PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 24/00

A construction process for artifact generators using a CASE tool

Luiz Paulo Alves Franca
Arndt von Staa

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 24/00
Editor: Carlos J. P. Lucena July, 2000

A construction process for artifact generators using a CASE tool *

Luiz Paulo Alves Franca
Arndt von Staa

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

A construction process for artifact generators using a CASE tool

Luiz Paulo Alves Franca, Arndt von Staa

Departamento de Informática, PUC-Rio
Rua Marquês de São Vicente 225
22453-900 Rio de Janeiro, RJ, Brazil
Tel: +(55)(21) 274-4449
{franca, arndt}@inf.puc-rio.br

Abstract

Artifact generators are a way to reduce production costs and time to market, and to increase reliability of artifacts within a given application domain. However, in order to achieve a more widespread usage of artifact generators, the difficulty and the cost of their development and maintenance must be reduced. This report presents a low-cost artifact generator development and maintenance process. The process departs from an example of a simple artifact within the target application domain. Given this example and considering all possible artifacts of the domain, all commonalties and variabilities are identified, as well as the properties that the specification of each specific target artifact must satisfy. The example is then modified in order to contain specific generator tags at all variable points. These tags establish the transformation rules that must be applied to the specification in order to generate the application. We have used a CASE tool, which allows the programming of the specification editors and of the generators. By means of another tool, the tagged example artifact is transformed into a component, which is combined with the transformation library, yielding the generator code to be internalized by the CASE tool. The process has been successfully used to transform specifications into applications, components and documentation.

Keywords

Artifact generator, application generator, CASE, generator development process

Resumo

Geradores de artefatos permitem reduzir os custos e o tempo para a disponibilização, e aumentar a confiabilidade de artefatos dentro de um domínio de aplicação. No entanto, para que se possa disseminar mais o emprego de geradores de artefatos, devem ser reduzidos os custos e a dificuldade do seu desenvolvimento e manutenção. Este relatório apresenta um processo de baixo custo para o desenvolvimento e a manutenção de geradores de artefatos. O processo inicia com um exemplo de um artefato simples visando determinado domínio de aplicação. Dado este exemplo e levando em consideração o conjunto de todos os possíveis artefatos dentro deste domínio, são identificadas todas as partes comuns e todas as partes variáveis. São identificadas também as propriedades a serem satisfeitas pelas especificações de cada artefato alvo. A seguir, o exemplo é modificado de modo que venha a conter marcadores de geração em cada um dos pontos variáveis. Estes marcadores estabelecem as regras de transformação a serem aplicadas ao gerar um artefato específico a partir de sua especificação. Para programar os editores das especificações e o geradores, utilizamos uma ferramenta CASE. Por intermédio de outra ferramenta, o exemplo devidamente marcado é transformado em um componente a ser combinado com a biblioteca de transformações, produzindo, assim, o gerador a ser internalizado na ferramenta CASE. O processo foi utilizado com sucesso para transformar especificações em aplicações, componentes e documentação.

Palavras Chave

Gerador de artefatos, gerador de aplicações, CASE, processo de desenvolvimento de geradores.

1 – Introduction

An artifact generator is a software tool that produces an artifact from its specification (fig.1). An artifact is any item created during the enactment of some development or maintenance process, such as an executable application, a component of some application, an artifact skeleton, or documentation.

Artifact generators have been used in several domains for a long time; for example, in 1985 Jenkins [1] surveyed commercially available generators. Usually, though, generators are not flexible, restricting thus the domain of possible applications.

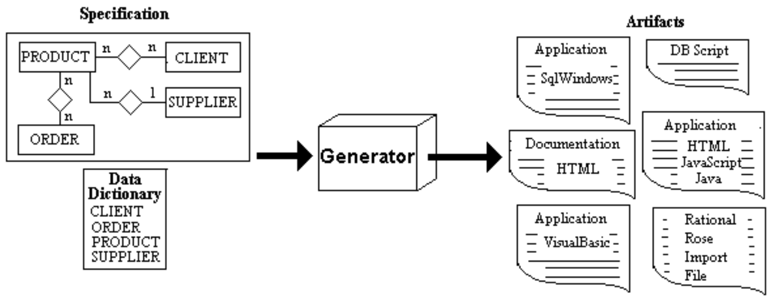


Figure 1. Schematic view of an artifact generator

In order to circumvent these restrictions, an organization could opt to develop its own collection of generators. One possibility would be to develop a highly parameterized generator capable of generating a broad set of possible applications. Usually this would not be a reasonable choice for the sheer complexity of the parameter set. Another possibility would be to develop a set of specific generators, each of which targeted to a specific and reasonably narrow domain. This option would be reasonable as long as the difficulty and effort to develop and maintain the generators is low. Furthermore, the generated artifacts must be complete, since just generating skeletons introduces maintenance difficulties, which reduce the cost effectiveness of the development and usage of such generators.

In this report we present an artifact generator development and maintenance process, which allows the fast and low cost development of generators. The development process envisages generator developers having only superficial knowledge of formal languages, parsing and compiling. Due to this fundamental requirement we have used an example driven approach.

The remaining of this report is organized as follows. In section 2 we describe the environment used when developing the artifact generator. In section 3 we present an overview of the example driven development process. In section 4 we detail the steps of this process. In section 5 we discuss some related literature. In section 6 we discuss a possible evolution of this work.

2 – Generator development environment

In this section we describe typical architectures and tools used while developing an artifact generator.

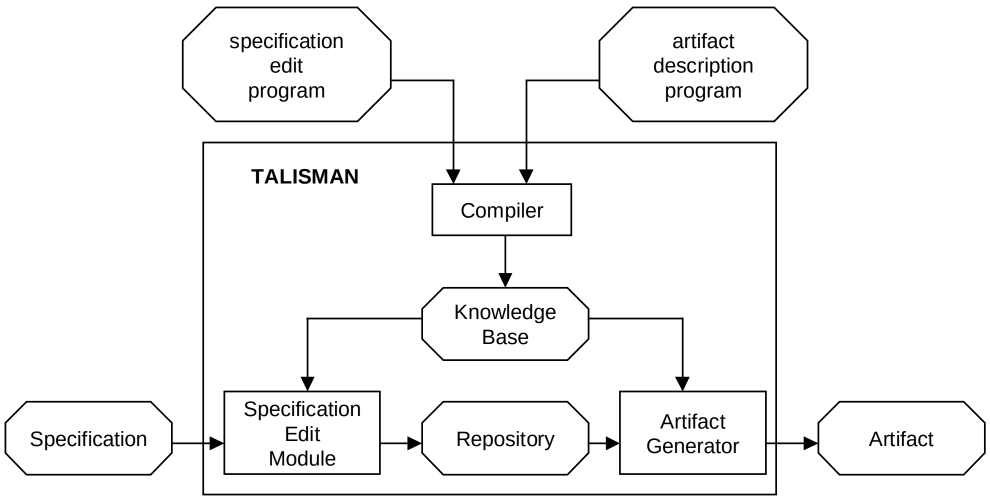
Most of the generators use some form of parser to analyze the specification in order to generate the artifact [2, 3, 4]. This approach requires the generator developer to define a specification language, develop an analyzer for this language, and to develop a transformer converting the resulting abstract syntax tree into the target artifact. These tasks require a fair knowledge of formal languages and compiler construction principles, which are not commonplace in most organizations. Due to this, we have followed another approach.

Our approach is based on meta-models. Using a representation language driven specification editor of some CASE tool, the specification is created in a repository at sufficiently low granularity. The representation languages needed by the generator are instantiated by means of simple programs bound to the meta-models of these languages. The repository is organized in accordance to these same meta-models. Recent works presented at the International Symposium on Constructing Software Engineering Tools [5, 6, 7] confirm the viability of this approach. The customization capability of CASE tools permits their use as the basis for the generating environment. In order to achieve this, the CASE tool should satisfy following requirements [5]:

- It should provide a repository containing all data relative to the instantiated meta-models in use.
- It should provide interfaces that allow ample access to the meta-data of the repository.
- It should provide mechanisms to edit the available meta-models, adding or modifying their meta-data.
- It should provide a variety of meta-data driven meta-editors (i.e. diagram editors and data dictionary editors).
- It should provide the possibility to create and execute scripts that explore and manipulate the repository content.

Using a CASE tool with these characteristics allows:

- The definition of the required specification modules. These specification modules provide means to edit diagrams and data dictionaries that correspond to the specification of the artifact to be generated. Both the diagram and the data dictionary editors are customized to the needs of the generator being developed.
- The definition of the generator modules. These modules constitute the script that explores and extracts data from the CASE's repository, producing the target artifact.



specification edit program
artifact description program
TALISMAN
Compiler
Knowledge Base
Specification
Specification Edit Module
Repository
Artifact Generator
Artifact

Figure 2. Using TALISMAN as an environment for the construction of artifact generators

In our process we have chosen the meta-CASE Talisman [8] (fig. 2) to support both the editing of specifications and the generation of the corresponding artifact, since it satisfies the necessary requirements and is available in our laboratory. It should be noted that any CASE tool satisfying the above mentioned requirements should be as good a choice. In [5, 6] the usage of Rational Rose [17] as a generating environment is discussed.

Both the specification editor and the artifact description program are files contain code written in the Talisman language. The Talisman compiler transforms these files and stores the executable byte code in its knowledge base. After the compilation, these programs become part of the instantiated CASE tool. The specification editor module (e.g. an entity-relationship diagram plus data dictionary) records the specification of the artifact in the repository. When the user activates the artifact generator, the repository is explored, accessed, and the retrieved data is formatted generating the code of the artifact. This code may be composed of several fragments, each of which written in a different language. The specification editor program (Figure 3) defines the contents and organization of each data dictionary entry screen. These screens contain a subset of the objects and relations of Talisman's meta-model. The artifact description program (Figure 4) defines the composition of the artifact based on the connection of its fixed parts (independent from the specification) with the data extracted from the specification.

```
BeginForm "Edit Attribute"
   Title "Name:" ;
   Name ;
   Title "Descripion:" ;
   Text  TextDescr ;
   Title "Observation" ;
   Text  TextObserv ;
   Title "Type" ;
   String Type;
   Title "Length" ;
   String Length;
EndForm
```

Figure 3. Portion of a specification edit program

```
Title "<div id=\"header\">";
Call "fNomeClasse"( oCLASS );
Title "window";
Title " </div> ";
Title " <div id=\"screen\"> ";
Title " <table>    <tr> ";
Title " <td align=\"right\"> <b> ";
Call "fIDName"(oCLASS,1);
Title " </b>   </td> ";
Title " <td>  <input type=\" ";
Call "fIDType"();
Title "name=\" ";
```

Figure 4. Portion of an artifact description program

3 – Example-driven approach

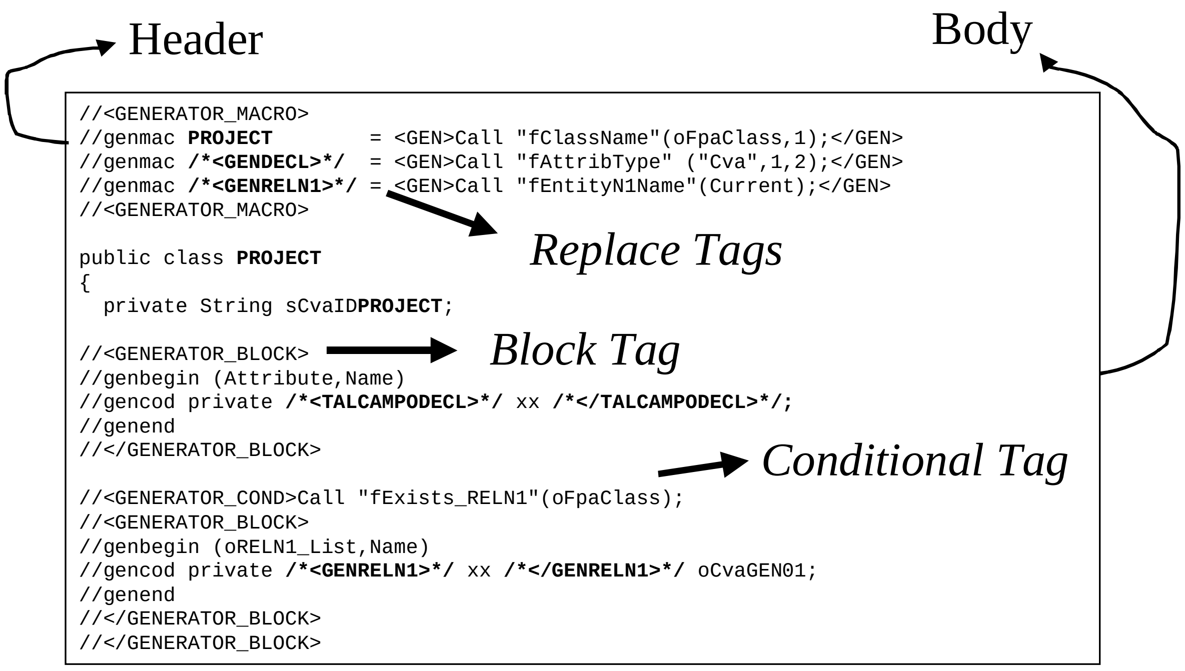
In this section we describe, in general terms, how the example-artifact is transformed into a generator component.

Even though CASE tools facilitate the construction of a generator, the proposed solution still entails a complex task, which is coding the artifact description program. This program tends to be quite complex because it must encapsulate the mapping between the specification and the implementation domain. As the conceptual distance between these domains grows, the programming effort grows too. Furthermore, this type of programming is time-consuming, error-prone, and sometimes it is not clear what code should be generated, enticing a fair amount of prototyping. Examining the generator programming process, we realized that this step could be simplified in the following way.

1) Develop a simple example-artifact covering all aspects of the target domain. This artifact must be thoroughly verified and validated, since the generator will replicate its content and structure. Once this correct example-artifact is available, it is analyzed in order to determine variable parts that depend on the specification and fixed parts that are simply replicated. Using the framework terminology, the variable parts are the *hot spots* [9] and, by analogy, the fixed parts are named *frozen-spots*. Transformation tags are placed at each hot spot. These transformation tags describe how the specification is accessed and transformed in order to produce the corresponding code within the target artifact. The file containing tags is called the *artifact meta-description file*. This file serves as an intermediate level between the specification and implementation domains. Three types of transformation tags are needed:

   - *Replacement*: the string defined by the tag is replaced by a call to a function that retrieves information from the repository.
   - *Block*: a sequence of commands is executed with respect to each of the elements of a given set (e.g. all entities of an entity relationship diagram).
   - *Conditional*: depending on the execution result of a function, the consequent block transformation is performed.

2) Transform the artifact meta-description file into its corresponding artifact description component. This transformation has been completely automated. A utility program (GENDES) reads the artifact meta-description file and replaces each tag by the corresponding transformation rule. In figures 7 and 8, we show some examples of transformations performed by GENDES.



```
//<GENERATOR_MACRO>
//genmac PROJECT        = <GEN>Call "fClassName"(oFpaClass,1);</GEN>
//genmac /*<GENDECL>*/  = <GEN>Call "fAttribType" ("Cva",1,2);</GEN>
//genmac /*<GENRELN1>*/ = <GEN>Call "fEntityN1Name"(Current);</GEN>
//<GENERATOR_MACRO>

public class PROJECT
{
  private String sCvaIDPROJECT;

//<GENERATOR_BLOCK>
//genbegin (Attribute,Name)
//gencod private /*<TALCAMPODECL>*/ xx /*</TALCAMPODECL>*/;
//genend
//</GENERATOR_BLOCK>

//<GENERATOR_COND>Call "fExists_RELN1"(oFpaClass);
//<GENERATOR_BLOCK>
//genbegin (oRELN1_List,Name)
//gencod private /*<GENRELN1>*/ xx /*</GENRELN1>*/ oCvaGEN01;
//genend
//</GENERATOR_BLOCK>
//</GENERATOR_BLOCK>
```

Figure 5. Fragment of the artifact meta-description file

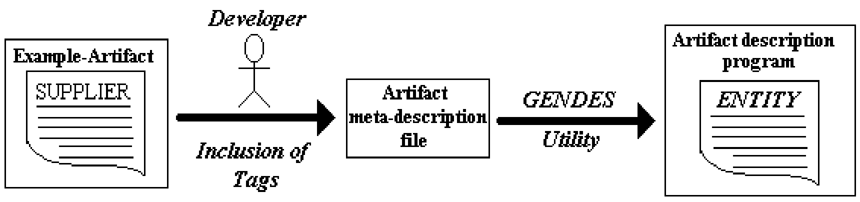
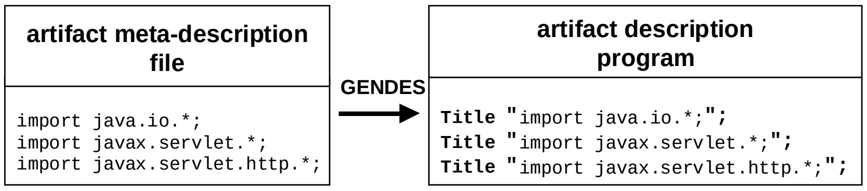


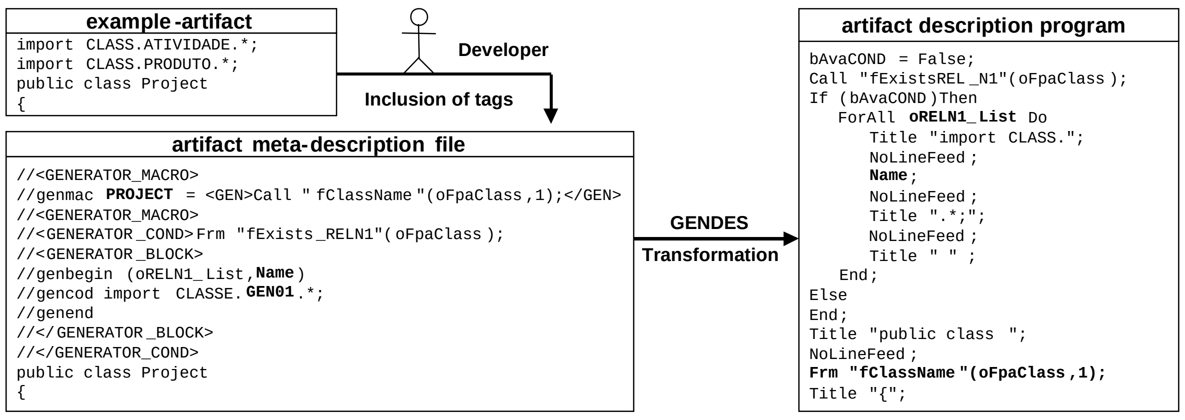
Figure 6. Automatic generation of the artifact description program



```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
Title "import java.io.*;";
Title "import javax.servlet.*;";
Title "import javax.servlet.http.*;";
```

Figure 7. Frozen-spot transformation



```
import CLASS.ATIVIDADE.*;
import CLASS.PRODUTO.*;
public class Project
{
```

```
//<GENERATOR_MACRO>
//genmac PROJECT = <GEN>Call " fClassName "(oFpaClass ,1);</GEN>
//<GENERATOR_MACRO>
//<GENERATOR _COND>Frm "fExists_RELN1"( oFpaClass );
//<GENERATOR _BLOCK>
//genbegin (oRELN1_ List ,Name )
//gencod import CLASSE. GEN01 .*;
//genend
//</ GENERATOR _BLOCK>
//</GENERATOR_COND>
public class Project
{
```

```
bAvaCOND = False;
Call "fExistsREL _N1"( oFpaClass );
If (bAvaCOND )Then
   ForAll oRELN1_List Do
      Title "import CLASS.";
      NoLineFeed ;
      Name ;
      NoLineFeed ;
      Title ".*;";
      NoLineFeed ;
      Title " " ;
   End;
Else
End;
Title "public class ";
NoLineFeed ;
Frm "fClassName "(oFpaClass ,1);
Title "{";
```

Figure 8. Tag (hot spot) transformation

4 – Construction process for Artifact Generators

In this section we describe the steps of the process that should be enacted while developing an artifact generator program.

Following are the steps of the artifact-generator development process:

a- Construction of a target example-artifact

This step consists in the construction of a correct and simple target example-artifact within the desired problem domain. This example will be the basis to generate similar artifacts. In one experiment we have built an example capable of handling the basic database operations (include, update, delete, retrieve) using a three layer client server architecture and using a Web browser as user interface.

b- Creation of the artifact meta-description file

Analyzing the example-artifact, we identify the specification dependent parts (hot spots) and the parts which are replicated in the end product (frozen-spots). While identifying hot spots, also the required transformations are specified. For example, if the title of a table maintenance window should be the name of this table, this name must be retrieved from the repository when building the artifact. In [3] a Statechart Simulator is described, in which 84% of the code correspond to frozen-spots.

This step terminates when all variable parts of the sample-artifact are marked with appropriate transformation rule tags. The result of this step is the artifact meta-description file and a list of properly specified transformation rules.

c- Automatic generation of the artifact description program

Using the GENDES program, the artifact meta-description file created during the previous step is transformed into the artifact description component.

d- Creation of a specification editor program

The specification editor must capture all necessary data for all of the transformation rules and hot spots identified during step b. Talisman allows the configuration of specification entry windows, which may contain titles, text and string fields, as well as relation lists relating the current object to other objects. In a relation list each entry can be edited or selected from a list (dictionary) of objects. The entry windows may contain data from several different objects and are initialized according to the contents of the repository. Finally, Talisman contains a diagram meta-editor capable of editing several different diagrammatic representation languages. Each element of a diagram is related to a specific object. Entry windows may display data in accordance to the contents of several diagrams.

e- Implementation of the transformation rules

During the creation of the artifact meta-description file, step b, the tags are bound to transformation rules. These rules are implemented as functions that access and manipulate the repository in order to perform the required transformation. These functions must be programmed in Talisman's native language and must conform to the specification defined during step b. Once coded and adequately verified and validated, they are registered in a library. These functions are heavily reused while building one or more generators, even when the target languages vary. The set of functions is small and does not grow. The library was constructed during our initial experiments. Also during this step the generation control module is coded.

f- Compilation of the specification editor and artifact description programs

The artifact descriptor component, the library of transformation rules and the control module are combined to form the artifact descriptor program. This program and the specification editor program are compiled using Talisman's internal compiler and stored in its knowledge base.

g- Editing the specification of an artifact

Activating the editor program allows the creation and maintenance of the specification of a target artifact. During the editing of a specification, the user must provide all the information required by the specification editor in order to create or maintain the specification of the target artifact.

h- Artifact generation

Once the specification has been completely and correctly edited, the generator program may be activated to produce the artifact's code in conformance with its specification. Depending on the type of artifact to be generated, it may be necessary to insert an extra step in order to create the utilities required to post-process the generated artifact. For example, during an experiment generating applications written in Java, we had to create a utility to copy the generated Java files to the directory structure coherent with the needs of the Visual J++ project used to compile the source code.

5 - Related work

In this section we will examine some other proposals found in literature, which are similar to our proposal.

Identification of fixed and variable parts

The identification of fixed and variable parts determines the transformations and the points where they should be applied. In the process proposed by Cleaveland [2], one of the steps requires the identification of *variants* and *invariants*. Based on this identification method, the product description is built using a language that allows handling of fixed and variable parts in the program. Coplien et al [11] describe the Scope, Commonality and Variability (SCV) analysis to be used by domain engineers. According to them, when commonalties and variabilities are well defined, there is an excellent opportunity for development automation. Pree [10] describes a hot spot mining method to identify a framework's hot spots.

Source language independent transformation

In our proposal the generated product is a result of the combination of fixed and variable parts, the latter obtained transforming the information contained in the specification of the target artifact. This construction technique reduces the need to create very low granularity (abstract syntax tree granularity) transformation rules to generate each sub part of the final product; consequently, the transformations are indifferent to the low-level semantics of the end product. Different experiments (generating Visual Basic, SQL-Windows programs and HTML document files) showed that this technique is independent of the programming language of the target artifact. In [2, 3], coding the product description file required creating commands to handle the fixed and variable parts separated from the programming language of the target artifact. One of the most successful examples of this approach is the development technique called "Framing Software Reuse" [13]. This technique is based on the construction of applications starting with the building of a hierarchy of modules, and defines the composition process through a module connection language. The analysis of industry related projects showed high productivity rates and high reuse during the development of COBOL programs [14].

Utilization of a CASE tool

Prado et al [12], showed the benefits of associating a CASE tool to a transformation system. The CASE tool increases the quality of the specification, since it makes it possible to use different textual and diagrammatic styles for visualizing specifications. [5, 6, 7] describe the use of CASE tools as generator construction environment.

6- Conclusion and future development

The proposed process has been used to build generators capable of generating database browsing and updating applications from their data models. The generated applications are able to carry out basic operations such as maintenance of tables extracted from the model, preserving the 1:N and N:N relations defined in the model. The following generators have been built:

- A three layer client-server architecture: composed of an interface layer (HTML + JavaScript), a business layer (Java servlet), and a database layer (ODBC and SQL).
- A client-server architecture using VisualBasic.
- A client-server architecture using SQLWindows.
- An html version of the specification documents.

These experiments have demonstrated experimentally the language independence and architecture independence of the generator building process. Furthermore, the library of transformation rules could be reused in all of these experiments.

We are planning to improve the GENDES utility, which transforms the product meta-description file into the artifact description component, enabling it to generate a specification verification program. This would allow application developers to verify completeness and other properties of a specification prior to generating the application. This should reduce problems found while generating and using artifacts from a new or modified specification.

Although artifact generators usually increase productivity and maintainability, they are not as widely used as they could. Many of the generators focus a very narrow application domain, others generate skeletons increasing the cost and difficulty of maintaining the generated artifact. Finally, using development processes based on compiler construction techniques, leads to costly and difficult generator development and maintenance processes.

One way to augment the utilization of artifact generators could be achieved facilitating their customization. However, raising the flexibility by means of parameterization may render the generator extremely complex and cumbersome to use, since it will be prepared for several potential customizations, some of which might never be used [16].

Another way to broaden the use of artifact generators is through the simplification of the construction process. Bringing down the difficulty and the cost of constructing and adapting generators increases the feasibility of constructing generators that are specific to each type of application.

The proposed construction process has been simplified due to following factors:

- Utilization of a CASE tool
  The CASE tool we used integrates both the user interface required for editing the specification, as well as the artifact generator that builds the artifact in accordance to this specification. Due to the use of an easily configurable CASE tool, we eliminated a large part of the effort spent building the generator's components.

- Using a process based on an example-artifact within the problem domain

Using SCV analysis based on a correct example, facilitated the identification of the set of transformation points and their nature assuring the exact reproduction of the example-artifact from its original specification. It also became easy to determine the exact composition of the specification. Thus, the specification will only contain necessary fields, reducing the amount of work when developing a new artifact.

- Creation of an artifact meta-description file
  The artifact meta-description file reduces the impedance-matching problem [5, 7] between the specification and implementation domains. In this file, transformation tags mark all points that depend on the specification. The remainder of the file corresponds to frozen spots, which should be simply replicated in the target artifact.

- Path for easy application generator evolution
  Whenever it becomes necessary to generalize the generator to deal with a wider problem domain, it is simple to identify the additional transformation points and the corresponding modifications in the specification. To evolve the artifact generator it is necessary only to insert the necessary transformation points into the artifact meta-description file and repeat the automatic generation steps.

- Reuse of the transformation rules
  The first experiment led to the creation of a library of transformation rules. In subsequent experiments the need to create new rules eventually ebbed out, even when the target language was changed.

With this simplification of the artifact generator construction process, we expect to contribute to the more widespread use of artifact generators, making them a part of the conventional toolkit of software developers.

References

[1] Jenkins, M.; “Surveying The Software Generator Market”; *Datamation*, Sept. 1985; pp. 105-120.

[2] Cleaveland, C.; “Building Application Generators”; *IEEE Software* 5(4); 1988; pp. 25-33.

[3] Masiero, P.C.; Meira, C. A.; “Development and Instantiation of a Generic Application Generator”; *J. System Software* 23; 1993; pp. 27-37.

[4] Neighbors, J. M.; “Draco: A Method for Engineering Reusable Software Systems”; In Ted J. Biggerstaff and Alan Perlis (Eds.); *Software Reusability*; Addison-Wesley / ACM Press; 1989; pp. 295-319.

[5] Hohenstein, U.; “An Approach for Generating Object-Oriented Interfaces for Relational Databases”; In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools*; 2000; pp. 101-111.

[6] Plantec, A.; Ribaud, V.; “Using and Re-using Application Generators”; In *Proceedings of the First International Symposium on Constructing Software Engineering Tools*; 1999; pp. 59-65.

[7] Milicev, D.; “Extended Object Diagrams for Transformational Specifications in Modeling Environments”; In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools*; 2000; pp. 121-131.

[8] Staa, A.; *Talisman: Ambiente de Engenharia de Software Assistido por Computador, Manual de Referência*; Rio de Janeiro, Brazil; 1993; (in Portuguese).

[9] Pree, W.; *Design Patterns for Object-Oriented Software Development*; Prentice-Hall, NJ; 1996.

[10] Pree, W.; “Hot spot-Driven Development”; In Mohamed Fayad and Douglas C. Schmidt (Eds.); *Building Application Frameworks: Object-Oriented Foundations of Framework Design*; John-Wiley & Sons; 1999; pp. 379-393.

[11] Coplien, J.; Hoffman, D.; Weiss, D.; “Commonality and Variability in Software Engineering”; *IEEE Software*, 15(6); 1998; pp. 37-45.

[12] Prado, A.; Barrére, T.; Bonafe, V.; “CASE Orientada a Objetos com Múltiplas Visões e Implementação

Automática de Sistemas - MVCASE”; In *Proceedings of the XIII Brazilian Symposium on Software Engineering*; 1999; pp. 113-128; (in Portuguese).

[13] Basset, P.; *Framing Software Reuse*; Prentice-Hall, NJ; 1996.

[14] Grossman, I.; Mah, M.; “Independent Research Study of Software Reuse (Using Frame technology)”; *QSM Associates*; Sept. 1994; 75 pp.

[15] Floch, J.; “Supporting Evolution and Maintenance by Using a Flexible Automatic Code Generator”; In *Proceedings of the 17th International Conference on Software Engineering*; 1995; pp. 211-219.

[16] Lanergan, R.; Grasso, C.; “Software Engineering with Reusable Designs and Code”; In Ted J. Biggerstaff and Alan Perlis (Eds.); *Software Reusability*; Addison-Wesley/ACM Press; 1989; pp. 187-196.

[17] Rational Co.; *Rose 98 Rose Extensibility User’s Guide*, 1998.