A Generative Approach to Framework Instantiation

Vaclav Cechticky[1], Philippe Chevalley[2],
Alessandro Pasetti[3], and Walter Schaufelberger[1]

[1] Institut für Automatik, ETH-Zürich, Physikstr. 3, CH-8092, Zürich, Switzerland
{cechti,ws}@control.ee.ethz.ch
[2] European Space Agency, ESTEC, PO Box 299, 2200 AG Noordwijk, The Netherlands
philippe.chevalley@esa.int
[3] P&P Software, Peter-Thumb Str. 46, D-78464, Germany
pasetti@pnp-software.com

Abstract. This paper describes the *OBS Instantiation Environment,* which demonstrates a generative approach to automating the instantiation process of a component-based framework. The process is automated in the sense that designers configure and assemble the framework components using intuitive visual operations in a GUI-based environment. Their configuration actions are then used to automatically generate the framework instantiation code. Generative techniques for framework instantiation are not new but tend to rely on domain-specific languages or on bespoke specification encoding and compilation techniques. Though effective and powerful, they are comparatively complex and present a high barrier to entry for general users. The distinctive feature of the approach proposed here is instead its simplicity and its reliance on mainstream technology and tools.

1 Introduction

This paper describes the *On Board Software (OBS) Instantiation Environment,* which demonstrates a generative approach to automating the instantiation process of a component-based framework. A software framework is the heart of a product family. It offers the assets from which the applications in the product family are built. In earlier work [1], we conceptualised a software framework as an artefact consisting of three types of constructs: a set of *domain-specific design patterns*, a set of *abstract interfaces* and a set of *concrete components*. The design patterns define the architectural solutions to the design problems arising in the framework domain. The abstract interfaces define the adaptation points where the generic framework architecture is adapted to match the requirements of specific applications. The concrete components support the instantiation of the design patterns and provide default implementations for the framework abstract interfaces.

The process whereby an application is created by specializing a framework is called *framework instantiation.* It takes place in two 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

F. Pfenning and Y. Smaragdakis (Eds.): GPCE 2003, LNCS 2830, pp. 267–286, 2003.

application. The instantiation approach proposed in this paper only covers the second step. The paper therefore assumes that the framework provides 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 their components are configured differently.

The problem of framework instantiation has been the object of research for several years. Older solutions [16,17] relied on putting together a body of rules (also known as "recipes") to aid the developer in using the framework. More recent versions of this approach use agents to assist the framework instantiation process [7] but most current work looks at generative techniques [3] as a means to automate the framework instantiation process [11,12,14,20]. Such techniques rely on domain-specific languages (DSL) to specify the target application. Although effective and powerful, these techniques tend to be comparatively complex and to present a high barrier to entry for general users. The distinctive feature of the approach proposed here is instead its simplicity and its reliance on mainstream technology and tools that have the potential of bringing it within the reach of non-specialist users.

The downside of our approach is a certain lack of generality: the OBS Instantiation Environment described in this paper is targeted at one particular framework. The paper however identifies one design pattern and several guidelines that would facilitate its porting to other frameworks. The justification for this way of proceeding is a belief that, given the wide variety of frameworks and the lack of standardization at framework level, there is more practical value in providing a blueprint for the development of a simple, though framework-specific, instantiation environment than there is in constructing a general-purpose, but complex, instantiation environment for a generic framework.

The paper is organized as follows. The next section describes the motivation behind our work. Sections 3 to 6 discuss various aspects of our approach. Section 7 describes its use on a concrete case study. Section 8 addresses the issue of the generalization to other frameworks and section 9 concludes the paper.

The work described here was funded by the European Space Agency under research contract 15753/02/NL/LvH. All its results (including source code) are publicly available through a project web site[1].

2 Background and Motivation

We recently developed the *Attitude and Orbit Control System (AOCS) Framework* [1,2] as a prototype object-oriented software framework for satellite and other embedded control systems. The AOCS Framework exists in three versions: two research prototype versions[2] in C++ and Java and one industrial quality version in C++ commercialized by P&P Software GmbH. The work described here refers to the Java version. Porting to the industrial-quality version may follow in the near future.

[1] Currently located at: http://control.ee.ethz.ch/~pasetti/AutomatedFrameworkInstantiation/

[2] Freely available from: http://control.ee.ethz.ch/~pasetti/RealTimeJavaFramework/

The instantiation process for the AOCS Framework consists of a long sequence of instructions that configure the framework components and compose them together. Coding and testing this sequence is a conceptually simple but tedious and error-prone task. We have used abstract factories [5] to simplify the instantiation task but found the simplification thus achieved rather modest. A desire to automate this process was the first motivation for the development of the OBS Instantiation Environment.

The second motivation arises from the target domain of the AOCS Framework, namely embedded control systems. Control engineers have become accustomed to designing their systems in environments like Matlab® that provide easy-to-use GUI-based tools to define the control algorithms and to model and simulate their behaviour together with the dynamics of the system within which they are embedded. The Matlab suite includes facilities to automatically generate code implementing the algorithms defined by the designer. The Matlab approach to control system has gained immense popularity in the control community not least because it holds the promise to allow the software to be directly generated from a model of the control system.

In reality, this promise can only be partially kept. Matlab-like tools excel at modelling control algorithms but the software of a modern control system (see figure 1) is dominated by heterogeneous functions like unit management, command processing, housekeeping data generation, failure detection, failure recovery, and other functions for which Matlab provides no specific abstractions and which it is consequently unable to model effectively. More generally, no single commercial tool offers sufficient abstractions to model all aspects of a complex control system.

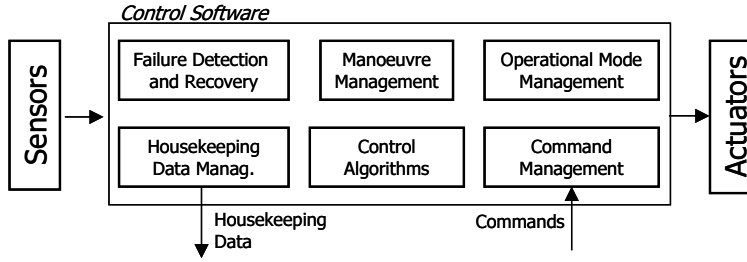

Fig. 1. Structure of typical control software

In our view, a software framework is ideally suited to define the overall architecture for a control application and to provide an umbrella under which components coming from commercial autocoding tools (suitably wrapped) can be combined with each other and with components coming from other sources to build the final application. We therefore see a software framework as complementary to a Matlab-based approach. However, we appreciate that the appeal of the latter largely lies in its GUI-oriented user interface and in the tacit premise that this will allow the control software to be developed directly by the control engineer with only minimal assistance from a software engineer. We believe that a framework-based approach will only be accepted in this community if it can be packaged in a similar way. This is precisely what we are trying to offer with the OBS Instantiation Environment.

Figure 2 shows how the OBS Instantiation Environment fits within the software development process we envisage for a control application. The final application is built by configuring and assembling components. A framework defines the architec-

ture within which they are embedded and assembled and provides a set of default components. Other components are manually coded on ad hoc basis while still others come from wrapping code automatically generated by tools like Matlab. The OBS Instantiation Environment provides the facilities for configuring and linking together these components and for generating the corresponding instantiation code.

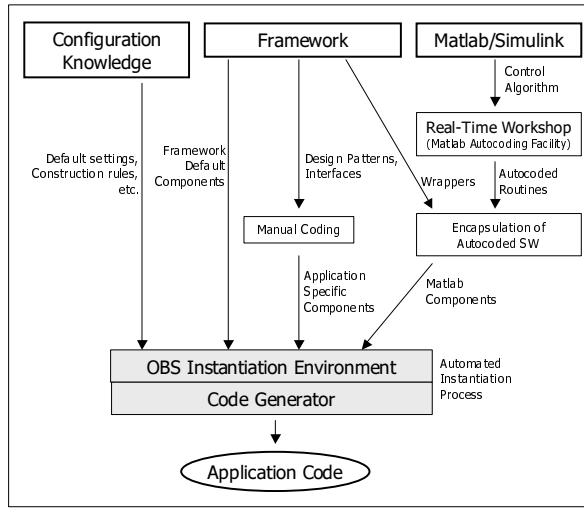

Fig. 2. Development process for embedded control software

Although our work concentrates on control systems, the situation in other domains is similar. Organizations increasingly build their applications by configuring and composing pre-defined blocks [3, 12]. Since these blocks are likely to come from different sources, a framework is required to provide the architectural skeleton within which they can be embedded. The problem then inevitably arises of how the framework is to be instantiated. In many embedded domains, this problem is exacerbated by the fact that the framework is used to build one-of-a-kind applications, which are specified by application engineers who are not software specialists and who therefore are not able or willing to take responsibility for a code-based instantiation process. Since the applications are unique, delegating responsibility for the instantiation process to a group of software specialists has a significant impact on total development costs. There is therefore a need to create an instantiation environment that is sufficiently user-friendly to allow the application specialists to take charge of the development of their software. The OBS Instantiation Environment provides a blueprint of how this objective can be achieved using simple and inexpensive technology.

3 Proposed Instantiation Approach

Three requirements can be inferred from the discussion above for a framework instantiation environment:

1. The environment should be based on mainstream technology. This is the only way to keep its cost low which is in turn essential to its practical adoption.
2. The environment should be easy to use. More specifically, whereas the job of instantiating a framework is traditionally left to a software specialist – and often to a specialist of the framework to be instantiated – the environment should make it possible for this task to be done by end-users. Hence, the environment should be seen as an enabling tool that empowers non-specialist end-users to take direct control of the development of their software.
3. The environment should allow the behaviour of the application under construction to be simulated. The success of environments like Matlab also depends on the fact that they let designers test their design by executing a (possibly incomplete) implementation in order to check its behaviour. An instantiation environment for a framework should offer similar simulation facilities.

The approach taken in the OBS Instantiation Environment to satisfying the above requirements is based on generative programming techniques [3]. Generally speaking, five steps are necessary to develop a generative programming environment for applications within a certain domain:

1. Definition of a formalism for specifying applications in the domain,
2. Definition of a common architecture for applications in the domain,
3. Development of configurable and customizable components to support implementation of a domain architecture,
4. Definition of a formalism to describe the configuration and customization of the components,
5. Development of a code generator to automatically transform an application specification into a domain configuration and to generate a concrete application from the domain configuration.

Step 2 is essentially equivalent to the development of a software framework for the target domain. Some of the components mentioned in step 3 may also be provided by the framework as encapsulation of recurring functionalities in the domain. In other cases, the framework will simply provide wrappers for code coming from other sources (legacy code, automatically generated code, etc). For the sake of simplicity, in this paper we will refer to all these components as *framework components*. This paper concentrates on steps 4 and 5, which are those covered by the OBS Instantiation Environment. Step 1 is discussed in section 6. The provision of simulation facilities is discussed in section 5.

The straightforward way to implement the above five steps is to define a domain-specific language through which the application specification can be expressed, and then to build a compiler that allows these specifications to be translated into source code. This solution, by itself, does not satisfy any of our requirements. The approach we chose is instead based on developing an *environment* where users can express their requirements in an informal manner through graphical means with the aid of context-specific information 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. The derivation of the application is still done in two steps - formal specification of the target application and its compilation - but

these steps are now hidden from the user who only interacts with a user-friendly environment.

Figure 3 shows how our solution is implemented. The figure is annotated with the key technologies behind our implementations. These are: XML for encoding information, XSLT programs for code generation, and Java bean builders for the component composition environment. All three technologies are widely known and well supported thus satisfying our first requirement.

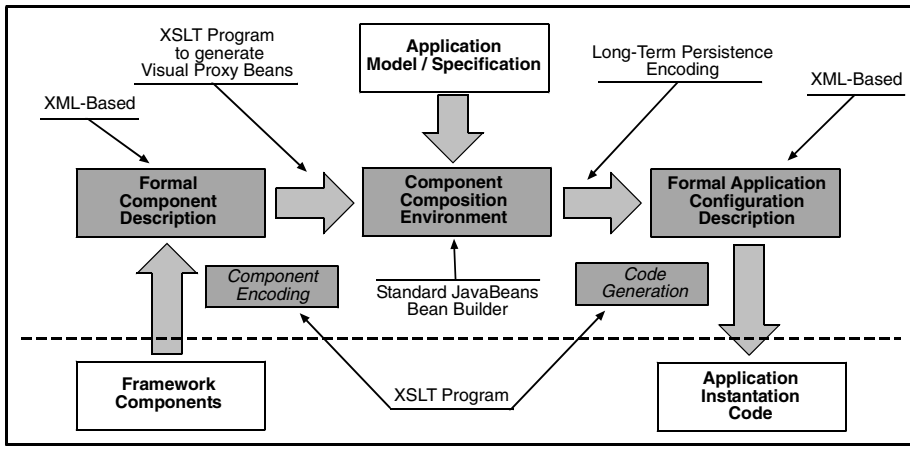

Fig. 3. Generative Approach to Framework instantiation

We use XML to decouple the way the framework components are implemented from the way they are configured (dashed line in the figure). XML grammars are defined to describe the framework components (or, more precisely, those of their characteristics that are relevant to the instantiation process) and to describe the application configuration defined by the user in the component composition environment.

Having selected XML as the encoding standard, we found it natural to use XSLT [4] to perform the code generation process. XSLT is a functional programming language that was developed to transform XML documents into other XML documents. It can more generally be used to manipulate XML-encoded data and to generate from them other textual documents. As already noted by other authors [21] XSLT programs can also act as simple, powerful, and easy-to-use code generators.

We use XSLT programs for two purposes. The designer defines the application configuration in the *component composition environment*. At the end of the configuration process, the selected application configuration is encoded in an XML document. An XSLT program is used to process it and to generate from it the application instantiation code (*code generation process*). This code is ready to be linked with the framework components to form the final application executable.

The second use of XSLT is less obvious. It stems from our choice of component composition technology. The component composition environment is the part of the framework instantiation environment where users configure and assemble the framework components. It represents the interface between the users and the instantiation environment. In order to satisfy our second requirement, it should be GUI-based. Its development is potentially one of the most complex and most expensive parts of a framework instantiation environment.

In order to avoid incurring such costs, we use as component composition environment a standard *bean builder* tool. Bean builders are commercial tools that offer sophisticated graphical environments where JavaBeans components can be manipulated [6]. They cannot be directly used for our purpose for several reasons: the target components may not be visualizable, they may be written in languages other than Java, their instantiation operations may not fit the JavaBeans model, etc.

Since the framework components cannot be imported in a bean builder, we construct *visual proxy components* that model the part of the behaviour of the framework components that is relevant to the instantiation process and that are additionally implemented as JavaBeans. The visual proxy components are then imported in a bean builder tool and designers perform the application configuration upon them. Their equivalence to the framework components (at least as far as the application instantiation process is concerned) means that designers can be given the illusion of manipulating the framework components when in fact they are operating upon their proxies. The second usage of XSLT envisaged in our approach is the automatic generation of the visual proxy components and other support components that support their configuration.

This section has presented the approach we propose from a general standpoint. The next section describes how we applied it to construct the OBS Instantiation Environment as an instantiation environment for the AOCS Framework.

4 The OBS Instantiation Environment

The first step in the construction of a framework instantiation environment must be a precise definition of what is meant by "framework instantiation". 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*:

1. Instantiation of a framework component,
2. Setting the value of a component property,
3. Setting the value of a static property,
4. Setting the value of an indexed property,
5. Linking an event-firing component to an event-listening component,
6. Adding a component to an *object list* (this is a kind of container component that can hold other components).

Note that the component properties can be either of primitive type or of class type. Thus, the second, third and fourth operations also cover the case of object composition. Note also that, 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* can thus be defined as the problem of translating a particular application specification into an ordered sequence of instantiation operations which, when executed, will result in the instantiation of an application that implements the initial specifications. The OBS Instantiation Environment solves the instantiation problem for the AOCS Framework.

The primary inputs to the instantiation process are the framework components. When suitably configured, they become the building blocks for the target application. The OBS Instantiation Environment consequently needs to manipulate them and needs to have access to information about them. 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 (including static and indexed properties), the events they fire and listen to, the object lists they maintain. This information is encoded using an XML grammar. For each framework component, an XML document describing its instantiation-relevant characteristics is automatically generated by a parser-like facility. Such documents are called *Visual Proxy Descriptor Files*.

The operations exposed by the components of the AOCS Framework adhere to certain naming conventions (roughly similar to those defined by the JavaBeans standard) that, to some extent, allow the semantics of an operation to be inferred from its name. These conventions in particular allow the instantiation operations to be recognized and identified. Hence, the OBS Instantiation Environment can automatically construct the visual proxy descriptor files by parsing the public API of the framework components.

As explained in the previous section, the OBS Instantiation Environment associates to each framework component a visual proxy component. Visual proxies must have two characteristics. Firstly, they must exhibit the same behaviour as their associated framework component during the application instantiation phase. Given the instantiation model adopted here for the target framework, this means that a visual proxy must:

1. Expose the same properties as its associated framework component,
2. Fire and listen to the same events as its associated framework component,
3. Expose the same object lists as its associated framework 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 and, during the instantiation phase, it is essentially equivalent to it. Secondly, visual proxies must be well suited to manipulation in a bean builder. In practice, this means that they must be implemented as visualizable JavaBeans. The OBS Instantiation Environment additionally complements them with beaninfo components, bean editor components and bean customizers. The beaninfos provide meta-information about the components that defines the way they are to be manipulated in the composition environment. The property editors define the way individual attributes of the visual proxy components are to be defined (for instance, they enforce constraints on their values). The bean customizers provide wizards that help the user configure components. Taken together, the visual proxies and their support components implicitly define a model of how the AOCS framework can be instantiated and of the constraints that the instantiation process must satisfy.

The visual proxy components, together with their support components (beaninfos and property editors) are automatically generated by XSLT programs that process the visual proxy descriptor files. Thus, the transition from the framework components to

their visual proxies is entirely automatic. Note also that whereas the visual proxies must be implemented in Java, no such restriction applies to the framework components. As already noted, the presence of an intermediate XML encoding separates the framework components from their visual proxies (but see the remark at the end of section 5). The process of construction of the visual proxies is sketched in figure 4 for a sample framework component.

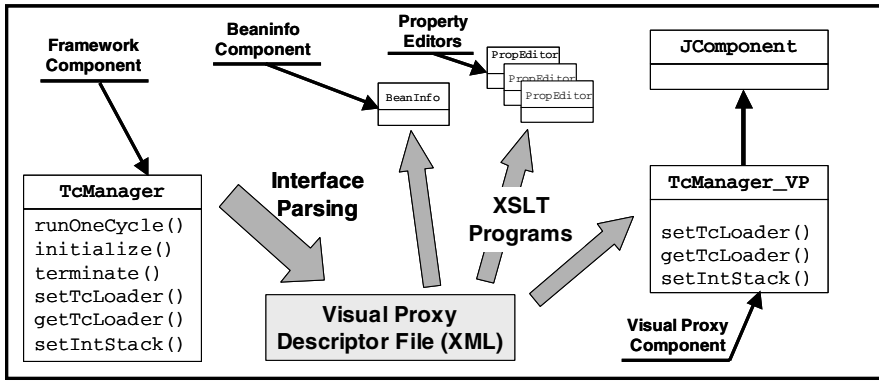

Fig. 4. Visual proxy generation process

The visual proxies are intended to be imported in the *component composition environment* (see figure 3). This is the part of the OBS Instantiation Environment where the designer configures and assembles the target application. In our minimalist approach, this environment is implemented by customizing a JavaBeans bean builder. For the OBS Instantiation Environment, we have used Sun's Bean Builder[3]. Although this bean builder is at present only available as a beta version, it was selected because it implements the long-term persistence mechanism and because it is expected to act, as its predecessor BDK did, as a kind of blueprint for future commercial bean builder products.

Figure 5 shows a screenshot of our composition environment. The top window offers palettes with the framework components. The bottom right window is the composition form where components are displayed and visually manipulated. The connection lines represent composition relationships between components. The component configuration is done using the property sheet in the bottom left window. Wizards (not shown in the figure) are provided to handle non-standard configuration actions (e.g. the additions of items into object lists). The wizards and the property editors are also responsible for enforcing the constraints on the instantiation process (e.g. constraints on the range of values of certain variables).

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 code generation problem is the problem of transforming the application configuration into source code. In the OBS Instantiation Environment, this is done in two steps.

[3] Available for free from: http://java.sun.com/products/javabeans/beanbuilder/index.html

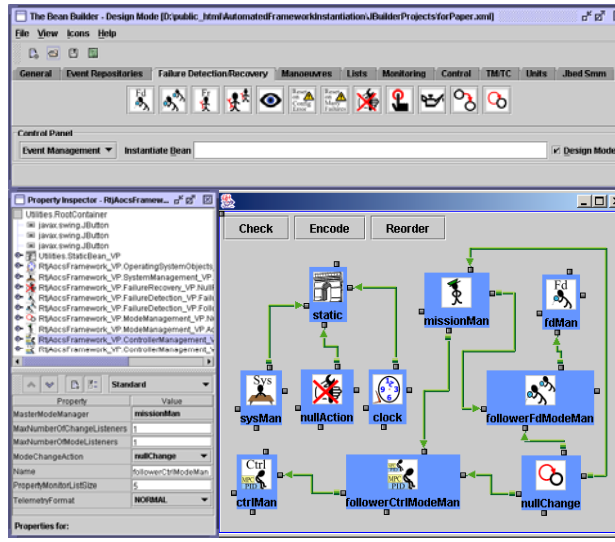

Fig. 5. Screenshot of composition environment of OBS Instantiation Environment

First, the application configuration that is defined through graphical means in the component composition environment must be encoded using some suitable formalism. Then, the encoded configuration description must be processed to generate the source code instantiating the application.

The first step is done using the *long-term persistence mechanism* [18]. This is a new feature of the Java 1.4 platform that allows the state of a set of components to be saved by recording the sequence of instructions that were executed to configure them. This is done by an *encoder* which examines the state of the target components and uses reflection techniques to work out which instructions were performed upon them to bring them to their current state. The relevance of such an encoding mechanism to a generative environment is obvious.

The Java 1.4 platform offers a default implementation of the encoder (the `XMLEncoder` class). The OBS Environment had to use a specially customized version for two reasons. First, the application configuration is defined by the designer in the component composition environment in terms of the visual proxy components whereas the application configuration must be expressed in terms of the framework components. Hence, the default encoder was extended to perform the translation from the visual proxies back to the framework components. Secondly, the order in which the configuration operations are performed upon the framework components must satisfy certain *ordering constraints.* The persisted image of an application configuration consists of an encoded list of instantiation statements. Enforcement of the ordering constraints is done by sorting this list. The criteria with respect to which the sorting is performed are domain-specific (they depend on the internal structure of the framework components) and can therefore be embedded within the environment. Enforcement of the ordering constraints during the persisting process means that designers are free to specify the instantiation operations in any order in the composition environment. This relieves them of the burden of complying with the constraints and

makes it easier for them to move back and forth in the instantiation process by doing and undoing configuration actions.

The codification and enforcing of the ordering constraints was one of the most complex problems we had to solve. At present we use a set of domain heuristics but the problem is conceptually similar to that found in graphical simulation systems where the simulation blocks instantiated by the user must be executed in some predefined order [19]. We plan to extend our environment to implement similar techniques for enforcing the ordering constraints on the instantiation sequence.

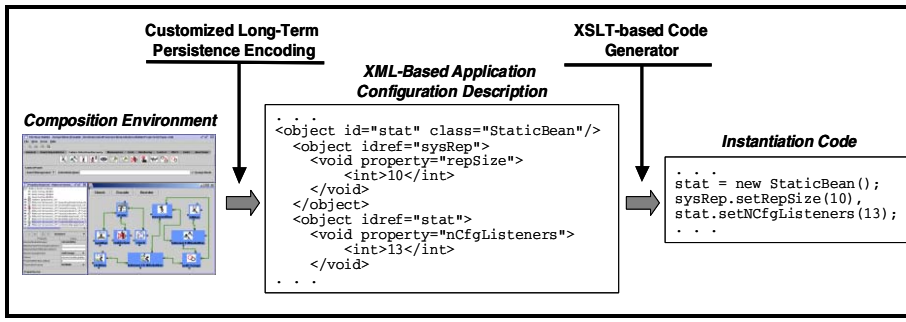

Fig. 6. Instantiation code generation process

The final generation of the source code is straightforward. The long-term persistence mechanism encodes the application configuration using an XML grammar where each element represents an instantiation statement. The XML document is processed by an XSLT program that translates the instantiation statements in Java source code. Translation into another object-oriented language would be equally straightforward. Figure 6 shows the code generation process in schematic form.

5 Simulation

The third requirement of section 3 calls for the instantiation environment to allow for the simulation of the application that is being configured. Simulation is understood here as the selective execution of operations on some of the components within the 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.

It is noteworthy that current work on framework instantiation does not seem to address the simulation problem. Given the crucial role that simulation plays in commercial tools like Matlab, this is a serious shortcoming. In our view, one of the benefits of a generative approach is that it makes it easy to switch from a model to its implementation and should therefore be easy to extend to cover simulation.

In keeping with the minimalist spirit of our approach, we have built simulation facilities upon existing tools and technologies. We have in particular exploited the ca-

pability 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 executing them has no effect.

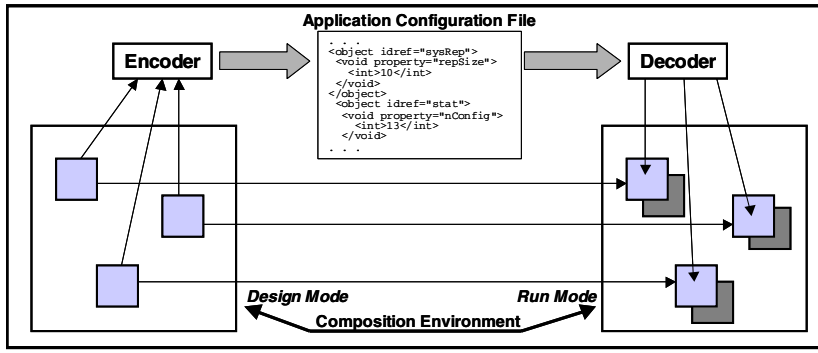

Fig. 7. Simulation concept

The previous section discussed how a customized long-term persistence encoder is used to generate an XML description of the application configuration defined by the user in the composition environment. This description can be used to generate the instantiation code (see figure 6) but it can also be used to dynamically construct and configure the components of the target application. This is exploited as shown in figure 7. When a transition into run-mode is detected, an encoding-decoding process is triggered resulting in the dynamic instantiation and configuration of the framework components (darkly shaded boxes in the figure) that are made to shadow their associated visual proxy components (lightly shaded boxes). 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 fundamental organizing principle of the OBS Instantiation Environment is the preservation of the illusion for the designers that they are manipulating the “real” framework components when in fact they are manipulating their much simpler visual proxies. Our simulation concept complies with this principle in that the simulation actions are ostensibly performed upon the visual proxies but are in fact internally re-routed to the dynamically created and configured framework components.

This simulation concept was demonstrated by endowing the OBS Bean Builder with the ability to perform a so-called “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 configuration 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.

Two remarks are in order. First, the use of XML to encode the framework component properties and the application configuration decouples the implementation of the framework components from that of the visual proxy components (dashed line in figure 3). This decoupling is weakened in the case of our simulation concept because, during a simulation, both the framework components and the visual proxy components exist side-by-side. Second, our simulation concept executes the application in a desktop environment. This introduces inevitable differences (e.g. with respect to timing) with respect to execution in the final embedded environment.

6 Software Development Process

As already noted, current work on generative approaches for framework instantiation tends to put the emphasis on the definition of a DSL. A DSL is useful because it, unlike the implementation language of the framework, "knows" about the framework characteristics (the abstractions it implements, the variation points it offers, the constraints it dictates, etc). For this reason, use of a DSL is probably inescapable in any automated framework instantiation environment. However, DSLs by their very nature tend to be idiosyncratic and thus put off non-specialist users.

Our solution to this dilemma is twofold. First, like many other authors [9, 11, 14, 20], we use XML to encode the framework characteristics and, like some authors [14, 20], we use XSLT to process this information. Both are mainstream technologies and their use helps in fostering acceptance of a generative approach. Secondly, and perhaps more importantly, we try to "hide" the DSL from the user. Conceptually, the composition environment of our OBS Instantiation Environment can be seen as a DSL for its target framework. The components it offers to the user are implementations of the framework abstractions and the environment constrains the operations that the user can perform upon these abstractions. Operations that are not allowed are simply not seen by the user. This is the advantage of using the visual proxies that only model the subset of the operations exposed by the framework components that can legally be used during the instantiation process. Similarly, there are constraints, like for instance the ordering of the instantiation operations, that the user ignores but that are automatically enforced by the environment. Thus, the graphical configuration operations performed by the user in the composition environment are effectively equivalent to writing a specification of the target application in a DSL but the user needs not be aware of this and the formal specification of the target application is automatically generated by the environment itself (it is contained in the application configuration file generated by long-term persisting the configured components in the composition environment).

We have also taken one further step in order to ease the transition of traditional users from manual assembly of the framework components to assembly in a composition environment and automatic generation of the instantiation code. Software development in our domain of interest (embedded control systems for satellite applications)

is articulated over three stages. In the first stage, the end-user, an *application engineer*, expresses the requirements to be satisfied by the application using informal English. In the second stage, the *software engineer* translates these requirements into a formal specification. Finally, in the third stage an architecture and an implementation are derived from the specifications.

This development process is somewhat at odds with the development model implied by the approach we propose. One motivation for our approach (see section 2) is to empower the application engineers to directly develop their own software. Given their intimate understanding of their requirements and of their domain, their natural way to operate is to proceed directly to the configuration of the application in the composition environment without first passing through a formal specification phase. This is, for instance, what experienced control engineers do when they use tools like Matlab. The simulation facilities of the environment are used to verify the correctness of the application as it is being constructed.

However, in order to preserve compatibility with the traditional approach, we have endowed the OBS Instantiation Environment with the capability to automatically generate the software specifications from the application configuration. This is done by an XSLT program that processes the XML-based *application configuration file*. The resulting specifications read like structured English text of the kind used in software specification documents in our field of interest. Essentially, the XSLT program associates pre-defined sentence templates to each configuration operation. The templates are user-defined but are intended to be invariant within the framework domain. The values of the template parameters are derived from the XML image of the application configuration. The information in the XML image is obviously formulated in terms of implementation-level concepts (e.g. the names of the interfaces implemented by a certain component). In order to be understandable for a human reader, the specifications must instead be formulated in terms of framework-level concepts (e.g. the name of the abstraction encapsulated by the abstract interfaces). The translation from implementation-level to framework-level concepts is contained in a *domain dictionary* [17] that is associated to the framework and whose purpose is precisely to provide a vocabulary to formally describe applications within the framework domain.

The application configuration file, the software specifications derived from it, and the visually configured components in the composition environment can thus be seen as three views on the same abstract specification of the target application and as three different specifications written in three different but equivalent DSLs.

7 Case Study

The OBS Instantiation Environment is a fully functioning instantiation environment for the AOCS Framework. In order to demonstrate its effectiveness, we used it to develop the control software for a “Swing Mass Model” (see figure 8). This is a laboratory equipment consisting of two rotating disks connected to each other and to a load with a torsional spring. A DC motor drives one of the disks. The goal of the controller is to control the speed and position of the other disk. The instantiated application is representative of a full control system application as conceptualised in figure 1 as it includes the following functionalities:

- Two operational modes,
- Processing of sensor data and stimulation of actuators
- Implementation of control algorithms,
- Failure detection checks on the main system variables,
- Failure recovery actions autonomously executed upon detection of failures,
- Autonomous provision of housekeeping data,
- Processing and execution of operator commands,
- Capability to execute speed profiles.

The instantiation required the configuration and composition of 75 components. The application construction was performed in several stages with intermediate configuration states being saved and then restored. This gave the designer the option to try alternative configuration approaches. The experience of using the environment was a positive one. The environment relieves the designer of much tedious and low-level work and its graphical interface makes its use easy and intuitive. Turn-around time from a configuration to the code implementing it is very short (less than a minute) which facilitates experimenting and prototyping.

The simulation capabilities built into the instantiation environment (the "configuration check", see section 5) proved particularly valuable as they allowed the designer to rapidly identify components whose configuration was still incomplete and which therefore required attention. The instantiation code generated by the environment was highly readable and could easily have been modified by hand if required. A formal description of the application was automatically generated using the facility described at the end of section 6.

Fig. 8. The Swing Mass Model

8 Generalization

The discussion in this paper – and much of the work behind it – is specific to one particular framework, the AOCS Framework, and to one particular generative environment, the OBS Instantiation Environment. However, there are three aspects of our

experience that are relevant to the problem of building other generative environments for other frameworks. They relate to the *technologies* we used, to a *design pattern* we applied, and to a number of *framework guidelines*.

Generative techniques have so far failed to find favour with industry. One reason for this failure is their tendency to rely on complex and unusual technology. Automatic configuration of pre-existing components is a natural field of application for generative techniques but industrial applications will come only if ways are found to achieve this aim using standard technologies. In our project, we have found two such technologies that have the potential to make the generative approach attractive to the ordinary software developer.

The first one is the *combination of XML and XSLT*. The suitability of XML as a way to encode the raw information that is to drive the generative process has already been widely exploited [9, 11, 14, 20] and the corresponding power of XSLT to build customizable code generators is attracting more and more attention [14]. Our experience is that once the choice is made to use XML to encode configuration information, XSLT should be the first option for implementing the code generator. It may not always be adequate because it is was devised for other purposes but, when found to be adequate, it will provide an excellent basis upon which to build a code generator quickly and at minimal cost.

The second technology is the *combination of bean builder tools and long-term persistence* as implemented in the Java 1.4 platform. Any non-trivial generative environment for frameworks will need a graphical interface where the user can manipulate the framework components. Bean builders can be useful for this purpose. They provide sophisticated, low-cost, customizable environments for component configuration and manipulation. The facilities offered by older versions of these tools for encoding configuration information were based on Java-style serialization or on rather primitive and poorly-tuneable code generators. The newer versions (of which Sun's Bean Builder, which we used in our project, is a prototype) can be expected to implement the long-term persistence mechanism. This generates an XML-based description of the operations that were executed to configure a set of components. This description is an excellent basis for a simple (and XSLT-based) code generator.

One problem with using standard bean builders as component composition environments is that, in most cases, the framework components cannot be directly imported into a bean builder tool because of language or other incompatibilities. In our project we have used the visual proxy mechanism to overcome this problem. We believe that this mechanism represents a design pattern – we call it the *visual proxy design pattern* – in the sense that the abstract ideas behind it could be beneficial in other contexts.

In general, the visual proxy design pattern is useful when there is a need to manipulate components in a visual environment for the purpose of generating configuration code for them. The components themselves may be awkward to manipulate directly for several reasons: (1) they may be too complex; (2) they may be written in a language that is incompatible with the chosen composition environment; (3) they may not have an "appearance" and may therefore be unsuitable for graphical manipulation. The design pattern calls for the construction of components – the visual proxy components – that only model the part of the behaviour of the original components that is relevant to the instantiation process. These components can be kept simple, can be made visualizable, and can be written in the desired language. Users can then be given the illusion of manipulating their own components when in fact they are operating upon their visual proxies. The illusion can be sustained

ing upon their visual proxies. The illusion can be sustained because the visual proxies and the base components implement the same configuration operations with the same semantics.

The visual proxy design pattern is a second result of our project with general applicability. The third and last one concerns a number of guidelines on how frameworks should be designed in order to facilitate their integration in generative environments for automating their instantiation. In principle, it is always possible to apply generative techniques to the instantiation process of almost any reasonably designed framework. However, one lesson of the work described here is that the cost of doing so can vary a great deal depending on how the framework is designed. More specifically, we have identified four guidelines that should be followed to ensure that a framework is "generative-friendly".

- *Component-Based Instantiation*. The framework should be designed so that the instantiation process can be expressed entirely in terms of configuration operations performed upon the components offered by the framework. This guideline is very natural in the case of a component-based framework. It allows the instantiation process to be expressed only in terms of the public API's of the framework components. This limits the amount of information that must be processed by the automated instantiation environment. Description of the external interfaces of the components is much simpler than description of their implementation and if only the former is needed, there is a clear gain in simplicity.

- *Instantiation Demarcation.* The framework should be designed so that the instantiation operations are clearly separate from the operations that are executed during other phases of the application operation. The point of using the visual proxy components is to replace potentially complex components (the framework components) with other components that, though equivalent to them from the point of view of the instantiation process, are functionally simpler and hence easier to manipulate in an autocoding environment. This approach is feasible only if the behaviour of the framework components can be neatly split between an instantiation-relevant sub-behaviour and another sub-behaviour that is not relevant to the instantiation process. The simplest way to ensure that this split is possible is to design the framework components so that the operations they offer can be separated between operations that are used only during the instantiation process and operations that are only used during the operational phases of the application.

- *Adherence to Naming Conventions.* The framework should be designed so that the names of the instantiation operations conform to some pre-defined naming patterns that allow the semantics of an operation to be inferred from its name. Manipulation of the framework components within an automated instantiation environment requires the environment to find out information about the components. Since the components are manipulated only through the operations they expose, the information the environment needs only concerns these operations. Use of naming patterns for the operations is arguably the simplest way through which this information can be encoded.

- *Instantiation Operation Independence.* The framework should be designed so that the instantiation operations are as far as possible independent of each other. In general, the instantiation sequence will have to satisfy some ordering constraints: some

operations can only be performed after some other operations have been performed; or the way some components are configured may be dependent on the outcome of previous configuration actions. This guideline recommends that this type of dependencies be as far as possible minimized. The instantiation sequence is defined by the user in a component composition environment. It is desirable to allow users to specify the instantiation operations individually since this simplifies their task. This means that the instantiation constraints must be imposed by the environment itself. The experience from this project is that imposing these constraints can give rise to significant levels of complexity. This justifies the introduction of this guideline.

It may be noted that most of these guidelines make sense even in the normal case of a framework that must be instantiated manually since they mostly tend to simplify the instantiation process and to clarify the boundaries between operations that must be performed as part of the instantiation process and operations that must be performed as part of the normal operation of an application. Since a framework is often used (i.e. instantiated) by people other than its designer, this type of demarcation can be useful independently of whether a generative approach to its instantiation is foreseen or not.

9 Conclusions and Future Work

In this paper, we propose a generative approach to framework instantiation and we demonstrate it by applying it to a particular framework. We claim that the distinctive feature of our approach is its simplicity and reliance on mainstream technologies. These features are important because they promise to bring a generative approach within the reach of most framework designers and users. In order to substantiate our claim, it is necessary to provide an estimate of how much effort would be required to port our approach to another framework.

The development of the OBS Instantiation Environment – from the start of the project to the execution of the case study described in section 7 – took place over a period of 8 months. The work was done by a senior engineer and a junior engineer allocated to the project at, respectively, 50% and 75% of their time. The total effort was therefore 10 man-months. Since this is a new concept, much of this effort went into false starts and trying out new ideas. Our estimate is that the re-implementation of the concept for a new framework which complies with the guidelines laid down in section 8 would require at most 4 man-months. This is regarded as a rather modest investment but it must be stressed that the estimate is heavily dependent on the way the framework is designed. If a generative approach to instantiation is envisaged, it is essential that this be kept in mind already during the framework development phase.

Our future work will follow two broad directions. On the one hand, we intend to further verify the validity of our approach by applying it to industrial test cases. Our experience to date is restricted to laboratory experiments (see section 7). The results we have obtained encourage us to try a more ambitious experiment where we port the environment to a new framework and apply it to the generation of operational software in an industrial context.

On the other hand, and in a more research–oriented line of work, we are looking at the possibility of extending the capability of the instantiation environment to perform

component *customisation* as well as component *configuration*. A component is configured by acting upon it through the operations it declares in its external interface. A component is customized if its internal implementation is modified. The OBS Instantiation Environment is, at present, only concerned with component configuration. The generative effort concentrates on the generation of the component configuration code alone.

We expect that the generation of customisation code will require the use of aspect oriented programming techniques. In particular, an idea we would like to explore is whether we can transform the problem of customizing a set of components with respect to a certain feature into an equivalent problem of configuring a meta-component that describes that feature. This would allow us to preserve much of the current approach that is geared towards component configuration with the important difference that some of the components that are configured in the environment would be meta-components (or maybe their visual proxies) and that the code that is generated from them is component customization code rather than application implementation code.

References

1. Pasetti, A.: Software Frameworks and Embedded Control Systems. LNCS Vol. 2231, Springer-Verlag, 2002
2. Pasetti A., et al.: An Object-Oriented Component-Based Framework for On-Board Systems, Proceedings of the Twelfth Data System in Aerospace (DASIA) Conference, Nice, France, May 2001
3. Czarnecki K., Eisenecker U.: Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000
4. Kay M.: XSLT – Programmer's Reference, Wrox Books, 2001
5. Gamma E., et al.: Design Patterns – Elements of Reusable Object Oriented Software, Addison-Wesley, Reading, Massachusetts, 1995
6. Englander R.: Developing JavaBeans (Java Series), O'Reilly and Associated, 1997
7. Ortigosa A., Campo M., Moriyon R.: Towards Agent-Oriented Assistance for Framework Instantiation, Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming, Systems, and Languages(OOPSLA 2000), Minneapolis, USA, Oct. 2000
8. Fontoura M., et al.: Using Domain-Specific Languages to Instantiate Object-Oriented Frameworks, IEE Proc.-Soft., Vol. 147, No. 4, August 2000
9. Swe Myat S., Yhang H., Jarzabek S., XVCL: A Tutorial, Proceedings of the Conference on Software Engineering and Knowledge Engineering (SEKE), Ischia, Italy, July 2002
10. Sztipanovits J., Karsai G.: Generative Programming for Embedded Systems, in: Batory D., Consel C., and Taha W. (eds.): Proceedings of the Conference on Generative Programming and Component Engineering (GPCE 2002), LNCS Vol. 2487, Springer-Verlag 2002
11. Czarnecki K., et al.: Generative Programming for Embedded Software: An Industrial Experience Report, in: Batory D., Consel C., and Taha W. (eds.): Proceedings of the 23rd Conference on Generative Programming and Component Engineering (GPCE 2002), LNCS Vol. 2487, Springer-Verlag 2002
12. Czarnecki K., Eisenecker U.: Components and Generative Programming, Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'99), Toulouse, France, 1999
13. Bryant B., et al.: Formal Specifications of Generative Component Assembly Using Two-Level Grammar, Proceedings of the Conference on Software Engineering and Knowledge Engineering (SEKE), Ischia, Italy, July 2002

14. Butler G.: Generative Techniques for Product Lines, Software Engineering Notes, Vol. 26, No. 6, November 2001
15. Anastasopoulos M., Gacek C.: Implementing Product Line Variability, Proceedings of the International Conference on Software Engineering (ICSE), Toronto, May 2001
16. Donohoe P. (ed): Software Product Lines – Experience and Research Directions, Kluwer Academic Publisher, 2000
17. Fayad M., Schmidt D., Johnson R. (eds.): Building Application Frameworks –Foundations of Framework Design, Wiley Computer Publishing, 1999
18. http://java.sun.com/products/jfc/tsc/articles/persistence/index.html
19. Giloi W.: Principles of Continuous System Simulation, B. G. Teubner Stuttgart, 1975
20. Oliveira T., Alencar P., Cowan D.; Towards a Declarative Approach to Framework Instantiation, Proceedings of the Workshop on Declarative Metaprogramming, Automated Software Engineering Conference, Edinburgh, Sept. 2002
21. Craig Cleveland J.; Program Generators with XML and Java, Prentice Hall, 2001