TABASCO: a Taxonomy-based Domain Engineering Method

LOEK CLEOPHAS
Technische Universiteit Eindhoven
and
BRUCE W. WATSON, DERRICK G. KOURIE, ANDREW BOAKE
University of Pretoria

We discuss TABASCO, a method for constructing *Domain-Specific Toolkits* (DSTs). We present TABASCO in the context of domain engineering and generative programming. We discuss the steps of TABASCO in detail, with a focus on the software construction side, giving examples based on actual applications of the method. In doing so, we show that TABASCO is a domain engineering method aimed at a particular kind of software domain, and how this method is applied in practice.

---

1. INTRODUCTION

In this paper, we discuss TABASCO, a method for constructing *Domain-Specific Toolkits* (DSTs). We show that TABASCO is a domain engineering method (in which generative programming techniques may be used) aimed at a particular kind of software domain, and show how this method can be applied in practice.

Domain engineering is often applied to large business software domains, but in TABASCO we apply it on a more restricted scale: to a family of closely related algorithms or data structures. This entails less risk, but still has a considerable number of advantages. This paper considers the steps of TABASCO and shows how they can be seen as domain engineering activities.

The most important step is the construction of a taxonomy of the algorithms from a particular domain. This makes the commonalities and variabilities between the algorithms explicit and provides correctness arguments for the various algorithms. Not only does this help in understanding the domain and the relationships that are part of it, but it also makes domain design (resulting in a toolkit design) more straightforward.

Once a toolkit has been designed and implemented based on the taxonomy structure, a *Domain-Specific Language* (DSL) may be developed. This language provides a mapping from user requirements to part(s) of the toolkit satisfying these requirements, thus simplifying toolkit use.

We discuss domain engineering and generative programming in Sections 2 and 3. Section 4 presents the TABASCO method. Following this overview, we discuss the taxonomy construction step of TABASCO in Section 5. In Sections 6 and 7 we discuss toolkit design, and DSL design and implementation in more detail. An overview of related work, including other applications of TABASCO, can be found in Section 8. Section 9 gives some conclusions and remarks as well as directions for future work.

2. DOMAIN ENGINEERING

Most current software engineering methods focus on the development of single software systems, i.e. they are *application engineering* methods. As a result, they are not well-suited for the development of reusable software.
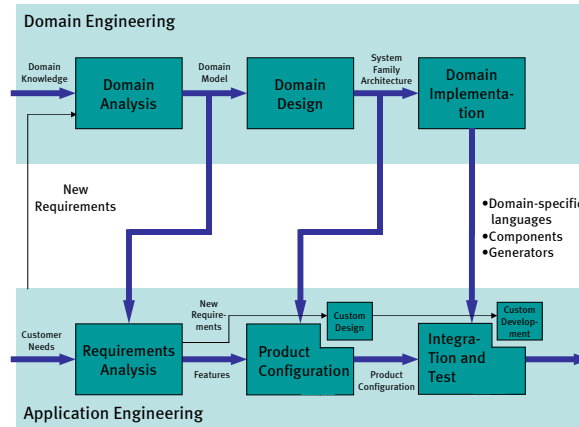

Figure 1: Domain engineering based software development [Czarnecki and Eisenecker 2000; Software Engineering Institute 1997]

In contrast, *domain engineering* emphasizes software reuse. It focuses on the development of generic software, from which concrete systems can be instantiated or parts of which can be reused in different systems. We consider domain engineering as in [Czarnecki and Eisenecker 2000; Software Engineering Institute 1997]. According to the former,

> Domain Engineering is the activity of collecting, organizing and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means for reusing these assets when building new systems.

The main idea is to capture domain knowledge in the form of a domain model and reusable assets (including software components) and to have a means for reusing them, to reduce production cost and increase quality.

Domain engineering is a three-step process consisting of *domain analysis*, *domain design* and *domain implementation*. The results—reusable assets and ways of reusing them—are used as part of traditional *application engineering*. Figure 1 depicts domain engineering and application engineering and their interactions.

The domain analysis step consists of selecting and defining a domain (*domain scoping*), collecting all relevant information from it and building a model (*domain modeling*). Such a *domain model* is based on an analysis of commonalities and variabilities of systems in a domain. A complete domain model may consist of a domain definition (defining scope and contents), domain dictionary (defining important concepts), concept models (capturing domain concepts in some modeling formalism and/or textual form) and feature models (describing features in the domain and ways to combine them). We will discuss domain modeling as part of TABASCO in Section 5, which includes a comparison of our taxonomy construction approach to the feature modeling approach.

In the domain design step, a system family architecture is developed based on the domain model. In addition, a production plan is developed, describing how concrete systems can be built using the components that will implement the family architecture. This production process can be manual, supported by tools, or fully automatic (as in the generative programming approach, discussed in the next section). It often involves a DSL which allows domain experts to specify their product requirements using domain concepts they are familiar with. An expression in such a DSL is then mapped to an actual software product. A system family architecture should be flexible, in the sense that the architectural skeleton can be adapted by using different components. The development of a systems family architecture based on a domain model—like almost any software architecture development—is still very much an activity based on experience and creativity: although there exist many methods and strategies for this task, none of them gives a precise set of rules to arrive at such an architecture (semi-)automatically. In Section 6, we will sketch how to get from domain model to architecture as part of TABASCO.

Following the design of an architecture, domain implementation takes place. As this depends on the implementation techniques and languages chosen, we do not discuss it in detail in this paper.

3.   GENERATIVE PROGRAMMING

Generative Programming [Czarnecki and Eisenecker 2000] is an approach to software development that seeks to automate the creation of products. For this, *generative* domain models need to be used: as part of the domain analysis, *configuration knowledge* has to be captured. Such knowledge specifies the validity of feature combinations, defaults, construction rules, optimizations and more.
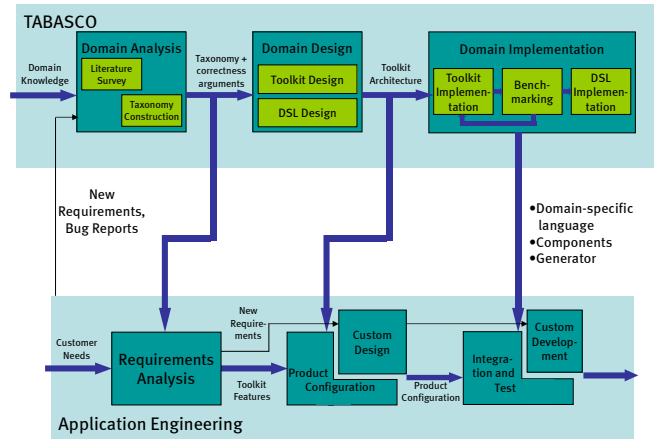
Figure 2: Overview of the TAxonomy BAsed Software COnstruction method

In addition to the modeling of product families, the domain design step needs to produce a formal specification of the configuration knowledge, i.e. the mapping from user requirements (i.e. product specifications, usually in the form of a DSL expression) to implementation components. Having the configuration knowledge captured in a formal way allows for the automatic generation of software components based on user requirements.

Furthermore, implementation of the configuration knowledge by generators—programs generating components or systems based on a user requirements specification—becomes part of the domain implementation step.

4. TABASCO

TABASCO—for TAxonomy BAsed Software COnstruction—is a method for developing reusable software components in the form of *Domain Specific Toolkits* (DSTs) for a restricted form of domains.

In the literature, domain engineering is often applied to business software domains, involving families of large and complex software systems. It seems to us that such applications of domain engineering entail a high risk, related to the complexity of the systems and third-party developer involvement. Such domains may therefore not be ready in general for the development of product families using domain engineering.

We think that applying a formal domain engineering approach to more restricted domains is useful. TABASCO restricts a domain from a family of software systems to one of related algorithms or data structures. It also assumes little or no third-party involvement, apart from the use of the resulting taxonomy, toolkit and DSL.

As a result of these restrictions, TABASCO is similar to DEMRAL—Domain Engineering Method for Reusable Algorithmic Libraries [Czarnecki 1998], [Czarnecki and Eisenecker 2000]. That method however does not construct as formal a domain model and has no direct pedagogic aim. In addition, the relative informality of the model means that the correctness-oriented underpinning of a DEMRAL toolkit is not as strong.

The TABASCO process consists of a number of steps: (1) selection of a specific domain, (2) literature survey, (3) taxonomy construction, (4) toolkit design, (5) DSL design, (6) toolkit implementation, (7) benchmarking and (8) DSL implementation. The two main goals of TABASCO are to bring order to a domain by taxonomizing problem solutions, and to create a toolkit of reusable software components based on the resulting taxonomy.

Figure 2 depicts TABASCO as a domain engineering method. As mentioned, a domain in TABASCO is quite restricted compared to general domain engineering. As a result, the amount of custom design and development in building an application using a TABASCO toolkit and/or DSL will usually be substantial.

We discuss various steps of TABASCO in detail in the next sections and clarify how they can be seen as domain engineering steps.

5. TAXONOMY CONSTRUCTION

The Merriam Webster's Collegiate Dictionary defines a *taxonomy* as "[an] orderly classification of plants and animals according to their presumed natural relationships" [Mish 1993, p. 1208]. Although this definition is biology oriented, we can create a classification according to essential details of algorithms or data structures from a domain as well. Our taxonomies deviate from the kind usually used in biology, since choice points or nodes in our taxonomies are not necessarily single-dimension choice points.

The main goal of taxonomy construction is to improve our understanding of algorithms and their relations, i.e. of their commonalities and variations. The construction is preceded by surveying the domain literature to find existing algorithms. Based on such a survey, one tries to bring order to the field by placing them in a taxonomy.

The various abstract algorithms in an algorithm taxonomy are derived from a common starting point. Details which refine more abstract algorithms and indicate variations are added to this starting point. The common starting point is usually a high-level algorithm whose correctness is easily shown. Associated with it are requirements in the form of a pre- and postcondition, invariant and a specification of (theoretical) running time and/or memory usage, specifying the problem under consideration. (For data structure taxonomies, representation invariants are used, and pre- and postconditions are absent.) The details separating the algorithms may be added for a number of reasons, resulting in the following categories:

— *Problem details* involve minor changes to pre- and postconditions, restricting algorithm input or output

— *Algorithm details* are used to specify variance in algorithmic structure

— *Representation details* are used to indicate variance in the representations and data structures used, either internally to an algorithm or influencing input and/or output representation as well. Note that they do not influence input or output *meaning*; such changes are indicated by *problem* details.

— *Performance details* are about variance in running time and memory consumption. Such details are added only to algorithms lower in the taxonomy, as they may not make sense for the abstract algorithms at the top.

When constructing a taxonomy, the most important detail types will be problem and algorithm details, followed by representation details: the main goal is to broaden our understanding of a particular domain, and we are therefore less interested in performance, apart from theoretical aspects. Since toolkits are based on taxonomies in the TABASCO method, the focus of the toolkit builder and thus of the toolkit will also be more on those aspects. For toolkit users however, performance details, followed by problem details, will be most important; they may not care about representation and algorithm structure at all, as long as the algorithmic components perform well for their problem. The effect is that a mapping from user requirements to toolkit components by means of a DSL is not entirely straightforward, something we discuss later on when describing DSL design and implementation.

We refrain from giving specific guidelines or rules for constructing taxonomies: the choice of details—including their granularity—and of how to structure a taxonomy depends on a person's understanding of and preference for emphasizing certain details of the algorithms in a domain. A taxonomy therefore is *a* taxonomy of a domain, but not *the* taxonomy of that domain, and taxonomists may end up with different taxonomies for the same field. Looking at taxonomy construction as a top-down process, the addition of problem, algorithm or representation details to an algorithm results in a new, refined algorithm solving the same or a similar problem. As a side effect of such detail additions, performance details may change as well. The goal of adding details to algorithms is to (indirectly) improve performance or to arrive at one of the algorithms from the literature. Associated with the addition of a detail, correctness arguments are given that show how the more detailed algorithm can be derived from its predecessor. To indicate a particular algorithm and form a taxonomy graph, we use the sequence of details in order of introduction. In some cases, it may be possible to derive an algorithm in multiple ways by reordering the application of some details. This causes the taxonomy to take the form of a directed acyclic graph.

5.1   A taxonomy of pattern matching algorithms

As an example, Figure 3 shows a taxonomy of keyword pattern matching algorithms. The (exact) keyword pattern matching problem can be described as "the problem of finding all occurrences of keywords from a given set, as substrings in a given string" [Watson 1995]. The top node represents the high-level algorithm for solving this problem (which only specifies that the set of all matches is assigned to the output variable), while algorithms further down in the taxonomy present refinements, with the algorithms close to or at the taxonomy graph leaves representing concrete algorithms. For example, variants of the Aho-Corasick algorithm are found in subtree (P, +, E, SP, AC), while Boyer-Moore algorithm variants are found in subtree (OKW, OBM, INDICES).

To give an idea of what kind of details labeling branches occur in the taxonomy, we consider the path labeled (P, +, E, SP, AC). The branch labeled (P) means that the matches are found by considering prefixes of the text in some order, and detail (+) indicates that the prefixes are considered in increasing length order. Detail (E) indicates that matches are registered by their endpoint, while detail (SP) indicates that the set of suffixes of the currently attempted match that are prefixes of a keyword is maintained to easily compute new matches. Finally, detail (AC) indicates that the longest suffix of the current attempted match that is still a prefix of a keyword is maintained (since the set of suffixes mentioned can be computed from this longest suffix). This is done by using a finite automaton and maintaining a state variable corresponding to this longest suffix. Using detail (AC-OPT), the value of this state variable is updated by a single transition of the so-called optimal Aho-Corasick automaton [Aho and Corasick 1975, section 6]. Using details (LS, AC-FAIL) instead, a linear search involving the so-called failure function Aho-Corasick automaton [Aho and Corasick 1975] is used. Finally, details (LS, KMP-FAIL) lead to a multiple-keyword version of the Knuth-Morris-Pratt algorithm [Knuth et al. 1977].
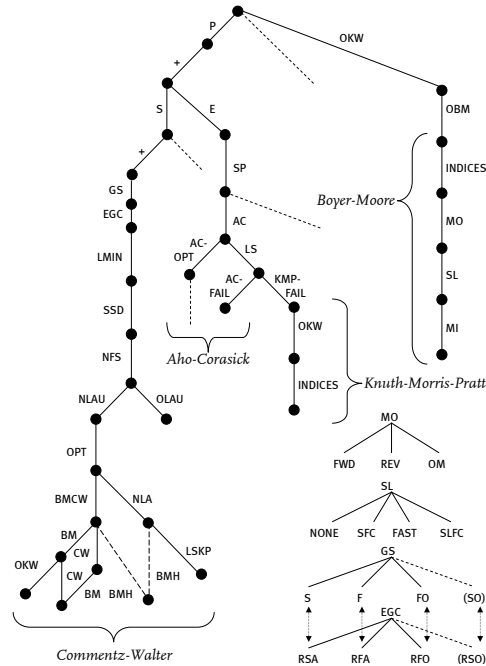
Figure 3: A taxonomy of keyword pattern matching algorithms.

The algorithm factorization involving these algorithms is discussed in detail in [Cleophas and Watson 2005], in which our specific notation for algorithms is used and the algorithms are shown to be instances of a common algorithm skeleton. The taxonomy, including formal correctness arguments for detail additions, is discussed in [Cleophas et al. 2004], [Cleophas 2003], [Watson and Zwaan 1996], and [Watson 1995].

5.2   Taxonomies and feature models

Since a taxonomy is constructed based on an understanding of the algorithms in a domain, their commonalities and variations, a taxonomy represents an understanding of a problem domain: it is a kind of domain model as described in Section 2. In [Czarnecki and Eisenecker 2000], such models are constructed based on feature models. Feature models consist of feature diagrams—depicting the variable as well as the common features in the domain—as well as constraints, defaults and optimizations regarding the combination of such features— forming the configuration knowledge. In taxonomy-based domain models, the configuration knowledge often is not described explicitly, but is implicit in the taxonomy and correctness arguments.

Depending on the problem domain, one might choose the taxonomy construction approach over the feature modeling one, or vice versa. For a field like that of sorting algorithms, in which the working of the various algorithms is not very complex and their relationships are already well known, it might not be necessary or worthwhile to construct a taxonomy. A feature model with a description of the configuration knowledge might be a better solution in this case. On the other hand, feature models are not well-suited for the description of pre- and postconditions and correctness arguments belonging to algorithms or data structures. For domains in which the algorithms are more complex or their relationships to each other are not clear—as was the case for keyword pattern matching before the construction of our taxonomy—it is wise to use the taxonomy approach.

5.3   Advantages of taxonomy construction

At the beginning of this section, we mentioned the main goal of taxonomy construction. Here we give an overview of its goals and advantages, some of which have already been mentioned:

—It makes it easier to compare algorithms. We can determine commonalities and variabilities by comparing the paths from the taxonomy root to the algorithms. In the literature, it is often hard to compare algorithms due to unnecessary details or difference in language or presentation.

—It gives a clear and correct presentation of the algorithms in a problem domain. Unnecessary details and complex presentations often make algorithms hard to understand or even cause them to be incorrect.

—It brings order to a field. The taxonomy can be used as a teaching aid or as a survey of the problem domain.

—It is well suited for discovering new algorithms, as it may reveal gaps in the taxonomy. In developing new algorithms to fill these gaps, the use of formal methods in the taxonomy construction process may prove useful.

—It allows us to direct the derivation process according to requirements, since every algorithm is formally specified. This would be hard to accomplish using a feature model.

—It gives us a formal specification of the algorithms. For each algorithm, we have pre- and postcondition, invariants and performance information. We also have a specification of the algorithm interface. Thus, we have specifications of the requirements satisfied by each algorithm. Furthermore, related algorithms will have related specifications. The formal requirements, together with the valid feature combinations and construction rules that are implicit in the taxonomy, form the configuration knowledge. Capturing the configuration knowledge in this way is useful for mapping user requirements to toolkit components later on, as discussed in Section 7.

Apart from these advantages, the taxonomy's availability aids in the construction of a DST, as we will see.

It should be noted that automated theorem provers could be used to prove the correctness of individual algorithms. Such an approach however does not have many of the other advantages mentioned here.

6. TOOLKIT DESIGN

As noted in Section 2, the development of a system family architecture is mainly driven by experience and creativity. Domain design is not a straightforward task, but the restricted form of our domains and domain models helps: our domains consist of closely related algorithms and data structures, and our domain model consists of a taxonomy which formally relates the algorithms or data structures to one another, grouping them according to their commonalities. The restriction of domains naturally results in restricted complexity and high levels of homogeneity in the domain models. In turn, the homogeneous and rather uncomplicated domain models make the domain design step less complex.

We do not claim that using the TABASCO method, the design of an algorithm or data structure toolkit directly follows from the domain model. Some choices still have to be made whose outcome may depend on the designer's experience and/or creativity. The use of the TABASCO method does make the task of domain design more straightforward however. The taxonomy makes the algorithm commonalities and variabilities explicit. To get from such commonalities and variabilities to a high-level design based on implementation language constructs and design patterns [Gamma et al. 1995], design techniques such as multi-paradigm design [Coplien 1998] are applied: in the latter work, Coplien shows which (C++) language constructs and which design patterns can best be used for particular commonality and variability kinds.

Depending on whether commonalities and variabilities occur in name, behavior, data structure, fine or gross algorithm, default values or state values, Coplien's work suggests the use of specific C++ mechanisms or design patterns. The C++ mechanisms include inheritance, overloading, virtual functions, containment, structs and enumerations. The design patterns used are those dealing with commonality and variability, including Strategy, Template Method, Singleton, Bridge, and Adapter. It would go too far to describe the mapping from specific commonality and variability kinds to specific constructs here, and we refer to the works cited for more information. Some examples will appear in the next section however, to illustrate how applying the approach to a taxonomy domain model makes the design task more straightforward.

6.1   The design of SPARE TIME

As an example, we consider part of the design of SPARE TIME, a pattern matching toolkit based on the taxonomy represented in Figure 3. This example is a revised version of one given in [Cleophas and Watson 2005].

Since C++ was chosen as the implementation language for SPARE TIME, our discussion of the design will be aimed at using language and design constructs available in C++.

Looking at Figure 3, one can see taxonomy parts corresponding to the Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick and Commentz-Walter keyword pattern matching algorithms. The algorithms in the taxonomy solve two similar problems, namely multiple and single keyword pattern matching, but resulting in slightly different interfaces: the first will have a match function taking a set of keywords, the match function of the second will take a single keyword. The toolkit therefore contains an abstract class *PM* and two classes *PMSingle* and *PMMultiple* for single and multiple keyword pattern matchers, each with their own function *match*.

For single-keyword pattern matching algorithms, the toolkit includes three approaches: brute-force/naïve pattern matching, Knuth-Morris-Pratt pattern matching, and Boyer-Moore pattern matching. Since all these have related operations, but differ in algorithm as well as some data structure and state, the suggested C++ construct is the use of inheritance with virtual functions [Coplien 1998, p. 151]. Thus, the *match* function in class *PMSingle* is a virtual one, and classes *PMBFSingle*, *PMKMP* and *PMBM* inherit from it. The use of inheritance with virtual functions in such a way corresponds to the *Strategy* design pattern [Gamma et al. 1995, p. 315]. For multiple-keyword pattern matching, the approaches are brute-force/naïve, Aho-Corasick and Commentz-Walter pattern matching, resulting in classes *PMBFMulti*, *PMAC* and *PMCW* inheriting from *PMMultiple*.
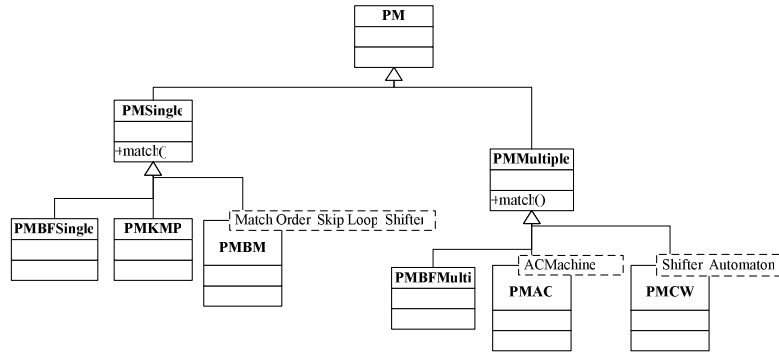
Figure 4: High-level class design of SPARE TIME

As we saw in Section 5, our taxonomy contains three variants of the Aho-Corasick (/multiple keyword Knuth-Morris-Pratt) algorithm: these are the Failure function Aho-Corasick and Goto function (Optimal) Aho-Corasick algorithms [Aho and Corasick 1975; Watson 1995], as well as a multiple keyword generalization of the Knuth-Morris-Pratt algorithm [Watson 1995]. These algorithms share the same algorithm skeleton and differ only in the state update used, that is in the type of automaton used. This corresponds to a representation detail as mentioned in Section 5. Since the general algorithm structure and behaviour are common to all three, Coplien [Coplien 1998, p. 149] suggests the use of templates. Class *PMAC* therefore is a template class accepting a parameter indicating the particular type of automaton to use in function *match*. Although not precisely corresponding to the description of the pattern, this part of the design can also be seen as an instance of the *Template Method* design pattern [Gamma et al. 1995, p. 325].

Since there is only one single keyword Knuth-Morris-Pratt variant in our taxonomy, the corresponding implementation is a straightforward one: class *PMKMP* is a child class of *PMSingle*, implementing function *match*.

The figure also shows many Boyer-Moore algorithm variants to be part of the taxonomy, and the same holds for Commentz-Walter-like algorithm variants. A design for the parts of the toolkit corresponding to these taxonomy parts can be obtained in a similar way as for the parts discussed above.

The resulting design (depicted in Figure 4) uses merely two different match functions for the whole toolkit, resulting in a clean and easy to understand interface.

Once a toolkit design has been constructed based on the taxonomy, implementing the toolkit is straightforward, as the specific algorithms have been derived in a common, abstract format as part of the taxonomy construction. We do not discuss toolkit implementation here, also because of its dependence on implementation language.

In addition to simplifying the design task, the taxonomy gives us structural invariants for the toolkit classes. These increase confidence in algorithm correctness and safety, and are therefore implemented by the classes: each has a class invariant member function which returns a Boolean to indicate an object's conformance to this structural invariant. These functions have been useful in understanding the code as well as debugging it.

More details on the design and implementation of SPARE TIME (as well as its predecessor SPARE PARTS) can be found in [Cleophas and Watson 2005; Watson and Cleophas 2004; Cleophas 2003; Watson 1995].

6.2   Advantages of taxonomy-based toolkit design

Taxonomy-based toolkit design in our opinion has more advantages than simply making the design process more straightforward. The advantages include the following:

—As mentioned, the availability of the taxonomy structure simplifies the toolkit design task. The structure of the toolkit (especially its grouping of algorithms by common properties) guides the toolkit architecture.

—In addition, the taxonomy's homogeneity results in a highly homogeneous design. Such a design may be easier to understand than an ad hoc design, especially given the availability of the taxonomy alongside the design.

—A toolkit's homogeneity and coherence (resulting from the grouping of algorithms based on their commonalities) make the client interface to such a toolkit homogeneous and easy to understand and use.

—The uniformity in presentation and implementation style that is possible due to the taxonomy, gives greater confidence in the accuracy of relative performance when comparing the different algorithms in the toolkit. This accuracy is often not present in existing algorithm collections due to the sometimes vastly differing styles used to implement and/or present the individual algorithms, as mentioned before.

—The taxonomy-basedness also makes the rest of the code (apart from the client interface) easier to understand and debug. This helps toolkit users as well as toolkit builders or extenders.

—The toolkit and taxonomy serve as examples of toolkit design and implementation techniques; the taxonomy-basedness makes it more straightforward to design and implement them, since it guides the design structure.

—The correctness arguments that are included in the taxonomy give us confidence in correctness and safety of the resulting toolkit, as pointed out in the previous section. Given the taxonomy presentation of the algorithms and their correctness arguments, implementing the abstract algorithms can be painless and fun as well.

—Related to the correctness arguments, the taxonomy in fact gives a complete formal specification of the requirements satisfied by each algorithm (including pre- and postcondition, invariant and specification of (theoretical) running time and/or memory usage). As a result, we have an explicit requirements specification for each toolkit component. This helps in understanding such components, in creating a mapping from user requirements to toolkit components (see Section 7) as well as in keeping track of changing requirements.

7. DSL DESIGN AND IMPLEMENTATION

Once a toolkit has been designed and implemented, one could say that the process is finished: after all, a reusable toolkit has been developed that can be used in application engineering. This disregards the difference between user requirements and what the toolkit components provide (in terms of requirements satisfied by them). This difference makes it hard for the average user—who may know nothing about the algorithms or data structures—to effectively and easily use toolkit components.

To overcome this problem, a DSL may be developed. Such a DSL or 'little language' can be tailored towards the domain concepts that the average user is interested in or familiar with and allows him or her to specify requirements that a requested toolkit component should satisfy. Usually, such DSLs put a focus on performance and problem details due to the user being interested in these, as discussed in Section 5. The toolkit components on the other hand will in general be structured based on algorithm and representation details. A mapping is therefore needed to get from DSL expression to one or more toolkit components.

As indicated in Section 3, such a mapping is based on the configuration knowledge captured during domain modeling. The information on valid component combinations and on construction rules therefore comes from the taxonomy (or feature diagram) and associated information. The information on defaults and efficiency needed for the mapping comes from benchmarking—hence the benchmarking substep in Figure 2. Based on this information, the mapping that forms the core of a DSL is encoded.

In the most basic form, the DSL will not be explicit, and the mapping consists of some paragraphs of text specifying which toolkit components to use when. For SPARE TIME such a text may include sentences like 'if performance independent of keyword set is required then use the Optimal Aho-Corasick algorithm implemented by $PMAC\langle ACMachineOpt\rangle$, else ...' (based on [Watson 1995, p. 309]).

Alternatively, an explicit DSL may be designed which users can use to specify requirements a component is expected to satisfy. A DSL expression (i.e. user requirements specification) should then be automatically processed to map to a certain toolkit component. This can be done by means of a generator, which may not only select a toolkit component but even generate new components based on parts already available in a toolkit.

For example, a small DSL for specifying sorting algorithm components was developed, based on a sorting toolkit and sorting algorithms feature diagram [Cleophas and Frishert 2002]. The grammar of this DSL allows a user to specify requirements such as size of array to be sorted, variance in values, running time performance and so on. In this case, the DSL and mapping from user requirements (DSL expression) to toolkit components by means of a generator were implemented using C++ template metaprogramming techniques, but other techniques may be used instead, such as scripting languages and/or makefiles. The parameters specified in a DSL expression are compared to a decision table at compile time. The first row in the table that matches the parameters (and/or the default parameter values in case a user does not specify any) determines which sorter component is returned.

Taking it one step further, the generators may be generic, i.e. not developed for a single DSL, but for DSLs in general. We have recently developed such a generic generator, an experimental tool called FUEL [Ouwens 2004]. It takes a DSL description—an XML file containing a grammar and requirements-to-components mapping—and expression, and generates an appropriate component based on a template file. A new DSL can easily be created by supplying different XML and template files to FUEL. FUEL is implemented using the Ruby scripting language.

As an example, we consider SPARE FUEL, a DSL for SPARE TIME. We give part of the XML file for it:

```
...
 <codefiles>
  <file src="sparefuel.src" target="sparefuel.h"/>
 </codefiles>
 <algorithms>
  <alg id="CWNFS" name="CW-[NFS, Factoracle]" classname="PMCW<CWShiftNFS, RFactoracle>">
   <param>cw/cws.hpp</param>
```

```
  </alg>
 </algorithms>
 <rules>
  <rule name="alphabet" value="English" default="CWNFS">
   <rule name="memory" value="high" default="CWNFS">
    <rule name="length" value="3" default="ACOpt">
     <rule name="setsize" value="1" default="CWNFS"/>
   ...
```

The codefiles part specifies the names of template file and output file to be generated. The algorithms part introduces identifiers for the algorithms, which are represented by a name, class name and source file. The rules section, lastly, encode the mapping. In the example, for an English alphabet, with a high amount of memory available and a minimal keyword length of 3, by default, algorithm ACOpt is chosen. In case the keyword set size is specified to be 1 however, this default is overridden and algorithm CWNFS is selected instead.

Using the example XML and template file `sparefuel.src` referred to in the example XML, the call

```
./fuel.rb sparefuel alphabet=English memory=high length=3 setsize=1
```

generates the following include file for the toolkit user.

/* Chosen algorithm: CW-[NFS, Factoracle]
    Parameters: alphabet=English, memory=high, length=3, setsize=1   */
#include "cw/cws.hpp"
typedef PMCW<CWShiftNFS, RFactoracle> PMAlg;

The example is based on SPARE TIME, written in C++, but FUEL is language-independent; it can deal with template files from other languages as well.

It should be empasized that FUEL is experimental. Apart from its use for SPARE FUEL, it has only been applied to small example toolkits. As a result, the example given here is also rather simple. It is conceivable that the current tool is not general enough to encode DSLs for other TABASCO domains. An important restriction of FUEL for example is that the order in which parameters appear on the command line should match the nesting order in which they appear in the XML file. We plan to remove this restriction in future versions and use it to develop other DSLs. This will show whether the tool is general enough for its application in TABASCO or not.

8. RELATED WORK

The type of taxonomy development and algorithm derivation used in TABASCO has been used by Jonkers for garbage collection algorithms [Jonkers 1983] and by Marcelis for attribute evaluation algorithms [Marcelis 1990].

The TABASCO approach was previously described in Watson's PhD thesis [Watson 1995] and in a paper by Cleophas and Watson [Cleophas and Watson 2005]. The taxonomy construction step of TABASCO in considered in more detail in those publications. In both, TABASCO had not yet matured into the domain engineering method presented in this paper. A poster with an outline of TABASCO was presented to Dutch industry practitioners and researchers at the JACQUARD2005 event [Cleophas et al. 2005].

The TABASCO approach has been (completely or partially) used for various algorithmic or data structure domains. In particular, a lot of work was done or is being done as part of FASTAR. FASTAR (*Finite Automata Systems—Theoretical and Applied Research*; `http://www.fastar.org`) involves finite automata-related research at the Technische Universiteit Eindhoven and the University of Pretoria. The work includes taxonomies of algorithms for *Minimal Acyclic Deterministic Finite Automata* (MADFA) construction [Watson], approximate keyword pattern matching [Bosman 2005], 2-dimensional pattern matching [van de Rijdt 2005], as well as FIRE LITE, a taxonomy-based toolkit of finite automata construction and minimization algorithms [Watson 1995], and the keyword pattern matching example discussed in this paper. A taxonomy and toolkit of regular tree language algorithms—generalizing FIRE LITE and SPARE TIME from strings to trees—are currently planned.

The TABASCO approach has also been used to create a taxonomy of Lempel-Ziv compression algorithms [Kouwenberg 2003] and a taxonomy [Barla-Szabo 2002] and toolkit [Koopman] of graph representations.

9. FINAL REMARKS

In this paper, we presented TABASCO, our TAxonomy-BAsed Software COnstruction domain engineering method. We gave a brief overview of domain engineering and generative programming. We described TABAS-CO, our approach to the construction of DSTs for a restricted kind of domains, namely that of algorithms solving (or data structures used for) the same or closely related problems. The steps of TABASCO were explained and motivated in detail, with an emphasis on the software construction steps of the method. In describing

our method, we gave examples from fields to which TABASCO has been applied, including keyword pattern matching and sorting algorithms.

As future work, we see the (further) application of TABASCO to other algorithmic or data structure fields—such as those mentioned in Section 8. Such work should lead to more explicit and more general guidelines on how to get from taxonomy or feature diagram to domain-specific toolkit. In particular, we hope to gain more knowledge on the step from domain model to domain design, which as indicated still involves some amount of creativity and experience. Having others apply TABASCO to algorithmic or data structure fields will also provide us with feedback regarding the usability and effectiveness of TABASCO for domain engineering.

We also plan to apply the experimental generic DSL generator FUEL to other domains. This research should indicate whether FUEL is general enough to be applied to any algorithmic or data structure field, or that it needs to be ammended to reach this goal.

REFERENCES

AHO, A. AND CORASICK, M. 1975. Efficient string matching: an aid to bibliographic search. *Commun. ACM 18*, 333–340.

BARLA-SZABO, G. 2002. A taxonomy of graph representations. M.S. thesis, Dep. Comput. Sci., University of Pretoria.

BOSMAN, R. 2005. A Taxonomy of Approximate Pattern Matching Algorithms in strings. M.S. thesis, Dep. Math. Comput. Sci., Technische Universiteit Eindhoven.

CLEOPHAS, L. AND FRISHERT, M. 2002. A Taxonomy and Toolkit of Sorting Algorithms. Assignment report for course 2R870 Software Construction, Dep. Math. Comput. Sci., Technische Universiteit Eindhoven.

CLEOPHAS, L., VOSLOO, I., AND WATSON, B. W. 2005. TABASCO: A Taxonomy-based Domain Engineering Method. In *Proceedings of JACQUARD2005: First Conference for the Software Engineering Community*. Poster Paper.

CLEOPHAS, L. AND WATSON, B. W. 2005. Taxonomy-based software construction of SPARE Time: a case study. *IEE Proceedings—Software 152,* 1 (February).

CLEOPHAS, L., WATSON, B. W., AND ZWAAN, G. 2004. Automaton-based sublinear keyword pattern matching. In *Proceedings of the 11th international conference on String Processing and Information REtrieval (SPIRE 2004)*. LNCS, vol. 3246. Springer.

CLEOPHAS, L. G. 2003. Towards SPARE Time: A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms. M.S. thesis, Dep. Math. Comput. Sci., Technische Universiteit Eindhoven.

COPLIEN, J. O. 1998. *Multi-Paradigm DESIGN for C++*. Addison-Wesley, Reading, MA.

CZARNECKI, K. 1998. Generative programming: Principles and techniques of software engineering based on automated configuration and fragment-based component models. Ph.D. thesis, Dep. Comput. Sci. Automation, Technical University of Ilmenau, Germany.

CZARNECKI, K. AND EISENECKER, U. W. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

JONKERS, H. 1983. Abstraction, specification and implementation techniques, with an application to garbage collection. *Mathematical Centre Tracts 166.*

KNUTH, D., MORRIS, J., AND PRATT, V. 1977. Fast pattern matching in strings. *SIAM J. Comput. 6,* 2, 323–350.

KOOPMAN, T. A generative graph toolkit. M.S. thesis, Dep. Comput. Sci., University of Pretoria. To appear in 2005.

KOUWENBERG, J. J. 2003. A taxonomy of Lempel-Ziv compression algorithms. M.S. thesis, Dep. Math. Comput. Sci., Technische Universiteit Eindhoven.

MARCELIS, A. 1990. On the classification of attribute evaluation algorithms. *Sci. Comput. Program. 14*, 1–24.

MISH, F. C., Ed. 1993. *Merriam Webster's Collegiate Dictionary*, 10th ed. Merriam Webster, Springfield, MA.

OUWENS, T. 2004. The Performance of SPARE Time. Internship Report, Software Construction Group, Dep. Math. Comput. Sci., Technische Universiteit Eindhoven.

SOFTWARE ENGINEERING INSTITUTE. 1997. Model-Based Software Engineering. WWW pages, http://www.sei.cmu.edu/mbse/.

VAN DE RIJDT, M. 2005. A Taxonomy of Approximate Pattern Matching Algorithms in strings. M.S. thesis, Dep. Math. Comput. Sci., Technische Universiteit Eindhoven.

WATSON, B. W. Constructing minimal acyclic deterministic finite automata. Ph.D. thesis, Dep. Comput. Sci., University of Pretoria. To appear.

WATSON, B. W. 1995. Taxonomies and toolkits of regular language algorithms. Ph.D. thesis, Dep. Math. Comput. Sci., Technische Universiteit Eindhoven.

WATSON, B. W. AND CLEOPHAS, L. 2004. SPARE Parts: A C++ toolkit for String PAttern REcognition. *Softw. Pract. Exper. 34,* 7 (June), 697–710.

WATSON, B. W. AND ZWAAN, G. 1996. A taxonomy of sublinear multiple keyword pattern matching algorithms. *Sci. Comput. Program. 27,* 2, 85–118.