# An Algorithm for Variability Identification by Selective Targeting

Anilloy Frank
Institute of Technical Informatics,
Graz University of Technology
Inffeldgasse 16, 8010 Graz, Austria
Email: frankanilloy@gmail.com

Eugen Brenner
Institute of Technical Informatics,
Graz University of Technology
Inffeldgasse 16, 8010 Graz, Austria
Email: brenner@tugraz.at

***Abstract*—Large companies have large embedded software systems, where common and reusable software parts are distributed in various interrelated subsystems that also have lots of uncommon and non-reusable parts. The approach finds software parts that may or may not be reusable in a particular application engineering project. It is the task of application engineering to figure out whether the identified components and variants are directly reusable and reuse them in application engineering. In Software Product Lines, the identified reusable common and variable components should be generalized and stored into asset bases. In real life, it may be too much effort and costs to generalize application level assets into domain assets and it is just more feasible to try to find reusable common and variable components directly from existing applications. The proposed approach is selectively targeting the component-feature model instead of an inclusive search to improve the identification. We explore the components and their features from a predefined component node list and the features node vector respectively.**

***Keywords*—Design Tools; Embedded Systems; Feature Extraction; Software Reusability; Variability Management.**

## I. INTRODUCTION

This paper is an extension of the conference paper [1], and aims at providing a greater insight into the algorithm for managing software variants of embedded systems. It presents a semi-automatic approach to identify reusable parts.

Embedded systems are microcontroller-based systems built into technical equipment mainly designed for a dedicated purpose. Communication with the outside world occurs via sensors and actuators [2]. Although this definition implies that embedded systems are used as isolated units, from 2006 it is observed that there is a trend to construct distributed pervasive systems by connecting several embedded devices as indicated by Tanenbaum and van Steen [3].

The current development trend in automotive software is to map software components on networked Electronic Control Units (ECU), which includes the shift from an ECU based approach to a function based approach. Also according to data presented by Ebert and Jones [4] up to 70 electronic units are used in a car containing embedded software, which is responsible for the value creation of the car and consists of more than 100 million lines of object code.

Ebert and Jones presents data about embedded software, stating that the volume of embedded software is increasing between 10 and 20 percent per year as a consequence of the increasing automation of devices and their application in real world scenarios.

An industrially accepted approach in the automotive applications is Model Based Software Engineering (MBSE). Model-Driven Engineering (MDE) is the use of models as the main artifacts during the software development and the maintenance process. Model Driven Software Development (MDSD) is typically realized in a distributed system environment.

Most MDSD approaches follow the Model Driven Architecture (MDA) concept. In this concept, we start with the specification of a platform independent model, this is then transformed to a platform specific model by applying several generators. The layered Meta Object Facility (MOF) approach is used for creating the models. This approach is also used as the basis for the Unified Modeling Language UML.

While MDSD facilitates models for the abstract specification of system architectures, their platform specific artifacts are often realized by applying Component Based Software Engineering (CBSE) techniques. Models become artifacts to be maintained along with the code, by using model transformations and code generation.

MDE is related with the Object Management Group (OMG) initiatives, Model-Driven Architecture (MDA®) and Model-Driven Development (MDD®), which argue that the use of models as the main artifact on software development will bring benefits on software reuse, documentation, maintenance, and development time.

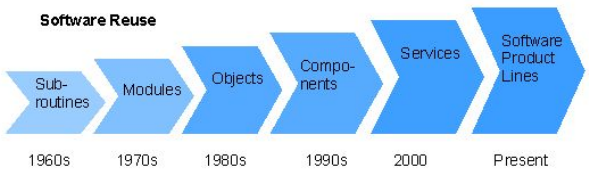


Fig. 1. Software reuse history.

Reuse of automotive embedded software is difficult, as it is typically developed for a small ECU that lacks both processing speed and memory of a general purpose machine. Moreover, the complexity of the embedded software is dramatically increasing. In view of this complexity, achieving the required reliability and performance is one of the most challenging problems [5].

Figure 1 shows a short history of the usage of reuse in software development. In the 1960s, reuse of software started with subroutines, followed by modules in the 1970s and objects in the 1980s. Around 1990 components appeared, followed by services at about 2000. Currently, Software Product Lines (SPL) are state of the art in the reuse of software. Today, many different approaches exist to the implementation of Software Product Lines, but the complexity still remains at unmanageable proportions.

Complexity management has become a vital factor in an organization. To save costs a company needs to minimize internal complexities arising from numerous factors like large products portfolios, regulations that necessitate component variations in different regions, requiring components from external sources like Original Equipment Manufacturers (OEMs), requirements for meeting certifications, and Virtual Organizations (VOs) [6]. It is also necessary to satisfy the range of customer requirements which determines external complexity. The dynamics involved is due to three major factors:

- *Globalization:* For companies to be present in all major markets and to be competitive the requirements of customers with different cultural, technological, economic, and legal backgrounds needs to be incorporated in products.
- *Evolving Technology:* With a need to reduce the time-to-market, technology is evolving at an extremely fast pace. The trend to launch new products quickly in the market is increasing, which necessitate for enhanced technology as well as convergence of technologies [7][8].
- *Increasing market influence:* The customers influence to determine a product's features and price is inducing the manufacturers to provide more and more product variants.

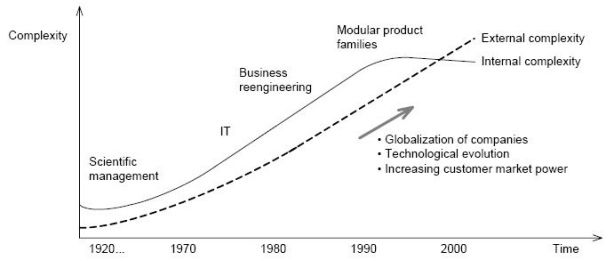


Fig. 2. Evolution of complexities [9].

Figure 2 depicts numerous methods and tools introduced in the past to limit the impact of rising external complexity onto internal complexity in manufacturing, information management, and processes.

With globalization, evolving technology, and increasing market influence the complexity revolving in reuse of embedded software is becoming extremely unmanageable mainly due to large number of variants. The proposed strategy is to introduce a variability identification layer that intends to facilitate software reuse. We start by analyzing the model structure. Based on this we form a concept to extract an element list to facilitate the identification of variability. The implementation section describes algorithm fragments of the different functional blocks. The evaluation of the proposed strategy is based on a technically advanced adaptation of a formal mathematical model [10], which is beyond the scope of this paper.

The rest of this paper is organized as follows. In Section II, a brief summary of related work by other authors is given, while in Section III, enumerates the objectives of the approach. Section IV presents the concept and approach, algorithm fragments, and evaluation. Section V discusses the contributions, while Section VI draws conclusions and future work.

## II. Related work

Usually, the product governs processes, manufacturing and information. The product is an interface between external and internal complexity. Designing modular products and applying module variants results in product families [11]. The interfaces between these modules need to be clearly specified. To address modular product families from a holistic perspective it needs to be managed in development and realization across the entire life cycle.

With so many modular product families now being in place, the following observations however indicate the following:

- *Increasing number of variants:* The number of variants continues to rise and is unmanageable in most companies. Due to cannibalization effects, new variants often do not substantially increase sales but only lead to redistribution from standard to special products. As a result, increased costs are not passed on to the selling price and the profit margin decreases [12].
- *Insufficient decision basis:* Many of the complexity effects cannot be captured using traditional accounting techniques, e.g., overhead calculation. The widely-used methods and lack of technical knowledge on the consequences can be misleading when it comes to decisions in variant management.
- *Unsuitable Methodologies:* Modular product families are treated with the same mechanisms as single products, which is unsuitable. Modular product families require a different approach to variant management than single products as interfaces and interactions among modules is crucial.

Planning a standardized architecture within an organization may address a part of these problems and facilitate reuse. With constantly changing requirements within the set of products, the variability needs to evolve. Many embedded systems are implemented with a set of alternative function variants to adapt to the changing requirements. Major challenges are in identifying the commonality of functionality, where the designs involve variability (ability to customize). In addition to variants, versions/releases of functional blocks also play an important role for the effective management over the entire product cycle.

Figure 3 depicts a scenario where well established software components tested for performance, safety, and reliability

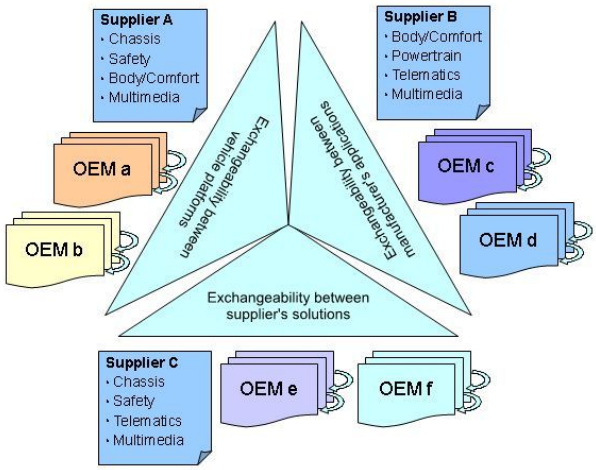


Fig. 3. External components are a hindrance to variability management.

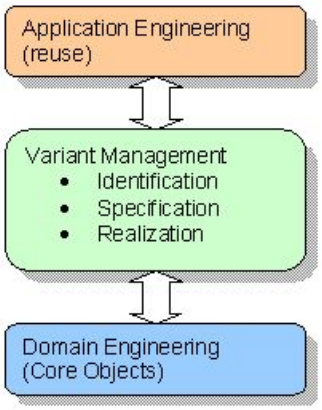


Fig. 4. Variability management in product lines.

procured from external sources and Original Equipment Manufacturers (OEMs) are causes for a hindrance in managing variability.

For achieving large-scale software reuse, reliability, performance, and rapid development of new products, a software product-line (SPL) is an effective strategy. A SPL is a family of products sharing the same assets allowing the derivation of distinct products within the same application domain.

An SPL is a set of software-intensive systems that share a common set of features for satisfying a particular market segment's needs. SPL can reduce development costs, shorten time-to-market, and improve product quality by reusing core assets for project-specific customizations [13][14].

The SPL approach promotes the generation of specific products from a set of core assets, domains in which products have well defined commonalities and variation points [15].

Enabling variability in software consists in delaying decisions at different software abstraction levels, ranging from requirements to runtime. The object-oriented approach to implement variability is based on the development of a frameworks of reusable software components described by a set of classes and by way instances of those classes collaborate.

One of the fundamental activity in Software Product Line Engineering (SPLE) is Variability Management (VM). Throughout the SPL life cycle, VM explicitly represents variations of software artifacts, managing dependencies among variants and supporting their instantiations [13].

To enable reuse on a large scale, SPLE identifies and manages commonalities and variations across a set of system artifacts such as requirements, architectures, code components, and test cases. As seen in the Product Line Hall of Fame [16], many companies have adopted this development approach.

As depicted in Figure 4, SPLE can be categorized into domain engineering and application engineering [17][18]. Domain engineering involves design, analysis and implementation of core objects, whereas application engineering is reusing these objects for product development.

Activities on the variant management process involves variability identification, variability specification and variability realization [19].

- The Variability Identification Process will incorporate feature extraction and feature modeling.
- The Variability Specification Process is to derive a pattern.
- The Variability Realization Process is a mechanism to allow variability.

To enable identification of variability for software components in a distributed system within the automotive domain [20][21], we enlist the specifications below:

- *Specification of components by compatibility*
  The product is tested using software functions of a certain variant and version. These products may exhibit compatibility issues between functional blocks, whilst using later version of the function may fail to perform as expected.
- *Extract, identify, and specify features*
  To enable parallel development, it is necessary to be able to extract features, and to identify and specify the functional blocks in the repository based on architecture and functionality.
- *Usability and prevention of inconsistencies*
  A process that tracks usability and prevents inconsistencies due to deprecate variants and versions in the repository is required.
- *Testing mechanism for validations*
  A testing mechanism for validations in order to maintain high quality for components and its variants has to be established.
- *Mechanism for simplified assistance*
  The developer has to be assisted by a process to intelligently determine whether a functional block or its variant should exist in the data backbone to avoid redesign of existing functions, thereby improving productivity.

Although variability management is recognized as an important aspect for the success of SPLs, there are not many solutions available [22]. However, there are currently no commonly accepted approaches that deal with variability holistically at architectural level [23].

Based on the challenges discussed and the concluded related work presented, the following objectives can be derived.

## III. OBJECTIVES OF THIS APPROACH

- *Objective 1: Support heterogeneous models containing hierarchically embedded software components containing the complete specification of specific functionality to foster reuse.*
  Breaking down the models into several components and logical clustering of components of the modeled software is not targeted. In contrast, the proposed methodology enables the identification of commonalities of components in heterogeneous models. For deployment and reuse purposes several partial models are treated as one artifact. Furthermore, the architecture should support reuse of these artifacts for the development of new functionalities.
  The challenge of the realized system of artifact heterogeneity should be based on existing component technologies that provides mature techniques, that are a consequence of the application independent and generic definition of the system specific components and ensures the portability of the proposed system on other platforms.

- *Objective 2: Enable dynamic configuration.*
  Each subsystem is modeled and simulated using a domain-specific simulation tool, while the co-simulation platform handles the coupling between these subsystems that enables holistic simulation of a system.
  The challenge for identifying variability of software components validating to numerous schemata of respective simulation tools and dynamically loading of plug-ins for specific set of components adhering to respective schemata at execution time in model interpretation architecture.

- *Objective 3: Enable shared usage of resources.*
  A scenario depicting the concept of virtual organization should have a clear method to tackle resource access, validation and verification of specific models.

## IV. ACTIVITIES FOR VARIABILITY IDENTIFICATION

Models confirming to numerous tools like ESCAPE®, EAST-ADL®, UML® tools, SysML® specifications and AUTOSAR® were considered. Although this concept is not limited to automotive domain alone.

### A. Project analysis

An analysis of the models exhibits a common architecture. Figure 5 depicts the textual representation that underlies several graphical models. The textual representation usually is given in XML, which strictly validates to a schema. A heterogeneous modeling environment may consist of numerous design tools, each with its own unique schema, to offer integrity and avoid inconsistencies. Developed projects have to be strictly validated to the schemas of these tools.

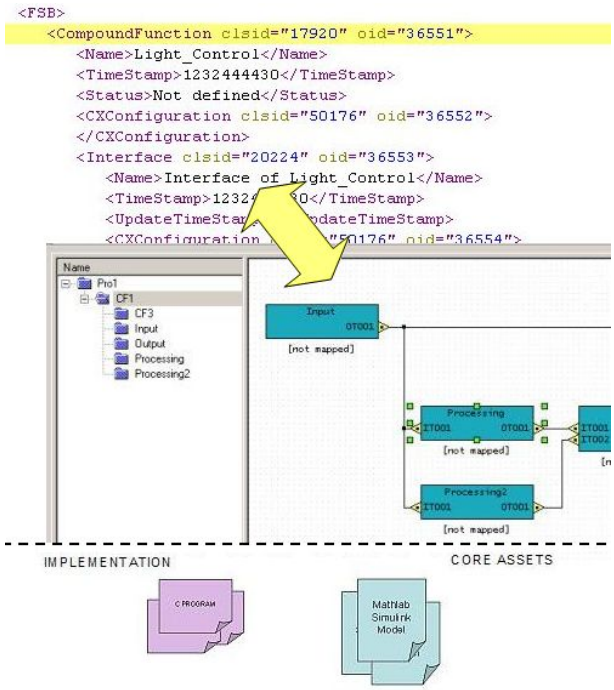


Fig. 5. Mapping textual and graphical representations.


Fig. 6. XML Nodes that are not significant for variability.

A closer examination of the nodes in the textual representation of models depicted in Figure 6 reveals some interesting information. The nodes outlined in rectangles provide important information regarding the identity, specification, physical attributes, etc. of a component, but are insignificant from the perspective of variant.

### B. Concept and approach

The basic concept to identify variability is depicted in Figure 7.

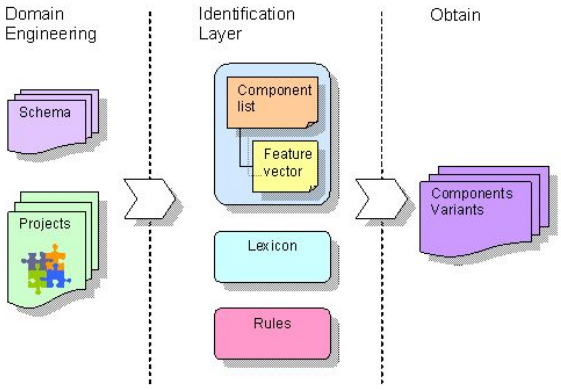


Fig. 7. Basic Concept.

The left side is a set of projects that have software components hierarchically embedded. These projects validate to the corresponding schemas. The middle layer is an identification layer with three functional blocks. A set of component lists is derived from the node list in the schema. Similarly, a feature vector is derived from the schema that corresponds to components. The second block is a customized parser that generates a relevant lexicon from the set of software components within a project. The third block is a set of rules (viz., mandatory, optional, exclude) to govern the identification of variability.

The basic concept can be extended to obtain a working model for the identification of variants. The work flow is depicted in Figure 8. The top layer here represents the domain or core assets. The middle layer is a semi-automatic identification layer for variants. A component list and a feature vector is derived manually from the schema of the project; a collection of elements that represent components and their descriptive features that significantly contribute to the identification of the component's variant.

The workflow can be further extended to adapt a heterogeneous environment, which consist of projects developed using several modeling and simulation tools. The identification layer is separated into two parts. Numerous component lists and feature vectors can be derived for each distinct schema as depicted in Figure 9, whereas a common lexicon and common rules govern the identification process.

### C. Implementation

In this section, several key aspects of the implementation are discussed. The focus is to describe the architecture of the identification layer, which forms the intermediate layer for adapting the core assets from domain engineering into application engineering.

The related approaches put on view a need for a generic methodology in identification of software components developed using several design tools.

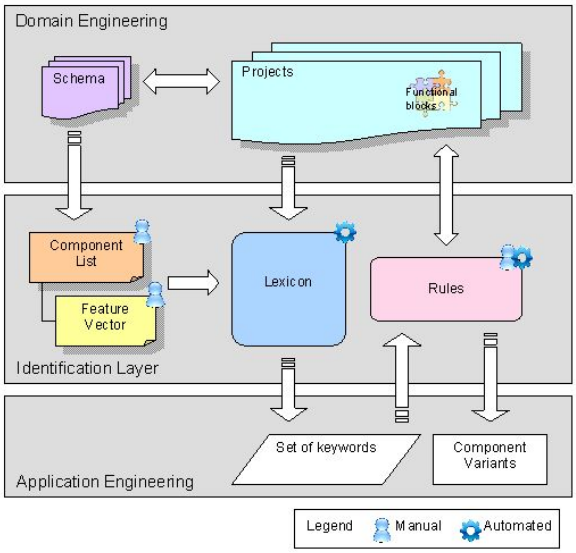


Fig. 8. Work flow of the identification process.

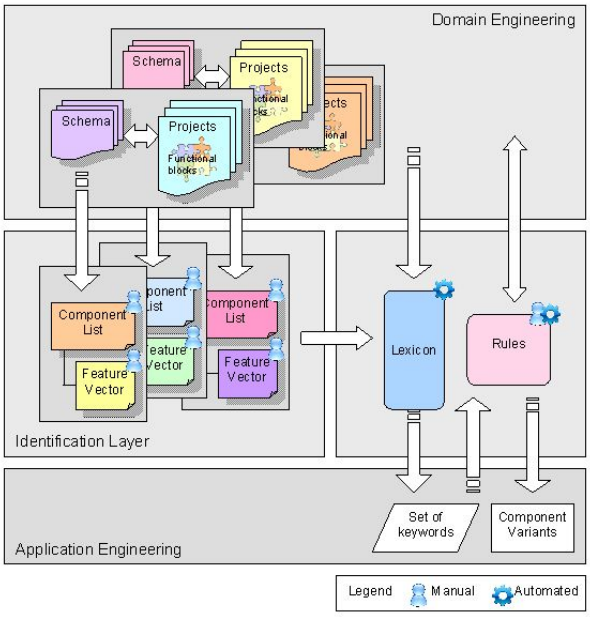


Fig. 9. Work flow of the identification process for heterogeneous systems.

*1) Component list and feature vectors:* As the project structure for each tool is well defined and strictly validated with corresponding schemata, these schemata can be used as basis for deriving the list that can identify components.

The results are summarized in Table I.

#### a) Component list

An example of the list of elements that characterize components derived manually from the schemata for design tool ESCAPE [24] is

```
"CompoundFunction HWFunction
SWFunction Parameter"
+ " StructureElement SWBubbleType
ParameterType ParameterTypeTerminal
IntDataType FloatDataType
TimeDataType AliasDataType
VariantDataType HWFunctionType
TypeInterface FunctionTypeTerminal
HWTypeTerminal"
+ " StructureElement DeviceMapping
DeviceType BusCAN BusSegment
MappedFunction"
```

The list is a delimited string with whitespace or any other delimiter.

A tool supporting multi-functional structures like ESCAPE has three views: Functional structure builder (FSB), Function type builder (FTB) and Hardware Structure Builder (HSB). Each view can have an independent list

- Component list for FSB
  FSB facilitates to build the structure of the model.

  ```
  "CompoundFunction HWFunction
  SWFunction Parameter"
  ```

- Component list for FTB
  FTB provides defining hardware and software types.

  ```
  "StructureElement SWBubbleType
  ParameterType ParameterTypeTerminal
  IntDataType FloatDataType
  TimeDataType AliasDataType
  VariantDataType HWFunctionType
  TypeInterface FunctionTypeTerminal
  HWTypeTerminal"
  ```

- Component list for HSB
  HSB that allows networking ECUs and mapping the software functions.

  ```
  "StructureElement DeviceMapping
  DeviceType BusCAN BusSegment
  MappedFunction"
  ```

Similarly, in a heterogeneous modeling environment each modeling tool will have its own schemata, and a corresponding list may be derived for each tool.

*b) Feature vector*

Similarly, the elements that characterize features of the software components are also derived manually from the schemata, which forms the feature vector and are enlisted below

```
"Name LongName DEScription
ConnectionSegment SourceTerminal
SinkTerminal Interface
CompoundTerminal HWTerminal
SWTerminal Input DataType"
```

*c) Algorithm to identify components within projects*

Using the string described in Section IV-C1a that characterize the software components nodes list within a project, the following algorithm can be devised.

**componentListString** ← string described in Section IV-C1a;

**Nodes** ← doc.GetElementsByTagName("*");

for each **Node** in the **Nodes** (Length($L_n$) $\geq$ 1), do

  if **Node.name** in **componentListString**, then

    **componentList** ← **Node.name**;

The order for matching the software components is $O(N)$.

The prototype dataset used for evaluation of this algorithm contained a total of 32909 nodes, of which only 1583 matches were the software components.

Similarly, using the string described in Section IV-C1b that characterize the features within software components, the following algorithm can be devised.

**featureVectorString** ← string described in Section IV-C1b;

**Nodes** ← **componentList**;

for each **Node** in the **Nodes** (Length($L_c$) $\geq$ 1), do

  if **Node.name** in **featureVectorString**, then

    **featureList** ← **Node.name**;

The order for determining the corresponding features within the software components is $O(N)$.

From the prototype dataset a total of 13353 nodes matches to the feature vector were found.

The results are summarized in Table II.

*2) Lexicon:* A simple customized parser has been devised which automatically extracts words from the text within the software components and features that match the component list and feature vector respectively.

**lexiconList** ← NULL;

**Nodes** ← **componentList** ∪ **featureList**;

for each **Node** in the **Nodes** (Length($L_{cf}$) $\geq$ 1), do

  **wordList** ← split(**Node.innerText**, delimiter) ;

  for each **word** in the **wordList** (Length($L_w$) $\geq$ 1),do

    if **word** not in **lexiconList**, then

      **lexiconList** ← **word**;

      **lexiconList.frequency** ← 1;

else

**lexiconList.frequency** ←

**lexiconList.frequency**+1;

End For;

A more sophisticated parser that discards non-words will further improve the Lexicon.

The Lexicon assists the user to choose from a set of relevant words along with their frequencies thereby improving the user experience.

*3) Rules:* In every case, a full match of software components to specification sets is not desired, but in many instances specification sets contain elements that are mandatory (contains all), optional (one or more) and exclude (omit). Providing rules to execute these features enhances the performance in the identification process.

**ruleContainAll** ←

Specification subset with Contain-all elements;

**ruleOptional** ←

Specification subset with Optional elements;

**ruleExclude** ←

Specification subset with Exclude elements;

**Nodes** ← **componentList** ∪ **featureList**;

for each **Node** in the **Nodes** (Length($L_{cf}$) ≥ 1), do

**wordList** ← split(**Node.innerText**, delimiter) ;

for each **word** in the **wordList** (Length($L_w$) ≥ 1),do

if **word** not in **ruleExclude**, then

if **word** in **ruleContainAll**, then

**variantList** ← **word**;

elseif **word** in **ruleOptional**, then

**variantList** ← **word**;

End For;

Using the rules enables to narrow down to a more realistic list of variants that matches the specification set.

*4) Transforming naming convention:* Moreover, the naming convention within an organization also lead to ambiguity in the identification of components when the number is large.

*a) Naming convention*

A list for a naming convention for a distributed business process is illustrated below

```
"WorkSpace DOMain GRouP PRoJect
FunctionBlock PartNo VARiant"
```

*b) Algorithm to transform names*

The string described above characterizes the naming convention within an organization, the scattered software components can be organized by splitting the names along a delimiter and transforming them into a hierarchical structure, then the following algorithm can be devised:

**nameConv** ← List described in Section IV-C4a;

**SWcompNameList** ← doc.readCompName("*");

for each **SWcompNameConv** in

**SWcompNameList** (Length($L_{nc}$) ≥ 1), do

**SWcompNameSplit** ←

split(**SWcompNameConv.name**, delimiter);

for each **SWcompNamePart** in

**SWcompNameSplit** (Length($L_{sn}$) ≥ 1), do

if not exist **SWcompNamePart0**, then

**RootElementNode** ← **SWcompNamePart**;

else

**ParentElementNode** ← **RootElementNode**;

for each **SWcompNamePart** in

**ParentElementNode.ChildNodes**

(Length($L_{cn}$) ≥ 1),do

if not exist **SWcompNamePart**, then

**ParentElementNode.addChildNode** ←

**SWcompNamePart**;

else

**ParentElementNode** ←

**ParentElementNode.ChildNodes**;

End For;

This algorithm can be further extended to assist the user to identify, search, and construct the names and display them as a hierarchy. A procedure to navigate and simplify the construction of such names will enable the user to quickly build long names uniformly over the entire project.

TABLE I. SUMMARY OF ELEMENTS IN SCHEMA OF THE SAMPLE DATA SET

| Schema | |
|---|---|
| **Description** | **Count** |
| Total elements collection | 171 |
| Components list | 23 |
| Features vector | 12 |

TABLE II. SUMMARY OF ELEMENTS IN PROJECT OF THE SAMPLE DATA SET

| Project | | |
|---|---|---|
| **Description** | **Count** | **Category** |
| Total elements | 32909 | all |
| Components | 1583 | 23 |
| Features within components | 13353 | 12 |

### D. Evaluation

A prototype of the architecture presented here has been implemented. The case studies targeted the design of model-based software components firstly in an industrial use case where the project model was developed using the design tool ESCAPE® [24], and secondly in a case study targeting the execution of specific paradigms based on the naming convention of AUTOSAR® [25].

The number of elements in schema and project of sample data set that was used to evaluate the implementation is summarized in Table I and Table II, respectively. It consists of a total of 32,909 elements. Of these a total of 1583 elements signify components which are categorized into 23 categories that form the Component List is summarized in Table III, where as a total of 13353 elements that signify features which are categorized in 12 categories that form the Feature Vector is summarized in Table IV.

Three different approaches were adopted to evaluate and determine the performance with respect to matches.

TABLE III. COMPONENT LIST DERIVED FROM SCHEMA

| Component List | |
|---|---|
| **Description** | **Count** |
| CompoundFunction | 58 |
| HWFunction | 182 |
| SWFunction | 46 |
| Parameter | 6 |
| StructureElement | 50 |
| SWBubbleType | 130 |
| ParameterType | 5 |
| ParameterTypeTerminal | 8 |
| IntDataType | 14 |
| FloatDataType | 2 |
| TimeDataType | 1 |
| AliasDataType | 1 |
| VariantDataType | 4 |
| HWFunctionType | 46 |
| TypeInterface | 181 |
| FunctionTypeTerminal | 580 |
| HWTypeTerminal | 91 |
| StructureElement | 50 |
| DeviceMapping | 10 |
| DeviceType | 4 |
| BusCAN | 3 |
| BusSegment | 3 |
| MappedFunction | 108 |
| | 1583 |

TABLE IV. FEATURE VECTOR DERIVED FROM SCHEMA

| Feature Vector | |
|---|---|
| **Description** | **Count** |
| Name | 7500 |
| LongName | 0 |
| Description | 0 |
| ConnectionSegment | 537 |
| SourceTerminal | 538 |
| SinkTerminal | 538 |
| Interface | 292 |
| CompoundTerminal | 269 |
| HWTerminal | 292 |
| SWTerminal | 302 |
| Input | 1543 |
| DataType | 1542 |
| | 13353 |

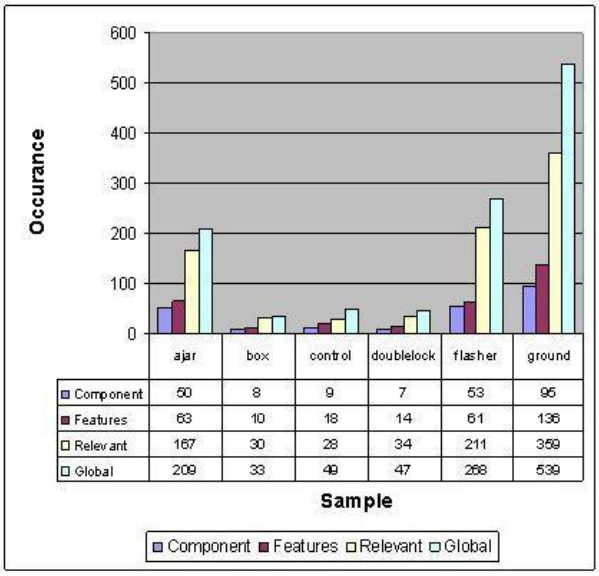


Fig. 10. Occurrence graph for a single element specification set.

- **Evaluation using a single element specification set**
  The first experiment was conducted on a single element specification set. A group of ten sets formed the input to determine the result set in both comprehensive (global) search and selective search as illustrated in Figure 10.
  The notion of comprehensive search is used, when scanning all occurrences of the specification set within projects, irrespective of whether they are components or features of those components. This can return a result set that contains false matches.
  The pattern of the results displayed similar behavior.

  **Observations**
  - The comprehensive search yields a result set that contains every occurrence of the specification set, even if these nodes do not characterize a component.
  - The nodes representing components yield a result set which is somewhat realistic, though these do not epitomize the complete set desired. This is often observed when the component nodes do not match, but their features collectively match the specification set.

- These nodes along with the feature set yield a more elaborate result set. A match contained by any node in a set of features would result in representing the component to which it belongs.

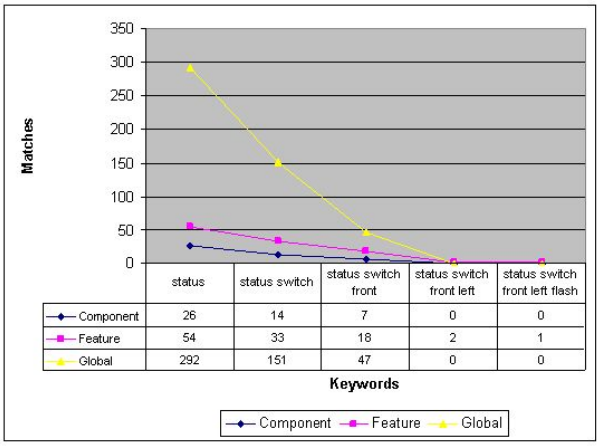


Fig. 11. Occurrence graph for multiple element specification sets.

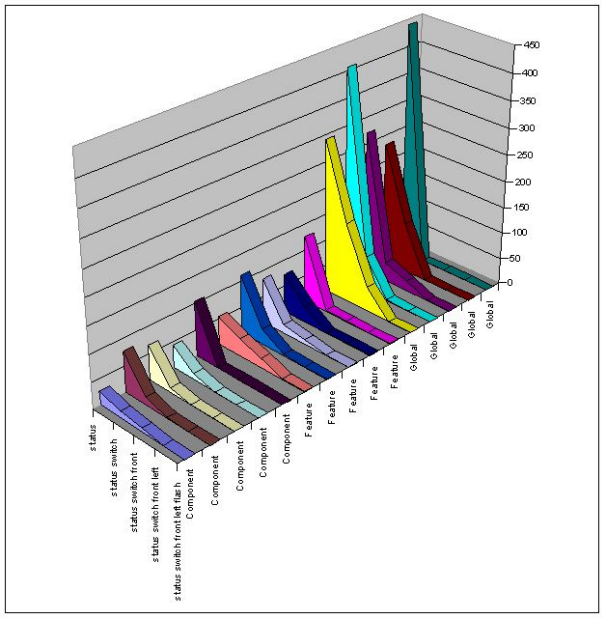


Fig. 12. Occurrence graph for different starting points.

- **Evaluation using multiple element specification set**
The second experiment was conducted using one up to seven element specification sets as a group illustrated in Figure 11.

  **Observations**
  - The comprehensive search often yielded large result sets, as it searches in individual nodes that are treated as atomic.
  - The exhibited behavior is similar to the varying size of the specification set. As observed in Figure 11, the selective component-feature search result set demonstrates a value when the size of specification set exceeds 3, because in this case the matches take place across the boundary of the feature within the component. On the other hand, the other methods return null result set as the search is only within the boundary of the element.
  - For any given size of specification set, the selective component-feature search returns a much smaller result set and is more precise.
  - Convergence is optimal with a specification set of size 3. If the size of the specification is too large, the result may be null for both methods as shown in Figure 11.

- **Evaluation using different starting points for elements in specification sets**
The third experiment was conducted searching for elements within specification sets using different starting points. Figure 12 depicts the result sets in comprehensive search and selective search.
To determine the effect of different starting points, a multiple-element specification set was used, where the orders of the elements were changed to obtain five sets.
The result set for this exhibits the same pattern as the two experiments above.

## V. CONTRIBUTIONS OF THIS APPROACH

- *Contribution 1: Model-based Variability Management for Complex Embedded Networks.*
The concept of Model-based Variability Management is proposed in the paper, which contemplates on the definition of a problem and specification of the cases. Furthermore the concept specified is used for feature extraction to extract spatial, functional, and name for the realization of new functionality. These models has been evaluated for data models in IV-D.

- *Contribution 2: A generic approach to envisage the identification of variability.*
The primary mechanism for determining commonality, allowing dynamic extension in the identification of variability of software components which are embedded in hierarchical model confirming to numerous tools like ESCAPE®, EAST-ADL®, UML® tools, SysML® specifications, and AUTOSAR®. The approach is based on the adaption of a formal mathematical model presented in the publication [10].

- *Contribution 3: An approach to visualize, navigate and simplify the unintelligible naming conventions.*
Mapping highly indecipherable naming conventions and transposing to hierarchical structures using predetermined delimiters, to assist the user to identify, search, and construct these names, comfortably displaying them as hierarchy, as well as having a procedure to navigate and simplify the construction of such names.

## VI. Conclusion

Managing variants is of utmost importance in today's large software bases as they reflect legal constraints, marketing decisions, and development cycles. As these software bases often grew from different sources and were developed by different teams using different tools it is in many cases very complicated if not nearly impossible to find artifacts that might be variants, both for historical reasons as for development purposes.

The algorithms presented here reflects both the capability to match keywords and to reflect the structure that characterizes a component. It can be applied directly to application engineering for identification of software component variants. Furthermore, it may also be applied for variability identification of software components in core assets of domain engineering in SPL. Our proposed method is capable of both aspects and therefore helps the developer to find matches even in large and heterogeneous databases.

The developed prototype is itself independent of a specific tool as it works on textual descriptions that typically are available in XML. The future work may comprise to extend the concept to specify and verify reusable components.

## References

[1] A. A. Frank and E. Brenner, "Variability identification by selective targeting of significant nodes," ICCGI 2012, The Seventh International Multi-conference on Computing in the Global Information Technology, 2012, pp. 148-153.

[2] C. Ebert and J. Salecker, "Guest editors' introduction: embedded software technologies and trends," Software, IEEE, Vol 26(3), 2009, pp. 14-18.

[3] A. S. Tanenbaum and M. Van Steen, "Distributed Systems: principles and paradigms (2nd Edition)," Prentice Hall, 2006.

[4] C. Ebert and C. Jones, "Embedded software: facts, figures, and future," Computer, IEEE Vol 42(4), 2009, pp. 42-52.

[5] D. Kum, G. Park, S. Lee, and W. Jung, "AUTOSAR migration from existing automotive software," The Proceedings of International Conference on Control, Automation and Systems, 2008, pp. 558–562.

[6] I. Foster and C. Kesselman, "The Grid: blueprint for a new computing infrastructure," Elsevier Series in Grid Computing. Morgan Kaufmann, second edition, 2004, pp. 672.

[7] B. L. Bayus, "Are product life cycles really getting shorter?" Journal of Product Innovation Management, Vol. 11 (4), 1994, pp. 300-308.

[8] S. Poole and M. Simon, "Technological trends, product design and the environment," Design Studies, Vol. 18 (3), 1997, pp. 237-248.

[9] A. Ericsson and G. Erixon, "Controlling design variants modular product platforms," Society of Manufacturing Engineers, Dearborn, MI, 1999.

[10] A. A. Frank and E. Brenner, "A generic approach for the identification of variability," ENASE2012, 7th International Conference on Evaluation of Novel Approaches to Software Engineering, 2012, pp. 167-172.

[11] T. W. Simpson, "Product platform design and customization: status and promise," Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol.18 (1), 2004, pp. 3-20.

[12] P. Child, R. Diederichs, F. H. Sanders, and S. Wisniowsk, "The management of complexity," Sloan Management Review, Vol. 33 (1), 1991, pp. 73-80.

[13] P. Clements and L. Northrop, "Software Product Lines: practices and patterns," Addison-Wesley, 2007.

[14] H. Gomaa and D. L. Webber, "Modeling adaptive and evolvable Software Product Lines using the variation point model," The Proceedings of the 37th Hawaii international Conference on System Sciences, 2004.

[15] E. Oliveira, I. Gimenes, and J. Maldonado, "A variability management process for software product lines," CASCON 2005, The conference of the Centre for Advanced Studies on Collaborative research, 2005, pp. 225 - 241.

[16] Product line hall of fame, "http://splc.net/fame.html," retrieved: 02,2013.

[17] F. Bachmann and P. C. Clements, "Variability in Software Product Lines," Technical Report -CMU/SEI-2005-TR-012, 2005.

[18] J. Bosch, "Design and use of Software Architectures: adopting and evolving a product-line approach," Addison-Wesley, 2000.

[19] L. A. Burgareli, Selma, S. S. Melnikoff, and G. V. Mauricio Ferreira, "A variation mechanism based on Adaptive Object Model for Software Product Line of Brazilian Satellite Launcher," ECBS-EERC 2009, First IEEE Eastern European Conference on the Engineering of Computer Based Systems, 2009, pp. 24-31.

[20] A. A. Frank and E. Brenner, "Model-based variability management for complex embedded networks," ICCGI 2010, The Fifth International Multi-conference on Computing in the Global Information Technology, 2010, pp. 305-309.

[21] A. A. Frank and E. Brenner, "Strategy for modeling variability in configurable software," PDES 2010, The 10th IFAC workshop on Programmable Devices and Embedded Systems, 2010, pp. 88-91.

[22] P. Heymans and J. Trigaux, "Software product line: state of the art," Technical report for PLENTY project, Institut d'Informatique FUNDP, Namur, 2003.

[23] M. Galster and P. Avgeriou, "Handling variability in software architecture: problem and implications," WICSA 2011, Ninth Working IEEE/IFIP Conference on Software Architecture, 2011, pp. 171-180.

[24] ESCAPE, "http://www.gigatronik2.de/index.php?seite=escape_produktinfos_de&navigation=3019&root=192&kanal=html," retrieved: 11,2012

[25] AUTOSAR, "http://www.autosar.org/download/conferencedocs/03_AUTOSAR_Tutorial.pdf," retrieved: 02,2013