Automated Generation of Product Variants in Mechanical Engineering

Bachelor Thesis

Submitted by:

Melanie Donabauer, 1155142

May 2015

Content

Abstract

For companies there is often the need to offer several products that are similar in their core functionality but differ slightly from each other. The reason why is, that each customer may have different requirements for such products. There exists a process called clone-and-own where existing products are usually used as references to create new variants from them. The problem with that approach is, that it has to be done manually which contributes to the reason why clone-and-own is very error prone. Therefore technologies are required that simplify the process of creating a wide variety of similar products.

Usually this is a topic in software engineering, but it is also of importance for mechanical engineering since it may be required to design and construct similar mechanical artefacts, e.g. variations of an existing robot with different arm lengths. There are little approaches that deal with these problems in the context of mechanical engineering. Therefore this thesis dwells on the issue whether it is possible to create several product variants from a mechanical artefact. For that purpose parsers will be implemented that take diverse files which describe parts of the mechanical product as input. These parsers will build an internal model and will write the relevant information back to a new file of the corresponding format. A crucial issue will be whether the produced output can be opened in the respective tools again and whether they may be reused for the parsers.

1 Introduction

This thesis deals with automatically creating new variants from existing products in the context of mechanical engineering which is usually a topic for software products, but as multiple similar mechanical artefacts need to be designed and constructed frequently, the desire for a product line comes up.

In the following chapters important terms are explained and an approach is described that allows to create new similar mechanical artefacts from existing ones.

1.1 Mechanical Engineering

The production of mechanical artefacts consists of several stages: product planning, engineering design, manufacturing, order management, production and procurement, customer delivery and service [1]. The design of such products is supported by several tools, e.g. CAD (computer-aided design) tools. The result of a designing process is usually represented in a STEP file which is the Standard for the Exchange of Product data model. It consists of geometric attributes, e.g. points, vectors, dimensions, etc., and non-geometric attributes.

1.2 Product Variants

Product variants of a product portfolio share common features that fit the needs of a specific market segment. Each variant is adapted according to the requirements of a particular customer. New product variants are usually added by the “Clone-and-Own” principle, where common functionality of the existing variants is merged and the required features are included. Another possible approach for creating product variants is Software Product Line Engineering [2].

1.3 Software Product Line Engineering

The aim of software product line engineering is to create several products that are similar to each other in their base functionality but can also provide different additional features depending on the requirements of the target group. New product variants can be added to the product line by selecting features that the customer desires [2].

In most cases it is necessary to plan the whole basic functionality and the set of features the product line should support beforehand. But this is nearly impossible since future customers might have slightly different needs than the product line can offer. Also the effort and the money spent on planning a software product line are usually very high. There are some companies that do not have a product line but still offer different variants of the same product. In those cases they start with an individual product that was copied and extended for different customers. Software product lines not only offer the benefit of easily creating new product variants but also provide means for easily maintaining them [3].

1.4 Clone-and-Own

Some techniques for creating a product portfolio require to build a configurable system, where parameters can be set depending on the customer's needs. Clone-and-own is a method that does not require the company to decide on the functionality of the product portfolio beforehand, but it may be extended incrementally, depending on the customer's requests. During the process several product variants are created that are similar to each other in their core functionality, but differ in features for each particular customer. The problem with this principle is that it does not provide an approach for systematic reuse of the product's features and does not address the problems with resulting feature interaction [2]. Also, it is difficult to maintain product variants as their number increases. Another issue is that adding new features that are common by more than one product variant needs to be done for all of them which comes with a huge effort.

1.5 ECCO

Instead of developing a software product line, most companies tend to create product variants by applying clone-and-own on two or more existing products by copying the relevant parts and extending the functionality with the desired features. The problem with this approach is that each product variant has to be maintained separately, bug fixes are not distributed for all product variants and features that are common for some variants have to be implemented for all of them. Maybe these disadvantages do not impose a problem if there are only a few product variants that have to be maintained, but as the number increases this task becomes even more difficult and vast. However, there are various reasons why companies decide to not rely on software product lines, one might be that the products are required to run on different operating systems. While dealing with the problems of this approach, an interest in creating a product line without much effort might arise [3].

ECCO [2] – Extraction and Composition for Clone-and-Own) – is a project that allows a systematic application of clone-and-own which enables the software engineer to incrementally generate a product portfolio from existing product variants. This tool deals with the two main challenges of clone-and-own: 1) extracting features from existing variants and 2) how the features depend on each other. These steps are usually difficult for human operators because it takes a huge effort to identify the differences and dependencies.

In the first step ECCO takes the existing product variants as inputs and compares them by analyzing what they have in common and in which functionalities they differentiate from each other. In the next step the software engineer is required to select the desired features that should be applied to the new product variant. After that ECCO searches for the relevant artifacts for the copying part and then supports the software engineer to manually complete the composition part by indicating where artifacts are missing or require modifications, e.g. if features did not occur in any product variant and still need to be implemented.

Before, it was mentioned that ECCO is able to incrementally create a product portfolio. This can be achieved by adding the newly generated product variant to the list of existing products and start the tool from the beginning.

1.6 Objectives

This thesis aims to derive similar but also diverse variants of existing mechanical products, or robot variants in particular, using ECCO - Extraction and Composition for Clone-and-Own [2]. A variant may consist of different files that describe the composition of the robot.

The approach consists of three steps: (1) Parsing of current file, (2) generation of an abstract syntax tree (AST), (3) writing the AST to a new file. For this purpose parsers for selected file types need to be implemented which should be able to generate abstract syntax trees based on the ECCO meta-model. For the purpose of this thesis there should be parsers that support Microsoft Excel files and STEP files. Ultimately the created abstract syntax tree is used as input for ECCO which is then required to create a new product variant.

Major issues of this thesis are to answer whether the variants that ECCO generates are readable by the particular tools and whether they can be used to generate new product variants.

2 Related Work

This section gives an overview on related works that describe similar approaches for creating product lines.

2.1 Ziadi et al.

[4] propose a three-step approach to identify common and variable features from the source code of product variants. At first a model is constructed from the input source code of each variant which is split into atomic parts in order to compare the variants with each other. Then an algorithm is applied that identifies the common features from the parts that were found in the first step. The last step requires to manually sort out the features that were identified as common by mistake. Additionally, features that are missing because they were not considered as common by the algorithm have to be added.

2.2 Xue et al.

In [5] an approach called Feature Location in Product Variants is introduced. It offers the possibility to locate code units which implement a certain feature that is shared by different product variants. This approach is split into several parts. First a concept called Software Differencing is applied in order to find common features among the product variants. After that Formal Concept Analysis is used that analyses shared functionalities and differences of product variants in order to group their features. To locate the code units which are responsible for implementing a particular feature, an information retrieval approach called Latent Sematic Indexing is used. This approach provides better results than when using information retrieval methods alone. The reason why information retrieval should not be applied without the pre-processing steps described before is, that the features and their implementation may be different for each product variant.

2.3 Frenzel et al.

[6] propose an approach for constructing product lines from similar product variants that is based on Gail Murphy's reflexion method. The reflexion method can be used to reconstruct a static architectural view, i.e. it describes the static components as well as their interfaces and dependencies on each other. Frenzel et al. take advantage on the similarity of product variants and apply the extended reflexion method incrementally on the product variants in order to compose the product variants into a software product line. They also take into account that their implementations and architectures are alike to a certain degree. This way they are able to compare the product variants incrementally and use existing mappings from previous variants.

2.4 Rubin et al.

[7] present a framework that is able to consolidate legacy product variants into software product lines that are based on common features between the variants. Their approach relies on locating and retrieving common and diverse artifacts. Composing the legacy products into product lines is done by comparing the features of the product variants and merging those that are similar to each other. Their framework allows various source meta-models as input, like UML, EMF, etc.

3 Approach

This section describes an approach for creating new variants from existing products in the context of mechanical engineering.

3.1 General

The figure below describes the overall process that is used for the approach.

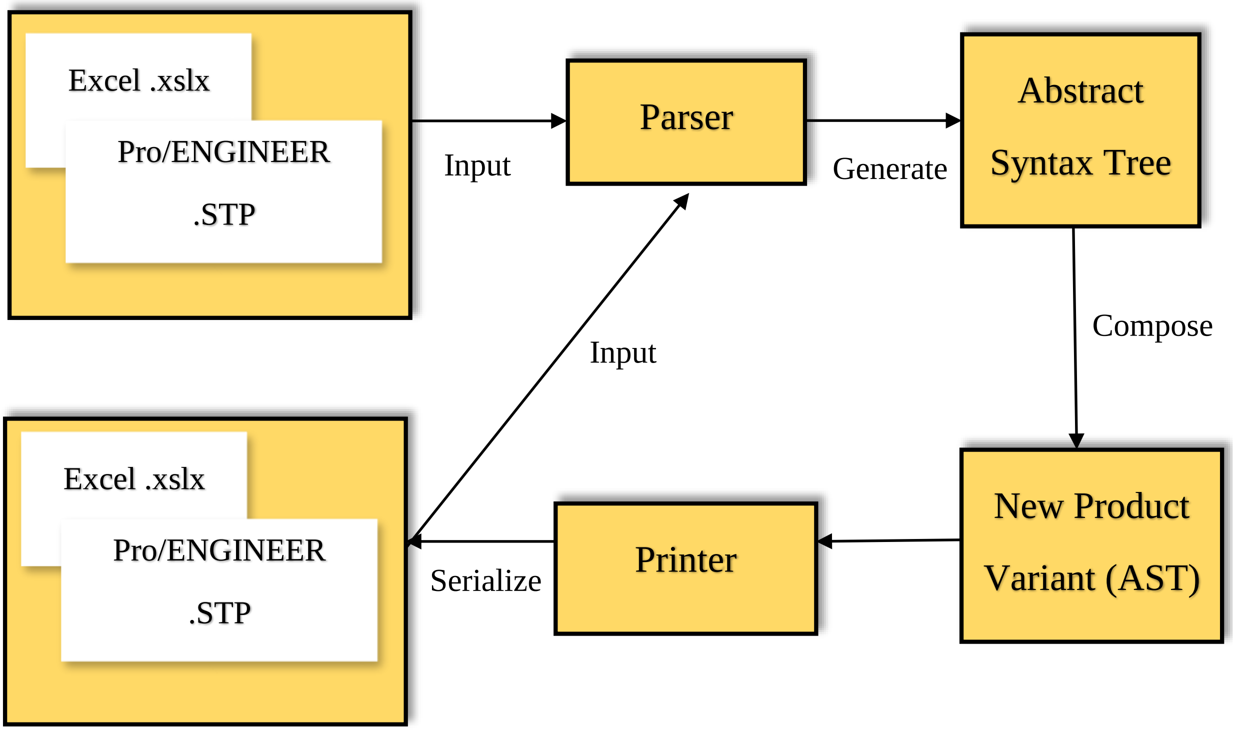


Figure 1: Overall process of the approach

Two parsers are implemented in order to show that product variants can be derived from mechanical artefacts as well. At first a parser for Microsoft Excel files which contain computations concerning the mechanical model is constructed. The other parser reads files that are obtained from the CAD tool ProENGINEER. For this purpose the STEP file format was considered as suitable since it has a simple and consistent specification.

Based on the ECCO meta-model an Abstract Syntax Tree (AST) is generated from the input file. The AST is provided as input for ECCO in order to extract associations and features between the product variants. Now ECCO is able to compose a new product variant from the existing ones which is again represented as an abstract syntax tree. In the last step the newly defined product variant is printed to a new file which should be valid for its respective file format, i.e. it should be possible to open it in the designated software tool and the system should be able to use the file as input in the next cycle.

3.2 Parser

In this section the implementation of the two parsers and the design of the data structure are explained.

ECCO meta-model

The figure below illustrates the ECCO meta-model that is required for comparing various product variants.

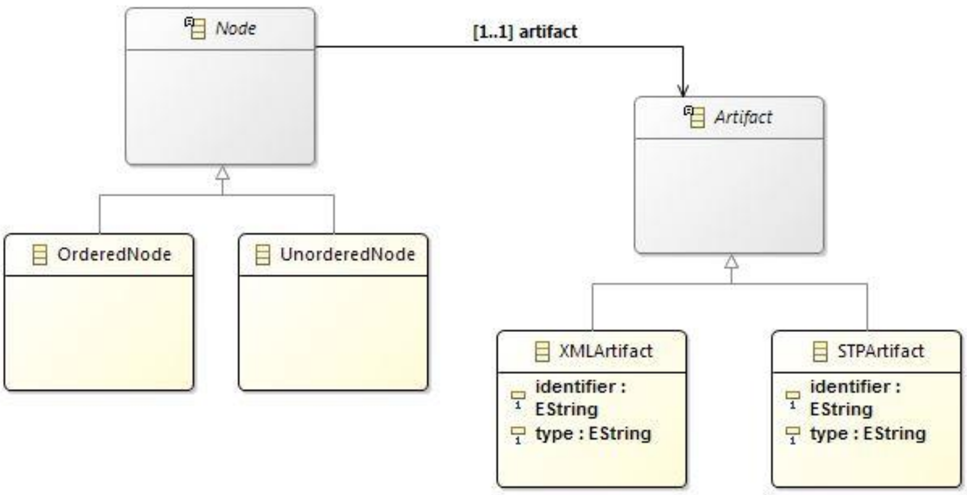


Figure 2: ECCO meta-model

The ECCO meta-model consists of the type "Node" and its derivatives “OrderedNode” and “UnorderedNode”. An “OrderedNode” is used if its children require a specific sequence, respectively “UnorderedNode” is chosen if the order is not of importance. Furthermore a node has to contain an artifact that reflects its identifier and type. An artifact may either be an “XMLArtifact” or a “STPArtifact” depending on whether an Excel file or a STEP file is parsed.

3.2.1 Microsoft Excel

A Microsoft Excel file is based on XML which makes it possible to export it as XML spreadsheet. For that reason it was decided to implement a general parser for XML in order to support other XML based file formats as well. Several test files were provided for a testing environment.

Architecture

At first it was intended to use the compiler generator Coco/R for the parsing part but it was soon clear that this approach had some disadvantages with respect to existing XML parsers. First of all it is not possible to validate a XML file with its XML schema. Another issue is that there is no way to automatically resolve references between elements. Therefore it was decided to use an existing API to process the XML based Excel files instead of implementing a parser with a compiler generator. There are three possible XML parsers that seem to be appropriate for this kind of task: (1) The streaming API for XML (StAX), (2) the Simple API for XML (SAX) and (3) the Document Object Model parser.

Table 1 compares the three XML APIs that were considered for the Excel parser.

Feature	**StAX**	**SAX**	**DOM**
API type	pulling	pushing	abstract syntax tree
Read XML file	✓	✓	✓
Elements access	sequential	sequential	globally

Write XML file	✓	×	✓
Add, Update, Delete Elements	×	×	✓

Table 1: Comparison of XML APIs

The Java API for XML Processing (JAXP) [8] is an API for Java to process XML files. It offers the developer the possibility to parse, transform, validate and query XML files using different APIs, like DOM, SAX, StAX, XPath, etc.

When using DOM the whole XML document is buffered in memory as an abstract syntax tree. However, the advantage of this API is that it provides simple access to elements in the tree. DOM is also convenient for adding, updating or deleting elements in the tree [9]. It was decided against the use of DOM because there is no need to add, update or delete elements in the original XML input file. There is also the issue that ECCO requires a predefined data structure, i.e. the nodes of the abstract syntax tree need to be of type "OrderedNode" or "UnorderedNode" that in return have to contain an artifact of the type "Artifact". Therefore it does not make much sense to maintain two abstract syntax trees in memory.

The two streaming APIs SAX and StAX only offer sequential access because the elements are not kept in memory. For that reason the XML file has to be parsed again to access a previous element. Also there is no possibility to add, update or delete elements to the XML file directly. However, there is a way to go around it, namely to store the whole XML file in a separate data structure which may be modified and written back to the file [9]. For the purpose of this thesis there is no need to read previous parsed elements and the parsers do not require to add, update or delete elements in the XML file.

The stream API StAX was preferred to SAX because it has a better usability for nested XML files and provides functionality for writing XML files.

For testing purpose it was decided to use a short example XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
  <Book nPages="300">
    <Title>0001</Title>
    <Author idref="P000" />
```

```
  </Book>
  <Book nPages="400">
    <Title>0002</Title>
    <Author idref="P234" />
  </Book>
  <Person id="P234" age="30">
    <FirstName>PersonFirstName</FirstName>
    <LastName>PersonLastName</LastName>
  </Person>
  <Child id="P000">
    <Name>child</Name>
  </Child>
</Root>
```

Figure 3: Example XML file

The example XML file in Figure 3 illustrates that it is possible to represent references between elements. An element with the attribute “id” can be referenced by any element that contains the attribute “idref” with the same value. Those references are used to establish dependencies between artifacts in ECCO. In order to check the validity of the XML document a corresponding XML schema was designed that defines elements that may contain IDs and references. Moreover it was decided to specify in the schema whether an element is supposed to be represented as an ordered or an unordered node. Another issue was that there should be a way to choose which element or attribute names are used in the identifier of an artifact. This is especially important if elements or attributes with the same name exist in the XML document. In Figure 3 the XML schema is shown that was designed for the XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="unqualified">

  <xsd:element name="Book">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="Title" type="xsd:string"/>
            <xsd:element name="Author">
               <xsd:complexType>
                  <xsd:attribute name="idref" type="xsd:IDREF"
                                 use="required"/>
               </xsd:complexType>
            </xsd:element>
         </xsd:sequence>
         <xsd:attribute name="nPages" use="required"/>
         <xsd:attribute name="ordered" fixed="true" use="prohibited"/>
      </xsd:complexType>
   </xsd:element>

 <xsd:element name="Person">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="FirstName" type="xsd:string"/>
```

```
                    <xsd:element name="LastName" type="xsd:string"/>
          </xsd:sequence>
          <xsd:attribute name="id" type="xsd:ID" use="required"/>
         <xsd:attribute name="age" use="required" />
         <xsd:attribute name="ordered" fixed="true" use="prohibited"/>
      </xsd:complexType>
       <xsd:key name="ElementIdentifier">
                    <xsd:selector xpath="//Person"/>
                    <xsd:field xpath="FirstName"/>
                    <xsd:field xpath="LastName"/>
       </xsd:key>
 </xsd:element>

 <xsd:element name="Child">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="Name" type="xsd:string"/>
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID" use="required"/>
      </xsd:complexType>
 </xsd:element>

 <xsd:element name="Root">
       <xsd:complexType>
          <xsd:sequence>
              <xsd:element  ref="Book" minOccurs="1" maxOccurs="unbounded" />
              <xsd:element  ref="Person" />
              <xsd:element  ref="Child" />
          </xsd:sequence>
          <xsd:attribute name="ordered" fixed="true" use="prohibited"/>
   </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Figure 4: Corresponding XSD file

In the XML schema IDs have to be of type “xsd:ID” and references are supposed to be of type “xsd:IDREF”. Some XML editors like XML Notepad 2007 are able to validate references that are specified in the XML schema.

```
<xsd:attribute name="idref" type="xsd:IDREF" use="required"/>
```

Elements may be specified to be ordered or unordered, therefore the attribute “ordered” was introduced in the XML schema. In the field “fixed” one can define whether the element is ordered (i.e. “true”) or unordered (i.e. “false”). The field “use” is specified with “prohibited” in order to avoid errors with a XML editor, since this attribute should not be used in the XML file.

```
<xsd:attribute name="ordered" fixed="true" use="prohibited"/>
```

In order to define the identifier of an element the XML schema key element with the name "ElementIdentifier" is used. The selector element describes the element for which it is applied to. The field elements determine the names that are used for the identifier. Furthermore it is important to be aware that the key element should be written in XPath syntax based on the root element, e.g. "//Person" searches for the first occurrence of such an element starting from the root element.

```
<xsd:key name="ElementIdentifier">
	<xsd:selector xpath="//Person"/>
	<xsd:field xpath="FirstName"/>
	<xsd:field xpath="LastName"/>
</xsd:key>
```

Streaming API for XML

StAX offers two possible ways for parsing XML documents. The first one is the cursor-based variant which is of advantage if efficiency is important. The alternative is event iterator-based where separate objects are created which offers the possibility to process the XML document in an object oriented way, i.e. code reusability and modularity can be improved. It was decided to use the latter approach for the purpose of this thesis.

The XML file is parsed based on specific tokens that allow to create the respective objects, e.g. a "StartElement" can be created from the current event if the parser encounters a START_ELEMENT token. The following table shows a subset of the tokens that are supported by StAX and that were considered to be useful for implementing the XML parser.

Token	Example
START_ELEMENT	<FirstName>
END_ELEMENT	</FirstName>
CHARACTERS	PersonFirstName
ATTRIBUTE	nPages="400"
START_DOCUMENT	<?xml version="1.0" encoding="UTF-8"?>
END_DOCUMENT	
PROCESSING_INSTRUCTION	

Table 2: StAX example of tokens

The following figure shows which classes are needed to implement a XML parser and also illustrates how to create a “StartDocument” object from the current event.

```
XMLInputFactory factory = XMLInputFactory.newInstance();

XMLEventReader eventReader = factory.createXMLEventReader(new

FileReader(xmlFile));
while(eventReader.hasNext()) {
        XMLEvent event = eventReader.nextEvent();

        switch(event.getEventType()) {
                case XMLStreamConstants.START_DOCUMENT:
                        StartDocument startDocument = (StartDocument) event;
                        break;
        }
}
```

Figure 5: StAX Java example

Implementation

The abstract syntax tree is composed as follows: The topmost node is the root element, the attributes and all other elements are its children or children of sub elements. As already mentioned, it depends on the attribute "ordered" whether the element is of type “OrderedNode” or “UnorderedNode”. The attributes of an element are added as unordered nodes to their parent.

The references between elements that were mentioned before are established with “ArtifactReference” objects that contain a source artifact and a target artifact. These references are reflected in the associations.

As mentioned earlier XPath is used for composing the identifier of the elements. The only disadvantage of that approach was that the DOM API had to be used because otherwise the whole document would have to be parsed again in order to find the specific element. But since the object tree is not needed after evaluating the XPath expression there is no need to keep it in memory.

3.2.2 Pro/ENGINEER

Pro/ENGINEER is a 3D CAD software tool that enables the construction and simulation of 3D models.

The design data of mechanical products is usually stored in STEP files which are based on the ISO standard 10303-21, the Standard for the Exchange of Product data model. The ISO standard offers compatibility on different computer systems and therefore enables simple exchange and manipulation of mechanical design data [1].

Architecture

While implementing a parser for the STEP file format two possible options came up: (1) using the JSDAI API [10], (2) building a compiler for the STP syntax with the compiler generator Coco/R [11]. JSDAI provides several libraries for reading, writing and runtime manipulation of object oriented data that are defined by an EXPRESS based data model, among them a parser for ISO 10303 files. Even though the JSDAI homepage offered plenty of tutorials, examples and documentations that described how to deal with the tool, the library did not work properly. For this reason it was decided to use Coco/R for implementing a parser for the STP file format.

Coco/R

The compiler generator Coco/R [13] was developed by the Institute for System Software of the University of Linz. This tool enables the developer to generate a scanner and a parser for a specific language or file format by using an attributed grammar as input. The attributed grammar is written in the compiler generator language Cocol/R that is provided by Coco/R. The Extended Backus-Naur Form by Niklaus Wirth [12] which can be used to define a context-free grammar, is the basis for the compiler language. An EBNF may contain terminal and non-terminal productions where a terminal production rule consists of other terminal or non-terminal tokens. Summarized, Cocol/R offers the possibility to easily implement a compiler for a programming language or a simple file format. It consists of two parts: (1) the scanner that is responsible for reading the input stream and transforming characters into the declared tokens, (2) the parser that receives the parsed tokens from the scanner and checks whether they are valid.

The description of the compiler may contain a set of characters that will be supported by the scanner. Additionally a list of tokens is required which indicate which character sequences will be transformed into the respective token. Parsing invalid characters or tokens lead to abortion of the process. The developer also has to specify the production rules that define the structure of the language.

The following figure describes the overall structure of the compiler description.

```
Cocol =
  [ Imports ]
  "COMPILER" ident
  [ GlobalFieldsAndMethods ]
  ScannerSpecification
  ParserSpecification
  "END" ident '.'
  .
```

Figure 6: Coco/R compiler description

At the beginning of the description the developer may import libraries and other external classes. Further variables and methods that the parser needs to implement can be added. In the next section the overall compiler is defined containing all supporting characters, tokens and production rules. A production rule defines the structure of a non-terminal symbol, i.e. it describes in which order other non-terminal or terminal symbols may appear. There must be production rules for all stated non-terminals. Each production is transformed to a method that parses the respective tokens.

The grammar of the compiler must be LL(1) compatible, i.e. productions are parsed from left to right and no alternative may start with the same token. If the compiler description contains such LL conflicts then Coco/R reports them. Those conflicts can simply be resolved by using LL resolvers. By checking semantic information the parser is able to decide which alternative should be taken, e.g. the developer may implement a Boolean function that looks ahead which tokens appear and returns true for the respective alternative.

```
IF(isTrue())
(
    …
)
```

Figure 7: LL(1) conflict resolver

In the figure below the symbols that may be used for defining production rules are described:

<table>
<tr><th>symbol</th><th>meaning</th><th>example</th><th></th></tr>
<tr><td>=</td><td>separates the sides of a production</td><td>A = a b c .</td><td></td></tr>
<tr><td>.</td><td>terminates a production</td><td>A = a b c .</td><td></td></tr>
<tr><td>|</td><td>separates alternatives</td><td>a b | c | d e</td><td>means a b or c or d e</td></tr>
<tr><td>( )</td><td>groups alternatives</td><td>(a | b) c</td><td>means a c or b c</td></tr>
<tr><td>[ ]</td><td>option</td><td>[a] b</td><td>means a b or b</td></tr>
<tr><td>{ }</td><td>iteration (0 or more times)</td><td>{a} b</td><td>means b or a b or a a b or ...</td></tr>
</table>

Figure 8: Coco/R meta symbols

Production rules may be parameterized with attributes which are written between the characters "<" and ">". In the following some examples are illustrated that describe the use of attributes:

```
Rule1<String text> = … .
Rule2<out String text> = … .
```

Figure 9: Definition of production rules

The difference between Rule1 and Rule2 is that the first one uses the String "text" only as an input parameter and the resulting method has the return type void. The last rule returns a newly composed String “text”.

It is also possible to add semantic actions to production rules by enclosing them in the characters "(." and ".). Semantic actions are pieces of code written in the relevant target language. They represent statements which define what the parser should do when parsing the respective token, i.e. in the generated parser class the statements are positioned after the validation of the parsed token. Semantic actions may also access variables and methods defined in the parser or in imported classes. The code is not checked by the compiler generator but is simply copied to the generated parser. Therefore it may be necessary to revise the code and run Coco/R again. The most outstanding advantage of semantic actions, at least in the case of this thesis, is that they can be used to create the abstract syntax tree in an easy way, i.e. it incrementally builds a model of the input file while parsing it.

Implementation

The ISO standard 10303-21 file format is composed of a header and a data section. The header section is declared with the keyword “HEADER”. It includes the definition of a file description and some file properties like the file name, time stamp, author and organization. The file schema defines the EXPRESS schema which provides the format of the data section. EXPRESS is a language for modelling product data which is specified in ISO 10303. The data section constitutes the graphical model of the STEP file, i.e. a list of entities is defined where each entity may reference to others in order to build graphical shapes and objects. The following example shows the header section and the data section with a subset of possible entities.

```
ISO-10303-21;
HEADER;
FILE_DESCRIPTION((''),'2;1');
FILE_NAME(
/* name */                    'PARAMETRICROBOT_ASM',
/* time stamp */              '2014-06-03T',
/* author */                  ('virtualbox'),
/* organization */            (''),
/*  pre-processor_version  */  'PRO/ENGINEER  BY  PARAMETRIC  TECHNOLOGY
CORPORATION, 2011080',
/*  originate  system  */       'PRO/ENGINEER  BY  PARAMETRIC  TECHNOLOGY
CORPORATION, 2011080',
/* authorization */  ''
);
FILE_SCHEMA(('CONFIG_CONTROL_DESIGN'));
ENDSEC;
DATA;
#2=DIRECTION('',(0.E0,0.E0,1.E0));
#3=VECTOR('',#2,5.E2);
#4=CARTESIAN_POINT('',(-2.5E2,0.E0,-2.5E2));
#5=LINE('',#4,#3);
#6=DIRECTION('',(1.E0,0.E0,0.E0));
#7=VECTOR('',#6,5.E2);

…..

#1367=(GEOMETRIC_REPRESENTATION_CONTEXT(3)GLOBAL_UNCERTAINTY_ASSIGNED
_CONTEXT((
```

```
#1366))GLOBAL_UNIT_ASSIGNED_CONTEXT((#1360,#1364,#1365))REPRESENTATIO
N_CONTEXT(
'ID6','3'));
#1371=AXIS2_PLACEMENT_3D('',#1368,#1369,#1370);
#1375=PRODUCT_DEFINITION('design','',#1374,#1344);
ENDSEC;
END-ISO-10303-21;
```

Figure 10: STEP file example

Each entity is defined with a unique identifier specified in the format “#x” where x may be any positive integer less than 2^{63}. These unique identifiers can be used to reference other entities. Entities are declared with a name definition and some subsequent attributes enclosed in parentheses to parameterize them, e.g. “#4=CARTESIAN_POINT(‘’,(-2.5E2, 0.E0,-2.5E2))”. The file format also includes some special attributes like “$” which indicates that the attribute value is unset. Derived attributes that were declared in the super type are marked with the character “*”. STEP also supports enumerations, Boolean and logical values which are enclosed in dots, e.g. “.FALSE.”. When implementing the STEP parser with Coco/R those characteristics have to be considered.

The internal data structure is defined with the ISO standard as a root node with its children header node and data node, which contain properties of the header section and respectively the entities of the data section. All nodes except for the ISO standard, the header and data node are unordered because the sequence of their children does not matter. At first view such a STEP file looks rather flat, i.e. there are less parent-child relationships than for the XML abstract syntax tree. But there are a lot of references between the entities, which can be used for creating a little bit more structure in the tree. So if an entity references another entity a parent-child relationship may be established between them, provided that it is not referenced by any other entity.

In the following an overview of the compiler description for ISO 10303-21 is given. The first part to specify is a set of characters that may appear in simple text. For this purpose the symbol “char” is declared which allows any character except for carriage return, linefeed, and some other characters which will be used as tokens (see later).

```
CHARACTERS
/* supported characters */
    tab             = '\u0009'. /*  9 = tabulator */
    lf              = '\u000a'. /* 10 = line feed */
    cr              = '\u000d'. /* 13 = carriage return */
    char              = ANY - cr - lf - ';' - ',' - '\'' - '(' - ')' - '#' - '=' - '*' - '$'.
```

Figure 11: Grammar - Supported Characters

As mentioned before tokens are the terminal parts of production rules. They specify the string literal that may be read by the parser.

One of the most important tokens declared in the compiler description is “characterSequence” which responds to one character or a sequence of characters. Other important tokens are responsible for determining the beginning and end of the standard, the header and data section, as well as some entities for the header section. There were also defined some tokens for enumerations and Boolean values that appeared in the test files. The rest of the tokens are used for general entities like “CARTESIAN_POINT”.

However, not all entities and attributes which are available in the standard were defined in the compiler description since that would exceed the extent of this thesis. The following figure shows a list of a subset of the tokens that are supported by the implemented parser.

```
TOKENS
    /* terminal tokens */
    characterSequence   = char {char} .

    isoStandardStart    = "ISO-10303-21".
    header              = "HEADER" .
    fileDescription     = "FILE_DESCRIPTION" .
    fileName            = "FILE_NAME" .
    fileSchema          = "FILE_SCHEMA" .
    endSection          = "ENDSEC" .
    data                = "DATA" .
    isoStandardEnd      = "END-ISO-10303-21" .

    true            = ".T." .
    false           = ".F." .
    made            = ".MADE." .
    milli           = ".MILLI." .
    metre           = ".METRE." .
    radian          = ".RADIAN." .
    steradian       = ".STERADIAN." .
    unspecified     = ".UNSPECIFIED." .
    ahead           = ".AHEAD." .
    pCurveS1        = ".PCURVE_S1." .
    pCurveS2        = ".PCURVE_S2." .
    continous       = ".CONTINUOUS." .

    /* Geometric Shapes */
    direction               = "DIRECTION" .
    vector                  = "VECTOR" .
    cartesianPoint          = "CARTESIAN_POINT" .
```

Figure 12: Grammar - Subset of supported tokens

Of course there are further supported entities that are not shown in Figure 12. The next step is the definition of production rules for the tokens that were declared previously. The figure below demonstrates an example of the definition of a production rule for one of the supported entities.

```
Vector<out Node vectorNode>                  (. String text = "",  type = "";
                                                StringBuilder identifier = new StringBuilder();
                                                List<Node> referenceNodeList = new ArrayList<>();
                                                List<String> unresolvedHashTags = new ArrayList<>();
                                             .)
    =   vector                               (. type = t.val; .)
        leftBrace
        Text<out text> /* description */     (. identifier.append(text); .)
        comma                                (. identifier.append(t.val); .)
        ParseReferences<identifier, referenceNodeList, unresolvedHashTags>
        comma                                (. identifier.append(t.val); .)
        characterSequence /* length */       (. identifier.append(t.val);  .)
        CreateNode<out vectorNode, identifier.toString(), type, referenceNodeList, unresolvedHashTags>
        .
```

Figure 13: Grammar – Vector production Rule

In this example a production rule for the entity “VECTOR” is shown. The method which is generated by Coco/R returns the created ECCO node that represents a “VECTOR” entity. Additional production rules which were necessary for parsing this entity are “Text”, “ParseReferences”, “ParseReferenceNode” and “CreateNode”.

```
Text<out String text>                  (. StringBuilder sb = new StringBuilder(); .)
    =   simpleQuote                    (. sb.append(t.val); .)
        [
            characterSequence          (. sb.append(t.val); .)
            [
                hash                       (. sb.append(t.val); .)
                characterSequence          (. sb.append(t.val); .)
            ]
        ]
        simpleQuote                    (. sb.append(t.val); .)
                                       (. text = sb.toString(); .)
        .
```

Figure 14: Grammar - Text production rule

The production “Text” returns a character sequence that conforms to the STEP file standard. It encloses the text in simple quotes and might as well contain a hash token (i.e. “#”).

```
ParseReferences<StringBuilder sb, List nodeList, List unresolvedHashTags>          (. String hashTag = "";
                                                                                      Node referenceNode = null;
                                                                                   .)
    =   ParseReferenceNode<out referenceNode, nodeList, sb, unresolvedHashTags>
        {   IF(isReference())
            (
                comma                                                              (. sb.append(t.val);  .)
                ParseReferenceNode<out referenceNode, nodeList, sb, unresolvedHashTags>
            )
        }
        .
```

Figure 15: Grammar - Production rule for references

```
ParseReferenceNode<out Node referenceNode, List nodeList, StringBuilder sb, List unresolvedHashTags>        (. String hashTag = ""; .)
        =   HashTag<out hashTag>                        (. referenceNode = this.getNode(hashTag);
                                                            if(referenceNode != null) {
                                                                if(sb != null) {
                                                                    sb.append(referenceNode.getArtifact().getType() +
                                                                        "(" + referenceNode.getArtifact().getIdentifier() + ")");
                                                                }
                                                                if(nodeList != null)
                                                                    nodeList.add(referenceNode);
                                                            }
                                                            else {
                                                                sb.append(hashTag);
                                                                unresolvedHashTags.add(hashTag);
                                                            }
                                                        .)
.
```

Figure 16: Grammar - Production rule for parsing a reference node

“ParseReferences” parses references to other entities which is defined by the unique number of the referenced object, e.g. “#2”. It has to be checked whether the reference was already parsed, therefore all entities along with their entity number are stored in a hash table. If the entity already exists it can be replaced by the identifier of the referenced object, e.g. “DIRECTION('',(0.E0,0.E0,1.E0))”. If the corresponding entity has not appeared yet, then the number is stored as unresolved. As soon as the number is used by any following entities the reference will be resolved, i.e. the reference is replaced by the identifier.

```
CreateNode<out Node node, String identifier, String type, List referenceNodeList, List unresolvedHashTags>
    =   rightBrace                      (.  STPArtifact artifact = new STPArtifact(identifier, type) {
                                                @Override
                                                public String toString() {
                                                    STPArtifact artifact = (STPArtifact)this;
                                                    return artifact.getEntityArtifact();
                                                }
                                            };
                                            node = new UnorderedNode(artifact) {
                                                    @Override
                                                    public String toString() {
                                                        return this.getUnorderedNode();
                                                    }
                                            };
                                            if(referenceNodeList != null) {
                                                for(Object n : referenceNodeList) {
                                                    if(n instanceof Node)
                                                        this.setParentReference(node, (Node)n);
                                                }
                                            }
                                            if(unresolvedHashTags != null) {
                                                for(Object tag : unresolvedHashTags) {
                                                    if(tag instanceof String)
                                                        this.addReferenceNode((String)tag, node);
                                                }
                                            }
                                        .)
```

Figure 17: Grammar - Production rule for creating nodes

“CreateNode” generates an ECCO node from the given entity type and identifier. Additionally it establishes internal references but only for entities that were resolved.

The following table describes the parsing process for “#3=VECTOR('',#2,5.E2);” and the referenced entity “#2=DIRECTION('',(0.E0,0.E0,1.E0));”.

Token / Production	**Literal**	**Action**
vector	VECTOR	type = “vector”
leftBrace	(	
Text	‘’	identifier = “”

comma	,	identifier = “,”
ParseReferences	#2	identifier = “,DIRECTION('',(0.E0,0.E0,1.E0)) “
comma	,	identifier = “,DIRECTION('',(0.E0,0.E0,1.E0)),“
characterSequence	5.E2	identifier = “,DIRECTION('',(0.E0,0.E0,1.E0)),5.E2“
CreateNode	)	STPArtifact artifact = new STPArtifact(identifier, type); vectorNode = new UnorderedNode(artifact);

Table 3: Example of parsing process

3.3 Serialization

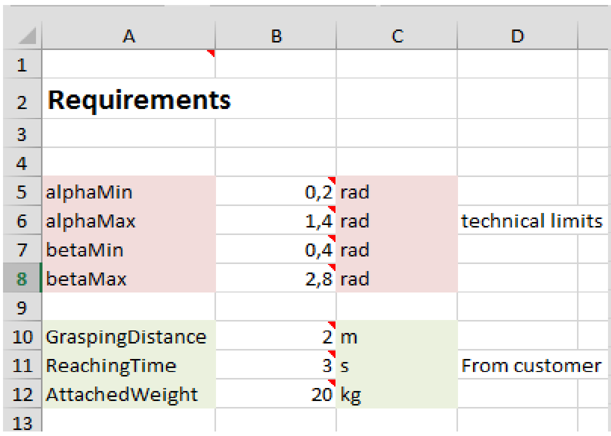
The last step of the overall process of this approach is to transform the ECCO model that was created in the previous chapter into a file that corresponds to the file format of the input file. The serialization can be performed in several possible ways. One way is to iterate through the model and retrieve the necessary information. Another option is to overwrite the String method of the ECCO Node and ECCO Artifact for the concrete objects. As can bee seen in Figure 17 it was decided to use the latter approach.

4 Evaluation

This section serves as basis for proving that the implemented parsers work as intended and deliver useful output that can be opened in the designated programs again. Several test files were given in order to build a testing environment in the context of mechanical engineering.

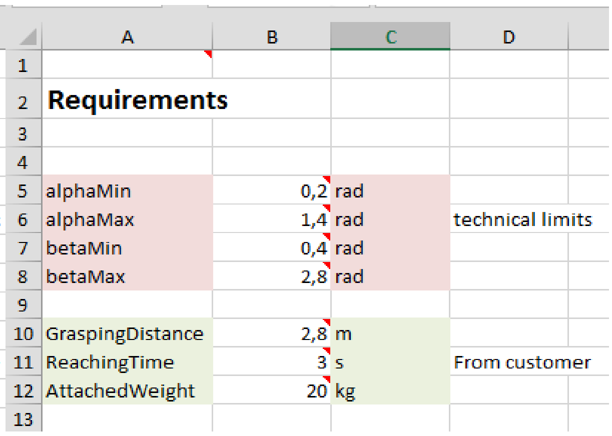
4.1 Microsoft Excel

In order to prove that the Excel parser works in the context of mechanical engineering, a testing scenario is described. The following figures show the requirements for two robot variants calculated in Microsoft Excel. These robots are similar to each other but differ in their grasping distance (2 metres versus 2.8 metres).



	A	B	C	D
1				
2	**Requirements**			
3				
4				
5	alphaMin	0,2	rad	
6	alphaMax	1,4	rad	technical limits
7	betaMin	0,4	rad	
8	betaMax	2,8	rad	
9				
10	GraspingDistance	2	m	
11	ReachingTime	3	s	From customer
12	AttachedWeight	20	kg	
13				

Figure 18: Robot variant 1



	A	B	C	D
1				
2	**Requirements**			
3				
4				
5	alphaMin	0,2	rad	
6	alphaMax	1,4	rad	technical limits
7	betaMin	0,4	rad	
8	betaMax	2,8	rad	
9				
10	GraspingDistance	2,8	m	
11	ReachingTime	3	s	From customer
12	AttachedWeight	20	kg	
13				

Figure 19: Robot variant 2

As already mentioned before, Excel files can be converted into XML spreadsheets. The following figure shows one of the robot variants opened in XML Notepad 2007.

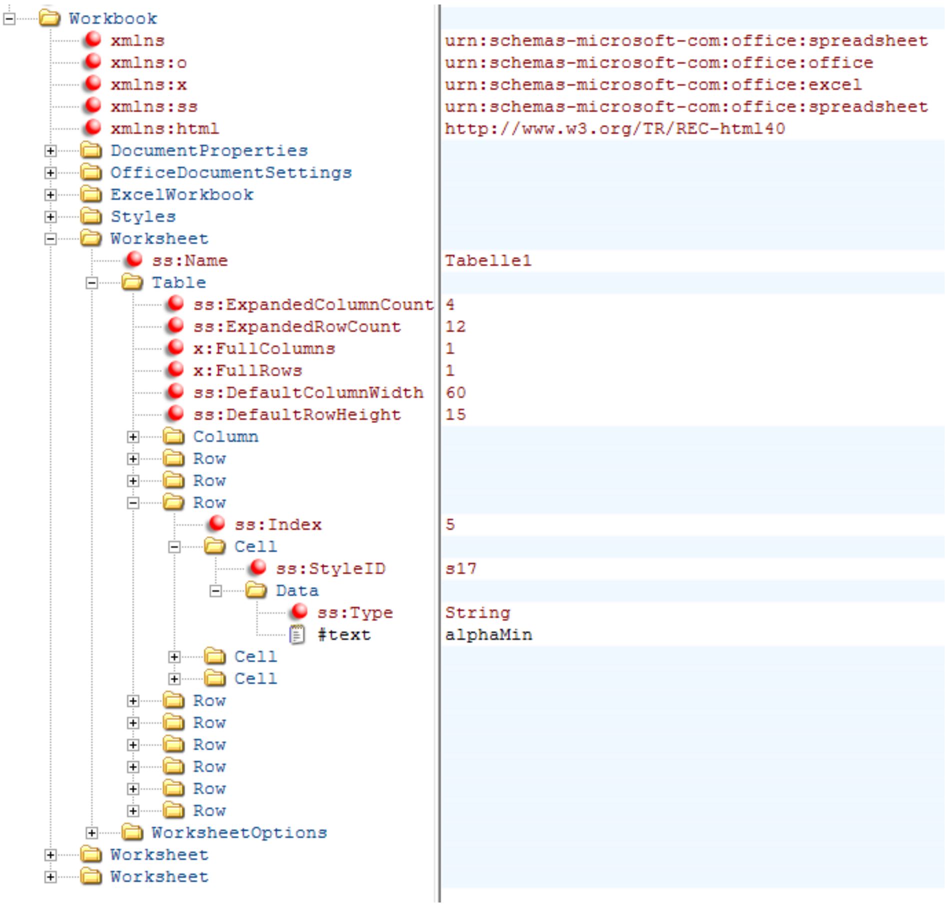


Figure 20: XML file shown in XML Notepad 2007

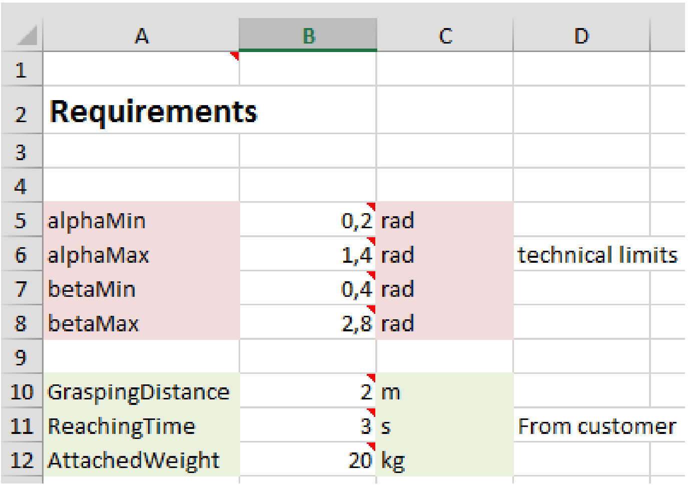
In order to specify references, artifact identifiers and unordered/ordered elements as described earlier, a minimal XML schema for Excel was defined.

```
<?xml version="1.0"?>
<?mso-application progid="Excel.Sheet"?>
<xsd:schema elementFormDefault="qualified" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Workbook">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="DocumentProperties" minOccurs="0">
                <xsd:element name="OfficeDocumentSettings" minOccurs="0">
                <xsd:element name="ExcelWorkbook" minOccurs="0">
                <xsd:element name="Styles" minOccurs="0">
                <xsd:element name="Worksheet" minOccurs="0" maxOccurs="unbounded">
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Figure 21: XML schema for Excel

However, it is not necessary to declare these properties because the program will also deliver useful output without them. They are only used to improve the output of ECCO.

At the end of the processing lifecycle an output of the new product variant is generated in the XML file format. The figure below shows that the output file can be opened in Microsoft Excel again. This proves that it is possible to generate a new Excel product variant with the use of ECCO.



	A	B	C	D
1				
2	**Requirements**			
3				
4				
5	alphaMin	0,2	rad	
6	alphaMax	1,4	rad	technical limits
7	betaMin	0,4	rad	
8	betaMax	2,8	rad	
9				
10	GraspingDistance	2	m	
11	ReachingTime	3	s	From customer
12	AttachedWeight	20	kg	

Figure 22: Excel - Generated output of robot variant

4.2 Pro/ENGINEER

For the purpose of showing that the parser for the STEP file format for the tool Pro/ENGINEER works, a simple robot variant was prepared that has a base, a shoulder, an upper and a lower arm, as well as a wrist. The robot variant is shown in the next figure.

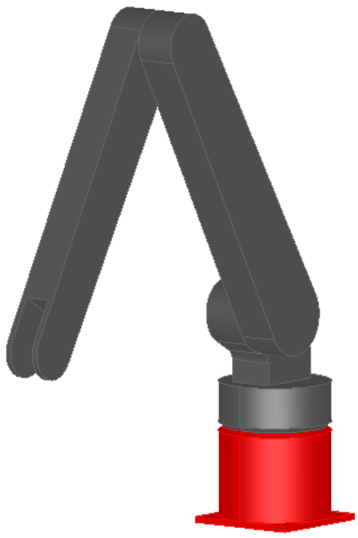


Figure 23: Graphical view of input robot variant

The corresponding STEP file contains over a thousand lines, therefore it was decided to leave it out at this point since it is similar to the file that was described in the previous chapter.

The STEP parser uses the STEP file as input and generates a model that can be used by ECCO to generate new product variants. The result of the parser is a new STEP file with the input robot variant.

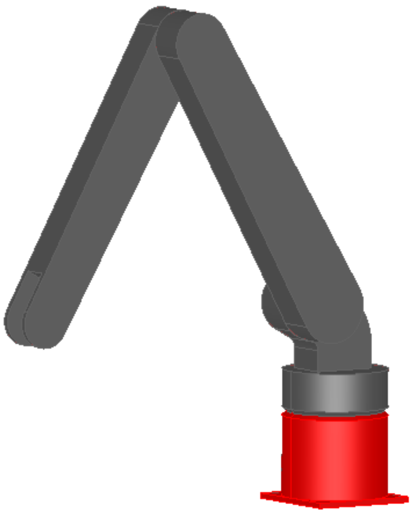


Figure 24: STEP - Generated output of robot variant

5 Conclusion

Usually the term clone-and-own comes up in the context of software engineering. This thesis describes how an automated way of this method can be applied to products in mechanical engineering, i.e. it explains how new variants of a product can be produced with little effort and little steps that have to be done manually unlike the process in the common clone-and-own approach.

This thesis showed that it is possible to create internal models from simple file formats that can be used to extract information and to generate new product variants. Additionally it is possible to transform these models into the original file format again, so that it can be opened in the corresponding tools. Both parsers – the Excel parser for Microsoft Excel and the STEP parser for Pro/ENGINEER – work properly with the files that were given for testing purpose. They deliver the expected results, namely the produced result does not differ from the given input in a way that when opened in the corresponding tools no differences can be seen. This is necessary to produce new variants from existing products and to consolidate them to a product line.

This thesis provided knowledge about how it is possible to create new variants in an easy and automated way with the help of ECCO and the implemented parsers that were introduced. Also new parsers can be easily implemented by using XML parsers for XML based file formats, or the compiler generator Coco/R for simple file formats where the size of the grammar does not go beyond a certain scope.

References

[1] D. Hislop, Z. Lacroix, G. Moeller. Issues in Mechanical Engineering Design Management. In SIGMOD Record, Vol. 33, No. 2, June 2004.

[2] L. Linsbauer, E. Haslinger, S. Fischer, R. Lopez-Herrejon, A. Egyed. ECCO – Extraction and Composition for Clone-and-Own. In IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014.

[3] L. Linsbauer, R. Lopez-Herrejon, A. Egyed. Recovering traceability between features and code in product variants. In 17th International Software Product Line Conference, 2013.

[4] T. Ziadi, L. Frias, M. da Silva, M. Ziane. Feature Identification from the Source Code of Product Variants. In 16th European Conference on Software Maintenance and Reengineering, 2012.

[5] Y. Xue, Z. Xing, S. Jarzabek. Feature Location in a Collection of Product Variants. In 19th Working Conference on Reverse Engineering, 2012.

[6] P. Frenzel, R. Koschke, A. Breu, K. Angstmann. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. In Software Quality Journal, Vol. 17, no. 4, pp. 331-366, 2009.

[7] J. Rubin, M. Chechik. Combining Related Products into Product Lines. In FASE, Vol. 7212 of Lecture Notes in Computer Science, pp. 285-300, 2012.

[8] Java API for XML Processing (JAXP). Online in Internet: URL: https://jaxp.java.net/, May 2015.

[9] T. C. Lam, J. J. Ding, J. Liu. XML Document Parsing: Operational and Performance Characteristics. In Computer by IEEE Computer Society, Vol. 41, Issue 9, pp. 30-37, September 2008.

[10] JSDAI. Online in Internet: URL: http://www.jsdai.net, May 2015.

[11] The Compiler Generator Coco/R. Online in Internet: URL: http://www.ssw.uni-linz.ac.at/Coco/, May 2015.

[12] N. Wirth. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?. In Communications of the ACM, Vol. 20, Number 11, November 1977.

[13] H. Mössenböck. The Compiler Generator Coco/R. User Manual. June 2004. Online in Internet: URL: http://www.ssw.uni-linz.ac.at/Coco/Doc/UserManual.pdf

List of Figures

List of Tables