Variability in Automation System Models

Gerd Dauenhauer, Thomas Aschauer, Wolfgang Pree

C. Doppler Laboratory Embedded Software Systems, University of Salzburg
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract. Model driven engineering as well as software product line engineering are two approaches that increase the productivity of creating software. Despite the rather mature support of the individual approaches, tools and techniques for their combination, promising product specific customization of models, are still inadequate. We identify core problems of current approaches when applied to automation system models and propose a solution based on an explicit notion of variability embedded in the core of the modeling language itself.

1 Introduction

Model driven engineering (MDE) is becoming increasingly popular for developing complex software intensive systems. Prominent examples include the Object Management Group's Model Driven Architecture [1] initiative, targeted mainly at generating executable software, and MATLAB/Simulink [2], which is widely used for example in the automotive industry for designing control algorithms. Both approaches allow the user to define the behavior of a system in terms of a high-level model which is then transformed into low-level implementation code.

Besides executable code, MDE may also be targeted at other artifacts such as configuration files. Our group, for example, cooperates with a provider of a specific kind of automation systems, called *testbeds* used for example in the automotive industry for developing combustion engines. Due to the ever changing measurement tasks during engine development, testbeds must be highly flexible and customizable. This flexibility is achieved through configuration parameters of the automation system software. Instead of the laborious and error prone process of manually configuring the automation system – a typical configuration comprises tens of thousands of individual parameter values – we apply MDE to let the users work with models of testbeds and to automatically derive configuration data. Figure 1 shows the tool chain.

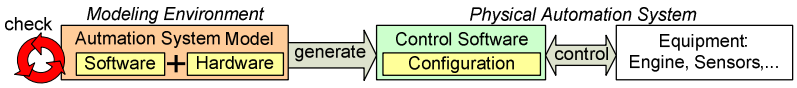

Fig. 1. Model driven engineering for configuration parameter generation**.**

Since a testbed can be used for different measurement tasks, its automation system software has to be configured accordingly. Think for example of a testbed for diesel and gasoline engines. If a diesel engine is operated, all gasoline related hardware and software parts of the testbed must be disabled, and vice versa. In order to derive configuration parameters, the model must be modified each time a different task is to be performed. For a typical testbed, however, different usage scenarios can be anticipated. Instead of manually *modifying* the testbed model each time the measurement task changes, the testbed model could already incorporate these usage scenarios, so the model would allow *choosing* between predefined model *variants*.

Expressed in terms of software product line engineering (SPLE) [3], such a testbed model represents a *product line*, from which specific *products* can be generated by making decisions at *variation points*. While the product line comprises the union of all possible testbed model variants, a product represents a specific testbed model from which configuration parameters can be generated. Variation points in our case are for example the choices between diesel and gasoline fuel. Products are created from reusable *assets* according to selections made at variation points; in our case these assets are for example model fragments representing diesel and gasoline fuel subsystems. Assets may also contain variation points, i.e. they may be parameterized. For example a diesel fuel subsystem may support weight based or flow based measurement of fuel consumption. We will use these terms throughout the rest of the paper.

SPLE is concerned with two major aspects: (a) the technical representation of variability, i.e. representing assets and variation points, and (b) feature modeling, i.e. the representation of dependencies among variation points in terms of conditional expressions. Assets used in SPLE often are source code fragments, and variation points are represented as #IFDEF-like annotations within the source code. No generally applicable equivalent however is available for the case of graphical models. In the rest of the paper we thus focus on the representation of variability in a modeling evironment for testbed automation systems and only briefly touch feature models. We describe commonly used workarounds for combining SPLE and MDE and then present our own approach, which we think is applicable to other domain, too.

2 Problems of Current Approaches

Although software product line engineering techniques are already applied to model driven engineering, we consider the current approaches inappropriate for automation system models. The problems stem mainly from the fact that SPLE is implemented as an add-on to existing MDE tools. This section uses examples to describe two conceptual approaches of applying SPLE approaches to MDE, and highlights their shortcomings when it comes to modeling both, software and hardware aspects which is essential for our domain. In testbed models, software aspects are represented in a graphical dataflow model; hardware aspects are represented in an electrical wiring model.

2.1 Using Dedicated Model Elements to Represent Variation Points

This approach is based on positive variability where a model comprises the union of all model elements used in any of the testbed's variants. Decisions made in the feature model are used to configure the model. Feature selection may be done through parameterization of the model in place or by creating a new model from a subset of the existing source model through some model transformation. Modeling a union model however is not always straightforward as shown in figure 2 a). The semantics of dataflow models for example usually forbids connecting multiple output signals to one input signal, so we cannot simply define both connections and choose between the connections depending on the feature model. We must find workarounds instead.

An existing model can be parameterized in place through removing or changing individual model elements, for example by setting the output value of a constant block in a dataflow model to a certain value. Model elements representing variation points may be explicitly marked as fixed, optional, or variable. In case of UML, for example, stereotypes may be used, as in the PLUS approach [4]. Whole parts of a model can be enabled or disabled through model elements representing switches, for example routing output signals of multiple source model elements into one input signal of a target model element as in the Koala approach [5]. Figures 2 b) and c) show how variant specific constant values in a dataflow model can be represented in both ways. As a result of the feature selection, the existing model is altered at its variation points, reflecting the choices made.

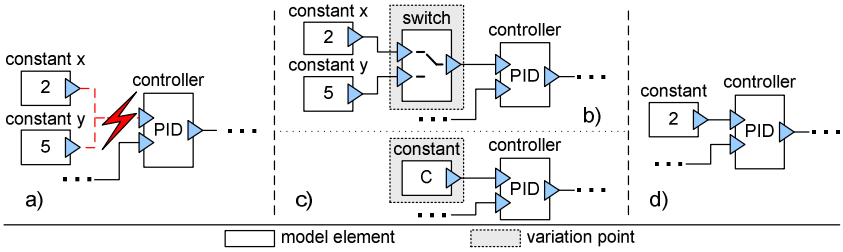

Fig. 2. A dataflow model with variation points.

Figure 2 d) shows how the final model for the example could look like if value "2" was chosen in the feature model. If the source model was figure 2 b), the constant y would be removed, along with the now unnecessary switch. If the source model was figure 2 b), the unspecified value "C" would simple be replaced by "2".

Although figure 2 b) is a possible solution to the problem of representing variability within the model, it requires additional blocks that do not stem from modeling the domain, but from the particular technical representation of variability. Such additional blocks increase the size of the model and are likely to lead to obfuscated models or "accidental complexity", as Brooks calls it [6]. The second solution shown in figure 2 c) is less complex, but one still cannot fully understand the model until the choices made in the feature model are clear, i.e. until the value "2" is specified. In the case of dataflow models, however, this approach still is a practically used solution [7].

In contrast to the dataflow model before, we now consider an example specifying a testbed's electrical wiring. Suppose a testbed supports two operation modes, one

requiring a pressure sensor, and a second one requiring temperature sensor. Both sensors would be connected to the same plug of an I/O device. In reality, of course only one sensor may be connected to the I/O device at a time. Similarly to the case of the dataflow model, a modeling environment might prevent multiple connections to a single electrical plug in the model. As a consequence, the model fragment in figure 3 a) could not express the fact that both connections are valid in principle, but not at the same time, similarly to the dataflow example in figure 2 a).

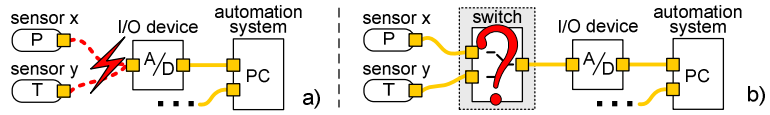

Fig. 3. An electrical wiring model with variation points.

In contrast to the dataflow example before, it is problematic to represent the variation point by introducing an artificial switch component as shown in figure 3 b), since the model would not reflect the structure of the physical testbed anymore. This is particularly important since testbed models are not just used to derive configuration data, but also to document the system's current state to provide guidance for testbed maintenance. Even if we would not require a hardware model to accurately represent its real world counterpart, modeling an artificial switch still introduces accidental complexity.

2.2 Creating Products by Merging Assets in Multiple Model Fragments

An alternative technique for representing positive variability is using multiple assets representing model fragments. One specific testbed model can then be created by merging these fragments into a single model, as illustrated in figure 4.

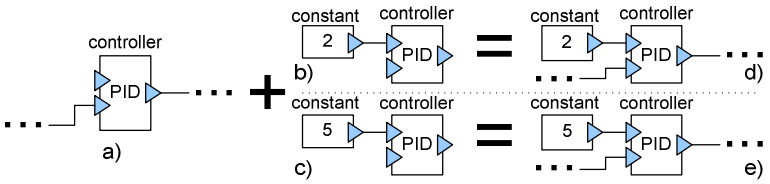

Fig. 4. Dataflow models created by merging multiple fragments.

We use the dataflow example from the previous section; the electrical wiring example can be modeled analogously. Figure 4 a) represents a fragment containing the common functionality where one of the controller's inputs is not yet connected; figures b) and c) are two additional fragments containing the variant specific elements to be merged in. The complete model shown in figure 4 d) is the result of merging fragment a) and b), while model e) is created from fragment a) and c). In order to be able to merge fragments automatically, e.g. by a dedicated external SPLE tool, models of the fragments must contain some uniquely identifiable shared elements.

In our example, fragment b) and c) both contain a PID controller. Since the common model fragment a) also contains an equal controller block, they can be merged

unambiguously by partially replacing the definition of the controller in the common fragment. If no such shared element could be identified, merging could not be performed automatically. As a consequence, fragments must be kept in sync, which may not always be straightforward if they are stored in different files, edited by different users. Another consequence of using multiple, technically independent assets is that the "big picture" of the model is lost. Consistency checks for example, e.g. compatibility between dataflow signals, can be performed only locally for each fragment; the overall consistency can not be checked until they are merged together according to a particular selection in the feature model. The resulting cycle of "feature selection, model merge, and consistency check" is cumbersome. In practice, this situation usually is avoided by encoding such technical requirements into the feature model, leading to complex feature dependencies that are intermixed with business decisions. Maintaining the feature model then becomes difficult and error-prone, since in-depth technical knowledge as well as business-specific knowledge is required.

3 A Modeling Language with Built-In Support for Variability

In the previous section we used examples to identify problems that arise when using a modeling language that does not provide direct support for representing variability. We now describe how a modeling language could support variability from a user's perspective. Our approach is related to the modeling approach using multiple model fragments as described in section 2.2. Instead of using technically independent assets, such as different files, we treat them as conceptual entities defined *within one single model*, in order to avoid the disadvantages of that approach.

We start with a model defining the common behavior. We then explicitly define the variation points with additional fragments and define their behavior as an extension of the common model. Picking up the example from figure 2, figure 5 shows how our approach can be applied to define the dataflow model: a) describes the common behavior, b) and c) are each defined by *incrementally* defining the variations.

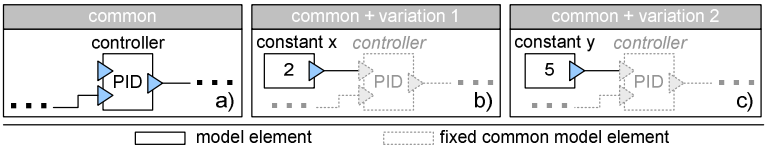

Fig. 5. Explicit variations in the dataflow models.

Note that the first input port of the PID controller in fragment a) is not yet connected. Incremental fragments b) and c) only define the constant block and the connection to the PID controller in the common model. The PID controller drawn in dashed lines and any other model elements defined by the common fragment are *fixed* since the model fragment for a variation may only introduce additional behavior.

Instead of introducing fragment specific model elements as used in figure 5 that differ only in their parameter values, we explicitly provide a means for representing fragment specific parameter values. Figure 6 shows how our example could be repre-

sented more concisely. Fragment a) again is the common behavior, now however it already contains a constant block with a default value explicitly marked as modifiable in variants. Fragments b) and c) only redefine the parameter value. They do not introduce additional model elements anymore.

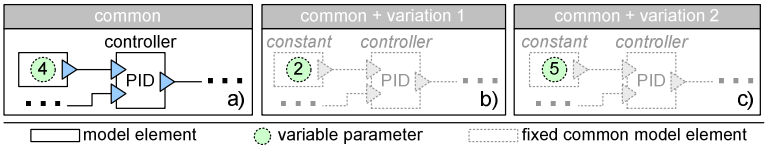

Fig. 6. Explicit variation of parameter values.

Figure 7 shows a hardware example corresponding to figure 3. Note that no artificial switch component from figure 3 b) is needed anymore, and both wire connections from sensors x and y to the I/O device's input plug can be represented without contradictions. Again, a) represents the common behavior, where models b) and c) represent the fragments using different sensors, where the common behavior is fixed.

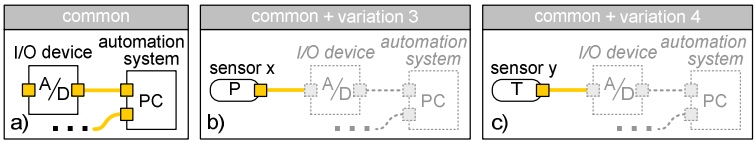

Fig. 7. Explicit variation in the electrical wiring model.

The main difference to the merging approach described in section 2.2 is that in our approach the common model and the fragments all *share the same model elements*. There are no model elements that must be kept in sync in order to merge fragments correctly. In our example, variations 3 and 4 both reference the same single I/O device element in the model. Another major difference is that in our approach, the fragments describing the variations extend the common model fragment and as such more consistency checks can be performed *in context*. For example, in variations 3 and 4, not only the fact is modeled that sensor x or y is connected to the I/O device, but in addition, the whole signal chain from the sensors to the automation system can be traced and checked for consistency, e.g. for proper encoding of sensor data on bus messages between the I/O device and the automation system. By keeping all fragments within the same model, we however face the disadvantage that individual fragments can not be used outside their anticipated scope. Additionally, model fragments can not be created in a distributed environment without further support of the modeling environment. We think that such restrictions are acceptable, though.

3.1 Variation Points and Feature Modeling

Although feature modeling is not the main focus of scope of this paper, we however briefly describe how our modeling approach affects feature modeling. Similarly to the approach in section 2.2, we create a complete testbed model by merging together

multiple model fragments. The fragments to merge may be selected manually or they may be defined by choices made in the feature model. Feature models are used to express logical dependencies among variation points in a model. Fragments from figure 4 b) and c) for example cannot be merged, since they contain contradicting definitions for constant values. The same holds true for our fragments 6 b) and c). While in the conventional approaches for combining SPLE with MDE such dependencies have to be modeled explicitly, in our approach these *technical* conflicts can already be derived from the model. Table 1 shows the tabular representation of model elements and fragments from our example in figure 5 and 6.

Table 1. Variants in tabular form describing their model elements.

Model element	**common**	**+ var. 1**	**+ var. 2**	**+ var. 3**	**+ var. 4**
constant	4	***2***	***5***	4	4
controller	✓	✓	✓	✓	✓
sensor x				✓	
sensor y					✓
I/O device	✓			✓	✓
automation system	✓			✓	✓
…	…	…	…	…	…

Model elements such as constant, controller, I/O device, and automation system are used in the common fragment, but sensor x and sensor y are not. The constant is used also in variation 1 and 2 representing dataflow model fragments with a value of "2" and "5" respectively. It is also implicitly used in variation 3 and 4 with the default value defined in the common fragment. From these definitions, one can automatically derive that variation 1 and 2 cannot be used simultaneously. The controller is used unmodified in variation 1 and 2 and again implicitly used in variation 3 and 4. Note that the connections between model elements representing signal flows or electrical wires are also model elements but are skipped here; they would also be explicitly marked present or missing in each of the fragments.

Note that the model with its variation points does not yet define a complete feature model. For example we described "common + variation 1", as well as "common + variation 3", but did not make statements about whether these variations can be active simultaneously. But since there are no conflicts visible in the table, these fragments may *technically* both be used together. We currently do not care whether merging these fragments make sense from e.g. a *business perspective* though. We think that manually creating a testbed model from fragments like this, i.e. by enabling "columns" from the table is a possible first step to a full-featured product line support in our modeling environment. We thus consider presenting such a tabular view to the users as the means for configuring testbed models; a feature model, however, could be defined on top of this technical basis later on.

3.2 Variation Points of Model Elements

So far we have described how a model can be created by enabling model fragments, i.e. by selecting which model elements to use, how to connect them and what their

parameter values should be. As motivated in the introduction, model elements may themselves come in multiple variants. Their variants are chosen in the same way as described above. After all, model elements and the testbed model are not different in a technical sense; a testbed model itself is also just a model element that could in principle be used in a model of a factory. As an example, consider a testbed model that contains a diesel fuel system model element. This system may come with weight or flow based measurement of fuel consumption and as such defines one variation point.

4 Model and Variability Representation

In order to sketch how we implement variability, we first introduce the language's existing implementation core. Similar to other modeling languages such as the UML [8] with MOF [9] as its core, our modeling language uses a set of core of primitives. Our language, however, is based on a unification of *cla*sses and ob*jects* known as clabjects [10, 11]. White boxes in figure 8 represent a simplified view on our modeling language core, which is sufficient for our discussion here.

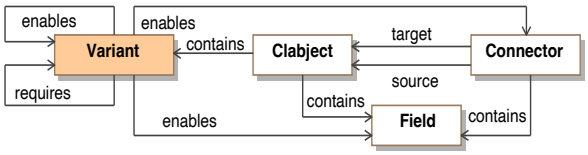

Fig. 8. Clabject based modeling language core**.**

Clabjects are used to represent model elements that users work with, for example sensors and I/O devices. A clabject may represent either a type or an instance. A clabject can be associated with another clabject by means of a connector, for example an I/O device may contain electrical plugs. A connector also may represent either a type, e.g. specifying cardinality, or an instance, i.e. a link between two clabjects representing instances. A connector may either represent composition of a clabject and its contained clabjects, or it may represent general associations between clabjects. Both, clabject and connector can have fields for representing parameter values, for example the value of the constant in our dataflow example. Again, a field may be either a type specifying e.g. the data type, or an instance specifying a value. Each of the basic elements, clabject, connector, and field, has another relation that allows defining *subtype* and *instantiation* relations between elements; these relations are skipped here to reduce clutter. Although the modeling language core does not define semantics of a specific domain, our modeling environment however ensures that the models are consistent, i.e. that for example the connector instances are established between compatible clabject instances and that the model adheres to the multiplicity constraints.

The examples in section 3 informally introduced the kinds of variability we support in our modeling language. These are: (1) enabling/disabling connections, for example between the sensors and the I/O device in figure 7, (2) enabling/disabling model elements, for example the different sensors in figure 7, (3) enabling/disabling variant specific field values, for example the constant values in figure 6, and (4) ena-

bling/disabling of variants of contained model elements, for example the fuel consumption measurement. The basic model is thus extended with an explicit notion of one or more variants; one variant is always implicitly defined as the common variant. The clabject representing e.g. a whole testbed thus contains the union of all clabjects, connectors and fields used in any of these variants. The *enables* relations between a variant and a subset of the clabjects, connectors and fields now explicitly define which of these parts it requires. Two additional relations are defined for the variant: an *enables* relation is used to chose between variants of a contained clabject, while the *requires* relation is used to represent the DAG of variant dependencies.

Using this basic mechanism, we can represent the different kinds of variation easily: (1) can be represented straightforwardly by an *enables* relation between a variant and a connector instance. (2) can be represented in the same way by an *enables* relation between a variant and a connector instance; note that the containment of clabjects is represented by connectors, too. (3) can be represented by an *enables* relation between the variant and a specific field value. (4) can be represented by an *enables* relation between the clabject's variant and a variant of a contained clabjects.

5 Related Work

Voelter and Groher [12] use the terms *negative* and *positive* variability to describe how models are constructed. Negative variability, uses a model from which unnecessary features are selectively removed to get a specific model. Positive variability, in contrast, uses a core model to which features are added.

SPLE in MDE often is done by configuring models according to the choices made in an external tool. pure::variants *Connector* is such a tool for MATLAB/Simulink [13]. It imports Simulink blocks as assets into the separate feature modeling tool. For a certain feature selection the corresponding Simulink blocks are added, removed, or their parameters are set, and also signals, i.e. connections between blocks, can be created or deleted. Thus this product supports positive as well as negative variability.

BigLever provides an analogous commercial *Bridge* solution [14] for integrating Telelogic's Rhapsody [15] UML and SysML modeling tool into their Gears SPLE tool. Elements in a Rhapsody model are turned into variation points, managed by Gears. Thus this commercial product supports negative variability.

Creating models from fragments seems to be less well supported. Voelter and Groher [12], for example describe a solution based on positive variability using aspect oriented software development. Among their assets are models that are merged or *woven* together according to a feature model. Straw et al. [17] show how UML-like class models can be merged, while Herrmann et al. [16] present an algebraic view on model composition.

6 Conclusion

In this paper we motivated why a modeling environment for automation systems should support variability of models. We first described why we consider existing

SPLE approaches insufficient in this context. We introduced our alternative approach from a user perspective first and briefly outlined how it could be integrated seamlessly with our clabject based modeling language core. We already have implemented the modeling environment and demonstrated its applicability to the domain. We did however not yet implement the variability support. Although we described variability using examples from the engine testbed domain, we expect that our approach can be applied to other sufficiently complex domains as well.

References

1. OMG Model Driven Architecture, http://www.omg.org/mda
2. The MathWorks MATLAB/Simulink, http://www.mathworks.com/products/simulink
3. Clements, P., Northrop, L., Northrop, L. M.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional (2001)
4. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison Wesley, Redwood City (2004)
5. van Ommering, R., van der Linden, F., Kramer, J., and Magee, J.: The Koala Component Model for Consumer Electronics Software. Computer vol. 33, 3 (2000)
6. Brooks, F. P.: No Silver Bullet Essence and Accidents of Software Engineering, In: Computer, vol. 20/4, pp. 10—19, IEEE Computer Society Press (1987)
7. Dziobek, C., Loew, J., Przystas, W., Weiland, J.: Von Vielfalt und Variabilität – Handhabung von Funktionsvarianten in Simulink-Modellen, In: Elektronik Automotive, vol 2, pp. 33—37, WEKA Fachmedien GmbH (2008)
8. Object Management Group: Unified Modeling Language Superstructure, v 2.1.2 (2007)
9. Object Management Group: Meta Object Facility, http://www.omg.org/mof
10. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling, In: Proceedings of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools, pp. 19—33, Springer-Verlag (2001)
11. Aschauer, T., Dauenhauer, G., Pree, W.: Multi-Level Modeling for Industrial Automation Systems. 35 th Euromicro Conference on Software Engineering and Advanced Applications, to appear (2009)
12. Voelter, M. and Groher, I. 2007. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In Proceedings of the SPLC'07, pp. 233-242, IEEE Computer Society (2007)
13. pure systems: pure:variants Connector for MATLAB®/Simulink®, http://www.pure-systems.com (2009)
14. BigLever Telelogic Rhapsody® Gears™ Bridge, http://www.biglever.com/extras/Rhapsody_Gears_Data_Sheet.pdf (2009)
15. Telelogic Rhapsody®, http://modeling.telelogic.com/products/rhapsody, (2009)
16. Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., and Voelkel, S.: An algebraic view on the semantics of model composition. In Proceeding of the 3rd European Conference on Model Driven Architecture, Foundations and Applications, pp. 99—13. Springer-Verlag (2007)
17. Straw, G., Georg, G., Song, E., Ghosh, S., France, R., and Bieman, J.: Model Composition Directives, In Proceedings of the 7th UML Conference, pp. 87—94, Springer-Verlag (2004)