Generative Programming for Embedded Software: An Industrial Experience Report

Krzysztof Czarnecki,[1] Thomas Bednasch,[2] Peter Unger,[3] and Ulrich Eisenecker[2]

[1]DaimlerChrysler AG, Research and Technology, Ulm, Germany
[2]University of Applied Sciences Kaiserslautern, Zweibrücken, Germany
[3]Technical University of Ilmenau, Germany
czarnecki@acm.org, Thomas.Bednasch@epost.de, Punger@gmx.de, Ulrich.Eisenecker@t-online.de

Abstract. Physical products come in many variants, and so does the software embedded in them. The software embedded in a product variant usually has to be optimized to fit its limited memory and computing power. Generative programming is well suited for developing embedded software since it allows us to automatically produce variants of embedded software optimized for specific products. This paper reports on our experience in applying generative programming in the embedded domain. We propose an extended feature modeling notation, discuss tool support for feature modeling, describe a domain-independent system configuration editor, and comment on the applicability of static configuration in the area of embedded systems.

1 Introduction

An embedded system is any computer that is a component in a larger system and that relies on its own microprocessor [Wol02]. The products that contain embedded software usually come in many variants and thus the requirements on the embedded software also vary. Sources of variation include different product features and different supported hardware, e.g., on-board satellite software may support different services and different fuel injection systems impact automotive engine control. At the same time, in order to reduce recurring hardware costs, embedded software usually runs on platforms with reduced amount of memory and processor capability and with minimum support hardware. As a result, embedded software usually needs to be optimized for minimum memory footprint and for maximum speed to perform its required computations and actions within its real-time deadlines.

Generative programming is well suited for developing embedded software since it allows us to automatically produce variants of embedded software optimized for specific products based on a set of reusable components. However, a practical approach has to take into account the special requirements of industrial embedded software development, such as requirements on the programming languages being used or the need to support validation and verification procedures.

This paper reports on our experience in applying generative programming in the embedded domain. It also proposes a technology projection that is geared towards satisfying the current requirements of industrial embedded software development. The case studies used to illustrate our points are described in Section 3. The specific contributions of this paper include (1) extended requirements on feature modeling (Section 4), (2) comparison of different approaches to tool support for feature modeling (Section 5), (3) concept of a domain-independent system configuration

D. Batory, C. Consel, and W. Taha (Eds.): GPCE 2002, LNCS 2487, pp. 156–172, 2002.

editor based on feature modeling (Section 6), (4) combination of the configuration editor with a template-based generation approach (Sections 7-9), and discussion of the applicability of this approach in the industrial context (Sections 10). Section 11 discusses related work. We conclude with ideas for future work in Section 12.

2 What Is Generative Programming?

Generative programming builds on system-family engineering (also referred to as product-line engineering) [CN01, WL99, Par76] and puts its focus on maximizing the automation of application development [Cle01, CE00, BO92, Cle88, Nei80]: given a system specification, a concrete system is generated based on a set of reusable components.

The means of application specification, the generators, and the reusable components are developed in a domain-engineering cycle and then used in application engineering to create concrete applications. The main domain engineering steps are

- domain analysis, which involves domain scoping, finding common and variable features and their dependencies, and modeling the structural and behavioral aspects of the domain concepts
- domain design, which involves the development of a common architecture for the system family and a production plan
- domain implementation, which involves implementing reusable components, domain-specific languages, and configuration generators.

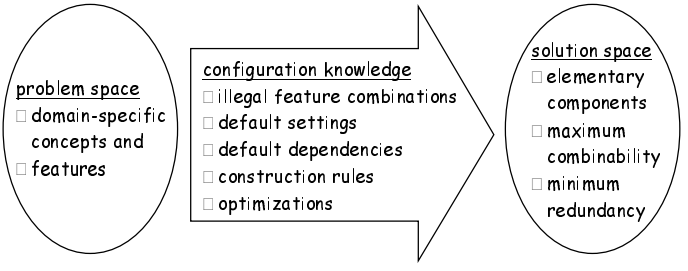

Fig. 1. Elements of a generative domain model [CE00]

The key to automating the assembly of systems is a generative domain model that consists of a problem space, a solution space, and the configuration knowledge mapping between them (see Fig. 1). The solution space comprises the implementation components and the common system family architecture, defining all possible combinations of the implementation components. The implementation components are designed for maximum combinability, minimum redundancy, and maximized reuse. The problem space, on the other hand, consists of the application-oriented concepts and features that application engineers use to express their needs. This space is implemented as a domain-specific language (DSL). The configuration knowledge specifies illegal feature combinations, default settings, default dependencies (some defaults may be computed based on some other features), construction rules (combinations of certain features translate into certain combinations of implementation components), and optimizations. The configuration knowledge is

implemented in the form of generators. The generated products may also contain non-software artifacts, such as test plans, manuals, tutorials, maintenance guidelines, etc.

Each of the elements of a generative domain model can be implemented using different technologies, which gives rise to different *technology projections*:

- Components can be implemented using, e.g., generic components (such as in the STL), component models (e.g., JavaBeans, ActiveX, or CORBA), or aspect-oriented programming approaches.
- Generators can be implemented using preprocessors (e.g., template processors), application generators, built-in metaprogramming capabilities of a language (e.g., template metaprogramming in C++), or extendible programming systems.
- DSLs can be implemented using new textual or graphical languages (or language extensions), programming language-specific features, or interactive wizards.

The choice of a specific technology depends on its technical suitability for a given problem domain, mandated programming languages, existing infrastructure, familiarity of the developers with the technology, political and other considerations. We present one specific technology projection geared towards embedded software.

3 Case Studies

This paper is based on our experience in working with internal DaimlerChrysler business units from the space, aerospace, and automotive domains. The main example for this paper comes from the satellite domain. The European Space Agency has issued the Packet Utilization Services (PUS) standard [PUS], which defines a service-based interface for communicating with a satellite and a number of generic services, such as housekeeping (periodically sending telemetry reports about the status of various satellite functions to the ground) or storage control (storing telemetry data when there is no connection to the ground). The satellite case study described in this paper is based on a project at DaimlerChrysler Research and Technology to develop a prototype of a product-line architecture implementing the PUS standard for Astrium Space. Astrium Space is a leading European satellite manufacturer and was part of the DaimlerChrysler AG until 2001. The purpose of this product-line architecture is to avoid the re-development of the higher-level, application-oriented communication software for each new satellite from scratch.

Another example which is discussed in this paper is the OSEK/VDX standard [OSEK] defining a realtime operating system for embedded automotive applications. OSEK/VDX is defined as a statically configured, application-specific operating system. The number and names of tasks, stack sizes, events, alarms, etc. used by an application are statically defined using the configuration language OIL [OIL]. An OSEK/VDX implementation provides a generator that takes an OIL specification and generates the application-specific tasking and communication support (usually as C code). We have conducted an experiment to express the configuration space defined by OIL using feature models.

4 Extended Feature Modeling

During domain analysis, the scope of a product line is captured as a feature model [KCH+90,CE00]. A feature model describes the common and variable features of the

products, their dependencies, and supplementary information such as feature descriptions, binding times, priorities, etc. Features are organized into feature diagrams, which reveal the kinds of variability contained in the product configuration space. An example of a feature diagram is shown in Fig. 2. It describes a simple model of a car. The root of the diagram represents the concept *car*. The remaining nodes are features:

- *Mandatory features*: Every car has a body, transmission, and engine.
- *Optional feature*: A car may pull a trailer or not.
- *Alternative features*: A car may have either an automatic or a manual transmission.
- *Or-features*: A car may have an electric engine, a gasoline engine, or both.

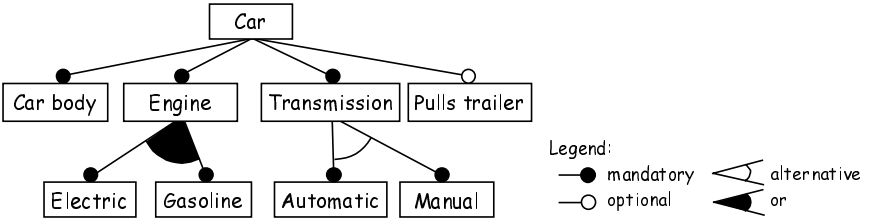

Fig. 2. A sample feature diagram of a car

A different representation of a product configuration space is to use parameter tables (e.g., as in the FAST method [WL99] or [Cle01]). In our experience, feature diagrams allow better modularity in large configuration spaces than flat parameter tables. Thanks to the hierarchical organization of feature diagrams, parameter sets that are pertinent to just one class of variants can be hidden in one branch of a feature diagram and need not be considered when configuring a variant outside this class. In this context, it is important to note that a feature diagram is not just a part-of hierarchy of the software modules, but a hierarchical decomposition of the configuration space. Features in a feature diagram need not correspond to concrete software modules, but may represent abstract properties such as performance requirements and aspectual properties affecting several software modules such as synchronization.

The original feature diagram notation from [KCH+90] is insufficient to effectively model the variabilities in our case studies. The following subsections list the shortcomings and propose extensions to overcome them. The proposed extensions are geared towards representing feature models using a MetaCASE environment as described in Section 5. We are currently working on an alternative solution to the presented one which is based on annotating feature models with partial or full specializations of other feature models at the meta-level and base level.

4.1 Cardinalities

[CE00, p. 117] suggests to model multiplicity using a subfeature (see Fig. 3) rather than extending the feature notation with cardinalities. The reason behind this decision was to keep the notation as simple as possible and to avoid cluttering it with all the mechanisms known from structural modeling such as UML class diagrams.

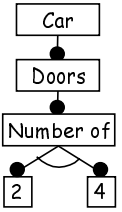

Fig. 3. Representing multiplicity in a feature diagram

Unfortunately, this approach does not necessarily work when the feature that needs some multiplicity is the root of a subtree containing variable features. In that case, our intention could be not just to specify the number of doors as in Fig. 3, but to express that a specific configuration may include several different variants of the same feature. This is demonstrated in Fig. 4, which shows a fragment of the feature diagram from our satellite software case study. According to the PUS standard, the interface to a satellite consists of a number of applications, which implement services consisting of subservices. As the diagram shows, every satellite has a packet router application. It may have one storage control application or not. And it may have zero or more user defined applications, which is expressed by the cardinality "*" next to the feature `UserDefinedApp`. Each of the user defined applications may implement a different selection of PUS-predefined services (the service and subservice numbers are defined in the PUS standard) and user-defined services. Finally, the PUS standard defines some of the subservices as mandatory and some as optional. In the diagram, the filled or empty circles could have been alternatively rendered as the cardinalities 1 or 0..1, respectively.

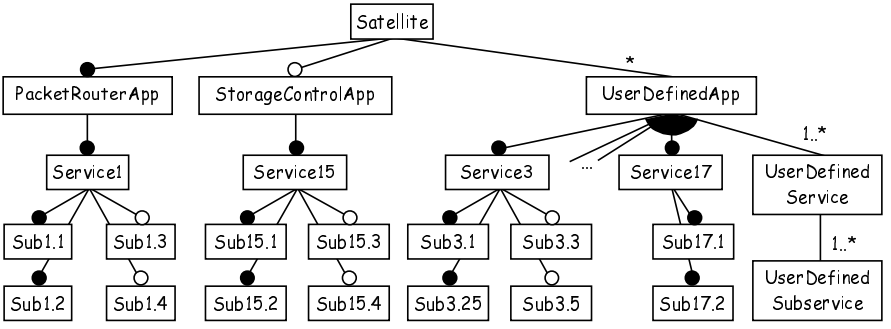

Fig. 4. Fragment of a feature diagram representing the configuration of a satellite interface according to the PUS standard

4.2 Attributes

The next extension was to introduce attributes. A feature may contain a number of attributes, just like a UML classifier. An exploded representation of the feature `StorageControlApp` is shown in Fig. 5.

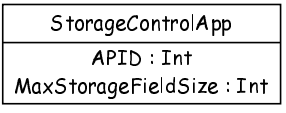

Fig. 5. Exploded representation of a feature showing its attributes

The reason for introducing attributes was to allow for a more concise representation of feature diagrams. Of course, each attribute could be represented as a subfeature, but this quickly leads to very large diagrams. With attributes, the tree structure is primarily used to organize a parameter space in a hierarchical way. If certain parameters apply to all system variants with a given feature, then they can be represented as attributes in that feature. Also, attributes include a type specification. Enumeration attributes allow us to reuse the same enumeration type in different places and this way we do not need to draw the same alternative subfeatures multiple times. Finally,

attributes allow for a concise representation in the configuration tool (see Section 6). Currently, we work on a more uniform feature model representation, which avoids attributes as a separate concept and allows to achieve the goals described in this section using an extended form of features.

4.3 Reference Attributes

In addition to integer, float, boolean, string, enumeration and list attributes, we also need reference attributes. Cardinalities allow us to include several different configurations of feature subtrees in a system, and reference attributes allow us to model graph-like connections between the different configurations of feature subtrees. As an example, consider the partial feature diagram of the configurable real-time operating system OSEK/VDX in Fig. 6. An OSEK/VDX configuration for a particular application consists of a number of tasks, resources, events, alarms, and counters. The feature `Task` contains the reference attribute `Resource`, which specifies (using the attribute specification notation of UML) that a given task in a OSEK/VDX configuration can have zero or more specific resources. Despite these graph-like relationships, the feature-subfeature decomposition remains hierarchical, which is important for aiding the configuration process described in Section 6.

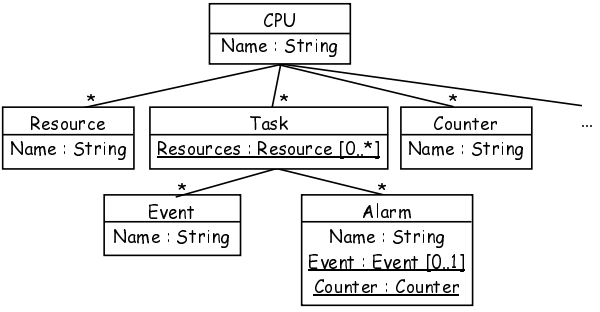

Fig. 6. Fragment of a feature diagram representing the configuration space of the OSEK/VDX real-time operating system (reference attributes are underlined)

4.4 Metamodeling for Supplementary Information

A feature model has a feature diagram and usually contains additional information such as feature descriptions, rationale for each feature, stakeholders and client programs interested in each feature, examples of systems with a given feature, configuration constraints between features, default dependency rules, binding times and modes, open/closed attributes, and feature priorities (see [CE00]). From our experience, not all of these are needed in every project, and different projects often have their own additional kinds of supplementary information. For example, a project may have its own binding site model with product-specific binding times and sites. This problem can be solved by allowing developers to edit the metamodel of the feature modeling notation in order to tailor it to the project-specific needs. This capability needs to be supported by the feature modeling tool.

5 Feature Modeling Tool Support

Feature modeling can be performed using UML tools, MetaCASE tools, and dedicated feature modeling tools. We discuss each of these choices as next.

UML tools. A feature model notation can be created using the standard UML extension mechanisms, e.g., using a stereotyped class to represent features and the containment relationship to represent the feature-subfeature relationship (see e.g., [GFA98, Cla01a, Cla01b]). This approach, although practicable, has the drawback that current major UML modeling tools do not sufficiently support profiles. In particular, it is not possible to add constraints to a profile and have them checked automatically by the tool while creating a model. As a result, such tools cannot check whether the feature model being created is actually a valid feature model or not.

MetaCASE tools. MetaCASE tools genuinely support metamodel editing and allow us to create new notations. Examples of MetaCASE tools are MetaEdit+ form MetaCase Consulting [MC] and the Generic Modeling Environment (GME2000) from the Vanderbilt University [LMB+01]. We have used GME2000 to define a feature modeling notation and to perform the feature modeling for the satellite case study. (GME2000 is freely available at http://www.isis.vanderbilt.edu).

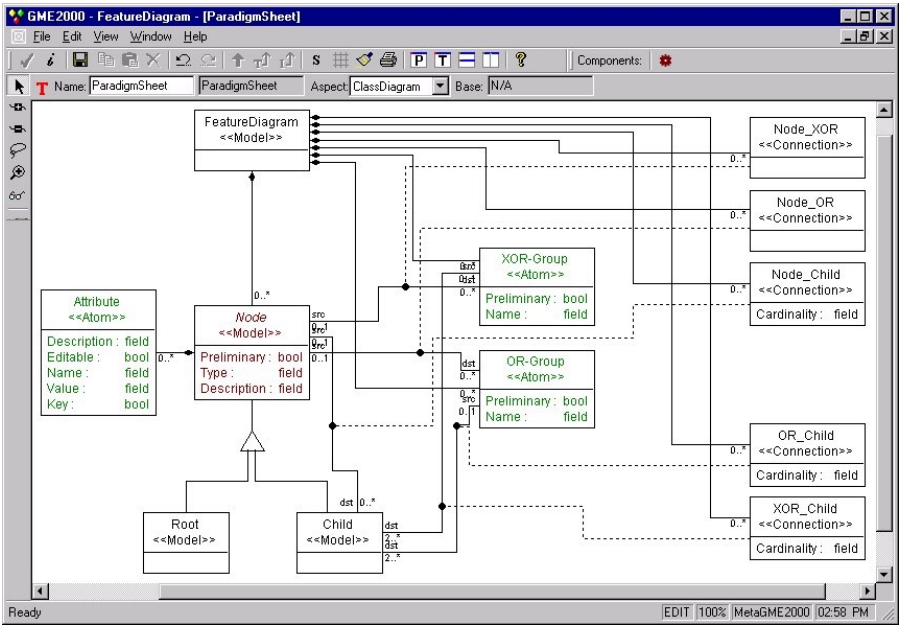

Fig. 7. UML metamodel of a feature modeling notation in GME2000

Fig. 7 shows the UML metamodel of our feature modeling notation in GME. (Some aspects of the notations, such as visualization, constraints, and attributes are not shown in this view.) According to our metamodel, a `FeatureDiagram` contains a number of `Nodes` (with the specializations `Root` and `Child`). A `Node` may have a number of `Attributes`, and `Nodes` and `Attributes` have the field `Description`. Furthermore, we have `XOR-Group` and `OR-Group` to represent

groups of alternative and or-features, respectively. Finally, connections with `Children` at the one end (i.e., `Node_Child`, `OR_Child`, and `XOR_Child`) have the field `Cardinality` with the default value 1.

The metamodel also contains a number of constraints in the Object Constraint Language (OCL) [UML, chapter 6], such as that a feature diagram may only contain one root, a child or a group has to have one incoming connection, a group has to have at least two children, and no cycles are allowed. These constraints are enforced automatically by GME when creating a concrete feature diagram.

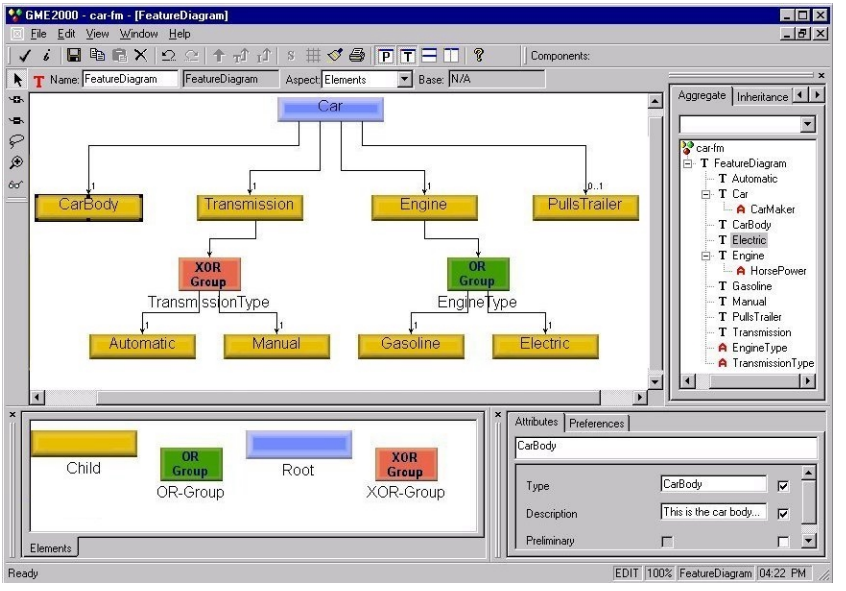

Fig. 8. Example of a feature model in the MetaCASE tool GME2000

Fig. 8 shows the sample feature diagram of a car from Fig. 2 using the notation defined in Fig. 7. The *Aggregate* tab on the right shows that `Car` has the attribute `CarMaker`, and `Engine` has the attribute `HorsePower` (there is a separate view in which attributes can be added to the features[1]). Please note that based on the currently loaded metamodel, the GME environment offers only the desired modeling elements. The validity of the feature diagram can be checked by activating the OCL constraint checker through the *File* menu.

We found using a MetaCASE tool such as GME2000 particularly useful for prototyping new notations (such as experimenting with the extensions described in Section 4) because it allows us to quickly evolve a notation without reprogramming a dedicated tool.

---

[1] The *Attribute* tab in Fig. 8 shows the meta-attributes of the currently selected feature-model element. They are defined as fields of the metamodel elements, e.g., `Description` in Fig. 7, and are used to model the additional information described in Section 4.4. They should not be confused with the feature attributes described in Section 4.2, such as `CarMaker` or `HorsePower`. The latter attributes are instances of the `Attribute` atom in Fig. 7.

Dedicated feature modeling tools. A dedicated feature modeling tool is designed to provide optimal support for feature modeling. Compared to general-purpose modeling tools, it supports drawing and layouting feature trees in the concise notation from Fig. 2. An example of such a tool is Ami Eddi ([Sel00, Bli01, ESBC01]), which is freely available at www.generative-programming.org (see Fig. 9).

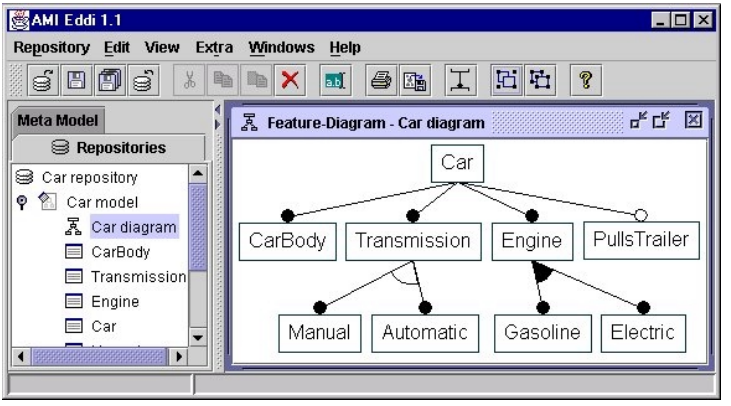

Fig. 9. Dedicated feature modeling tool Ami Eddi

An important feature of Ami Eddi is its metamodeling capability, which allows the user to extend the modeling notation with a user-defined list of additional information to be recorded with each feature (see Section 4.4). In GME2000, this capability corresponds to adding new fields to the metamodel elements (such as `Description` in Fig. 7). Future work on Ami Eddi includes the ability to annotate feature models with partial or full specializations of feature models at the metalevel and base level. This extension will provide the capability to appropriately model cardinalities and attributes without extending the notation with these concepts.

6 Configuration Editor

A feature model defines the configuration space of a product. We have developed ConfigEditor, a graphical configuration editor that helps an application engineer to create a specific product configuration based on a feature model.

In ConfigEditor, a configuration is constructed top-down by successively selecting the features of the currently loaded feature model according to its variabilities. The configuration process starts at the root, which is automatically inserted into a configuration. Whenever a new feature is added to a configuration, its mandatory subfeatures are added automatically. This is demonstrated in Fig. 10, which shows the initial configuration of a satellite (according to the feature model in Fig. 4). The feature tree being constructed is displayed in the left part of the ConfigEditor window. The tabbed panes on the right are related to the currently selected feature in the tree under construction. The *Attribute*-tab pane allows us to edit the attributes of the selected feature. A separate window shows the description of the currently selected feature (it displays the contents of the description field of the selected feature as provided in the feature model).

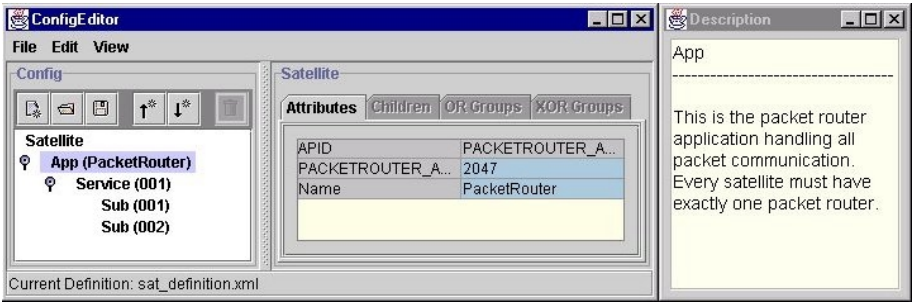

Fig. 10. The minimal satellite configuration in ConfigEditor. The *Attribute*-tab pane shows the attributes of the packet router application

The *Children*-tab pane allows us to add subfeatures not belonging to any group and having a cardinality other than 1. Fig. 11 shows this pane for our satellite example. According to Fig. 4, satellite may have an optional storage control application and zero or more user defined applications. These choices appear in the list on the *Children*-tab pane and the corresponding cardinalities are shown in brackets. The feature tree on the left shows the result of adding one storage control application and two user-defined applications. Please note that both newly added user-defined applications are marked with the red symbol "fill". This marking gives the application engineer a visual clue that both features require further editing. In our example, the reason is that according to Fig. 4, a user defined application has a group of or-features and at least one of them needs to be added to the current configuration.

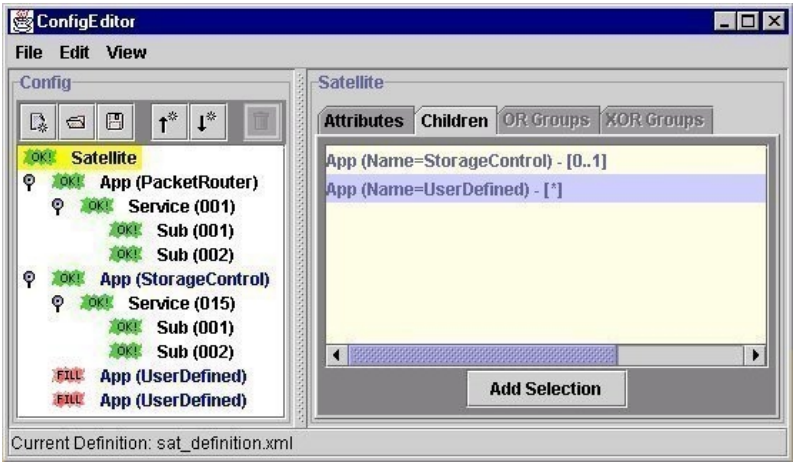

Fig. 11. Satellite configuration after adding the storage control and two user applications

Features belonging to a group of or-features or alternative features can be added using the *OR Group*- or the *XOR Group*-tab pane. Fig. 12 shows the *OR Group*-tab pane for our example. Because a feature may have more than one group of or-subfeatures, the pane provides a drop-down menu to switch between them. The list below the drop-down menu shows the list of or-features in the currently selected group. One or more different or-features from the list can be added to the current

configuration. The same or-feature can be added as many times as specified by its cardinality. The XOR Group-tab pane works similarly, with the difference that only one alternative feature can be selected from the list.

ConfigEditor accepts a feature model definition in a simple exchange format in XML. The idea is to provide a set of import filters that can transform XML exports of feature models from different modeling tools to our XML format. The current implementation of ConfigEditor includes an import filter for the XML export of a feature model from GME2000. A concrete configuration created using ConfigEditor is also saved in an XML format.

A configuration editor should also support checking validity of a configuration using additional constraints and automatically completing it by computing defaults. In our satellite example, the inclusion of some optional subservices (e.g., the telecommand subservice 15.5) requires the inclusion of other subservices (e.g., the corresponding telemetry subservice 15.6). Such dependencies are not represented in the feature diagram and need to be expressed as additional constraints. A constraint checker for ConfigEditor is currently being implemented.

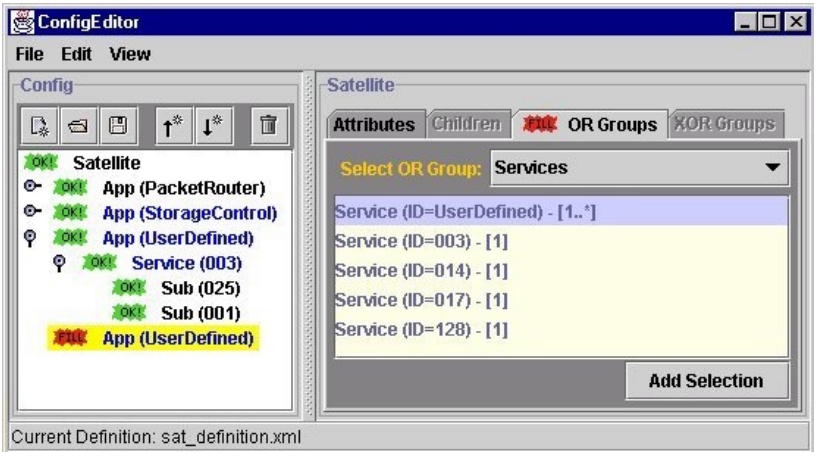

Fig. 12. Satellite configuration after adding service 3 to the first user-defined application and before adding services to the second user-defined application

7 Product-Line Architectures and Template-Based Generators

A product-line architecture (PLA) [Bos00] defines the components needed to build each of the products in the product line. In our satellite example, the architecture consists of a number of Ada83 packages representing domain abstractions such as up/down communication link, on-board storage manager, packet router, service decoder, components providing the different services, etc.

A PLA for embedded software is usually required to include only those components in a given product configuration that are actually used. This helps to minimize resource consumption such as memory and computing power and to

minimize recurring hardware costs. Additionally, this strategy reduces amount of code that needs to go through quality assurance procedures.[2]

In our satellite case study, the configuration of applications and services determines which service-providing components need to be included in a specific system. Other components requiring configuration are the service decoder and the storage control manager. For example, the service decoder needs to be able to decode communication packages only for the services included in the configuration.

We found template-based code generation [Cle01] particularly suited for the static configuration of embedded PLAs. In this approach, an arbitrary text file such as a source program file in any programming language or a documentation file are instrumented with code selection and iterative code expansion constructs. Such an instrumented file called is a *template*. A *template processor* takes a template plus a set of configuration parameters and generates a concrete instance of the template.

Compared to a programming-language-based generation approach, a simple text-based template approach has a number of properties that significantly ease its adoption on industrial project in the embedded software area. Most importantly, the approach is programming-language-independent. From our experience, organizations developing safety-critical embedded software are very reluctant to introduce major changes to their existing development procedures. In particular, they are significantly slower in adopting new programming languages than it is the case in other domains. Two major programming languages currently used in the embedded domain are C and Ada. For our satellite case study, we were mandated to use Ada 83 rather than Ada 95 in order to stay compatible with some radiation-tolerant processors for which no Ada 95 compiler exists. Another advantage of the template approach in the industrial context is the possibility to easily templatize existing source code. Finally, the template approach encourages generating readable source code. On our satellite project, this property, which is often considered to be counterproductive in the context of code generation, turned out to be actually critical for the adoption of the generation approach. By generating well-formatted, readable source code, the existing validation and verification procedures of a development organization can be applied to the generated product, just as to a manually created one. In our view, the high cost of validation and verification of the generator itself may be economical for general-purpose generators such as programming-language compilers, but it appears too high for in-house, product-line-specific generators.

8 Code Generation Using TL

For our satellite case study, we have used Template Language (TL) [Cle01] and the freely available TL processor (http://craigc.com/). TL provides code selection and iterative code expansion constructs and it can also be extended with user-defined functions. The configuration parameters for a TL template are stored in an XML file and the TL constructs can conveniently access them using XPath [XP] expressions. For example, the following code snippet uses the TL if-construct to decide whether to

[2] In fact, the quality assurance guidelines of several manufacturers form the space and aerospace domain forbid the inclusion of unused code in the safety-critical software embedded in their products. This is usually not the case in the automotive domain.

include the definition of the constant MAX_STORAGE_OBJECTS in the generated code or not:

```
#if "/Satellite/App/Service/Attribute[@Name='ID' and @Value='015']"#
  MAX_STORAGE_OBJECTS: constant integer
    := #"//Attribute[@Name='MAX_STORAGE_OBJECTS']/attribute::Value"# ;
#fi#
```

The TL constructs are enclosed in #...# and shown in bold. The if-condition is an XPath expression which checks in the XML configuration whether any application of the satellite includes the service with the ID 015. The value of the constant is also retrieved from the XML configuration (the XPath expression looks for any attribute with the appropriate name). Assuming an XML configuration that includes service 15 and a MAX_STORAGE_OBJECTS attribute with the value 100, the TL processor would generate the following code:

```
MAX_STORAGE_OBJECTS: constant integer
 := 100 ;
```

A larger example of a template from the satellite case study is shown in Table 1. This is an excerpt from the template implementing the service decoder procedure, which calls the appropriate service package depending on the service and subservice number provided as a parameter. The template code is shown on the left and the corresponding generated code is shown on the right. The template uses nested pairs of for-constructs to iterate over the services defined in the XML configuration as subnodes in of /Satellite/PUS/ and over the subservices of each service. One such nested pair is used to generate the list of service packages to be included using the WITH statement. Another pair is used to generate the nested CASE statement that calls the appropriate service procedure. Assuming that the XML configuration includes service 1 with subservices 1 and 2, and service 15 with subservices 1, 2, and 3, we get the code shown in the right column in Table 1 (the code generated by the for-constructs is shown in bold).

9 Putting It Together

The entire generative process supporting an embedded product line is shown in Fig. 13. It is interesting to note that the same configuration that is used to generate the embedded software is used to configure the testing and simulation environment. In the case of our satellite example, the ground station software that needs to communicate with the satellite is configured with the same configuration as the satellite software. This way, the ground station can only send packets that are understood by the satellite. In addition to generating the embedded software, we can use the same XML satellite configuration and the same template technology to generate the documentation for a concrete satellite configuration.

10 Applicability of the Approach

Static configuration is not always the best choice for embedded software product lines. In some cases, dynamic configuration can be more economical. The choice between these two approaches depends on the production volume of the products containing the embedded software, the resource limitations of the target platform, and whether the software embedded in a product needs to be maintained during its

lifetime or not. High production volumes usually imply the opportunity to significantly reduce recurring hardware costs by applying static configuration and cutting down on memory and processor power. In this case, the approach described in this paper may be well suited.

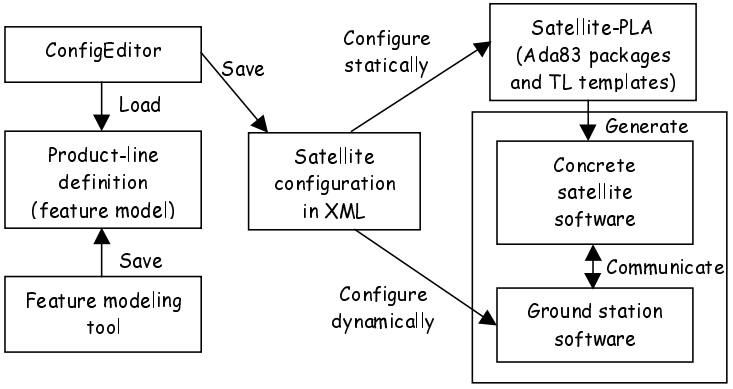

Fig. 13. Overview of the entire modeling and configuration chain

If the individual products need to be maintained over their lifetime, the cost of logistics for keeping track of static configurations may outweigh the savings on hardware. For example, this is the case for control functions in the automotive area, where the usual strategy is to have just one version of a software component for all different car models. Typically, an electronic control unit (ECU) comes with embedded functions (such as ABS, ESP, or engine control) capable of supporting different car models (i.e., the whole product-line code is put on one ECU). The idea is that the repair shops do not have to carry different variants of an ECU for different car models, but only one, which significantly reduces logistic costs.[3] Consequently, automotive functions use dynamic configuration parameters for both tuning the software to the mechanics of a car (so called *end-of-line configuration*) and activating the necessary program code for a given car model. The car model identification is broadcast on the network in the car on start-up and the ECUs configure themselves automatically.

The situation is different for basic components that need to be installed on several ECUs in different configurations. This mandates the use of static configuration. An example of such a basic component is the operating system. Consequently, the automotive real-time operating system OSEK/VDX is set up as a generator emitting application-specific tasking and communication support for each ECU as a set of C functions. Thus, our approach is applicable to basic components that need to be included in different configurations on several different ECUs.

Finally, even in the case of relatively low production volumes, there still might be memory and computing power constraints mandating the use of static configuration. This is the case in the satellite domain, where the production volume is usually less

[3] Currently, most repair shops do not have the equipment and expertise to flush the ECU memory, and software updates are usually done by replacing the entire ECU. This situation may change in future.

then 10 units per year, but the software and hardware has to be optimized for minimum power consumption. Also, in contrast to cars, satellites do not undergo any maintenance. As a result, our approach is well suited for this domain.

Table 1. An excerpt from the service decoder template and the corresponding generated code

Decoder procedure template

```
WITH Ada.Exceptions;

#for i "/Satellite/PUS/Service"#
 #for j "$i/Sub"#
  WITH Service#"$j/../@ID
    "#_#"$j/@ID"#;
 #end#
  -- next service type
#end#
SEPARATE (App)
PROCEDURE Decode(
       S :IN Pus.Servicetype;
       P :IN Pus.Ptr ) IS
BEGIN
  ...
  -- select the service provider
  CASE S.Typ IS

#for i "/Satellite/PUS/Service"#
   WHEN #"$i/@ID"# =>
     CASE S.Sub IS
     #for j "$i/Sub"#
       WHEN #"$j/@ID"#=>
        Service#"$j/../@ID"
          #_#"$j/@ID"#.Exec(P);
     #end#
       WHEN OTHERS => NULL;
     END CASE;
   -- next case
#end#
   WHEN OTHERS => NULL;
  END CASE;
  ...
END Decode;
```

Generated code

```
WITH Ada.Exceptions;
  WITH Service001_001;
  WITH Service001_002;
  -- next service type
  WITH Service015_001;
  WITH Service015_002;
  WITH Service015_003;
  -- next service type
  ...
SEPARATE (App)
PROCEDURE Decode(
       S :IN Pus.Servicetype;
       P :IN Pus.Ptr ) IS
BEGIN
  ...
  -- select the service provider
  CASE S.Typ IS
   WHEN 001 =>
     CASE S.Sub IS
       WHEN 001=>
         Service001_001.Exec(P);
       WHEN 002=>
         Service001_002.Exec(P);
       WHEN OTHERS => NULL;
     END CASE;
   -- next case
   WHEN 015 =>
     CASE S.Sub IS
       WHEN 001=>
         Service015_001.Exec(P);
       WHEN 002=>
         Service015_002.Exec(P);
       WHEN 003=>
         Service015_003.Exec(P);
       WHEN OTHERS => NULL;
     END CASE;
   -- next case
   ...
   -- next case
   WHEN OTHERS => NULL;
  END CASE;
  ...
END Decode;
```

11 Related Work

Various extensions to feature modeling were previously proposed (e.g., [CE00, HSV00]), but none of them addresses the problems solved by the extensions presented in this paper.

The use of a configuration tool to produce input for a generator is not new, but we are not aware of any other GUI-based general-purpose configuration editor based on feature modeling. OSEK/VDX is probably the most prominent example of using a

configuration tool and a generator in the automotive embedded domain. OSEK/VDX implementations include an application generator that takes an OIL file and generates the application-specific code. The OIL files can be written manually, or they can be created using graphical configuration tools, which most commercial OSEK/VDX implementations provide. However, the OSEK/VDX configuration space is hardwired in these tools, while our configuration editor is general-purpose and based on feature modeling. Thus, our technology can be used to rapidly create configuration support for new system families.

The extended design space approach of Baum [Bau01] comes closest to our work. The concept of an extended design space defines the system family configuration space and covers all the elements of the configuration knowledge from Fig. 1. The design space is stored in a database system and can be manipulated using an editor called Reboost. A graphical representation such as in our feature modeling approach is missing, however. The concept of an extended design space has been validated by applying it as a configuration front-end for QNX, a commercial operating system for embedded applications. The case study provides a tool for creating specific configurations, called D-Space-1. However, the tool can only produce QNX configurations and is not a general-purpose configuration editor.

12 Conclusions and Future Work

Our experience shows that generative programming is well suited for the industrial use in the embedded domain. However, the specific technologies being used need to fit industrial constraints such as mandated programming languages, existing validation and verification procedures, and economics of supporting system variants.

Our future work will include formalizing the representation techniques of the generative domain model and completing the modeling and configuration tools to support all the elements of the generative domain model including consistency constraints, default dependency rules, and rules for computing implementation features.

Acknowledgements. The authors would like to thank Craig Cleaveland and the anonymous reviewers for providing valuable comments on the paper.

References

[Bli01] Frank Blinn. Entwurf und Implementierung eines Generators für Merkmalmetamodelle. Diploma thesis, I/MST, University of Applied Sciences Kaiserslautern, Zweibrücken 2001 (in German)

[BO92] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. In *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, October 1992, pp. 355–398

[Bos00] J. Bosch. *Design and Use of Software Architecture: Adopting and evolving a product-line approach*. Addison-Wesley, 2000

[Bau01] L. Baum. A Generative Approach to Customized Run-time Platforms. Ph.D. thesis, University of Kaiserslautern, 2001 (Shaker, Aachen, 2001, ISBN 3-8265-8966-1)

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000

[Cla01a] M. Clauß. Modeling Variability with UML. In online proceedings of the GCSE 2001 Young Researchers Workshop, 2001, http://i44w3.info.uni-karlsruhe.de/~heuzer/GCSE-YRW2001/program.html

[Cla01b] M. Clauß. Untersuchung der Modellierung von Variabilität in UML. Diploma Thesis, Technical University of Dresden, 2001, (in German)

[Cle01] C. Cleaveland. *Program Generators with XML and Java*. Prentice-Hall, 2001

[Cle88] J. C. Cleaveland. Building Application Generators. In *IEEE Software*, no. 4, vol. 9, July 1988, pp. 25-33

[CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001

[ESBC01] U. Eisenecker, M. Selbig, F. Blinn, K. Czarnecki. Feature Modeling for Software System Families. In *OBJEKTspektrum*, No. 5, 2001, pp. 23-30 (in German)

[GFA98] M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse (ICSR5)*. P. Devanbu and J. Poulin, (Eds.), IEEE Computer Society Press, 1998, pp. 76-85, see www.intecs.it

[HSV00] A. Hein, M. Schlick, R. Vinga-Martins. Applying Feature Models in Industrial Settings. In *Software Product Lines: Experience and Research Directions. (Proceedings of The First Software Product Line Conference - SPLC1)*. P. Donohoe, Kluwer Academic Publishers, 2000, pp.47-70

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990

[LMB+01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, P. Volgyesi. The Generic Modeling Environment. Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001, http://www.isis.vanderbilt.edu/

[MC] Domain-Specific Modeling: 10 Times Faster Than UML. Whitepaper by MetaCase Consulting available at http://www.metacase.com/papers/index.html

[Nei80] J. Neighbors. Software construction using components. Ph. D. Thesis, (Technical Report TR-160), University of California, Irvine, 1980

[OIL] OSEK/VDX System Generation – OIL: OSEK Implementation Language Version 2.3, September 10th, 2001, http://www.osek-vdx.org/

[OSEK] OSEK/VDX Operating System Specification Version 2.2. OSEK/VDX Technical Committee, September 10th, 2001, http://www.osek-vdx.org/

[Par76] D. Parnas. On the design and development of program families. In *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, 1976, pp. 1-9

[PUS] Packet Utilisation Standard. European Space Agency, ESA PSS-07-101 Issue 1, 1994

[Sel00] Mario Selbig. A Feature Diagram-Editor – Analysis, Design, and Implementation of its Core Functionality. Dimploma Thesis, I/MST, University of Applied Sciences Kaiserslautern, Zweibrücken 2000

[UML] OMG Unified Modeling Language Specification, Version 1.4. OMG, September 2001, www.omg.org

[WL99] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Reading, MA, 1999

[Wol02] W. Wolf. What Is Embedded Computing? In *IEEE Computer*, January 2002, 136-137

[XP] XML Path Language (XPath), Version 1.0. W3C Recommendation, November 16, 1999, www.w3.org