GENERATIVE PROGRAMMING FOR SPACE APPLICATIONS

V. Cechticky, A. Pasetti

**ETH-Zentrum, Institut für Automatik, Physikstr. 3, 8092 Zürich, Switzerland*
***P&P Software GmbH, Peter-Thumb Str. 46, Konstanz, 78464, Germany*
cechti@aut.ee.ethz.ch, pasetti@pnp-software.com

Abstract

Generative programming is a software engineering paradigm that, given a particular requirements specification, allows an application implementing those requirements to be automatically generated by configuring and customizing the components and the architecture provided by a software framework. Generative programming raises the level of abstraction at which an application is defined and implemented. It can potentially increase the ease of development and the reliability of the final application and is therefore of interest to domains – like space – where there is a need to contain costs while maintaining or enhancing overall quality. This paper is divided into two parts. The first part introduces the generative programming paradigm in general. The second part presents the results of a study done by the authors for ESA-Estec (contract number 15753/02/NL/LvH) to apply generative programming techniques to automate the instantiation of the AOCS Framework. The AOCS Framework is a prototype framework for on-board control applications.

1 From Software Frameworks to Generative Programming

Software frameworks [Gam95, Fay99] are a software reuse technology that promotes the reuse of entire architectures within a narrowly defined application domain. They propose an architecture that is optimized for all applications within this domain and provide configurable components that support their instantiation. Framework can and have been used as stand-alone artifacts to facilitate the development of applications that are developed manually but they can also serve as a basis upon which to build a generative programming environment for the automatic development of applications within the framework domain. The term “generative programming” is used here in the sense of [Kza00] to designate a programming paradigm that allows an application to be automatically constructed from its specifications. The concrete realization of a generative environment aimed at a specific domain requires the following steps:

1. Definition of the boundary of the target domain to be covered by the environment.
2. Definition of a model of the target domain that describes the common and variable properties of the applications in the domain and their mutual relationships.
3. Definition of a formalism for specifying applications in the domain.
4. Definition of a common architecture for applications in the domain.
5. Development of the configurable and customizable components required to support the implementation of the domain architecture.
6. Definition of a formalism to describe the configuration and customization of the components.
7. Development of a generator that can automatically transform an application specification into a domain configuration and that can then generate a concrete application from the domain configuration.

A framework is a fundamental constitutive element of a generative programming environment because it covers steps 3 and 4: it defines the common domain architecture and it acts as a repository for customizable and configurable components to build applications. Steps 1 and 2 and 5 to 7 will often require the definition of a *domain-specific language* (DSL) through which the specification of the target application can be expressed and of a *domain-specific compiler* capable of translating an application specification expressed in this language into source code that implements the specification. The domain-specific language will normally be aimed at a narrow target and will therefore offer abstractions and constructs that, being specific to that domain, make it easier to define applications than would be the case if some general-purpose language were used. Application development therefore becomes faster. Applications also become more reliable because their implementation is guaranteed to be compliant with their specification since the implementation is automatically generated from the specifications.

The ultimate objective of a generative programming environment is to provide a tool that allows the application specialist to directly specify his system and to automatically generate the application code from these specifications. In the space domain – as in many other embedded domains – the application specialist is often not a software specialist. Thus, for instance, the engineer that defines the control system of a satellite is usually not a software specialist. This gives rise to a two-stage development process with the specifications being written (usually in informal language) by the application specialist and their implementation being delegated to a software engineer who is normally not an application specialist. Since the functionalities of space applications are increasingly located in software, the resulting situation is one where the heart of an application – its software – is developed by persons who have only a limited understanding of the application itself. This is clearly undesirable and the interface between application and software specialists is a common source of specification misunderstandings and cost and schedule overruns. A generative programming environment can remove this interface and can accordingly bring substantial benefits both in cost and schedule terms.

The generative programming approach is not new to space projects. In the control domain, tools like Matlab already allow a control engineer to define a control system using abstractions with which he is familiar within an intuitive and easy-to-use environment. The environment additionally includes autocoding facilities that can automatically generate the code implementing the design defined by the user. A concrete example of the application of generative techniques to a space mission is the on-board control software of Proba [Mel02].

Tools like Matlab however are aimed at wide domains and therefore are unsuitable to model more than a fraction of a spacecraft or ground subsystem. Full automatization of the software development process for an entire on-board or on-ground subsystem requires the availability of a generative programming environment that is aimed at that particular subsystem. The next section describes work being done under an ESA research contract to prototype such an environment for the AOCS subsystem.

2 The Automated Framework Instantiation Project

In June 2002, P&P Software together with ETH-Zurich as subcontractor started a project for ESA-Estec (contract number 15753/02/NL/LvH) to investigate automated instantiation techniques for software frameworks[1]. The framework instantiation process is the process whereby a software framework is adapted to the needs of a specific application in its domain. The objective of the project was to prototype a generative environment for framework instantiation.

The framework instantiation process takes place in two basic steps: (1) the application-specific components required by the application are constructed. Their construction is guided and constrained by the need to adhere to the framework design patterns and to implement the framework interfaces; (2) the application-specific components and the framework components are configured and composed together to construct the final application. The instantiation approach considered in our project only covers the second step: it was assumed that the framework provides

[1] The views expressed in this paper are those of its authors only. They neither commit nor represent official ESA thinking.

all the components required to instantiate the target application. This assumption is not unrealistic: a mature framework will normally offer a sufficient complement of default components implementing all or nearly all functionalities required by applications in its domain. This assumption would also typically be satisfied in embedded domains where there is a need to construct several variants of the same basic product. These variants are built from the same pool of components but differ from each other because they configure the components differently.

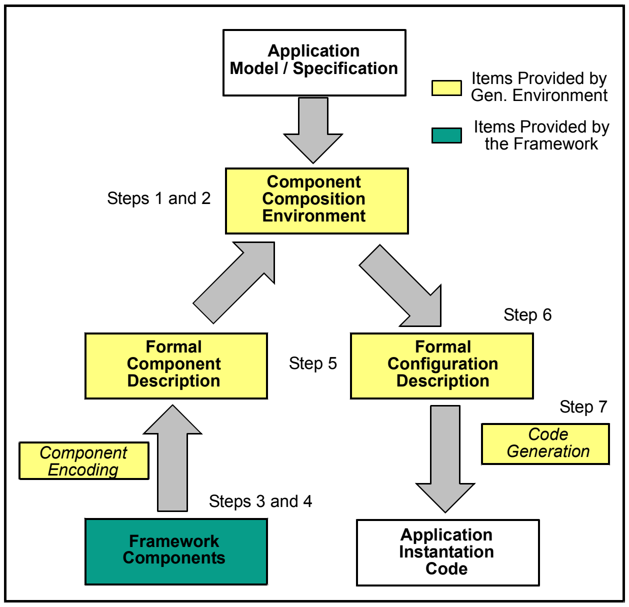
Application Model / Specification
Items Provided by Gen. Environment
Items Provided by the Framework
Steps 1 and 2
Component Composition Environment
Step 6
Formal Component Description
Step 5
Formal Configuration Description
Step 7
Code Generation
Component Encoding
Steps 3 and 4
Framework Components
Application Instantiation Code

Figure 1: Generative Environment for Framework Instantiation

Figure 1 shows how the general structure of a generative environment as given in the previous section can be adapted to cover the task of automating the instantiation of a software framework. The figure is labeled with the steps identified in the previous page. The flow of activities shown in the figure has two inputs. At the top there is the specification of the target application. At the bottom left-hand corner there are the components offered by the framework. The target application must be constructed by suitably configuring the framework components. The component configuration process is performed in a component composition environment. This imports the formal descriptions of the components and outputs a formal description of the target configuration (perhaps expressed in DSL). The application instantiation code is derived from this formal description (code generation process).

Since software frameworks are not standardized, the investigations and prototyping activities performed in our project had to be referred to one particular software framework. We chose as our sample framework the *AOCS Framework* [Pas01, Pas02]. This is a research prototype framework that models the AOCS domain. It is component-based and object-oriented and uses as adaptation mechanisms inheritance and object composition. It exists in two versions, one coded in C++ and one coded in Java. The Java version [Cec02] was used for this project. At present, instantiation of an application from the Java framework requires code to be written that specifies how each component should be configured and how different components should be linked together. The objective of the project was to automate this instantiation process.

Since the Java version of the AOCS Framework is organized as a set of (non visualizable) JavaBeans, it was initially believed that automation of the instantiation process might be achieved simply by importing the framework beans into a commercial bean development tool (e.g. Visual Age, CodeWarrior, Jbuilder, etc) and by then using the bean configuration and bean composition facilities of such a tool to construct the application. However, it quickly emerged that this is not possible because the instantiation operations of an on-board application are too complex to be reduced to those typical of bean-based applications. This initial approach is described in a dedicated project web site [Jbp02].

The alternative approach that was eventually adopted was based on generative programming techniques. As already mentioned, the basic way to automate the instantiation process is to develop a domain-specific language that allows the application specification to be expressed in a formal way and to build a framework-specific compiler for this language that allows a specification expressed in the language to be transformed into an instantiation sequence for the target framework. This however requires the user (the person writing the application specifications) to learn the formal language which, in practice, will often be difficult. The project therefore took a slightly different route and developed a GUI-based *environment* where the user can express his requirements in an informal manner using graphical means with the aid of context-specific information automatically provided by the environment itself. The environment is then responsible for translating the requirements implicitly formulated by the user into a formal description of the target application and for translating (compiling) this description into an instantiation sequence. Thus, the derivation of the application is still done in two steps - formal specification of the target application and compilation of the specifications - but these steps are now hidden from the user who only interacts with a user-friendly and GUI-based environment. This environment is called the *OBS Instantiation Environment.* It is described in detail dedicated web site [Afp02]. The next section gives an overview of its architecture and the technologies it uses.

3 Overview of the OBS Instantiation Environment

A generative system for a framework is necessarily tailored to its target framework. The degree of reusability of the environment as a whole is therefore very low. The one-of-a-kind nature of this type of environment and their intrinsic complexity mean that there is a risk that their development, although technically feasible, will often be resisted on economic grounds. Reliance on existing technology is probably the only way to make their development attractive from a practical point of view. For this reason, in this project, an effort was made to base the OBS Instantiation Environment on mainstream technologies. Figure 2 illustrates the structure of the OBS Instantiation Environment. It reworks figure 1 to show which particular elements are offered by the environment and which technologies are used to realize them. As indicated in the figure, the OBS Environment is based on three basic principles:

- Formal encoding of component properties and of the application configuration uses XML grammars.
- The code generation is performed using an XSLT program.
- The application configuration is done using a standard Java bean builder environment.

Adoption of these three principles reflect the constraint to base the OBS Instantiation Environment on mainstream technology in order to contain its development costs. The choice of XML as an encoding format for the data describing the framework components and the application configuration has two advantages. Firstly, XML is a widely used and widely understood technology which is very well supported at the level of processing tools (many of which are of high quality and available for free as public domain software). This helps minimize costs. Secondly, once XML is adopted, it is natural to consider XSLT programs as code generators. The code generator is often one of the most complex and most expensive element of a generative programming environment. The use of a standard language like XSLT gave a decisive contribution to contain the costs of the OBS Instantiation Environment. Note that XSLT programs are normally used to transform an XML document into another XML document. However, the experience of this project is that they can also be used as simple but effective code generators.

Another complex and expensive element of a generative environment for frameworks is the configuration tool where the designer defines the application by configuring and assembling the framework components on the basis of the particular requirements of his application. Complexity and cost normally derive from the desire to make this tool as user-friendly as possible and, ideally, to make it GUI-based. The choice made in our project was to use as component configuration and assembly tool a standard bean builder tool for JavaBeans-based applications. Note that, as already mentioned, it is in general not possible to directly use a bean composition tool as application instantiation environment. This project accordingly used a special type of transformation of the framework components to make use of a standard bean builder tool possible.

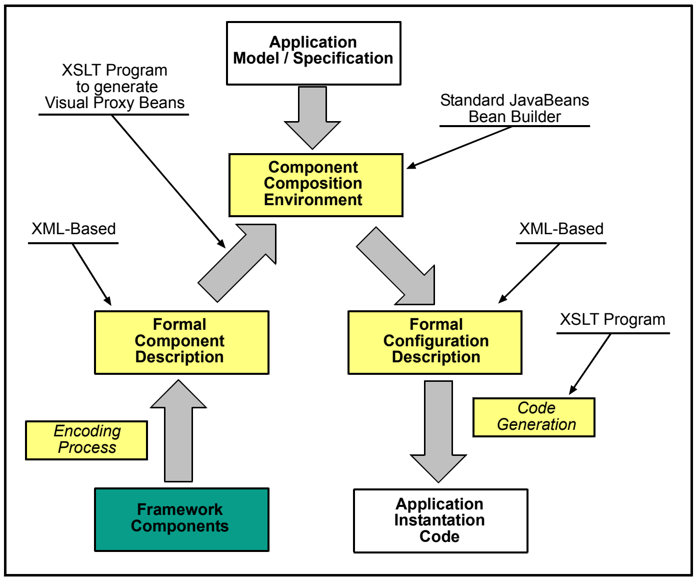
Application Model / Specification
XSLT Program to generate Visual Proxy Beans
Standard JavaBeans Bean Builder
Component Composition Environment
XML-Based
XML-Based
Formal Component Description
Formal Configuration Description
XSLT Program
Code Generation
Encoding Process
Framework Components
Application Instantiation Code

Figure 2: Technologies used in the OBS Instantiation Environment

3.1 The Instantiation Problem for the AOCS Framework

The OBS Instantiation Environment is concerned with automatizing the instantiation of the AOCS Framework. The first conceptual step in its development must accordingly be a more precise definition of what exactly is meant by "instantiation of the AOCS Framework". This is necessary because a process can only be automatized if it is precisely defined. In the case of the AOCS Framework, the instantiation of an application from the framework consists in performing an ordered sequence of the following six *instantiation operations*:

- Instantiation of a framework component
- Setting the value of a component property
- Setting the value of an indexed property
- Setting the value of a static property
- Linking an event-firing framework component to an event-listening framework component
- Adding a component to an object list

Note that the component properties can be either of primitive type or of class type. Thus, the second operation also covers the case of object composition.

The definition of an application instantiated from the AOCS Framework thus reduces to the definition of an ordered set of operations of the type listed above. In accordance with the component-based character of the AOCS Framework, all the instantiation operations can be expressed in terms of the methods declared by the external interfaces of the framework components. The instantiation sequence can therefore be encoded as an ordered set of method calls performed upon the framework components. The *instantiation problem* addressed by the OBS Instantiation Environment is the problem of translating a particular application specification into an ordered set of instantiation operations which, when executed, will result in the instantiation of an application that implements the initial specifications.

3.2 Component Encoding Process

The primary input to the instantiation process are the framework components. These components - suitably configured - are the building blocks for the target application. The OBS Instantiation Environment consequently needs to manipulate these components. It therefore needs to have access to their description. The process whereby a formal description of the components is constructed is called *component encoding process* (see figure 2).

Since the OBS Instantiation Environment is only concerned with the instantiation process, it only needs information about the part of the framework components that comes into play during the instantiation process. Given the instantiation model adopted here for the target framework, the only characteristics of the framework components that need to be encoded are:

- The properties they expose
- The events they fire
- The events they listen to
- The object lists they manage

This information is encoded using an XML grammar. In the case of the AOCS Framework, the framework components take the form of Java classes. For each Java class, an XML document is generated that describes the instantiation-relevant characteristics of the component that the class represent. For reasons that will become clear in the next subsection, these XML documents are called *Visual Proxy Descriptor Files*.

The visual proxy descriptor files are automatically constructed by a dedicated Java application. This application needs a way to determine which properties are exposed by each class, which events it can fire or listen to, or which object lists it manages. In the AOCS Framework, this type of information is stored in the names of the methods implemented by the class. Naming conventions are used that define the behaviour of the class during the instantiation phase. For the property and event part of this behaviour, the naming conventions are the same as those prescribed by the JavaBeans standard. For the object list part, a framework-specific naming convention is used. The application that constructs the visual proxy descriptor files therefore only needs to have access to the API implemented by a class. It is implemented as a parser that processes all the methods implemented by the class and, by checking whether their signature conforms to certain patterns, determines which properties it exposes, which events it fires or processes, which object lists it manages. This information is then encoded as an XML document.

3.3 The Visual Proxy Components

The application designer interacts with the OBS Instantiation Environment primarily through its *component composition environment* (see figure 2). This is the tool that allows him to configure the framework components and to assemble them to construct the target application. Ideally, it would be desirable to set up this composition environment to directly manipulate the framework components. This is in general not possible because framework components are not designed to be manipulated in a composition environment. The alternative approach taken in our project is to replace the framework components with proxy components that mimic their behaviour during the instantiation phase. These proxy components are called *visual proxies*. The visual proxies are the components that are imported in the composition environment and they are the components that are actually manipulated by the designer. However, since the proxy components, *as far as the instantiation process is concerned*, are equivalent to the framework components, the designer has the illusion of manipulating the latter.

Thus, to each framework component a visual proxy component is associated. A visual proxy component must have two chief characteristics: (1) it should exhibit the same behaviour as its associated framework component during the application instantiation, and (2) it should be capable of being imported, displayed, manipulated, and configured within the component composition environment. Given the instantiation model adopted here for the target framework (see section 3.1), the first requirement implies that a visual proxy should:

- Expose the same properties as its associated application component.
- Listen to the same events as its associated application component.
- Fire the same events as its associated application component.
- Expose the same adder methods as its associated application component.

Compliance with the above means that, for the purposes of application instantiation, a visual proxy component exposes the same API as its associated framework component. The visual proxy components are constructed from their visual proxy descriptors. Since the latter are implemented as XML documents, the construction of the classes from which the visual proxy components are instantiated is best done using an XSLT program. As discussed below, the composition environment is based on a JavaBeans bean builder tool and the second requirement is therefore satisfied by implementing the visual proxies as visualizable JavaBeans components. The visual proxies are therefore implemented as Java beans and, in order to facilitate their manipulation, they are complemented by beaninfo components, bean editor components and a bean customizer. The bean customizer is common to all visual proxy beans whereas the bean editors and the beaninfo are specific to each visual proxy bean. They too are generated by XSLT programs. Figure 3 shows the conceptual path from a framework component to its visual proxy component and its associated support beans.

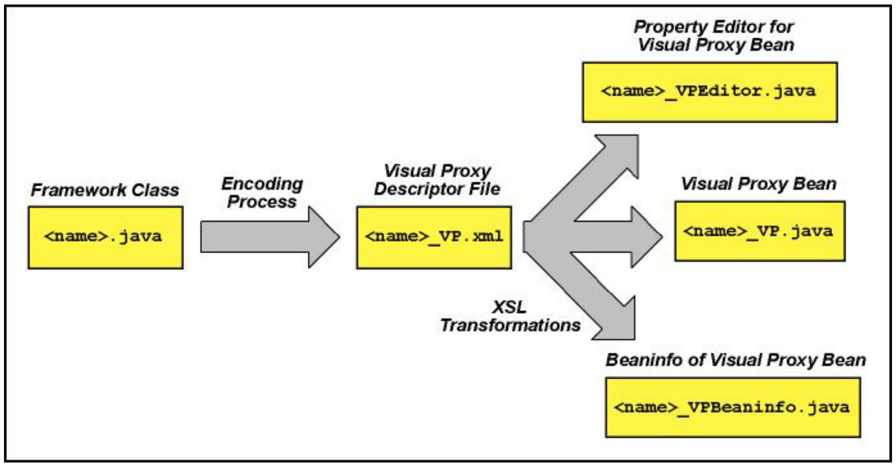

Figure 3: From Framework Components to their Visual Proxies

3.4 The Component Composition Environment

The *composition environment* is the part of the OBS Instantiation Environment where the designer configures and assembles the components to construct the target application. As explained above, in the concept proposed here, the designer manipulates the visual proxies of the framework components. These visual proxies are implemented in Java and have the form of visualizable JavaBeans components. For this reason the obvious choice for the composition environment is to select a standard bean builder tool. Several bean builder tools were evaluated in the project and the one eventually selected was Sun's Bean Builder [Sbb03]. It should be stressed that the version of the Bean Builder available at present is a beta version and the Bean Builder is not yet very stable. However it is expected that upgrades will be released soon and that they would be usable with no or very little changes in the OBS Instantiation Environment.

The spirit of this project is to use as far as possible mainstream technology. Hence, it would be desirable to avoid making the OBS Instantiation Environment dependent on a particular bean builder tool. This objective could only be partially achieved. The OBS Instantiation Environment relies, among other things, on the ability of the component composition tool to generate an XML-based encoding of the target configuration using the long-term persistence mechanism. For this and for other reasons, it is therefore not compatible with current commercial bean builder tools which came on the market before long-term persistence was added to the Java platform. However, Sun's Bean Builder can be reasonably expected to act, as their earlier BDK did, as a kind of blueprint for future commercial products. Hence, it seems likely that, in the future, the OBS Instantiation Environment might be easily upgraded to use other types of bean builder tools.

3.5 Application Configuration and Code Generation

The operations performed by the designer in the component composition environment result in the definition of an *application configuration*. The application configuration defines which framework components are to be included in the target application and how they are to be configured. The application configuration is defined in a stepwise fashion by the designer in the composition environment but this definition must be formalized if it is to serve as an input to the code generation process. In the OBS Instantiation Environment, the configuration of the target application is formalized using the long-term persistence mechanism [Ltp03] provided by the Java 1.4 platform as implemented in as specially customized version of the default XML encoder. The output of the persistence process is an XML document that describes the configuration actions performed by the designer in the composition environment and that consequently describes the configuration of the target application. This XML document is called the *Application Configuration File*. This file is finally processed by an XSLT program that generates from it the instantiation code for the target application. The code generation process is shown at conceptual level in figure 4.

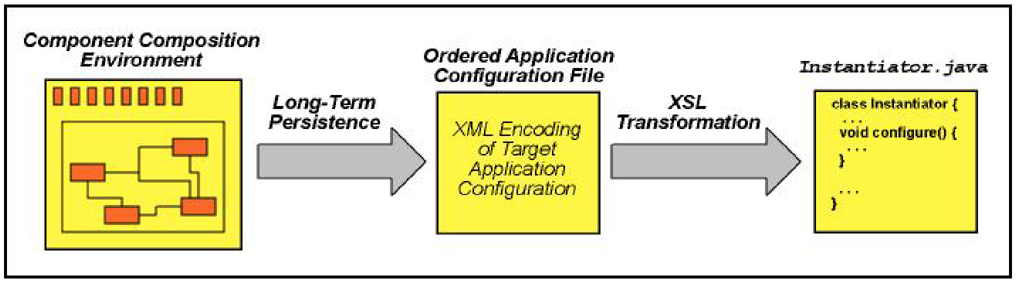
Component Composition Environment
Long-Term Persistence
Ordered Application Configuration File
XML Encoding of Target Application Configuration
XSL Transformation
Instantiator.java
class Instantiator {
...
void configure() {
...
}
...
}

Figure 4: The Code Generation

3.6 Application Simulation

The model for the OBS Instantiation Environment is provided by tools like Matlab that let non-expert users build a control system by configuring pre-defined blocks and offer facilities to automatically generate the code that implements the design defined by the user. One of the strengths of Matlab-like environments is that they let the designers test the configuration they are building by simulating it. Simulation and design are interleaved and designers can, at every step in the design process, pause and perform simulations to test selected aspects of the applications they are building.

For the above reasons, our project investigated the addition of simulation facilities to the OBS Instantiation Environment. The term "simulation" is understood as the selective execution of operations on some of the components within the instantiation environment and the monitoring of the resulting change in their observable state. Simulation is seen as a debugging tool to help designers verify whether their configuration actions satisfy their requirements. An important consequence is that it should be possible to simulate an incompletely configured application because debugging is especially valuable during the configuration process.

In keeping with the minimalist spirit of the approach adopted in this project, simulation facilities are built upon existing tools and technologies. In particular, they exploit the capability of JavaBeans-based bean builder to operate in two modes: *design mode* and *run mode*. In design mode, the components are configured. In run mode, they are executed. The bean builder used in the OBS Instantiation Environment has a built-in run-mode but this, by itself, is useless because the components it manipulates are the visual proxy components and these have no run-time behaviour associated to them: they just exist to be configured and therefore performing a simulation upon them has no effect. Visual proxy components, however, are in a one-to-one correspondence to framework components and each configuration action performed upon them has a counterpart on the framework components.

The OBS Instantiation Environment uses a customized long-term persistence encoder to generate an XML description of the application configuration defined by the user in the composition environment (the *Application Configuration File*). This description can be used to generate the instantiation code but it can also be used to dynamically construct and configure the components of the target application. This is exploited as shown in the figure 5. When a transition into run-mode is detected, an encoding-decoding process is triggered resulting in the dynamical creation and configuration of the framework components (orange boxes in the figure) that are made to shadow their associated visual proxy components (green boxes with dashed edges). The designer can then perform a simulation by asking for a certain operation to be executed upon the application components. The request is intercepted and re-routed to the underlying framework components. The state of the framework components can then be inspected to ascertain the effect of the simulation action.

The resources available to this project were insufficient to fully implement simulation facilities upon the OBS Instantiation Environment. The project instead defined a simulation model and demonstrated its effectiveness with a partial application to the OBS Instantiation Environment. This consisted in endowing the OBS Bean Builder with the ability to perform a *configuration check*. All framework components expose an operation through which they can be asked to check their own internal configuration and to report whether or not they are configured. Typical implementations of this operation verify that all the required plug-ins have been loaded, that settable parameter have legal values, that these values are consistent with each other, etc. A configuration check is useful at the end of the instantiation process (to verify that all components are ready to enter normal operational mode) but it is not itself a configuration operation and it is therefore not modelled by the visual proxy components. Thus, performing a simulation check is a simple form of simulation because it involves executing an operation upon the configured application components and observing the result. Other more complex forms of simulations could be built in a similar fashion. consisting of a *configuration check*. In the configuration check, components

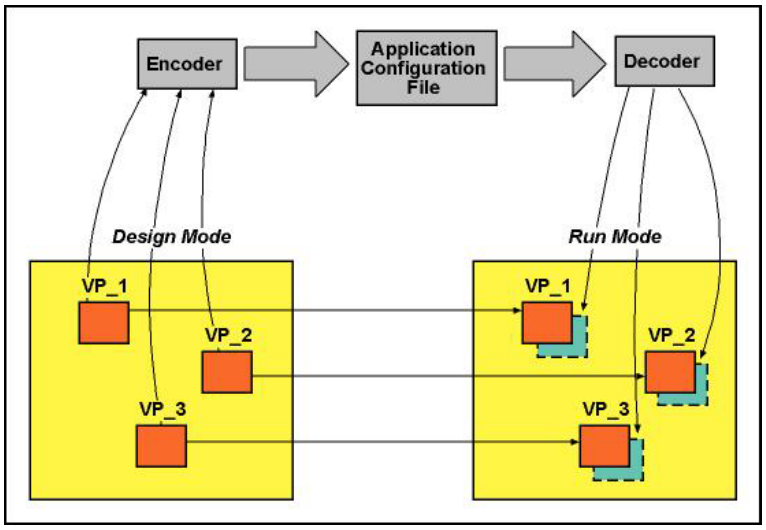

Figure 5: The Application Simulation Concept

3.7 The Case Study

A case study was performed as a way of demonstrating and testing the effectiveness of the framework instantiation concept proposed in this project and embodied in the OBS Instantiation Environment. The OBS Instantiation Environment is built upon the AOCS Framework. The case study accordingly consists in the identification of a target application within the domain of this framework and in the use of the OBS Instantiation Environment to instantiate it. The target application selected for the case study is the same target application selected in the Real Time Java Project [Cec02]. This consisted in the "SMM Application", namely a control software for a so-called Swing Mass Model or SMM. The instantiation approach originally adopted for the SMM Application was based on the development of a set of abstract factories that constructed and partially configured the components required for the application. In this project instead the instantiation was entirely performed in the OBS Instantiation Environment. The application components were configured using the configuration wizards and the property sheets provided by the environment. The total number of components used for the application was 75. The time required to fully construct the application was in the order of a few hours. The work was done in several stages. The instantiation process could be thus broken up in stages because partial configurations could be saved and later restored. The simulation facilities (the configuration check, see section 3.6) played a very important role in speeding up the instantiation process. The designer could check the completeness of the configuration process by performing periodic configuration checks that gave valuable indications as to which components still required attention. The time required for a complete configuration check is in the order of tens of seconds which allowed this facility to be used frequently.

3.8 Component Customization vs Component Configuration

A comprehensive programming environment for applications instantiations from frameworks should cover the *customization* of the components as well as their *configuration*. A component is configured by acting upon it through the operations it declares in its external interface. For instance, a component might expose an operation to set the value of one of its parameters. Such an operation might be called during the configuration phase if it is desired to change the default value of the parameter. A component is customized if its internal implementation is modified. Consider for instance the case of a component that is first developed for operation in an environment where there is no pre-emption. Its use in a multi-tasking environment where pre-emption may occur will normally require that the component be customized with the addition of synchronization code.

The OBS Instantiation Environment is aimed at automating the process of component configuration. It assumes that the components provided by the framework are used "as is" without changes to their implementation. The possibility of automatically customizing the components is of course attractive and was investigated in the course of the project. The conclusion was that the most efficient way to do it is to use aspect oriented programming (AOP) techniques [Kic97]. AOP provides a way to inject into an existing piece of code some new properties either by automatically transforming it or by extending the compiler to introduce the desired properties into the object code. In most aspect-oriented languages, a particular type of customization is encapsulated in an aspect that can in turn be seen as a meta-component. The problem of *customizing* a set of framework components can then be tranformed into the problem of *configuring* a set of meta-components. In the context of this project, this is an important consideration because the OBS Instantiation Environment is geared towards facilitating the configuration of a set of components. The transformation of the customization problem into a configuration problem therefore opens the way to extending the use of the OBS Instantiation Environment to dealing with customization issues. The practical implementation of this approach will be explored in a future project.

4 Conclusions

In recent years, there has been a tendency to emphasize the role of commercial autocoding tools in the development of software for on-board applications. Space applications however are too complex to be covered by any single such tool. As an example, consider the AOCS. Typical AOCS applications implement: control algorithms, control logic, interfacing to the hardware, data handling. For each of these functionalities, one can find good commercial modelling tools with autocoding facilities but there is no tool that is capable of covering *all* AOCS functionalities. The situation is similar for other elements of a space system. Space applications tend to be *multi-domain* and therefore it is normally not possible to find commercial modelling/autocoding tools that cover them in their entirety.

The approach explored in this project represents one way to handle such a situation. The solution it proposes is sketched in figure 6. Several commercial tools are used to model different parts of the same application. The models in these tools are translated into code by autocoding facilities that are attached to the tools. The code coming out of the tools must be embedded within a *software framework* that provides the architectural skeleton for the application. The framework provides a set of *wrappers* that allow the code generated by the commercial modelling tools to be encapsulated in components suitable to be integrated with each other. In addition to wrapping, the autocoded components will normally have to undergo a *customization* process in order to achieve full compatibility with the other parts of the application and to implement non-functional requirements (e.g. synchronization, compliance with error handling strategy, mapping of internal variables to database variables, etc). An instantiation environment like the OBS Instantiation Environment can cover the tasks shown in yellow in the figure. It be used to automate the component configuration and customization process and to generate the final application code. The experience from the Automated Framework Instantiation project is that generative environments can be built on top of a software frameworks with a minimum of effort using mainstream technology. In future work, we hope to further validate the practicality of the vision behind figure 6.

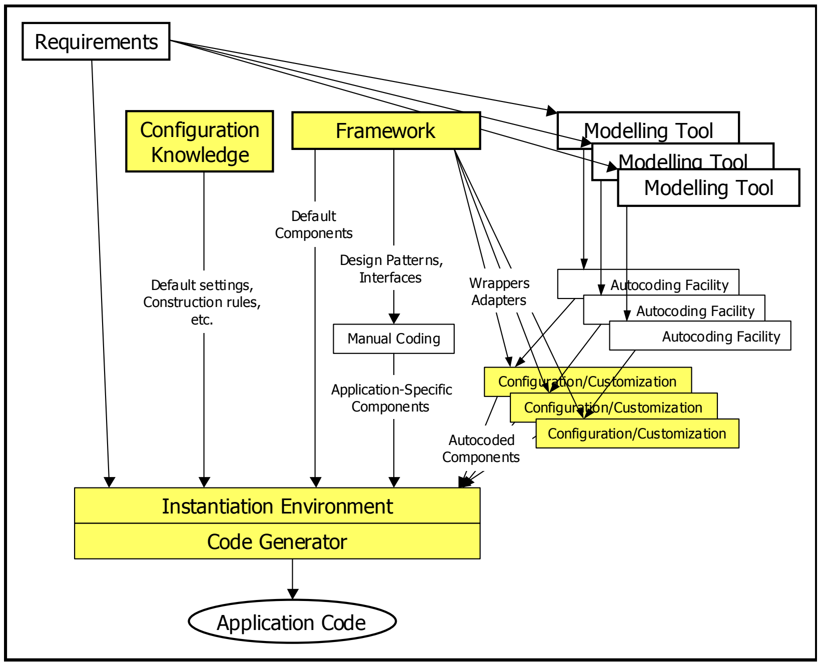

Figure 6: Vision for a fully automated Application Generation Process

5 References

[Afp02] Automated Framework Instantiation, Project Home Page, control.ee.ethz.ch/~pasetti/AutomatedFrameworkInstantiation
[Cec02] V. Cechticky, A. Pasetti*, Real-Time Java for On-Board Systems*, Proceedings of the Data Systems In Aerospace Conference (DASIA), Dublin, May 2002
[Cza00] K. Czarnecki, U. Eisenecker, *Generative Programming,* Addison-Wesley, 2000
[Fay99] M. Fayad, D. Schmidt, R. Johnson (eds) *Building Application Frameworks – Object Oriented Foundations of Framework Design*. Wiley Computer Publishing, 1995
[Gam95] E. Gamma et. al., *Design Patterns*, Addison Wesley, 1995
[Jbp02] JavaBeans Framework Project, control.ee.ethz.ch/~pasetti/JavaBeansFramework

[Kic97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J Loingtier, J. Irwin, *Aspect-Oriented Programming,* European Conference on Object-Oriented Programming (ECOOP `97, Springer-Verlag), Finland, June 1997

[Ltp03] The XML Encoder: http://java.sun.com/products/jfc/tsc/articles/persistence4/index.html

[Mel02] K. Mellach, F. Teston, S. Ekholm, D. Bernaert, *Proba: Avionics Architecture Development and Validation*, Proceedings of Data Systems In Aerospace Conference (DASIA), Dublin, May 2001

[Pas01] A. Pasetti, *et al, An Object-Oriented Component-Based Framework for On-Board Software*, Proceedings of the Data Systems In Aerospace Conference (DASIA), Nice, May 2001

[Pas02] A. Pasetti, *Software Frameworks and Embedded Control Systems*, Springer-Verlag, 2002

[Sbb03] Bean Builder Home Page: java.sun.com/products/javabeans/beanbuilder/index.html