Generative Programming and Components: theory and practice

Matteo Bordin
Department of Pure and Applied Mathematics
University of Padua
via G. Belzoni 7, I-35131 Padova, Italy
mbordin@studenti.math.unipd.it

Abstract. This paper investigates on the possible advantages of applying generative programming in a component based development process: if a component oriented approach is applied, then generative programming can be used to automatically compose and assemble components. In part one of this paper, I present the application of Generative Programming from an engineering point of view, using a simple but complete didactic example based on C++ template metaprogramming. Finally, part two is a description of a generative approach to develop component for the hard real-time software domain, this time using the Ada 2005 programming language.

PART ONE – Theory: generative programming and component based software engineering

1. Introduction

The idea of formally describe software and leave a machine produce the corresponding code has always been intriguing: this is because a generative approach allows for correctness by construction and fast (and cheap) development. In the last years, the emerging of visual design languages (for example the *de-facto* standard UML) has revitalize researches in the field. While these languages has still a lot to prove for what regards functional code (even if Intentional Programming could deliver valid solutions in this area), they has shown to be reliable and usable to describe the architecture of a software system. This fact is of particular importance nowadays: in a world of complex, distributed, multithreaded systems, software architecture is at least as important as pure functional code. In such a domain, Component Based Software Engineering (CBSE) has shown to be a good way to produce reusable software, eliminating part of the idiosyncrasies typical of a pure object oriented approaches. As stated in [1], Generative Programming can be seen as a way to automate components selection and assembly on demand: if this approach is used, then it is finally

possible to generate code from a specification which has been designed to be reusable, leading to a faster, cheaper and safer way to produce code.
In part one of this paper, a simple use of generative programming to automate component assembly is shown, using the by now famous didactic example of car product line (see [1]) based on C++ template metaprogramming. In part two, the approach previously described is evolved and applied in the hard real-time concurrent software domain.

2. The engineering approach

After the explosion of Object Oriented Design (OOD) and Programming (OOP) almost 20 years ago, many researchers and everyday users have argued about the inability of this approach to fully support software reuse, encapsulation and information hiding. Sometimes these problems can be solved reducing the expressivity of design and programming languages, in other cases the problem relies on the semantic of object oriented languages (see [2]). However, for our purpose, the main drawback of OOD is its focus on just single systems: to improve software reuse it is necessary to shift the focus to entire system families. In particular, OOD/OOP main deficiencies in this area are:

- *No distinction of engineering for reuse and engineering with reuse:* as stated in [3], the difference between developing software based on components and design new reusable component is huge: in fact they are two totally different development processes. Generative programming can positively affect both processes.
- *No domain scoping phase:* in the development process of object oriented system, just one stakeholder is present. It is necessary to analyze the entire domain to fully discover new potential customers and improve reuse.
- *No distinction between-inter and intra-application variability:* intra-application variability is the use of different variants of an object in the same application, while inter-application variability is the use of variants in different applications. In both case dynamic polymorphism is used, even if it is the OO semantic just for intra-application variability.
- *Fixed implementation of variability modeling:* using OO notation, as UML, it is necessary to specify at design time how variability is applied, i.e. if inheritance, class parameterization, templates are used. To avoid this problem, feature diagram are introduced later in this paper.

Component based software engineering addresses these problems: it indicates a way to achieve real software reuse by introducing independent and configurable software entities (components) which can be combined to build a complete system. In relation to this, Generative Programming is a software engineering paradigm in which a highly customized software can be automatically build on demand, eventually using elementary and reusable components ([1]) (fig 2.1). But how can Generative Programming help to improve a development process based on components? If components are designed to fit an entire family of software systems, then their assembly and configuration can be performed using generative programming from a formal specification, creating a customized software system. This is what generative programming should be for the component based application programmer: formally define the configuration of the system and let the tool produce the entire software architecture.

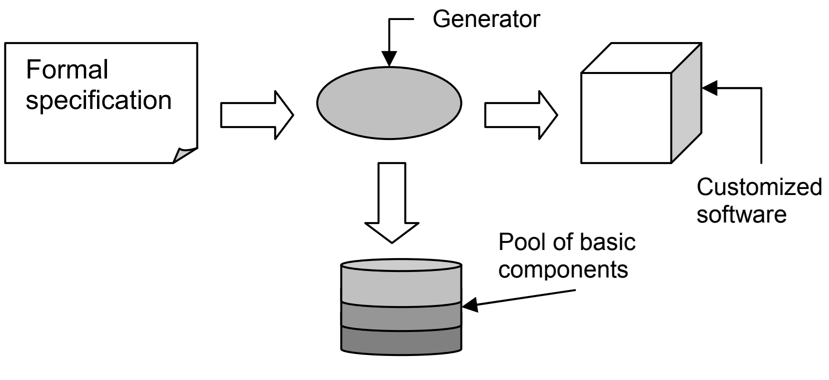

Fig. 2.1- Generative programming and components

If this approach is used, then the development process radically changes. Clearly our main interest is on development for reuse (also referred to as Domain Engineering), which is composed by three main activities:

- *Domain analysis:* in the first phase it is necessary to analyze the domain to discover and model features, i.e. identify common and variable features of the domain and their relations ([1]). This process is driven not only by technological, but also by economic and business aspects.
- *Domain design:* the purpose of this phase is to develop a common framework for the system family.
- *Domain implementation:* in the last phase, generators and components are developed.

These phases are analyzed in the following sections, using a simple example to help the reader. The example has been introduced by the authors of [1] and perfectly shows the advantages of using a component approach with generative programming: as a car is composed by many independent parts as engine, transmission, brakes, seats, etc., we can describe and build a software system using many reusable and independent components.

2.1 Domain analysis

The first necessary phase to achieve full software reuse is domain analysis. In this phase, the developer defines the problem space, i.e. the domain specific concepts and features. This information can be represented with a feature diagram. A feature diagram is a tree where the root represents the main concepts (in our case, a car), and each other node represents one of its features (engine, transmission, etc.). The edges connecting the nodes define the dependencies between features. They can be (compare with figure 2.1.1):

- *Mandatory feature:* a simple edge ending with a filled circle
- *Optional feature:* a simple edge ending with an empty circle (for example PullsTrailer).
- *Alternative feature:* edges connected with an empty arch represent alternative features (for example automatic and manual transmission). Exactly one of these features must be present.
- *Or*-feature: edges connected with a filled arch represent or features (for example a car can have a gasoline engine, an electric engine or both). At least one of these features must be present.

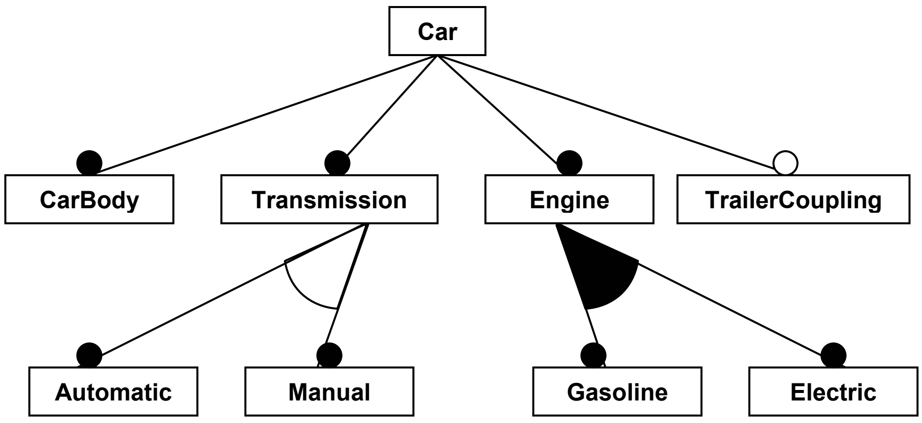

Fig 2.1.1 – Feature model for the example

While it is possible to combine different kind of feature, it is better to express all concepts with just these four kinds of feature, "normalizing" the diagram.

It should be noted that it is not possible to directly specify cardinality in feature diagram: asserting that a car has k wheels can be expressed as:

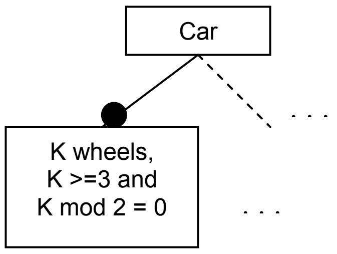

Fig. 2.1.2 – Cardinality in feature diagram

Some constraints can't be expressed by feature diagram: in this case, an additional (possibly formal) language is needed. The feature diagram, constraints and additional non-function requirements constitute a feature model.

The main advantage of feature diagram is the possibility to post-pone the definition of how variability is applied. Each node (apart from the root) is a possible variation point: for example the feature "transmission" can be automatic or manual. However, in this phase it is not needed to specify how to map (nor to design language or programming language) this variation: it can be mapped with inheritance, parameterized inheritance or static parameterization. Generally, variability is mapped differently at different levels. Using feature diagram the developer can describe domain analysis without binding it to a specific design method or programming language.

2.2 Domain design

Domain design provides for the mapping from the problem space defined in domain analysis to the solution space, i.e. a set of basic and reusable components: to improve reuse, the basic components must maximize their combinability and minimize redundancy. In this phase, it is necessary to define a set of rules which specify feasible combinations, dependencies and default values for the just designed components. This information is called configuration knowledge (see fig 2.2.1).

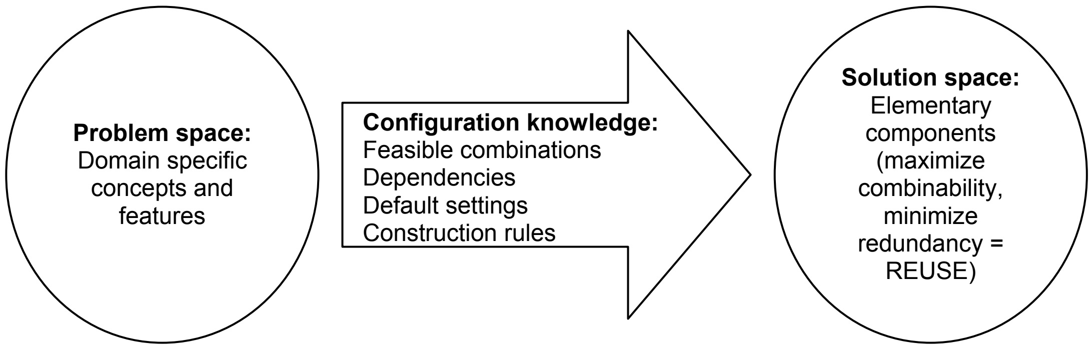

Fig 2.2.1 – From problem space to solution space

The domain design process generally requires to identify implementation components, design a layered architecture, define interfaces between layers and specify a component model or middleware to use. All this information descends from the domain analysis phase and must be encoded in a domain specification language (DSL). A generator (described in the next chapter) acts on this specification to produce component code and assembly.
Writing a formal specification requires preliminary steps starting from the feature model:

- *Identify the main functionalities in the feature diagram:* they are component categories (interfaces). In our example, the main functionalities are CarBody, Transmission, Engine and PullsTrailer.
- *Enumerate component divided by categories:* each component is an implementation of a particular interface. Note that no information about how to implement variance is needed. Figure 2.2.2 shows components categories and their division:

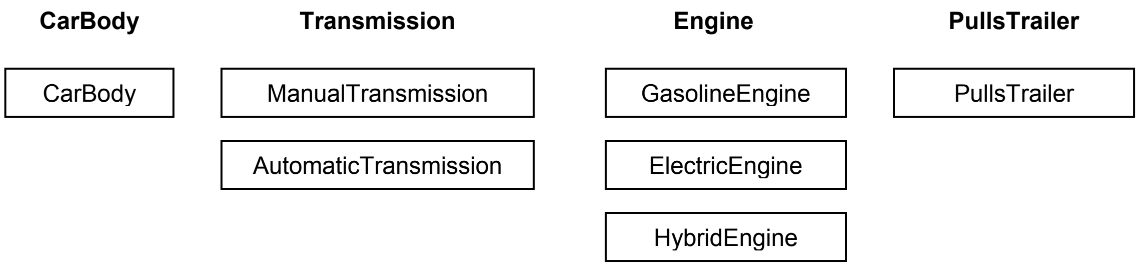

Fig. 2.2.2 Components categories

- *Identify dependencies:* specification of required interface(s). For example, the CarBody component requires an implementation of interface Engine.
- *Design a layered architecture:* all components are sorted hierarchical: this ordering can be used as a model to apply delegation.
- *Writing the specification:* finally the specification is written. At this point the architecture of our product is defined.

Now it is time to develop a generator for the specified grammar and produce all the code needed to represent components and constraints.

2.3 Domain implementation

The component architecture can be eventually generated from a document written in a proper DSL. In this phase, variability is applied in a specific way, depending on the provided generators. In addition to the generator, basic components could possibly be defined, in order to provide a library on which the generator can rely (as described in part two). For example, the GasolineEngine component is implemented this way:

```
template<class Config_>
struct GasolineEngine{
  typedef Config_ Config; //publish configuration type
}
```

GasolineEngine required interface is empty but it still need a type containing the configuration to propagate to the higher levels. The ManualTransmission component is implemented as:

```
template<class Engine> //specify here the required interface types
struct ManualTransmission{
  typedef typename Engine::Config Config; //publish configuration type
  Engine e; //allocate object for the required interface
}
```

Required interface types are mapped as template parameters and allocated as instances in the data structure. It should be noted that at this point a defined choice to implement variability is taken: since no late binding is needed, C++ templates are used. This crucial decision is taken while implementing the generator: since the concept of generator is so central in generative programming, it is better to explain more deeply their role, functionalities and implementation.

3. Generators

A generator is simply a program that transforms a higher level specification in a lower level one. Examples of generators are simple compilers, RMI compilers (rmic in Java), CORBA IDL compilers and many others. Nowadays, almost every UML tool or GUI builder provides for an automatic code generator. However the concept (and power) of generators goes far beyond, as they allow to:

- *Express intentionally system specification:* intentional descriptions are easy to understand, analyze, verify and maintain, i.e. they posses all good software qualities; they are expressed in a DSL and implemented with a generator.
- *Move verification from code level to intentional specification:* since the code produced by a generator is correct by construction, at least a part of verification can be moved from code level to the specification level, decreasing the cost of this phase. In order to achieve this result, generators correctness should be verified formally: this is one of the reasons of their high developing cost.
- *Compute an efficient implementation:* generators mapping of an intentional specification is efficient; in addition they relieve the programmer form performing such (probably complicated) transformations. The mapping from a higher level specification to a lower level one is called transformation; actually software development in general can be seen as a sequence of transformations and modifications of specifications. The process of transforming specification can be applied iteratively, eventually creating many intermediate-level specifications: this way many smaller and simpler generators are developed, each of them acting on a particular abstraction level.
- *Code factorization:* since generators operates on a defined set of small, configurable components, the resulting code is highly factorized, therefore easy to understand, maintain and reuse.

When a generator transforms a specification in a (usually) lower one, the transformation is called vertical. However it is possible to develop a generator able to transform a specification in a more efficient one, maintaining the same abstraction level: in this case the transformation is called horizontal. An example of horizontal transformation is code optimization by inlining, applied by common compilers. From now on, the term transformation indicates vertical transformation, which is more interesting for our purpose.

The impact of a generator (and, for extension, of generative programming) on the development process is huge. At the moment, effective software evolution is rare, even if evolutionary process model has proven to be useful. The main reason is that evolution usually changes high-level software properties (architecture) whose implementation is scattered across the entire system code: using a traditional approach, it is almost impossible to effectively perform design revisions. On the contrary, if evolution is applied on a high level specification and a generator expresses it at the lower abstraction level (code), then an evolutionary life cycle model can be efficiently applied. The ideal paradigm shift is shown in fig. 3.2:

a. Traditional approach: requirements are directly implemented in the source code.
b. Requirements must still be manually mapped in a DSL but, using domain specific abstraction, library and generators, source code is automatically generated. The generated code can include functional code (in addition to architectural).
c. Requirements are directly expressed in a machine-processable encoding. Then an interactive and iterative transformation system can produce the DSL specification needed to apply code generation.

The highest level of abstraction (c) can't be easily reached for one main reason: in an evolutionary life cycle model, not all design decisions are available at the same time, increasing the difficulty of applying transformations and designing generators. To make effective software evolution, the specification and the generator must be based on a set of domain specific components, which are combined to form the skeleton of the system. The approach described in part two fully reaches (and, at the moment, stops at) the middle level.

3.1 Implementing generators

There are three main ways to implement a generator:

a. *Generator as a stand alone program:* this is the most expensive way, yet not the most effective: all generative infrastructure, i.e. internal data representation, parsing, verification and generation engine must be developed and (not to be forgotten!) verified. In addition, integration can be problematic, due to different notation, data representation and implementation languages.

b. *Generator using metaprogramming:* using programming languages built-in metaprogramming facilities is clearly cost effective. An example is C++ template metaprogramming: if components are implemented as templates, then the compiler itself can produce the code for a particular assembly or realization. In addition, the configuration knowledge can be mapped on a template-based generator to perform static configuration (see next chapter). Template metaprogramming is easy to use, efficient (the compiler usually knows its job!) and fast. On the other hand, it is bounded to a particular programming language, limited to the expressive power of the implementation language and generally difficult to debug.

c. *Generator in a generative infrastructure:* in this case a common (and possibly platform independent) data format, parsing and transformation facilities are used. The infrastructure should support the developing of domain specific generators, with appropriate tools (for example debugger). This is clearly the best choice. A perfect example of such an infrastructure is Intentional Programming.

In our car-themed example, it is possible to implement a generator in a generative environment to obtain the previously described (ch. 2.3) implementation components. After that we can still use generative programming to automate component assembly.

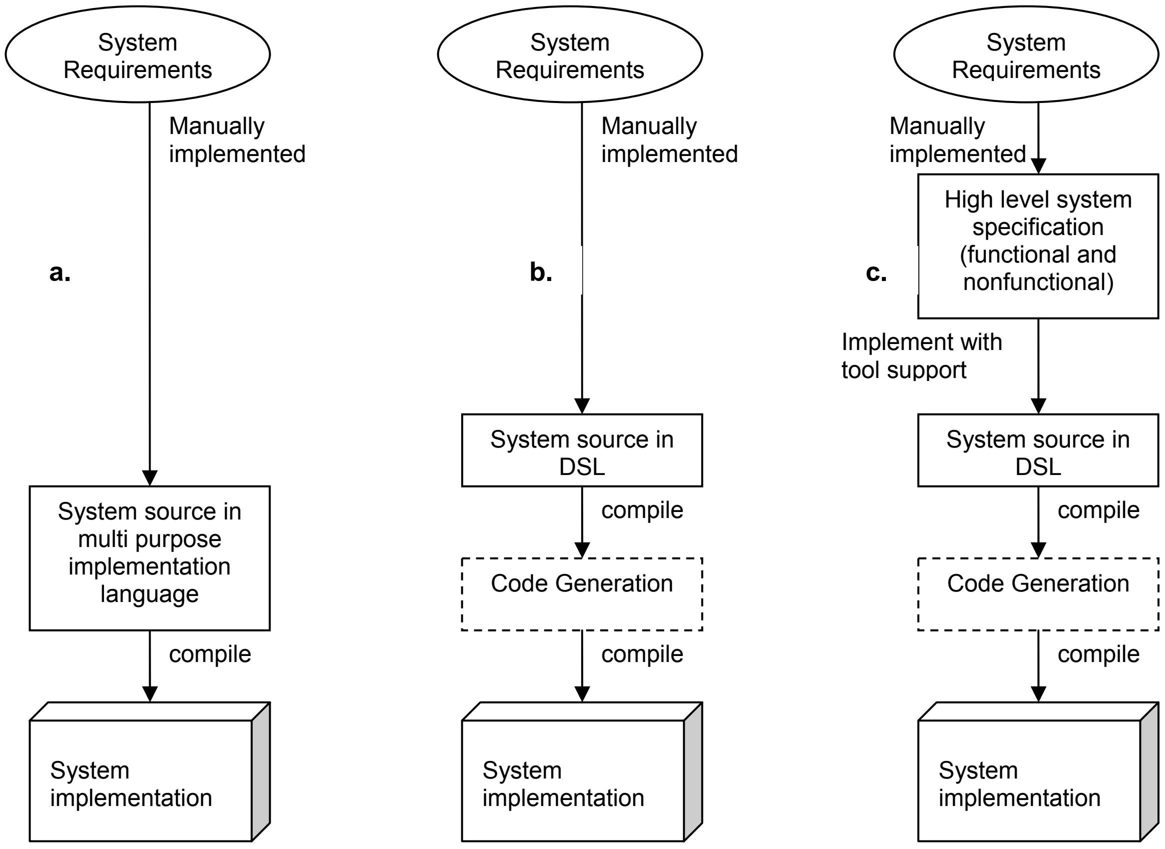

Figure 3.2 – Paradigm shift using generative programming

3.2 C++ template metaprogramming

Since C++ template metaprogramming is the chosen generative approach to automate component assembly (see next chapter), it is better to review it with some basic example.

C++ templates are a turing-complete language allowing performing computation at compile time. Template metaprogramming resembles in many ways functional programming. A typical example is the factorial function, which can be computed at compile time using templates:

```
template<int i>
struct Factorial{
  enum{ RET = Factorial<n-1>::RET * n; }
};
```

```
    template<>//template specialization
    struct Factorial<0>{
      enum{ RET = 1;}
    }
```

If a declaration as `Factorial<10>::RET` is encountered by the compiler, then 10 `Factorial` types are allocated, each of them instantiated with an integer and containing its factorial. The recursion is stopped at the invocation `Factorial<0>` because in this case the specialized template is instantiated. Another example of template metaprogramming is the static if statement:

```
    template<bool condition, class Then, class Else>
    struct IF{
      typedef Then RET;
    }

    //partial template specialization
    template<class Then, class Else>
    struct IF<false, Then, Else>{
      typedef RET Else;
    }
```

As in the previous example, the template can be seen as a function evaluated at compile time. In this case the return value is a type, for example: `IF<(1+2 > 4), int, float>::RET i;` The type of `i` is `int`.
Many more example of template metaprogramming (including static `SWITCH`, `FOR`, etc.) can be found in [1].

4. Using a generator to automate components assembly

It is now time to return to our car-themed example and show how generative programming can automate components assembly. After the development of implementation components designed in the domain design phase, component assembly can be performed manually defining feasible configuration: for example, if we want a gasoline powered car without pulls trailer and manual transmission, we can declare the configuration as follow:

```
    struct Config1{
      typedef GasolineEngine<Config1> Engine;
      typedef ManualTransmission<Engine> Transmission;
      typedef CarBody<Transmission> CarBody;
      typedef Car<CarBody> Car;
    }
```

The desired car is instantiated as `Config1::Car my_car`. It is clear that the work of defining possible feasible configurations this way is long and error-prone. A much smarter approach is to provide "abstract description" for feasible configurations: imagine that a gasoline engine is always coupled with a manual transmission and that an electric engine is always coupled with an automatic transmission; then it is possible to use a configuration generator to assembly the components (a configuration generator checks the specification of the system to build and assembles the implementation components). This way, only a small amount of information must be provided (the name of the combination), while the generator is responsible to do the entire job. The generator would look like this (note that CarGenerator is passed to build the engine instead of Configuration):

```
//all possible options
enum option{none, pullsTrailer};
//all possible combination
enum possibleComb{gasolineAndManual, electricAndAutomatic, hybridAndAutomatic};

//the generator, with default value
template<int _possibleEngine = gasolineAndManual, int _option = none>
struct CarGenerator{

//publish option
int option = _option;

//get engine type
//static SWITCH is used, see [1] for more detail
typedef SWITCH<_possibleComb,
      CASE<gasolineAndManual, GasolineEngine<CarGenerator> >,
    CASE<electricAndAutomatic, ElectricEngine<CarGenerator> >,
    CASE<hybridAndAutomatic, HybridEngine<CarGenerator> > >::RET Engine;


//get transmission type
typedef  IF<(_possibleComb  ==  gasolineAndManual),  ManualTransmission<Engine>,
AutomaticTransmission<Engine> >::RET Transmission;

typedef CarBody<Transmission> CarBody;

//the finished car
typedef Car<CarBody> Car;

};
```

At this point, an electric car without pulls trailer is simply selected as: `CarGenerator< electricAndAutomaitc >::Car c`. Since all the information needed to implement a configuration generator derives directly from the configuration knowledge, even the development of the generator could be (ideally) automated. One final note about the implementation: the careful reader has surely noticed that template metaprogramming has a static nature and that polymorphism is resolved at compile time rather than run time. However, if late binding is not needed, there is no reason to undertake the extra performance cost of dynamic dispatching.

5. Conclusions

The shift from object oriented to component oriented design and programming is necessary in order to improve software reuse. Clearly this change is not problem-free: the cost of developing a reusable infrastructure and system is higher than the usual. This cost can be lowered by using generative programming, which allows for automatic architecture code generation and components selection and assembly. Ideally, from the feature model it is possible derive implementation components (provided that a suitable generator is present), and from configuration knowledge it is possible to implement a configuration generator (see fig. 5.1). However, even if a generative infrastructure is provided, still the cost of software increases due to the development of domain specific basic components and generators.

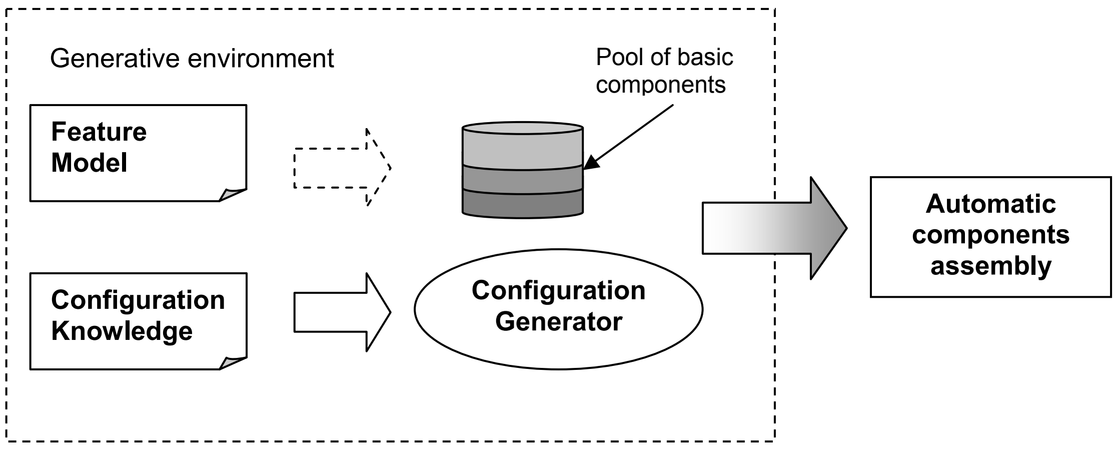

Fig. 5.1 – Global view

In our example, generative programming is mainly used to implement a configuration generator. This is due to the very simple domain I used: once the way to implement variability has been defined, the architectural code is easily derived from the feature diagram. However, restricting the impact of generative programming to just components assembly would be unfair: applying a generative approach on a whole component model would improve the quality of the developed components, in particular for what regards reuse, factorization and separation of functional and non-functional aspects; this is the path to true software evolution. Part two contains a description of an effective use of generative programming for component based software.
To fully complete the automatic generation of a software system, we still need a way to specify the functional code: this can be done using Intentional Programming (see [1] and [16]). Even if the research project have been abandoned by Microsoft in 2002, still a lot of work is done by other private software houses such as Intentional Software (www.intentsoft.com). However, the treatment of such a topic is outside the scope of this paper.

PART TWO – Practice: a generative approach for hard real-time components

1. The domain

An ideal domain for components and generative programming is represented by high integrity systems. The reason is simple: the high cost of verification (~60%-70% of total cost) for such systems would benefit enormously from reuse of already validated components. High integrity systems are usually divided into three categories, differing on the amount of damage taken if software fails:

- *Business critical:* software failure implies loss of money
- *Mission critical:* software failure implies mission's end or decay
- *Safety critical:* software failure implies possible loss of human lives

Among high-critical systems, we can also count hard real-time, multithreaded systems: in this case the architectural code includes also threads synchronization and communication, increasing its importance and complexity. Hard real-time concurrent system can be found mainly in avionics, transportation (trains, cars, etc.) and industrial systems.
The second part of this paper is as follows: first I introduce the Ada 2005 programming language, which is the implementation language for the components; then I propose a domain analysis for the domain; in the end, the basic components and generative environment are described.
All the ideas described in part two have been developed under the "HRT-UML" (Hard Real Time UML) technology project, co-founded by the European Space Agency (ESA). This project aims to define a UML profile suitable for hard real time (thus critical) system; for more information see [4].

1.1 Ada 2005 in a nutshell

Since Ada is not the most known programming language, the following section contains a brief introduction to its semantic. The described semantic is just a subset of the whole, but should be sufficient for our purpose. For a complete description of the Ada 2005 programming language see [5] and [6].
Ada 2005 is the descendent of Ada 95, which is itself a descendant of Ada 83; the language was originally developed under commission of the American Department of Defense. The original target of the language was the high critical, concurrent and real-time software domain. Nowadays, Ada 2005 is a type-safe, multi-purpose object-oriented programming language (Ada 95 had been be first OO programming language standardized by ISO), with particular attention to the semantics of the concurrent constructs. The syntax resembles Pascal one. The most interesting (at least for our purpose) constructs are:

Package

In Ada 2005 it is possible to define encapsulation structures in the form of a package. Packages are basically modules and perfectly suit the concept of component. Each package can have a public and private part.

```
package My_Package is

  procedure p(…
  type T is…
private
  --private declaration here
end My_Package;
```

Enumeration type

Ada 2005 allows describing safely enumeration type this way:

```
type An_Enumeration_Type is (Value1,…,ValueN);
```

It is also possible to define array types with index ranging on enumeration type:

```
type Array_Indexed_On_Enumeration is array (An_Enumeration_Type'Range) of
  An_Enumeration_Type;
```

Generics

Static polymorphism is implemented in Ada 2005 in the form of generics, which resemble of C++ template. Packages or procedures can be generics. Possible parameters for generics instantiation are: values, types, derived types (limited polymorphism), procedures, functions or other packages.

```
--generic procedure
generic
  type T;
```

```
procedure Swap(t1 : T; t2 : T);

--generic procedure with limited polymorphis
generic
  type Derived is new Base with private;
procedure P(T : Derived);

--generic package
generic
  int I;
  type My_Type;
  procedure p(T : My_Type);
package My_Package is…
```

Generics provide for compile time code generation (following the C++ template model).

Concurrency model
The concurrency model defined by the language is complex. We will provide just a basic description, but an interested reader could find a complete description in [6].

Task type
It is possible to define and instantiate a task type this way:

```
--task type declaration
task type My_Task_T(<PARAMETERS>);
--task type implementation
task body My_Task_T is
begin
  loop
        --task code here
  end loop;
end My_Task_T;

--task instantiation
T : My_Task_T(<PARAMETERS_VALUES>);
```

Protected objects
A protected object provides controlled access to shared resources. It is possible to access shared resources by function (read-only), procedure (read-write), or entry (read-write, synchronized on logical condition). Functions allows many concurrent readers, procedures grant mutual read/write access and entry statements maps the semantics of Dykstra's typed channel with guard [7]; entry statements are also used for threads suspension and to realize synchronization agents.

```
--protected type declaration
protected type My_Protected_Type_T(<PARAMETERS>) is
  function f(<PARAMETERS>)
  procedure p(<PARAMETERS>);
  entry e(<PARAMETERS>);
private
  --shared resources here
end My_Protected_Type_T;

--protected type implementation
protected body My_Protected_Type_T is
  function f(<PARAMETERS>) is…
  procedure p(<PARAMETERS>)is…
  entry e(<PARAMETERS>) when Condition is…
end My_Protected_Type_T;

--instantiation
P : My_Protected_Type_T(<PARAMETERS_VALUES>);
```

1.2 An ideal model for hard real-time concurrent systems

The following description derives from the HRT-HOOD design language, one of the ESA standards. HRT-HOOD is an object oriented method based on hierarchical composition of entities, particularly suited from hard real time systems. More information about the method and its semantic can be found in [10] and [11].
A concurrent system can be describe as a series of active entities (active object) synchronizing each other by means of shared resources. Active entities are basically composed by:

- *a control flow*. A control flow can be cyclic or sporadic: cyclic threads cyclically perform the same action, while sporadic ones are suspended waiting for a request to come. In both case, the normal execution can be interrupted to fulfill external requests. An object with a cyclic thread is called cyclic; similarly, an object with a sporadic thread is called sporadic.
- *a synchronization agent:* synchronization agents are modeled using protected types (see prev. chapter) and are used to manage communication between active objects. Each type of synchronization agent is suited to work with a particular type of object (cyclic or sporadic).

Each active entities has a nominal operation, i.e. its default operation (cyclic code in the case of cyclic object).
Communication between active objects is modeled as a deferred transfer of control (see [17]): invocations and their parameters are enclosed in a wrapper and posted in a protected queue owned by the synchronization agent of the called object; the fulfillment of the request is performed asynchronously by the called object, which retrieves the next request from the queue (fig 1.2.1). However it is still possible to define the level of synchronicity between the caller and the called. Request can then be:

- *Asynchronous:* the caller posts the request and continues its normal execution.
- *Loosely synchronous:* the caller posts the request and suspends itself: it will be released by the called just before it starts the fulfillment of the request.
- *Highly synchronous:* like the previous, but the caller is released at the end of request's fulfillment.

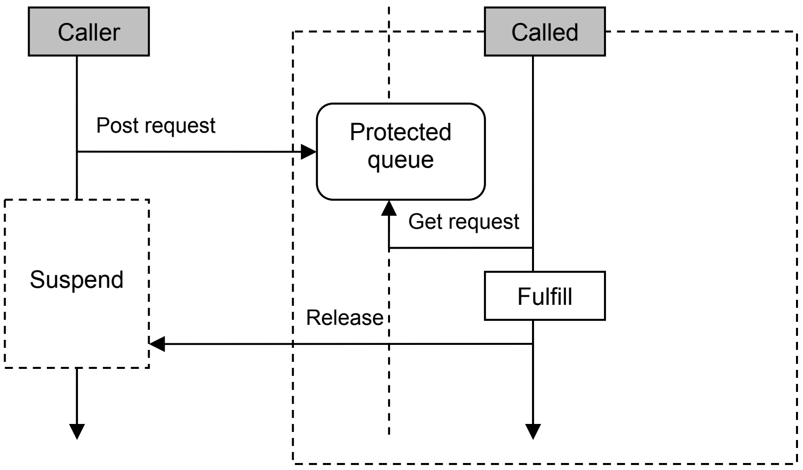

Fig. 1.2.1 – Highly synchronous call

This form of communication, if properly implemented, allows realizing deterministic synchronous communication, leading to static verification of system's behavior and schedulability. Clearly all this basic structure can be highly factorized, as explained in the next chapter.
One final note: for the implementation, the Ravenscar profile (see [12]) has been used. The Ravenscar profile is an Ada 2005 subset allowing for static analysis of concurrent systems. Ravenscar-compliant code has the following properties: static existence model, static synchronization and communication model, deterministic memory usage and deterministic execution model ([12]). Even if such a technicality could seem not important for our aim, it must be noted that the great effort on testing and quality needed by the component development process could be lighter using a proper semantic. The use of a totally deterministic concurrency model definitely improves the quality of the components.

2. The component model

The aim of the component model is to provide a set of basic entities that can be easily modified to accommodate designers' needs. Among the basic entities, there are a set of parametric thread types and synchronization agent types. They are composed to obtain hard real time archetypes of object (HRT archetypes): composition rules are well defined in the configuration knowledge. HRT archetypes can be cyclic or sporadic and are parametric on the functional code. In other words: a control flow must be composed with a synchronization agent in order to obtain an archetypes; the obtained archetype must be composed with functional code (a set of procedures/functions) and synchronization aspect to obtain the desired object. Synchronization aspects are information about the level of synchronization (see previous chapter) desired for a given procedure. Practically, the programmer just specifies the functional code and the synchronization level, and the model automatically builds the entire desired infrastructure (synchronization agent, etc.) using the generic facility of the language. In addition, a full set of small, independent and highly configurable components are provided: they are the low level building blocks of the model. For example, the basic library contains a component called Generic_Procedure: its required interface is the type of parameters for the procedure, the functional code and the synchronization aspect; with this information, all the synchronization architecture is build. I provide this little piece of code to make this concept clear:

```
generic
  type My_Param_T is new Param_T with private;
  procedure My_Functional_Code(P : My_Param_T);
  My_SA : Synchronization_Aspect_T;
package Generic_Procedure is
  --caller calls this function
  procedure Functional_Code(P : My_Param_T);
  --publish synchronization level
  SA : Synchonization_Aspect_T := My_SA;
  --eventually release the caller
  procedure Release;
private
  Manager : Synchonization_Manager_T;
end Generic_Procedure;
```

where My_Param_T is a wrapper type containing a record for each parameter; Synchronization_Aspect_T is an enumeration type ranging over the possible synchronization levels; Synchonization_Manager_T is the infrastructure needed to suspend the caller if the request is (loosely or highly) synchronous: it is realized using protected types. It should be noted that the required interface is explicitly mapped in the code in a defined location (generic parameters).
The Generic_Procedure component is instantiated this way:

```
--parameters profile
type My_Procedure_Params_T is new Param_T with
record
  I : Integer;
  B : Boolean;
end record;

--functional code
procedure My_Procedure_Code(P : My_Procedure_Params_T) is…

--component instance
package My_Procedure_Component is new Generic_Procedure(
        My_Procedure_Param_T,
        My_Procedure_Code,
        Loosely_Synchronous);

--publish the procedure for the caller
procedure My_Procedure(P : My_procedure_Param_T) renames
  My_Procedure_Component.Functional_Code;
```

For each method of the provided interface, an instance of the Generic_Procedure component is needed. A simplified feature diagram of HRT components is shown in fig. 2.1.
When the component is instantiated all the infrastructure is build at compile time by the template facility of the language. The component My_Procedure_Component is used to build the called object. The caller simply calls My_Procedure and, depending on the level on synchronicity, it could suspend. To release the caller, the called invoke My_Procedure_Component.Release at the proper moment.
Building blocks, as the Generic_Procedure component, are assembled automatically by the generative environment in order to build complex multithreading systems. The architectural and functional code to manage communication and synchronization is built on demand: the programmer just has to develop the proper functional code and specify the parameters to compose the system. Various levels of abstraction are possible: on top of the archetype level, it is possible to define interfaces (allowing for a form of static polymorphism) or directly components (see fig. 2.2). At the moment, implementation inheritance is not allowed, and only single interface inheritance is possible: however, support for multiple interface inheritance will be probably added in the near future. I will not go into full details of the mapping (which will possibly be the main subject of a future publication): the reader just need to know that components are

mapped as packages and interfaces as packages generic on functional code. This choice implies that only static polymorphism is used, allowing for a feasible verification.

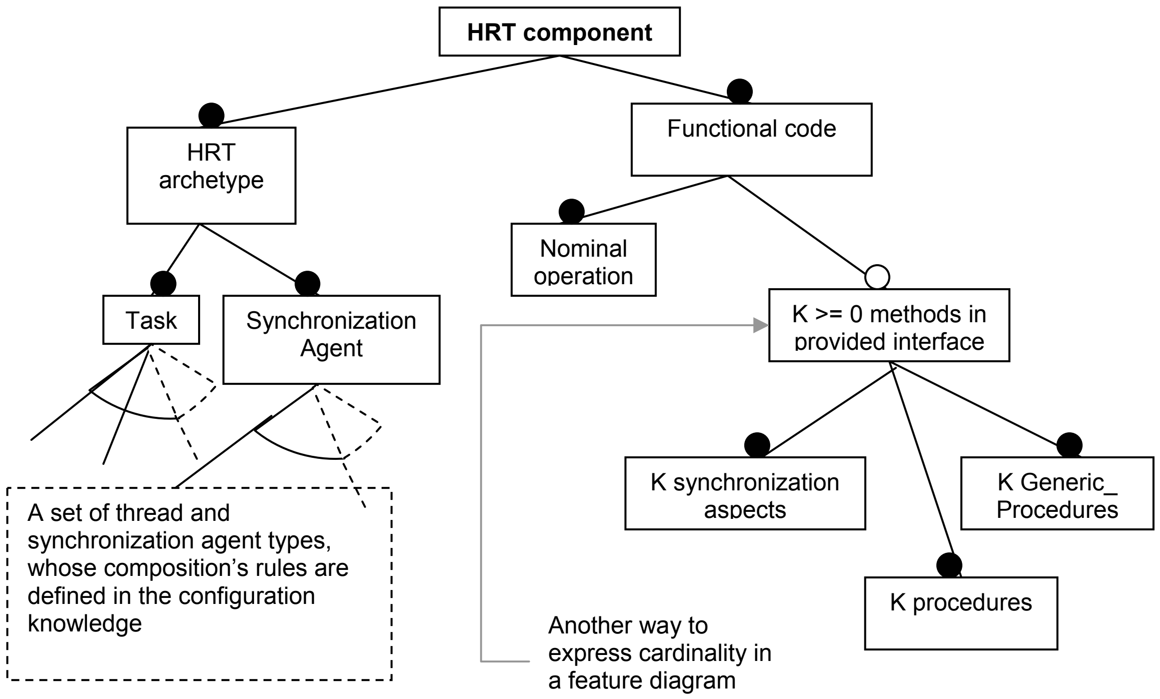
HRT component
HRT archetype
Functional code
Task
Synchronization Agent
Nominal operation
K >= 0 methods in provided interface
A set of thread and synchronization agent types, whose composition's rules are defined in the configuration knowledge
K synchronization aspects
K Generic_ Procedures
K procedures
Another way to express cardinality in a feature diagram

Fig. 2.1 – Simplified feature diagram for an HRT component

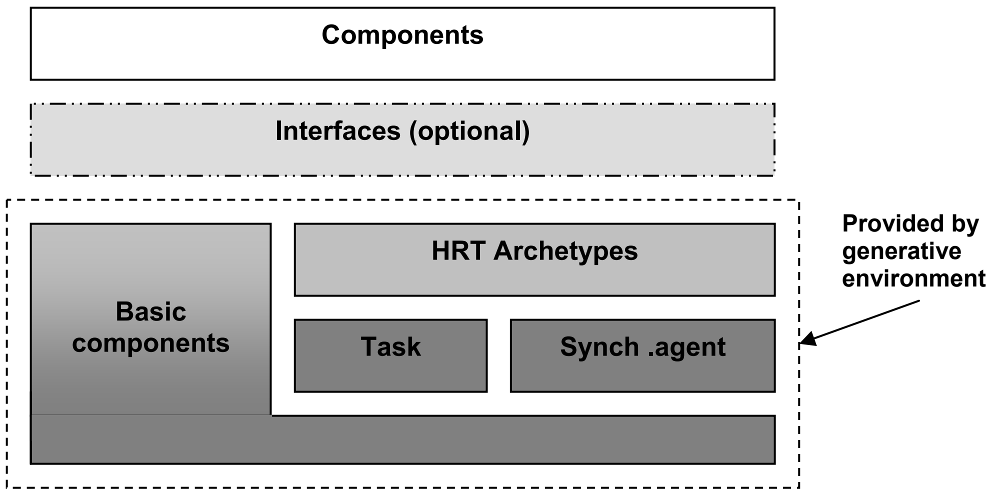

Fig. 2.2 – Architecture and abstraction levels

Briefly, a full component is a composition of a synchronization agent, a task (relying on the synchronization agent), an HRT archetype object (composed by both the task and the synchronization agent), a set of implementation of the Generic_Procedure component (relying on the archetype object) and functional code: these components are assembled in a layered way, improving encapsulation and information hiding. When an external request arrives, the execution (and synchronization) is delegated to the lower layer using the delegation pattern (see [22]). In particular, the Generic_Procedure component instance delegates request's queuing to the synchronization agent (via the HRT archetype) and caller suspension to the synchronization manager; similarly, once the threads has extracted the request from the queue (owned by the synchronization agent), it delegates the execution to the functional code and release the caller invoking `Release` on the corresponding Generic_Procedure instance. It should be noted that no performance hit is caused by using delegation, since Ada 2005 presents a semantic structure (`renames`) allowing for high performance delegation. For a graphical representation, see fig. 2.3. Many macro components are then assembled to build a full system.

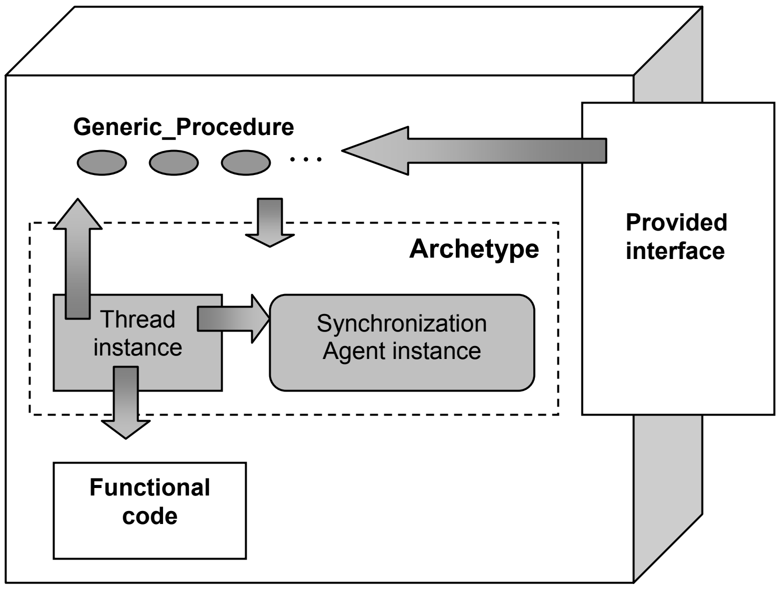

Fig 2.3 – Component's internal

Hard real-time components are finally mapped in the source code as an aggregation of package, providing public-visibility to the provided interface only. Instances of the same components can differ on real-time attributes, as execution priority, interval between cyclic execution, etc.: real time attributes are parameters for component instance.

2.1 Summing it all up: the Voter component

In the previous chapter I described how simple hard-real time component are defined: many different components can then be composed to define macro components. In this section, I provide a simplified example of such macro component, using the Voter example. In a typical high integrity system, fault tolerance can be performed using redundant hardware coupled with software. Imagine a controller of oxygen in a dangerous environment (a mine): clearly this system is safety critical. The controller cyclically controls the level of oxygen and if it is below a certain threshold, a warning message is sent. A way to improve safety is to deploy three different controller, compare their results and consider valid the survey only if at least 2 controllers over 3 agree. If a certain controller continues to supply results that don't agree, then it can be suspended, since the problem is probably caused by hardware failure. Such architecture is very common in high integrity systems. I won't go into the full detail on how a voting algorithm can be designed or how synchronization and communication between three independent tasks can be obtained: I prefer to concentrate on the software architecture solution.
The voter must contain a buffer on which the results from the controllers are memorized; in addition a voting task must be present. Once the buffer is full, it must trigger the voter task. Both the buffer and the voter are HRT components with a defined provided/required interface; delegation is again used to fulfill requests. To achieve reuse, we need to design such a voter as general as possible: its required interface contains at least the wrapper type of the results and an implementation of the interface Sender, witch provide for the communication of the result. The wrapper type of the result must be derived from the abstract class Result_Type and implement the `equals` method. Clearly the result type must contain additional information, such as the timestamp: two identical results too far in time from each other are different. A graphical representation of the component (this time using simplified UML) is represented in fig (2.1.1).

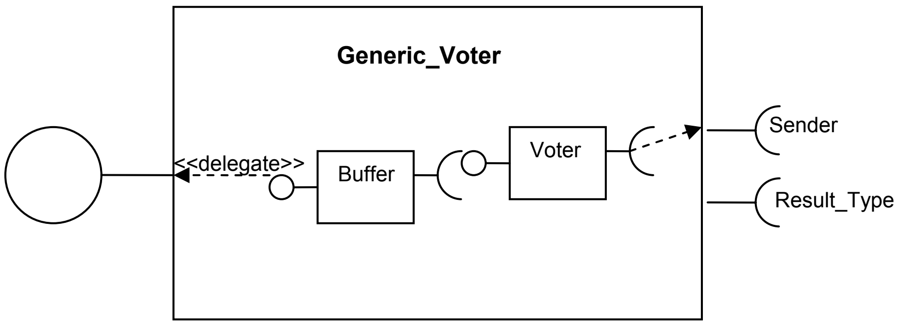

Fig. 2.1.1 – The Generic_Voter component

The voter component is mapped as a package generic on the result type and an implementation of the Sender interface. Additional parameterizations can be the type of voting algorithm, time interval to consider two results equal and many others. The following code represents the Voter component:

```
generic
  --required interface
  type My_Result_Type is new Result_Type with private;
  package My_Sender is new Sender(<>);
  --additional parameters...
package Generic_Voter is
  --provided interface
  procedure Vote(R : My_Result_Type);
  --additional methods...
```

The Vote procedure is invoked by each controller once they have performed their evaluation.
The Generic_Voter component is still a composition of HRT archetype instances, Generic_Procedure instances and so on: this simple example shows how the described component model can be expanded to define reusable domain-specific basic components. At this point, generative programming can be used to compose and instantiate many basic components as Generic_Voter.

3. The generative environment

The generative approach is an evolution of the one described in [8] and [9].
Given the importance of software verification and validation, the importance of correctly mapping the HRT-UML semantic to Ada 2005 is vital: producing code whose verification is not feasible would waste all the effort spent in developing a generative approach and environment. In addition, design semantic must be preserved in code (up to the generator) and executable (up to the compiler). For these reasons, polymorphism is mapped in a static way using generics and the communication model is mainly asynchronous (see [9]); similarly, this explains again the choice of producing code abiding by the Ravenscar profile (see ch. 1.2 of part two).
In order to produce a simple (to develop and verify) generator, the tools must be chosen carefully. Everything starts with an HRT-UML diagram and a formal description of non-functional requirements (worst case execution time, time-out, execution and ceiling priority, synchronization aspects, etc.). A preliminary validation (to check completeness and Ravenscar-compliance) is performed on the diagram. The model is first encoded in standard XMI (XML Metadata Interchange) and then it is mapped to XML keeping the useful information only. At this point, proper code generation is done on two steps.

Step one: using XML and XSLT as generative environment

The XML document just produced is transformed in Ada 2005 source code using XSLT. XSLT (eXtensible Stylesheet Language Transformation) is an XML application for specifying rules by which an XML document is transformed into something other. The output can be another XML document, an HTML page, SQL query and many others. XSLT has been used as a code generation engine in other application, for example the ESA project described in [14]. Using such a simple, low-cost, common and verified technology implies that a lot of development tools are available: in chapter 3.1 of part one this was described as one of the most important aspect of choosing an already available generative environment. In addition, it is possible to obtain a generator for other target platforms just by developing new stylesheets, thus keeping the data representation (XMI and XML) and the application performing the transformation (for example a Java program). Practically, XSLT transformation and XSLT stylesheets act as a compiler on the XML document.
The output of the first generative step is a set of Ada 2005 components (or interfaces) relying on the basic library, i.e. HRT archetypes.
Generative programming is applied at an earlier stage compared to the configuration generator described in part one: this allows dividing functional from non-functional code both on design and source code, improving simplicity and factorization. In addition, the design is free to evolve, since the functional code can still be used in a different architecture. Imagine for example to decide to change the way two components synchronize during a particular communication: it will be sufficient to change a Synchronization_Aspect in the design and (re)generate the code, leaving the functional code untouched. A component approach is fundamental to implement this separation, because entities are composed by general, configurable and virtually independent components.

Step two: compile-time components assembly with template

The components created in step one are generic packages: they still need to be instantiated and composed using the Ada 2005 compiler and the provided basic library. If a set of composition constraints is encoded in the XML document, then it is possible to produce a configuration generator for the user-defined components. However, since Ada 2005 does not support partial template specialization (generic overloading), a configuration generator should still be an XSLT stylesheet transforming an XML document representing the configuration knowledge. Unfortunately, this feature has not yet been fully considered in the HRT-UML project. The overall generative process is represented in fig. 3.1.

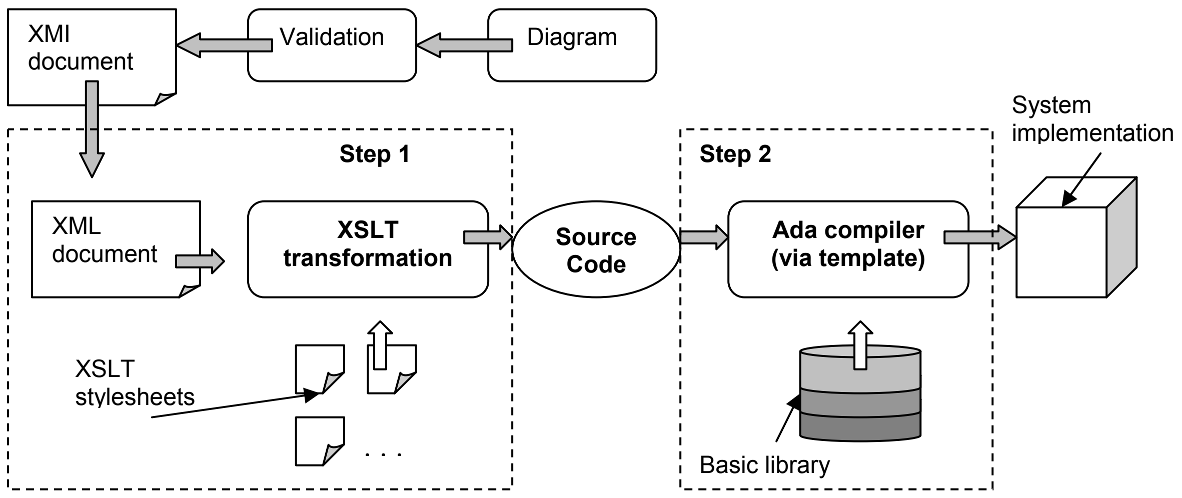

Fig 3.1 – Generative process

4. Conclusions

While the first car-themed example focused mainly on automatic component assembly, in the second part I've shown how generative programming can used to facilitate design evolution and to compose components: this is a clear step ahead towards automation of software development.
In order to successfully apply an evolutionary approach to software development, specification must be expressed intentionally and architecture's implementation must be divided from functional code: in this context, components are necessary to divide the system in many independent entities and generative programming is of great help to compose them and generate architectural code. This paradigm shift implies a subtle change in the development process: architecture must be factorize to define basic entities whose mapping to source code generates a set of independent, configurable and adaptable components. This should be done in two stages: first at the macro domain level, and then at the micro domain one. In our example, the macro domain is high integrity concurrent systems; however, to improve the development process, a similar approach must be applied to micro domains contained in the macro one, for example ground segment communication, on-board system, fault-tolerance infrastructure etc. The Generic_Voter briefly described in section 2.1 is an example of such micro domain specific components. This second step is at the moment totally absent from the HRT-UML project, and (in my personal opinion) should be considered in the next future: once a library of micro domain specific components is provided with the generative environment, engineering with reuse by generative programming can be applied. A similar divide and conquer approach must be used at any possible identified stakeholder layer. On the other hand, architecture and code factorization influences the meta model, eventually adding additional abstraction levels. In the original HRT-HOOD design method (which provides for a primitive Ada 95 mapping, see [11]), the only possible abstraction level was the objects one: to achieve factorization, task, synchronization agent, HRT archetypes and interface layers has been defined and mapped on the code. Increase expressivity and layering at the meta level is the key to define reusable domain-specific components. From this point of view, correctly mapping the meta level architecture to code is fundamental: the importance of developing a generator which produces a high-quality, high-performance, readable and semantically correct mapping must never be neglected.
Generative programming perfectly marries this scenario, being the ideal method to realize and compose component-based architectures: as the factorization of domain's features supplies applications families' building blocks in the form of components, generative programming is the tool to use to move to evolutionary software development and automatic components assembly, improving design reuse and effectiveness.

5. References

[1] K. Czarnecki, U. W. Eisenecker – Generative programming: Methods, Tools, and Applications, Addison Wesley
[2] C. Szyperski – Component Software: Beyond Object-Oriented programming, Addison-Wesley
[3] I. Crnkovic, S. Larsson, M. Chaudron - Component-based Development Process and Component LifecycleT, 27th International Conference Information Technology Interfaces (ITI), IEEE 2005.
[4] S. Mazzini, M. D'Alessandro, M. Di Natale, A. Domenici, G. Lipari, T.Vardanega – HRT-UML: Taking HRT-HOOD onto UML. In Rosen, J.P., Strohmeier, A., eds.: Reliable Software Technologies - Ada-Europe. Number 2655 in LNCS, Springer (2003)

[5] T. Taft, R. Duff, R. Brukard, E. Ploedereder, eds. - Consolidated Ada Reference Manual — International Standard ISO/IEC-8652:1995(E) with Technical Corrigendum 1. Volume 2219 of LNCS. Springer (2000) ISO/IEC 8652:1995.
[6] A. Burns, A. Wellings – Concurrency in Ada, Cambridge University Press
[7] S. Dabbs Halloway – Component Development for the Java Platform, Addison Wesley
[8] M. Bordin, T. Vardanega, - A New Strategy for the HRT-HOOD to Ada Mapping, Reliable Software Technologies.Ada-Europe 2005, Springer. LNCS(3555):51.66
[9] M. Bordin, T. Vardanega, - Automated Model-based Generation of Ravenscar-compliant Source Code, In: Proc. 17th Euromicro Conference on Real-Time Systems, July 2005, IEEE, 69.77
[10] HOOD User's Group, J.P. Rosen – HOOD: an Industrial Approach for Software Design
[11] A. Burns, A. Wellings – HRT-HOOD: A Structured Design Method for Hard Real-Time Systems, Elsevier Science
[12] A. Burns, B. Dobbing, T. Vardanega – Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems. Technical Report YCS-2003-348, University of York (UK) (2003) Approved as ISO/IEC JTC1/SC22 TR 42718.
[13] W. Zhao – A Unified Approach to component assembly Based on Generative Programming
[14] V. Cechticky, A. Pasetti – Generative Programming for Space Applications
[15] S. Chiba – Generative Programming from a Post Object-Oriented Programming Viewpoint
[16] C. Simonyi - The Death Of Computer Languages, The Birth of Intentional Programming. Microsofr Research Technical Report MSR-TR-95-52
[17] ESTEREL Technologies: Safety-, Mission- and Business-Critical Embedded Software. http://www.esterel-technologies.com/v3/?id=39425 (2005)
[18] J.W.S. Liu - Real-Time Systems. Prentice-Hall (2000)
[19] W3C: XSL Transformations (XSLT). http://www.w3.org/TR/xslt (2005)
[20] W3C: Extensible Markup Language (XML). http://www.w3.org/XML/ (2005)
[21] N. Halbwachs - Synchronous Programming of Reactive Systems. Kluwer International Series in Engineering and Computer Science, 215. Kluwer Academic Publishers (1993)
[22] E. Gamma, R. Helm, R. Johnson, J, Vlissides – Design Pattern: Elements of Reusable Object-Oriented Software, Addison Wesley