement of space decoupling is of no particular interest to this study, because such decoupling is already provided by operating systems in the form of network sockets, network addressing and transport protocols. Synchronisation decoupling is also of no interest, because the achievement of synchronisation decoupling simply depends on whether or not such functionality is provided by the communication platform. Time decoupling on the other hand, does not only depend on the availability of a message queue, but also on the implementation of components and the specification of their interaction patterns.

Achievement of time decoupling is not easy, as software systems which aim to be time decoupled should be developed using an event-driven programming model, in which event handler respond to incoming events, instead of method calls [23]. Also, for some use cases like those dealing with direct user interaction, time decoupling is not applicable.

Given the possible advantages of bus-oriented architectures, and their possible implementation issues the following question is proposed:

Research Question 1. What factors contribute to the successful employment of a bus-oriented architecture?

A bus-oriented architecture is deemed successful when its proposed benefits are achieved. The question above is broadly scoped, even though this study is primarily aimed at factors in achievement of component independence through time decoupling. This is done so that observations which are of high interest, but not directly related to time decoupling, are not discarded.

1.1 Complexity cost

The achievement of proposed benefits of bus-oriented architecture cannot be the only measure of its success. If in the process of achieving such benefits the architecture greatly increases system complexity, then companies might find other architectures to be a more viable option. Complexity affects the cost of maintenance [2], because the complexity of a system's source code affects its understandability [26, 22], which in turn affects system maintainability, since any code maintenance requires that the maintainer understand the source code [5].

Within bus-oriented architectures, part of its complexity lies in the scripting logic which specifies how messages are routed between components. Such logic can route a message based upon its content [14]. According to Hohpe and Woolf the primary issue with this type of coordination, which they call the *content-based (message) router* pattern, is that the scripting becomes a frequent point of maintenance [23]. Due to these frequent changes the complexity of this logic is of special interest, which is why this thesis provides an answer to the following question:

Research Question 2. What is the complexity cost of a bus-oriented architecture?

A case study of the ASF+SDF Meta-Environment is performed to answer Question 1 and Question 2. The Meta-Environment is a software system in which language definitions can be edited, checked and compiled [25]. It utilises the ToolBus software bus for inter-component communication and has been under development for over ten years, which enables it to provide insight into the evolutionary properties of bus-oriented architectures.

The Meta-Environment's evolution is analysed by mining its software repository. The premise of such *software repository mining* research is that it will shed light on changes that occur over time, for a given set of measured variables [24].

1.2 Contributions

This thesis provides factors which contribute to the achievement of time decoupling within bus-oriented architectures. It also provides results on the complexity cost of bus-oriented architectures.

These contributions are not only related to bus-oriented architectures, but also to service-oriented architectures. In their paper on architectural styles for service-oriented computing, Dillon et al. indicate the existence of the *broker*-style in which an intermediary component is involved in the interaction between other components [13]. According to Dillon et al., the WS-Notification specification family is able to achieve time decoupling. Software buses such as the ToolBus are brokers and can as such provide insight into the practical benefits of using these WS-Notification specifications.

Another broker-style implementation is that of Tuple- and Triple Space. Tuple Space and the extending Triple Space Computing claim to provide full decoupling in time [16]. In Tuple Space, processes can write, delete and read tuples from a globally persistent space located in a central component [16]. Triple Space adds, amongst other functionality, a publish/subscribe paradigm [13] in which processes can subscribe to triples which match a pattern and be notified when matching triples are written to the Triple Space. The Meta-Environment contains a very similar component called the `module-manager`, whose time decoupling features are introduced in Section 4.1.3.

1.3 Structure

The remainder of this thesis is structured as follows. Section 2 presents background on the ToolBus, the Meta-Environment, and the ToolBus in comparison to other software buses. Section 3 presents the research method, also discussing this study's solutions to issues commonly found in repository mining research. Section 4 presents the results. Section 5 presents a discussion on the results and their validity. Finally Section 6 concludes the work.

2 Background

Before establishing the background on several topics, this section provides definitions used throughout the remainder of this paper.

An *endpoint* is an entity which can participate in communication [1].

An *interaction* occurs when two endpoints exchange information [1, 33].

A *message* is a unit of information which is transported between endpoints during their interaction [1].

The next subsections provide the background on the ToolBus, the Meta-Environment, and relate the ToolBus to other software buses.

2.1 ToolBus

The ToolBus is the software bus used by the Meta-Environment for communication between its components. Components which are connected to the ToolBus are called *tools*. The messaging abilities of a tool are specified in a *tool interface definition*, which defines the messages which can be consumed by the tool, and the messages which are produced by the tool. A tool interface definition is a subtype of a process definition, which is described below.

Processes are runtime entities which perform atomic actions, to coordinate with other processes and tools. Process definitions specify the composition of such atomic actions using process algebra primitives, as shown in Table 1. Process definitions are instantiated into objects at runtime. Processes are not singletons, because the ToolBus can instantiate multiple (tool interface) process objects from a single process definition.

The ToolBus interleaves the coordination tasks performed by processes. Processes do not perform

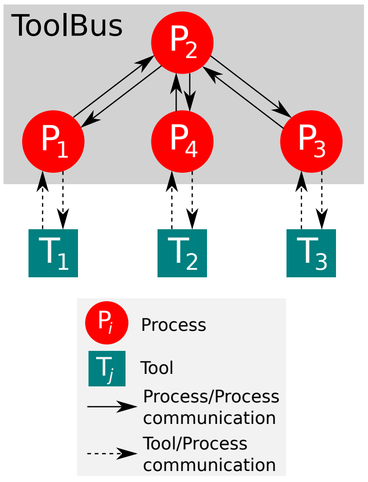


Figure 1: ToolBus scenario

any computation, and they do not modify the content of any message. One can say that the ToolBus splits coordination and computation, where coordination is performed in processes, and computation in tools.

Figure 1 shows a typical scenario in which processes P_i coordinate computation in tools T_j.

2.1.1 Communication

Processes can communicate using messages or notes or both. As the use of *message* is ambiguous, from here on out a ToolBus specific message is referred to as *toolbus-message*.

A toolbus-message can be send by a process to *one* other process. The sending process waits until a receiving process picks up the toolbus-message, even if such receiving process is not available at the time. Computation is blocked until the receiving process has fully handled the toolbus-message and returns control to the sending process.

A note can be broadcast from a process to *multiple* other processes. Notes are an implementation of the publish/subscribe pattern. In this pattern, subscribers have the ability to express their interest in an event, and are subsequently notified of any event, generated by a publisher, which matches their registered interest [14]. On broadcasting a note, it is placed in the note queues of processes which have previously subscribed to such notes. After broadcasting, the sending process can continue coordinating other messages without waiting for a reply from processes that received the note. Processes that received the note can also continue their coordination until they specifically request for the note to be retrieved from their process note queue. Thus, computation is not blocked.

Constructs similar to toolbus-messages and notes are used for communication between processes and tools. A process can ask a tool to perform some function and wait for a return value. A tool can communicate with its interface process using events which are placed in a queue similar to a note queue. The interface process can then later retrieve the value from the queue.

Message data The body of toolbus-messages and notes contains ATerms. ATerms contain data types like integers, reals, lists [36], etc. An example of an ATerm is `set-file-name("Article.tex")`, which combines the *identifier* `set-file-name` with a string *value* `"Article.tex"`.

ATerms have a function beyond simply being the content of a message body. Processes match the pattern of an ATerm to define which messages they can receive. It is through this pattern matching that the ToolBus achieves content-based routing. If some process sends ATerm `set-file-name("Article.tex")` then it can be received by some other process P_r, if P_r specifies the pattern `set-file-name(FileName?)`. The question mark following `FileName` is a wild card which matches any ATerm and assigns it to the `FileName` variable.

2.1.2 TScript

TScript is the scripting language used for establishing process definitions and tool interface definitions. Code listing 1 provides an example of a tool interface definition. Table 1 contains common primitives found within the TScript language.

The TScript example specifies a tool called the `module-manager`, which is executed using the command specified. The `ModuleManager` tool interface process executes the `module-manager` tool and then waits for one of three choices between alternative actions. What is interesting is that atomic actions can be nested into multiple choice (+) or other process-oriented primitives, an option which is used within the example to send a different message onto the ToolBus if there is no value assigned

```
tool module-manager is {
  command = "__PREFIX__/bin/module-manager"
}

process ModuleManager is
let
  MM : module-manager,
  Key : term,
  Value : term,
in
  execute(module-manager, MM?)
  .
  (
    rec-msg(mm-add-attribute(Value?))
    . snd-do(MM, add-attribute(Value))
  +
    rec-event(MM, attribute-changed(Value?))
    . snd-ack-event(MM, attribute-changed(Value))
    . snd-msg(mm-attribute-changed(Value))
  +
    rec-msg(mm-get-attribute(Key?))
    . snd-eval(MM, get-attribute(Key))
    .
    (
      rec-value(MM, attribute(Value?))
      . snd-msg(mm-attribute(Key, Value))
    +
      rec-value(MM, no-such-key)
      . snd-msg(mm-no-such-key(Key))
    )
  )
  *
  delta
endlet
```

Code listing 1: Example tool interface definition

to the given key.

Using these process-oriented primitives and the ATerm pattern matching functionality the ToolBus can achieve its content-based routing.

2.2 The ASF+SDF Meta-Environment

This study's primary artefact is the ASF+SDF Meta-Environment. The Syntax Definition Formalism (SDF) provides a syntax for programming language grammar specifications. Within the Algebraic Specification Formalism (ASF) one can utilise the SDF specification to specify rewrite rules for use within the domain of software analysis and transformation. The ASF+SDF Meta-Environment provides an integrated development environment for editing these ASF+SDF definitions, also known as ASF+SDF modules. It makes extensive use of the ToolBus.

2.2.1 Static deployment structure

The Meta-Environment's static deployment structure contains packages, programs, tools and libraries. In Figure 2 a coordination view[1] of the Meta-Environment is shown. The view shows interactions between tools and the ToolBus. It splits the Meta-Environment into three functional areas: a *kernel* area which contains tools that provide the primary system functionality, a *SDF* area which contains tools related to SDF grammar and an *ASF* area for tools related to term rewriting.

Functional areas were created, because the Meta-Environment aims to be an open architecture targeted to the design and implementation of term rewriting environments [37]. In practice this has lead to various different term rewriting formalism implementations which replace the ASF area, such as ELAN [37].

The next paragraphs will briefly discuss the structural elements found within the Meta-Environment.

Packages The Meta-Environment's development team releases multiple products. The notion of packages was adopted to ease the deployment process of such products. Packages are the most coarse-grained components found within the system. They encapsulate the source code of one or more programs, tools or libraries. Packages are not necessarily independent (e.g. if a program element within a package requires the use of the toolbus then the package depends upon the *toolbus* package) but because of their encapsulation multiple products can easily be created.

Packages are part of the static deployment structure. They are not part of the Meta-Environment's runtime architecture, as interconnectivity between tools specified in packages is handled by the ToolBus.

Programs Programs are runnable components of the system.

[1]The figure is obtained from the Meta-Environment's architecture documentation, located at: http://www.meta-environment.org/doc/books/meta-environment/architecture-meta-environment/architecture-meta-environment.html

Primitive	Description
`+` (e.g. A_1 `+` A_2)	Choice between two alternative actions (A_1 or A_2).
`.` (e.g. A_1 `.` A_2)	Sequential composition (A_1 followed by A_2).
`*` (e.g. A_1 `*` A_2)	Iteration (zero or more times A_1, followed by A_2).
`||` (e.g. A_1 `||` A_2)	Parallel composition (A_1 and A_2 at the same time).
`create(Process, Id?)`	Create `Process`, and provide the process `Id`.
`snd-msg(T)`	Send a toolbus-message of pattern `T`.
`rec-msg(T)`	Receive a toolbus-message of pattern `T`.
`snd-note(T)`	Send a note of pattern `T`.
`rec-note(T)`	Receive a note of pattern `T`.
`subscribe(T)`	Subscribe to notes of pattern `T`.
`unsubscribe(T)`	Unsubscribe to notes of pattern `T`.
`snd-eval(Tool, T)`	Request evaluation of `T` by `Tool`.
`rec-value(Tool, T?)`	Receive value of pattern `T` from `Tool`.
`snd-do(Tool, T)`	Request action `T` by `Tool`, without return value.
`rec-event(Tool, T?)`	Receive an event of pattern `T` from `Tool`.
`snd-ack-event(Tool, T)`	Acknowledge receiving event of pattern `T` from `Tool`.
`if ... then ... fi`	Guarded command.
`if ... then ... else ... fi`	Conditional expression.
`let ... in ... endlet`	Local variables.
`:=`	Assignment.
`rec-connect(T?)`	Receive a connection request from tool `T`.
`execute(Tool, Id?)`	Execute `Tool`, and provide the tool `Id`.

Table 1: Selection of ToolBus TScript primitives, as provided in [3].

Tools Tools were previously mentioned in Section 2.1. Tools are both part of the static as well as the dynamic structure of a ToolBus application. Static, because they are defined within the ToolBus's TScript, and are a subtype of programs. Dynamic, because they are a runtime entity which is instantiated and executed from the ToolBus.

Libraries Libraries are components referenced by the Meta-Environment's tools to perform some function for which no ToolBus connection is deemed necessary.

2.2.2 Programming languages

The Meta-Environment is written in various programming languages. The main programming languages are C, Java and ASF+SDF, where ASF+SDF is transformed into C code. Early versions of the Meta-Environment's user interface were created using Tcl/Tk, although these components were later replaced by components written in Java. TScript is used to define communication between system components.

2.3 Software buses

The software bus is the primary component found in any bus-oriented architecture. The remainder of this section describes three software buses, and the features they provide to achieve the proposed benefits of bus-oriented architectures. This positions the ToolBus [3] within the software bus domain. Other buses under observation are the Polylith software bus [32] and Information Bus [30].

2.3.1 Unconstrained language and platform

There are two notions to consider regarding the achievement of heterogeneity: the notion of *control integration* which is concerned with the communication and cooperation amongst components, and the notion of *data integration* which is concerned

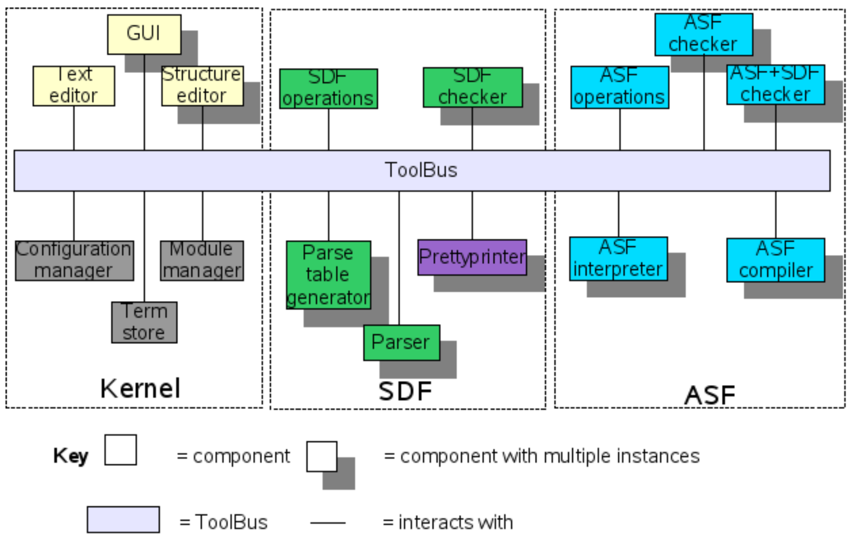


Figure 2: Meta-Environment coordination view

with the exchange of data structures amongst components [3]. Within software buses these data structures are represented as messages.

Control integration The Polylith software bus uses a module interconnection language (MIL) for its control integration. MILs have the purpose of establishing the overall program structure [12]. The Polylith MIL describes the static structure of the system. It is based upon a simple graph model of interconnection, with nodes corresponding to components and edges representing the bindings between their interfaces [32]. Once ready for deployment, the artefacts of a software system are packaged according to the MIL specification, using a concrete instance of a software bus abstraction. This process causes the Polylith software bus to be relatively static compared to the other systems.

The ToolBus utilises a scripting language called TScript, as described in Section 2.1.2. It is based upon process algebraic constructs, allowing the specification of atomic actions, alternative composition, sequential composition and parallel composition [17].

The Information Bus provides very limited control integration, because it simply routes messages based upon their subject [30].

Data integration All three systems propose their own message data format and use *adapters* for data integration purposes. The task of such an adapter is to map between the implementation language data format and the software bus message format and vice versa. The buses put restrictions upon the data types which can be used, such that they are interchangeable between endpoints implemented in different programming languages.

2.3.2 Adaptable execution location

The three architectures feature two very distinct solutions for ensuring that no change needs to be made to a component when its execution location changes. The Polylith software bus describes the location of components before packaging them into a software system, thus binding them together at a specific point in time. The ToolBus and Information bus allow for a component to register onto the software bus at any time. Within the ToolBus this is done using the `rec-connect(T?)` primitive (Table 1). The Information Bus utilises its publish/-subscribe mechanism as a discovery protocol [30]. Both solutions allow components to specify their execution location within their own configuration environment, and then bind themselves to the bus. In case of the ToolBus, the execution location of tools can also be defined before executing the ToolBus.

2.3.3 Independent components

Software buses provide component decoupling through their communication paradigm.

The Information Bus and ToolBus achieve space decoupling by using an intermediary communication platform. Within the Polylith software bus, space decoupling is achieved statically but removed during the packaging process.

The ToolBus and Information Bus can provide time decoupling through the use of message queues. Use of message queues requires that endpoints can subscribe to messages they are interested in. For this purpose the Information Bus allows endpoints to subscribe to message subjects. The ToolBus uses pattern matching to identify messages which should be received by a process [4].

Understanding the synchronisation decoupling provided by a software bus requires detailed analysis of its implementation. The ToolBus is decouples synchronisation of note retrieval, but sending a note to a note queue is a blocking operation. ToolBus-messages are coupled in synchronisation. For the Polylith software bus, synchronisation decoupling likely depends on the concrete software bus implementation provided by the software developer. The Information Bus utilises a publish/-subscribe mechanism similar to that of the ToolBus, and (most likely) synchronisation decouples the retrieval of published messages as well.

3 Research Method

A visual outline of the research method is provided in Figure 3. It displays artefacts and intermediate steps to create such artefacts. The steps of applying metrics, formulae, observation and analysis vary, depending on the hypothesis which is researched. First, a high-level overview of the artefacts and steps is provided. Thereafter, Section 3.1 and Section 3.2 provide the motivation for two hypotheses, and the concrete implementations of the research steps. Lastly, Section 3.3 discusses some overall issues in repository mining, such as revision selection and generated file handling.

The Meta-Environment's revisions form the initial artefact. The revision control system contains over 31000 revisions, which makes it impossible to observe each revision individually. Therefore, a bird-eye view of the system is needed. Such view is created by applying *metrics* to *revisions*, to create *raw data* which can then be summarised using *summarising formulae* into *statistics*. These statistics provide the bird-eye view of the system. Using this view, changes of interest are *observed*, both within the statistical representation itself, as well as within specific revisions. These *observations* are then *analysed* to come to a *discussion*, and subsequent conclusions.

3.1 Time decoupling

The introduction stated Research Question 1 as follows: "What factors contribute to the successful employment of a bus-oriented architecture?". This question was further scoped towards achievement of component independence as provided by a communication platform. The background showed that the ToolBus provides space decoupling and partial synchronisation decoupling. It also showed that notes are the primary means toward achieving time decoupling.

Time decoupling can be used to achieve component independence and reuse, which are development goals for the Meta-Environment. Therefore, the following hypothesis is proposed:

Hypothesis 1. The ToolBus's note primitives can successfully decouple tools in time.

Notes are the primary means toward achieving time decoupling. When notes are not used, then the hypothesis can be falsified. When notes are used this is not immediate proof in support of the hypothesis, because whether or not these notes are used *correctly* is unknown, therefore qualitative analysis is needed. The next paragraphs provide the research steps for this hypothesis.

3.1.1 Metrics and statistics

Whether or not notes are used is measured by counting the number of notes that are sent and received. Notes which are sent, but never received and vice versa, are not counted. Counting is the operation of finding `snd-note(T)` and `rec-note(T)` primitive (Table 1) occurrences, and for each occurrence adding 1 to the total.

The raw data is summarised into a plot which depicts the number of send note and receive note

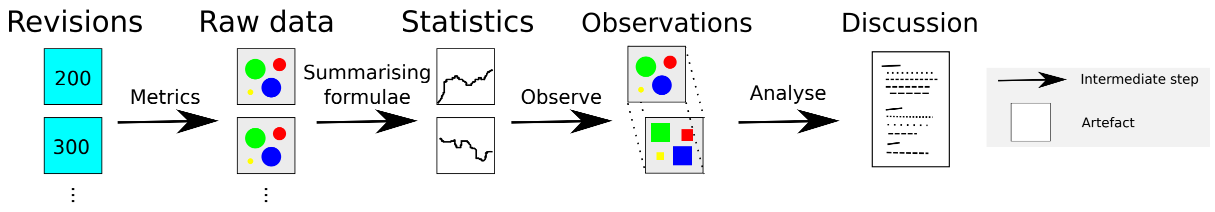


Figure 3: Research method

occurrences for every observed revision.

3.1.2 Observation and analysis

Quantitative analysis can verify that notes are used for communication within the Meta-Environment; however, it cannot be used to support or reject the hypothesis, because it does not provide any insight into the *usage* of these notes. Therefore, a qualitative analysis is performed to determine if note primitives truly achieve time decoupling. When a revision in which tools or groups of tools are time decoupled is found, then the hypothesis is supported.

3.2 Complexity cost

The introduction stated Research Question 2 as follows: "What is the complexity cost of a bus-oriented architecture?". In any architecture, most of its complexity is found within its components. Within a bus-oriented architecture, a portion of the complexity is located within its coordination script. The next paragraphs show why the complexity of such coordination script is likely to increase.

As noted in the introduction, a bus-oriented architecture claims to provide the ability to connect independent reusable components. Software reuse can significantly improve software quality and productivity [19], and as a result can be associated with reduced cost. But, building reusable components is not a free exercise. In a case study on reusing components in an industrial development project, Favaro found that as the size and complexity of reusable components increased it became harder to recoup on the initial investment of creating these components [15].

Small components are thus favoured from a reuse perspective. But cost related to connecting many small components to a software bus could be high, because each component requires additional scripting for defining its interface and for defining the information flow between itself and other components. With such an increase in the size of TScript comes additional complexity. This is an issue, as Hohpe and Woolf note, that the primary issue with content-based message routing is that the scripting becomes a frequent point of maintenance [23], and therefore a factor to take into account when calculating a project's cost.

Small reusable tools are to be expected, since reuse of Meta-Environment tools was a development goal from the beginning. If the Meta-Environment does contain tools which are mostly small, their effect on the complexity of TScript can be measured. To do so, the following hypothesis (which captures both the tool-size as well as the TScript size variable) is proposed:

Hypothesis 2. If the number of small tools connected to the ToolBus increases, the size of TScript relative to the size of TScript and tool-code combined, increases.

When a non-linear growth of the TScript percentage is observed, then the hypothesis is supported. The study only determines whether complexity cost remains relatively equal during the lifetime of the bus-oriented system. It will make no claims as to whether or not the percentage of TScript is better or worse than those found within other types of architecture, as such a claim would require the measurement of other different architectures.

3.2.1 Metrics

For the purpose of measuring the size of both TScript as well as tool-code, the non-commented lines of code (*NLOC*) metric is used. For which Conte et al. provide the following definition:

"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements." [10]

***NLOC* for complexity** TScript size is used as a measurement for complexity, even though the cyclomatic complexity metric devised by McCabe [28] is more commonly used for measuring the complexity of a software system. The cyclomatic complexity metric determines the number of paths through a computer program, with branches of these paths being located at statement such as `if` and `for`. If McCabe complexity is an acceptable complexity metric, then size is too, because several studies found that *NLOC* and cyclomatic complexity show positive correlation [35].

The correlation between size and cyclomatic complexity is likely to be strong with TScript, because almost every TScript primitive introduces a new execution path. In TScript the number of execution paths is determined not only by guard commands and conditional expressions such as `if` and `else` (Table 1), but also by composition primitives such as choice (+) and iteration ($*$), and by the pattern matching of messages, which allows for execution paths to differ depending on the content of a message. Therefore, only the lines which contain tool, process and variable declarations are not related to complexity. Given that the Meta-Environment's development team uses a common coding style for TScript, an even distribution of coordination per line of code can be assumed.

Interleaved execution As noted in Section 2.1, processes run interleaved at runtime. This in turn increases the number of execution paths, because it is unknown which coordination task is executed at what time. The ToolBus hides execution interleaving, therefore this study assumes that it does not affect the software developer's difficulty in understanding TScript.

3.2.2 Statistics

Based upon the raw size data, three statistics are calculated:

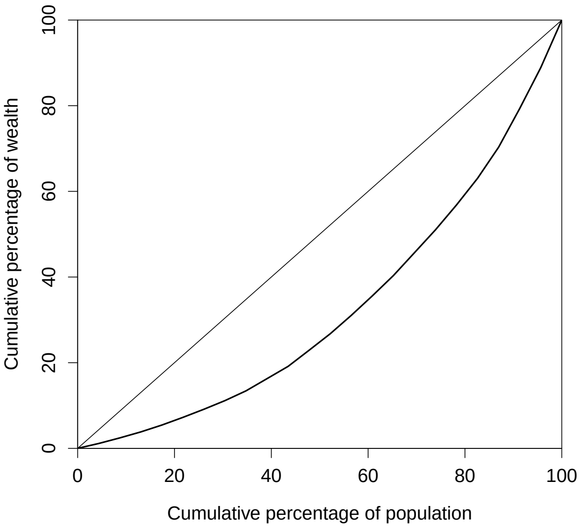


Figure 4: Lorenz curve example

- The *NLOC* per revision, per source code language.
- The percentage of TScript per revision.
- The equality of the tool size distribution per revision.

The first two statistics require no further explanation, but the last statistic does.

Tool size distribution The hypothesis requires a size distribution in which primarily small tools are connected to the ToolBus, because it is assumed that such distributions are commonly found in systems which are created with reuse in mind. The tool size distribution is interesting regardless of whether or not it exposes small tools, because different results could provide more material for discussion.

As noted in the introduction of Section 3, a bird-eye view is needed to detect interesting changes in the tool size distribution. Measures like the average and mean assume a Gaussian distribution which is uncommon for software systems [38]. In search of a statistic which does not require a priori knowledge about a distribution, Vasa et al. found that the Gini coefficient can successfully discover changes in software systems [38].

The Gini coefficient is a numeric representation of a Lorenz curve. In economics, the Lorenz curve

[27] is used to show the distribution of wealth in a subset of the world population, but it can equally be applied to other distributions. Figure 4 shows a Lorenz curve example of wealth distribution. The Lorenz curve plots on the y-axis the proportion of the distribution assumed by the bottom x% of the population [38]. The linear line represents the line of equality, because every part of the population has an equal part of the distribution ($x = y$). The non-linear line (Lorenz curve) represents the actual equality of distribution. When applied to $NLOC$, a plotted Lorenz curve would show for the bottom x% of tools, what y% of $NLOC$ they contain.

The Gini coefficient, defined by Italian statistician Corrado Gini [21], is derived from the Lorenz curve. Gini coefficient G is defined as:

$$G = \frac{A}{A + B} \tag{1}$$

where A represents the area between the line of equality and the Lorenz curve itself, and B represents the area below the Lorenz curve. The result of the equation is bound between 0 and 1, with 0 indicating perfect equality (e.g. multiset $[10, 10, 10]$) and 1 indicating complete inequality (e.g. multiset $[0, 0, 10]$).

The Gini coefficient itself is not of interest, but changes to it are, because those changes identify changes within the Meta-Environment.

3.2.3 Observation and analysis

Quantitative analysis provides a result which can be used for supporting or falsifying the hypothesis. Qualitative observation can lead to discoveries as to why the quantitative results came to be. For this purpose, the TScript files of major Meta-Environment releases are compared, to determine what effect changes which were performed between these releases had on the complexity of coordination.

3.3 Software repository mining

The Meta-Environment is analysed by mining its software repository. Software repositories include sources such as revision control systems, requirements tracking systems, bug tracking systems and communication archives [24]. For this study sources of information are the Meta-Environment's revision control system and release log. In Section 3.3.1 the revision selection method is discussed. Section 3.3.2 provides the method for identifying the software system's components and their size.

3.3.1 Revisions

The ASF+SDF Meta-Environment's Subversion revision control system repository, which is shared with other products, contains over 31000 revisions. Revision 190 is the first to contain source code of the Meta-Environment itself.

Given the set of revisions R found in the Subversion repository, this study is performed on the subset $\{R_{200}, R_{300}, R_n, \ldots, R_{31000}\}$, where n is the revision number in the Subversion revision control system. $\Delta n = 100$ is selected, to reduce the number of temporary spikes, caused by e.g. a wrong commit. The $\Delta n = 100$ still maintains adequate coverage to detect changes which affect the system for a longer duration.

3.3.2 Component structure and size

The Meta-Environment system component structure is specified in Makefiles and tool interface definitions. Makefiles define the system's programs and libraries and the source code files associated with them. As noted earlier, tools are a subset of programs. This subset is identified by matching the name of each program with the names found in tool interface definitions. If such a match exists, the program is deemed to be a tool.

The size of a component is measured in three ways, depending on the source code language:

C the $NLOC$ of C files referenced within the Makefile and the $NLOC$ of their header files are summed.

Java the $NLOC$ of all Java files is summed, because system packages composed of Java files contain at most a single tool or library. The type of the package is defined in a single Makefile found in the root of the package.

TScript the $NLOC$ of all files referenced by Makefiles is summed. TScript is associated with the package within which it resides, with the exception of tool interface definitions, which are associated with the tool for which they specify a ToolBus interface.

Measurement of component structure and size has previously been executed by Robles et al., which studied the size of software packages found in Debian Linux distributions [34]. In their study, Robles et al. calculated the *NLOC* of non-generated source code files, excluding files which were exactly the same. Their method was almost identical to the measurement of Java *NLOC* presented in this thesis. Unlike this study however, Robles et al. did not use Makefile nor file dependencies for determining which source code files were actually used within a software package.

Generated source code is also prevalent within the Meta-Environment's source repository; the source repository also contains unused copies of source code created during the transition from the CVS revision control system to Subversion. Both generated code as well as unused code copies should not be counted towards the size of components, because they do not affect the maintenance effort of the Meta-Environment. The following paragraphs describe the solution to these issues.

Generated source code In one version of the Meta-Environment, generated code makes up over 50% of all C source code [11]. Generated source code files are identified manually, because automatic recognition is deemed less reliable. To reduce the required identification effort, files which were found to be generated in one revision are deemed to be generated in all following revisions, i.e. if a file is found to be generated in R_{5000} it is assumed that it remains generated during the life-time of the system.

Unused source code The use of Makefiles for component structure detection ensures that source code which is no longer in use does not affect a component's size. When no longer in use, either the directory which contains these source code files is no longer referenced by the Makefile in the parent directory, or the Makefile itself does no longer reference these files.

4 Results

The following subsections provide the statistics and observations (as shown in the research method, Figure 3) for both hypotheses. Section 4.1 provides results concerning Hypothesis 1. Thereafter, Section 4.2 and Section 4.3 provide results concerning Hypothesis 2. A discussion of the results in provided in Section 5.

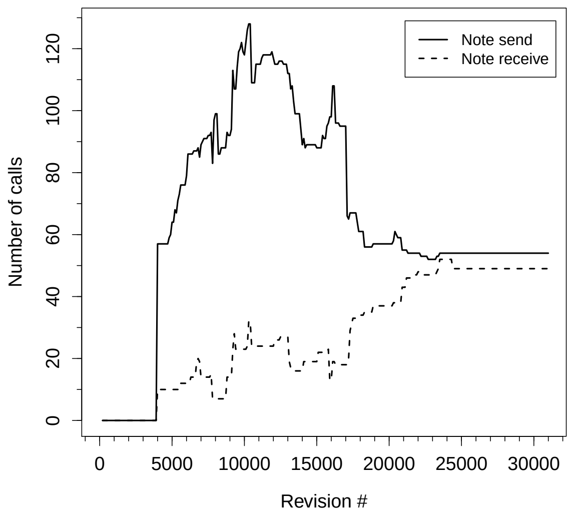


Figure 5: Number of snd-note(T) and rec-note(T) occurrences. For R_{200} to R_{31000}, $interval = 100$.

4.1 Time decoupling

Figure 5 presents the results on the amount of note communication found in the Meta-Environment. Notes are used before R_{4200}, but not visible in the measurement, because the opposite communication ends are not yet referenced by Makefiles. Findings indicate two primary usages (scenarios) of notes, which are discussed in the next paragraphs.

From R_{4200} to R_{17000} notes are far more sent than received, indicating that a receiving endpoint is called from multiple sending endpoints. Analysis of R_{10000} shows that the `ui-status` ATerm is sent from 99 locations to 7 receiving locations, accounting for 75% of all note communication. The `ui-status` ATerm is used to provide the Meta-Environment's user interface with information on the status of executing tools. Each of the 7 receiving locations is aimed at receiving specific text-messages, e.g. for errors, ordinary text-messages, etc.

From R_{17100} onward a significant rise and fall of receive and send note calls respectively is observed. This change coincides with the introduction of a *module-manager* tool, which was added as part of an architectural change during development of the Meta-Environment 2.0 version. The module-manager is further described in Section 4.1.3.

Scenarios in depth As part of the qualitative analysis the following paragraphs describe note usage scenarios which were abstracted from concrete usage scenarios found within the Meta-Environment's TScript. These descriptions and their implications are further discussed in Section 5. To aid in the understanding of these descriptions Figure 6 provides a visual representation. The subfigures use a UML2 Sequence Diagram notation, with the following additional semantics:

- Arrows represent both ends of communication between entities, thus `msg(T)` represents a `snd-msg(T)` primitive on the sending side and a `rec-msg(T)` primitive on the receiving side.
- If the content of a message is not important for understanding the diagram, `T` is used.

4.1.1 Basic coordination

The basic coordination scenario shown in Figure 6a is the fundamental scenario found within all versions of the Meta-Environment. The scenarios presented in Section 4.1.2 and Section 4.1.3, both use this scenario.

Two tools T_x and T_y communicate using their interface processes (P_x and P_y). The event published by T_x is communicated using various toolbus-messages, P_n sends a note to P_m which continues communication to P_y via toolbus-messages. Within the sequence diagram, P_n and P_m represent a variable number of processes, such that it is also possible for P_x and P_y to directly communicate using a note.

When the set of communication paths between P_x and P_y contains both paths which are and paths which are not decoupled, then these processes and their associated tools are only partly time decoupled.

4.1.2 Concurrent return coordination

The concurrent return coordination scenario is primarily used to provide the user interface with updated information about the state of ASF+SDF modules and operations performed on them. The coordination scenario is concurrent in the sense that information is send to the user interface whilst computation initiated by the user interface tool is still taking place inside other tools.

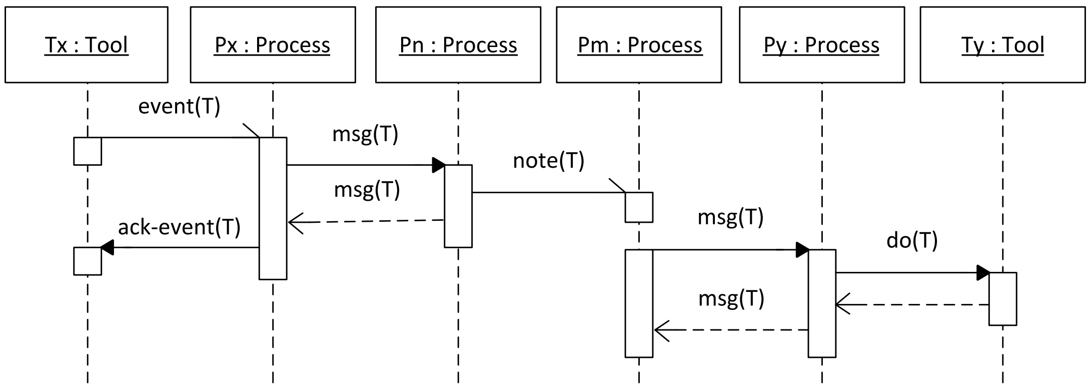
There are two slightly different implementations, of which the first is presented in Figure 6b. It depicts a single tool T_x which communicates using its interface process P_x. On receiving an event from T_x, P_x coordinates with a number of other processes (represented by P_n) and tools. When a tool associated with P_n completes a portion of its computation, it sends a toolbus-message to P_y which in turn sends a note to P_x. This implementation is for instance used to provide an updated ASF+SDF module dependency graph to the user interface.

The second implementation is slightly different. It is presented in Figure 6c. Within it P_y becomes the second tool interface process of T_x. P_n communicates with P_y using a note, after which P_y directly communicates with T_x. This implementation is primarily used for sending `ui-status` ATerms to the user interface, using the `Status-display` process.

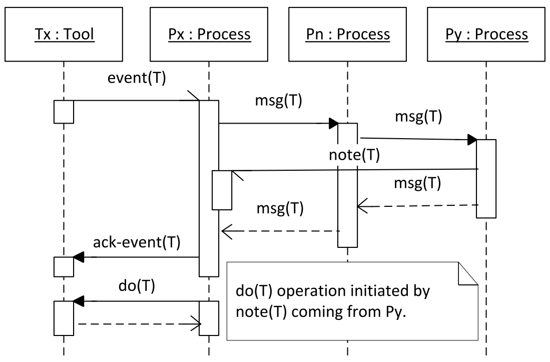
4.1.3 Module-manager

At R_{17100} the *module-manager* tool is introduced. One of the main responsibilities of the module-manager is to propagate state changes across ASF+SDF module dependencies, i.e. module A is reprocessed if module A depends upon B and B is reprocessed. For this purpose the module-manager stores the state (e.g. "unavailable", "editable", etc.) of ASF+SDF modules and provides events to *listener* processes when their state changes. These listener processes coordinate with tools to perform computation.

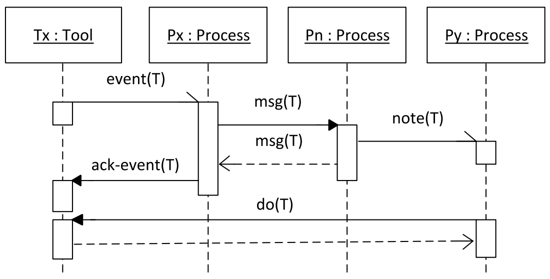
The scenario is presented in Figure 6d. A listener process, such as $P_{listener}$ subscribes to, and waits for notes to be published by the module-manager tool T_{mm}, via its process P_{mm}. The listener process acts as a façade in front of utility processes such as P_{util}. These utility processes implement the 'business process'. They call upon multiple tools to perform computation. After P_{util} has finished its business process, it notifies the module-manager of the new state, by changing an attribute located in the module-manager. After which T_{mm} triggers another `attribute-changed` event.



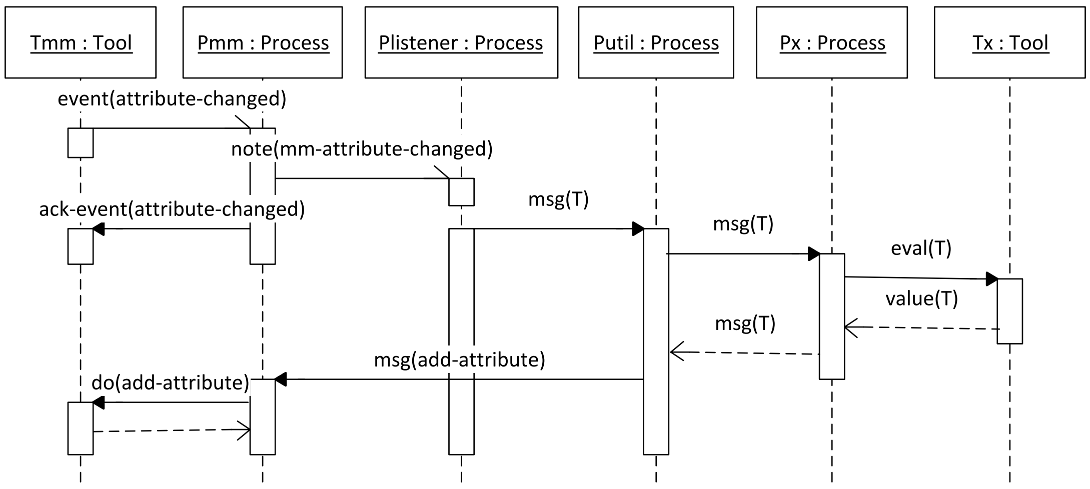
(a) Basic



(b) Concurrent #1



(c) Concurrent #2



(d) Module-manager

Figure 6: UML2 sequence diagrams of coordination scenarios.

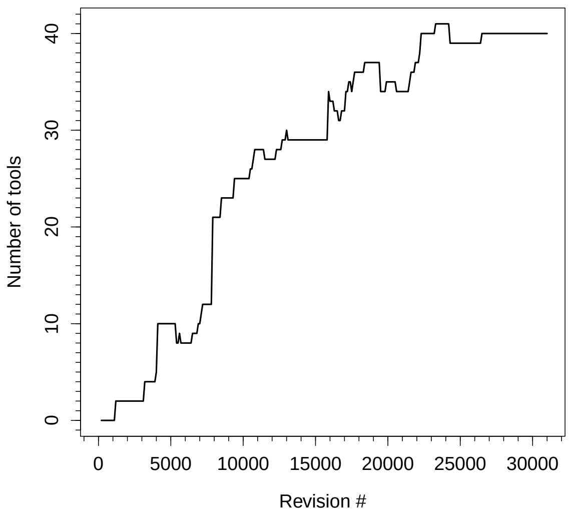


Figure 7: Number of tools. For R_{200} to R_{31000}, $interval = 100$.

4.2 Tool-size distribution

Figure 7 shows that the number of tools increases as the Meta-Environment evolves. Figure 8 presents the results of tool-size distribution measurement for four revisions, These distributions show that as the system evolves the distribution moves towards tools of mostly $NLOC < 1000$. Figure 9 confirms that the equality of the tool-size distribution remains quite constant from R_{8000} on out. This indicates that at no time during the system's evolution it contained a very different tool-size distribution than the ones shown in Figure 8 (with the exception of the top-left distribution). The listing below sheds light on the distribution changes indicated by the changing Gini coefficient.

$< R_{1400}$ Contains less than two tools. The resulting perfect equality is not depicted.

R_{1400} **to** R_{8000} Initial system evolution is chaotic. This is caused primarily by the small number of tools and their frequent change in size. Between R_{3200} and R_{4100}, a lot of new tools are added. Resulting in the distribution as shown at the top-left of Figure 8. The distribution of R_{8000} is shown at the top-right of Figure 8.

R_{8100} **to** R_{15800} The Gini coefficient slowly de-

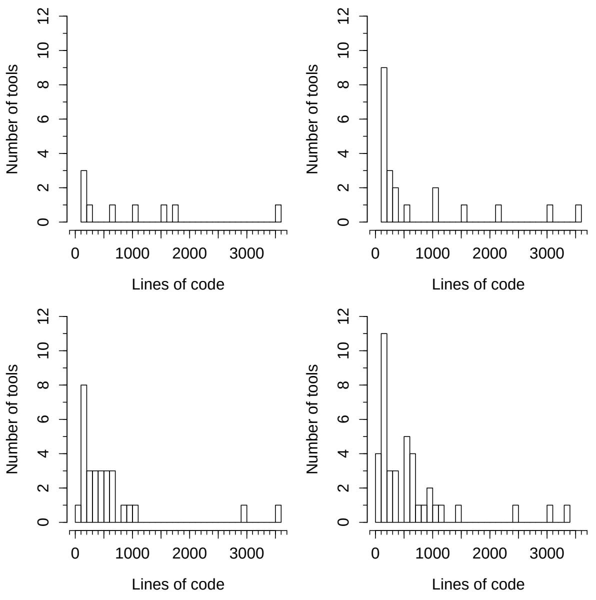


Figure 8: Tool-size distribution. For R_{4100}, R_{8000} (top row) and R_{15800}, R_{27500} (bottom row).

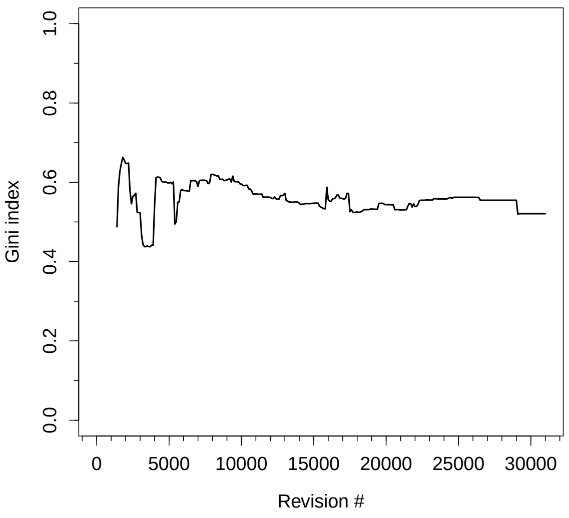


Figure 9: Gini coefficient of tool-size distribution. For R_{1400} to R_{31000}, $interval = 100$.

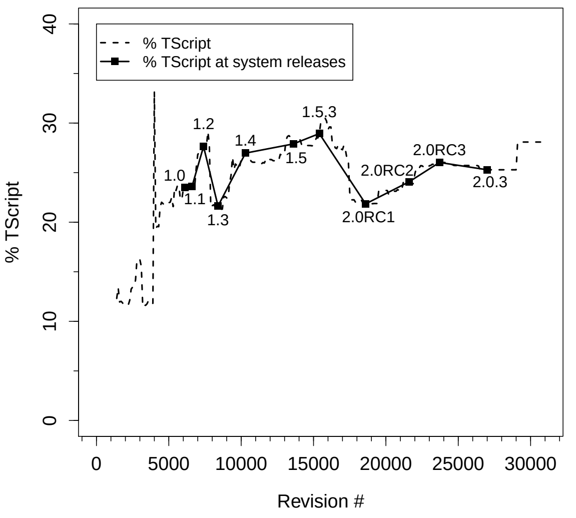


Figure 10: % TScript relative to size of TScript and tool-code combined. For R_{1400} to R_{31000}, $interval = 100$.

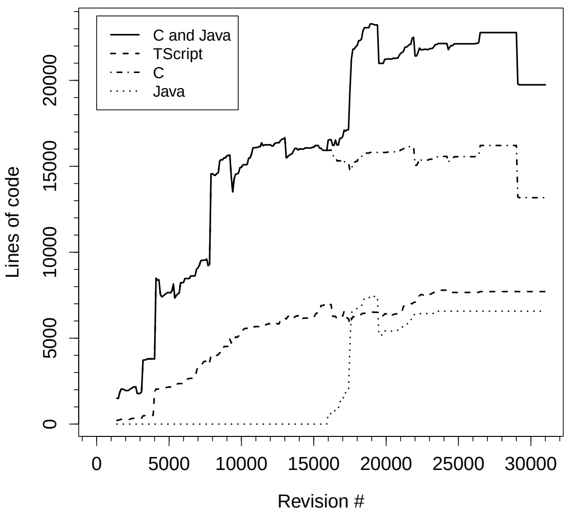


Figure 11: Lines of TScript, C and Java associated with tools. For R_{1400} to R_{31000}, $interval = 100$.

clines as the tool-size distribution becomes more equal (Figure 8, bottom-left). This is primarily caused by an increase of tools within the 100 to 1300 lines of code range, while no significant modifications are made to tools over 2000 lines of code.

R_{15900} **to** R_{17800} The Gini coefficient changes significantly, as various Java Meta-Environment user interface plugin tools are introduced.

R_{17900} **to** R_{29000} During this period the distribution remains fairly stable. The number of small tools increases, but this increase in equality is negated by the editor-plugin tool, which was introduced at R_{17600} and has slowly grown to 2500 lines of code (Figure 8, bottom-right).

$R_{29100} <$ A sudden decline, caused by the movement of code associated with a tool to a library.

4.3 Percentage of TScript

Figure 10 presents the percentage of TScript found in tools, compared to the total size of TScript and tool-code combined. Figure 11 presents the individual size of TScript and tool-code. Given that the tool-size distribution for $< R_{1400}$ could not be calculated, these and any remaining figures start at R_{1400}.

The TScript percentage peak at R_{4000} is caused by Makefiles referencing TScript before referencing associated C source code.

The sharp increases in C and Java *NLOC* occur for either of two reasons: (1) packages which were under development, but not previously referenced by the Meta-Environment, were added as a reference; or (2) Makefiles start referencing previously existing subdirectories, which contain large quantities of source code. TScript size is hardly affected by these referencing issues, because most TScript is located within the *meta* package, which is included from R_{190} onward. Measuring the size of previously unreferenced packages and source code files at an earlier time to reduce sharp increases is not possible, because Makefile identification of component types is needed for determining whether or not a source code file is part of a tool (see Section 3.3.2).

In Figure 10, the Meta-Environment's releases are labelled. The listing below provides an observation of changes taking place between these releases. It provides additional information for use within the discussion.

1.0 to 1.2 The growth in TScript is primarily attributable to the addition of processes to display ASF+SDF evaluator errors and the splitting and extension of coordination related to the modification of ASF+SDF modules. The splitting is comparable to the *extract method* refactoring [18] in that a single large process is refactored into a process which calls upon other smaller processes to perform the same functionality.

1.2 to 1.3 The Meta-Environment aims to provide an open environment for creating term rewriting engines [37]. For this purpose the development team added *hooks* [37], which are messages that are sent from the generic part of the Meta-Environment to a specific rewriting formalism (such as ASF). Hooks add another layer of coordination, thus causing an increase the size of TScript.

Further changes include (1) the removal of duplicate TScript by creating utility processes which are called by multiple other processes; and (2) further extension of functionality related to ASF+SDF modules.

1.3 to 1.4 Flexibility of the Meta-Environment is increased by extracting the storage of ASF+SDF module information found in the *module-db* tool into a separate *term-store* tool. This results in a net increase in TScript, because the old process for handling ASF+SDF module information is maintained to act as a wrapper for the *term-store* processes. The measured drop in C *NLOC* at R_{9300} is caused by the removal of the module-db tool.

Another major refactoring increases the flexibility of the Meta-Environment's user interface button actions. Whereas these actions were previously hard-coded into the user interface they can now be modified within the system's TScript.

1.4 to 1.5.3 Achievement of more flexibility was not a goal for these releases. The replacement of the Meta-Environment's Emacs text editor by a Java Swing implementation causes some TScript to be added. Additional functionality in the user interface which provides more errors and warnings to the user causes additional growth. The decrease of C *NLOC* at revision R_{13100} is attributable to the refactoring of the *structure-editor* tool, which implements user interaction with syntax trees.

1.5.3 to 2.0RC1 Various changes have a large impact on the Meta-Environment's architecture. At R_{17100} the *module-manager* tool is introduced.

The large increase in Java *NLOC* at R_{17500} is caused by the replacement of the user interface, previously programmed in Tcl/Tk, with various new Java SWING interface tools.

2.0RC1 to 2.0.3 Further 2.0 releases aim to transfer the functionality provided in version 1.5.3 to the new architecture. In addition, some new functionality is added. TScript growth at this point is primarily located within the utility processes, in which the 'business process' is further specified (as noted in Section 4.1.3).

5 Discussion

The discussion is split into four sections. Section 5.1 and Section 5.2 each provide a discussion on results for the two hypotheses individually. Afterwards, Section 5.3 provides a discussion on the transaction handling within the Meta-Environment. Section 5.4 discusses the validity of the results.

5.1 Time decoupling

The note usage results displayed in Figure 5 indicate that notes are used. Qualitative analysis presented in Section 4.1.1, shows that the basic scenario can provide time decoupling between processes and tools. The concurrent return scenario presented in Section 4.1.2, shows that use of such a basic scenario does not always lead to true time decoupling. The attempt to time decouple the user interface leads to a situation in which information which is meant for display to the user may be stored until a later time, when the usefulness of this information has long surpassed. The module-manager scenario presented in Section 4.1.3 shows true time decoupling, albeit not at the level of individual tools. Therefore, Hypothesis 1 is accepted.

Conclusion 1. Achievement of time decoupling within a bus-oriented architecture is possible.

The following section discusses how and why the development team used notes the way they did.

5.1.1 Note usage

Analysis of the Meta-Environment made clear that the use of notes is in no way meant to handle situations in which a tool is unavailable and would need to be invoked at a later time. This development choice is legitimate, because the Meta-Environment tools run on a single computer system, which makes the assumption that all tools are available a valid one.

Instead, the Meta-Environment uses notes to provide time decoupling for dealing with *fire and forget* situations. In these situations, tasks performed by tools can be executed concurrently, without blocking the computation of other tools, like for example the user interface. The `Status-display` process discussed in Section 4.1.2 provides a nice example, and there are other similar observed cases in which a basic coordination scenario is found. Such time decoupling is mostly unidirectional, which means that a tool T_x is not time coupled to other tools, but the other tools are time coupled to tool T_x.

The `Status-display` process is fully decoupled in time, because it receives notes exclusively, and no outward communication is observed. Of no other observed process can be said with absolute certainty that it is decoupled from tools other than its own, because other tool interface processes were found to send or receive at least a single toolbus-message. A fully time decoupled tool interface process implies a fully time decoupled tool, as long as no other time *coupled* tool interface processes interact with the tool and the tool communicates exclusively through the ToolBus. The `Status-display` process is not the only process which communicates with its tool, but still does present a case in which full time decoupling of a single tool is within reach.

The module-manager scenario changed the time decoupling of tools. The continuous triggering of listeners with notes that indicate an ASF+SDF module state change, and the subsequent modification of the the module-manager's state by utility processes, time decouples groups of tools. It allows a listener to pull new work from its note queue at any time, without being time coupled to any other listeners or tools not associated with the task at hand.

5.2 Complexity cost

In Section 4.2, the tool distributions shown in Figure 8 shows that the Meta-Environment contains mostly small tools. Figure 7 shows that the number of tools increases as the Meta-Environment evolves. These two results form the premise for Hypothesis 2. The results presented in Section 4.3 indicate that the size of TScript relative to the size of TScript and tool-code combined, does not increase in an exponential or cubic fashion over time. Hypothesis 2 indicated that such a relative increase was to be expected. Given the results the hypothesis is falsified.

Conclusion 2. The complexity cost of coordination scripting shows a close to linear growth relative to the growth of components which are coordinated.

In the introduction to Hypothesis 2, provided in Section 3.2, it was noted that every additional tool likely requires additional scripting for defining the information flow between itself and other tools. The Meta-Environment team has surmounted this obstacle by designing well defined paths of interconnection within TScript. This limits the possible interconnections and, therefore, the number of lines of TScript needed to provide such interconnection.

A good example is the module-manager, which acts as a Mediator [20] to reduce the number of interconnections between listener processes. It reduces the number of possible interconnections, because each additional listener processes adds only a single dependency between itself and the module-manager, instead of dependencies between itself and all other listener processes.

Another example would be the handling of button click actions, which are coordinated through various standard button handling processes until veering of to the process that performs the associated computation.

5.2.1 Language complexity

TScript is a complex language to work with. Its primitives all provide some sort of decision point, in terms of the execution path, the type of composition, etc. The `if-else` primitives and composition primitives such as `+` and `*`, are all relatively simple compared to the pattern matching of messages which takes place in `snd-msg/note` and `rec-msg/note` primitives.

The development team developed the Meta-Environment without using a TScript development environment, but such tooling could be beneficial to the development process. The lack of tooling was found to be a disadvantage whilst performing this study, because the interaction amongst processes and tools had to be observed by searching through text files, which proved to be time consuming and error prone. Similarly, developers of the Meta-Environment find the undisciplined message pattern hard to cope with [11]. Tooling could dynamically infer which ATerms can match to which other ATerms at runtime, and using this information provide the user with capabilities like (1) navigating from the message sending and receiving location to the other endpoint(s), and (2) creating process dependency graphs. Such advantages, which are often found with statically typed languages are useful for program design, maintenance and understanding [6].

5.3 Coarse-grained operations

The Meta-Environment consists of many fine-grained tool operations which are combined to perform some useful activity. These fine-grained operations share data dependencies with each other, which makes it difficult to achieve successful time decoupling, because tools depend too much upon the state within other tools.

When the use of notes became a more dominant choice within the Meta-Environment, developers noticed that a lot of race conditions took place. An example can be found in the user interface's text editor. Based upon the text content of this editor, various computational actions (such as checking the syntax of the text) are performed, whilst the user is still typing. The computational result is then send back to the user interface, which displays it to the user. The sequence in which these results are provided was not predetermined, causing results which were no longer accurate to be displayed to the user.

Possible solutions to this problem are to add a transaction mechanism or lock data to remove race conditions. Such a transaction management mechanism was therefore added to the Meta-Environment in the form of a `transaction-manager` tool. Manual management of transactions adds complexity to the system, as every location at which race conditions can take place needs to be identified and encapsulated within a transaction. A solution, which reduces this complexity and therefore makes the achievement of time decoupling easier, is to create more coarse-grained operations, such that the operations themselves become an implicit transaction boundary. Of course not all race conditions can be captured by creating coarse-grained operations, therefore a transaction management feature is still needed. The lack of a transaction facility within TScript is a big loss, as this requires the creation of additional tools, whilst clearly such a standard facility is needed for notes to be a usable feature.

There is a drawback to the creation of coarse-grained tool operations. Functionality which was previously shared between multiple tools, programmed in multiple languages, is now hidden inside a single tool. Therefore, such functionality can no longer be reused by tools written in different programming languages or for different platforms, and thus part of the system's heterogeneity is lost.

5.4 Threats to validity

Threats to validity concern issues which affect claims made in this thesis.

5.4.1 Construct validity

Construct validity concerns the question whether or not results of a study are an indication for the theoretical concept which is supposed to be measured.

This thesis uses complexity as a concept that defines the ease of understanding (part of) a system. Human understanding is still being researched, and no definitive methods for measuring understanding of software systems are available. This study works with the assumption that that the size of a system correlates with the time and effort required to understand such system.

5.4.2 Internal validity

Internal validity concerns issues with the inference of cause and effect in the study's analysis.

Revision selection In Section 3.3.1, the limitation of this study to every 100th revision was presented. Such a limitation could negatively affect the correctness of this study. Every 100th revision was selected to reduce the number of temporary anomalies, effectively increasing the range of revisions which can be considered within a given time duration. Decreasing the interval would result in a decrease in the range of observable revisions, i.e. instead of $\{R_{200}, R_{300}, R_n, \ldots, R_{31000}\}$ one could consider $\{R_{9000}, R_{9020}, R_n, \ldots, R_{15200}\}$. If this study were performed on such a limited subset, then despite the smaller interval the inference of this study would be incorrect, because the percentage of TScript would seem to grow as the number of small tools increases (as displayed in Figure 10). The downside of restricting research to every 100th revision is not that it decreases the validity of claims, but that these claims possibly could be further strengthened by examples which are located in unobserved revisions.

Message type inference Time decoupling of processes and tools was manually observed for a number of cases. The method of manually observing and tracing communication across processes increases the risk of not observing a single communication path which introduces time coupling between two system elements. Similarly, manual observation of TScript structure does not reveal all cases which are not well structured, increasing the risk that the whole TScript is deemed to be well structured, whilst in fact only parts are well structured.

A message type inference technique, which can be used to create dependency graphs (as suggested in Section 5.2.1), can increase result reliability. Although the ToolBus uses such an inference technique at runtime, it was not used for purposes of this study, because the deployment of every revision in the set of studied revisions was deemed difficult, due to continuous changes to the system's deployment configuration and C compiler.

Language expressiveness In calculating the percentage of TScript there has been made no attempt to differentiate between C code and Java code, even though McConnell, when combining results of various studies, found the expressiveness of Java to be 2.5 times higher than C [29]. But not every study provides equal results, as a study by Prechelt found that programmers who are assigned a programming task in C and Java produced programs with similar *NLOC* [31].

No differentiation is applied, because it is likely that the expressiveness of a language differs depending on the domain to which it is applied, to such an extend that the same language has different factors of expressiveness within the same system. This would require further qualitative analysis of all functionality found within the Meta-Environment, to come up with some factor whos reliability can be disputed as well.

Combined result synthesis Even though this study had no intention of synthesising the subject of time decoupling and complexity, it is useful to briefly determine the internal validity of the possible relation between them. Given the results and earlier discussion it is not possible to state that the application of time decoupling has an effect (positive or negative) on the complexity cost. The effect of TScript structuring as discussed in Section 5.2 cannot be separated from the effect of time decoupling, because the action of time decoupling and restructuring were not separated in the system's evolution, causing temporal precedence to be unobservable.

5.4.3 External validity

External validity concerns issues with the generalisation inferences of this study. Conclusions for which external validity is discussed include Conclusion 1 and Conclusion 2.

Distributed systems The non-distributed nature of the Meta-Environment could negatively affect this study's generalisation to distributed systems. The ToolBus library contains all functionality needed to work with remote tools. The locality of any tool is abstracted away from the processes that represent their interface. This makes it possible to deploy parts of the Meta-Environment onto multiple systems, without affecting the ToolBus or the Meta-Environment's functionality. Therefore, this study is representative for distributed systems.

Time decoupling The ToolBus is a software bus which (1) contains an intermediary component which takes part in the communication process, and (2) contains message queueing capabilities. These characteristics were found crucial for achievement of time decoupling. Therefore, the time decoupling claim of this study is only representative for architectures which contain a software bus that share these characteristics.

Within the WS-Notification specification family, the WS-BrokeredNotification specification defines an intermediary component called the NotificationBroker [8]. If the implementation of such a NotificationBroker includes a message queue, then the time decoupling claim of this study can also be applied to it.

Part of the use of time decoupling is for reliability purposes, because it enables components to be unavailable for a period of time. Using time decoupling for reliability purposes requires that (1) the software bus does not require components (tools) to be available for their message subscriptions to remain active, and (2) the software bus does provide a reliable message queue, such that messages are not lost after a period of time or a software bus restart. The first requirement is achieved within the ToolBus, but the second is not. Also, the Meta-Environment was not developed with reliability in mind, because it is not a distributed system. Therefore, this study is not representative for the use of time decoupling for reliability purposes.

Triple Space is comparable to the combination of a software bus and module-manager. Therefore, it can be used to provide time decoupling. Given that Triple Space does not require components to be available for subscriptions to remain active[2], it can also be used for reliable time decoupling.

Complexity cost TScript is a centralised scripting language, which defines coordination based upon message content (called content-based routing) using process-oriented constructs.

Not all software bus coordination scripting languages are centralised. Some software buses are no more than data transmission and subscription holding facilities, which allow components to subscribe to messages by their topic, content or type [14]. Their complexity characteristics are different, because these type of software buses scatter the messaging and subscription logic across the components which are connected to it. Such scattering also occurs when the coordination scripting language does not feature process-oriented constructs, because such process-oriented constructs are meant to replace part of the component complexity, thereby separating coordination and computation.

The knowledge and experience of developers affects the cost complexity of the system. Developer working on the Meta-Environment are academics which have at least a masters degree in a computer science related field. Most developers are either working towards or have obtained a Ph.D.

Given the above paragraphs, the generalisability of the complexity cost claim is limited to systems which share characteristics with the Meta-Environment and the ToolBus. This specifically includes systems which feature a software bus that applies content-based routing and provides process-oriented constructs, and systems that are developed by developers with a high level of knowledge and experience.

Within the WS-Notification specification family, the WS-BaseNotification specification defines content-based filters [7]. Using these filters, subscribing components can limit the events for which they receive notifications, based upon the content of such events. Another specification which deals which such filtering is the WS-Topics specification, which includes topic-based filtering [9] comparable to ATerm identifiers (Section 2.1.1). When combined both specifications provide content-based routing characteristics similar to that of TScript. They do not, however, provide any process-oriented constructs. Therefore, the complexity cost claim cannot be generalised to these two specifications.

The tooling support for the development of a coordination language might affect its complexity. Such tooling support, which was discussed in Section 5.2.1, could increase the understanding of the coordination of interaction. Tooling support for TScript is limited. It is only provided in the form of an application which shows messaging as it occurs, but it cannot generate a dependency graph. One can speculate that by providing better tooling developers with more limited knowledge and expe-

[2]As indicated in the WSMX Triple-Space Computing specification, retrieved on the 19th of August, 2010 and located at http://www.wsmo.org/TR/d21/v0.1/.

rience can achieve similar complexity cost.

6 Conclusion

A case study was performed on the Meta-Environment, which features a bus-oriented architecture based upon a content-based message routing software bus. The achievement of time decoupling and the complexity cost of coordination were observed. Time decoupling enables two components to participate in an interaction without both being available at the same time.

Results indicate that time decoupling is achievable within bus-oriented architectures and service-oriented architectures, but that it is difficult to time decouple a single component. Within the Meta-Environment, time decoupling only occurs between groups of components. A factor in such grouping is found to be the component design, which contains many fine-grained operations that share a lot of data dependencies, which in turn makes time decoupling of individual component more complex. Successful and less complex time decoupling of components requires that (1) more coarse-grained operations are specified, such that data dependencies are located within tools themselves, and (2) a transaction facility is provided for cases in which such data dependencies cannot be moved within components.

The primary complexity cost of a bus-oriented architecture is assumed to be located in the scripting that defines the coordination amongst its connected components. Results indicate that, in systems that share characteristics with the Meta-Environment, the complexity cost of coordination scripting shows a close to linear growth relative to the growth of components which are coordinated. The result is attributable to the structure which developers created in the coordination script. The structure is set up such that the number of possible paths of interaction between components are reduced, which in effect reduces the possible complexity.

6.1 Future work

This case study only examined a single system, reducing the external validity of conclusions and leaving many questions open for future research. As future work, other bus-oriented architectures and service-oriented architectures can be examined. This can also provide more information on the possible connection between granularity of operations and system heterogeneity.

The development of an application which provides a dependency graph depicting the communication amongst processes and tools could greatly strengthen the claims on time decoupling and complexity cost. It would also ease further development of systems that utilise the ToolBus, because it would provide better insight into the coupling of such systems.

Acknowledgements

I would like to thank members of the SEN1 group at Centrum Wiskunde & Informatica for providing me with insight into the Meta-Environment and ToolBus. A big thanks to Jurgen Vinju and Hans Dekkers who provided valuable feedback on my thesis. Furthermore, I would like to thank the other students at Centrum Wiskunde & Informatica: Jelle Geelhoed, Nico Schoenmaker and Waruzjan Shahbazian, for providing feedback, participating in discussions and pulling me through highs and lows experienced during this project.

References

[1] L. Aldred, W. M. P. van der Aalst, M. Dumas, and A. ter Hofstede. Understanding the challenges in getting together: The semantics of decoupling in middleware. Technical report, Business Process Management Center, 2006.

[2] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36(11):81–94, 1993.

[3] J. A. Bergstra and P. Klint. The toolbus coordination architecture. In *COORDINATION '96: Proceedings of the First International Conference on Coordination Languages and Models*, pages 75–88, London, UK, 1996. Springer-Verlag.

[4] J. A. Bergstra and P. Klint. The discrete time toolbus—a software coordination archi-

tecture. *Sci. Comput. Program.*, 31(2-3):205–229, 1998.

[5] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[6] G. Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

[7] OASIS International Standards Consortium. *Web Services Base Notification 1.3 (WS-BaseNotification).* October 2006.

[8] OASIS International Standards Consortium. *Web Services Brokered Notification 1.3 (WS-BrokeredNotification).* October 2006.

[9] OASIS International Standards Consortium. *Web Services Topics 1.3 (WS-Topics).* October 2006.

[10] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software engineering metrics and models.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986.

[11] H. A. de Jong and P. Klint. Toolbus: The next generation. In *FMCO 2002, Revised Lectures*, pages 220–241, 2003.

[12] F. DeRemer and H. Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM.

[13] T. S. Dillon, C. Wu, and E. Chang. Reference architectural styles for service-oriented computing. In *NPC'07: Proceedings of the 2007 IFIP international conference on Network and parallel computing*, pages 543–555, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] P. Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2): 114–131, 2003.

[15] J. Favaro. What price reusability? a case study. *Ada Lett.*, XI(3):115–124, 1991.

[16] D. Fensel. Triple-space computing: Semantic web services based on persistent publication of information. In *In IFIP Int'l Conf. on Intelligence in Communication Systems*, pages 43–53. Springer-Verlag, 2004.

[17] W. J. Fokkink, P. Klint, B. Lisser, and Y. S. Usenko. Automated Translation And Analysis Of A ToolBus Script For Auctions. In F. Arbab and M. Sirjani, editors, *Proceedings of IPM International Symposium on Fundamentals of Software Engineering 2009 (3).* Springer, 2009.

[18] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, July 1999.

[19] W. Frakes and C. Terry. Software reuse: metrics and models. *ACM Comput. Surv.*, 28(2): 415–435, 1996.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[21] C. Gini. Variabilità e mutabilità: contributo allo studio delle distribuzioni e delle relazioni statistiche. *Facoltá di Giurisprudenza della R. Universitá dei Cagliari*, anno III(parte 2a), 1912.

[22] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[23] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[24] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, 2007.

[25] P. Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, 1993.

[26] H. F. Li and W. K. Cheung. An empirical study of software metrics. *IEEE Trans. Softw. Eng.*, 13(6):697–708, 1987.

[27] M. O. Lorenz. Methods of measuring the concentration of wealth. *Publications of the American Statistical Association*, 9:209–219, 1905.

[28] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[29] S. McConnell. *Code Complete, Second Edition.* Microsoft Press, Redmond, WA, USA, 2004.

[30] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 58–68, New York, NY, USA, 1993. ACM.

[31] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10): 23–29, 2000. ISSN 0018-9162.

[32] J. M. Purtilo. The polylith software bus. *ACM Trans. Program. Lang. Syst.*, 16(1):151–174, 1994.

[33] D. A. C. Quartel, L. F. Pires, M. J. van Sinderen, H. M. Franken, and C. A. Vissers. On the role of basic design concepts in behaviour structuring. *Comput. Netw. ISDN Syst.*, 29 (4):413–436, 1997.

[34] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor. Mining large software compilations over time: another perspective of software evolution. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 3–9, New York, NY, USA, 2006. ACM.

[35] M. Shepperd and D. C. Ince. A critique of three metrics. *Journal of Systems and Software*, 26(3):197 – 210, 1994.

[36] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, 2000.

[37] M. G. J. van den Brand, P.-E. Moreau, and J. J. Vinju. Environments for term rewriting engines for free! In *RTA'03: Proceedings of the 14th international conference on Rewriting techniques and applications*, pages 424–435, Berlin, Heidelberg, 2003. Springer-Verlag.

[38] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the gini coefficient. In *ICSM*, pages 179–188, 2009.