Modeling in an Architectural Variability Description Language[1]

Theo Dirk Meijler[1], Silvie Schoenmaker[1], Egbert de Ruijter[2]

[1] University of Groningen, Department of Mathematics and Computer Science, PO Box 800, NL9700AV Groningen, The Netherlands
t.d.meijler@cs.rug.nl, silvie@home.nl
[2] Thales Naval Nederland, PO Box 42, NL7550 GD Hengelo Ov. The Netherlands
egbert.deruijter@nl.thalesgroup.com

Abstract. In order to handle the large amount of variability in product families, automated product derivation support is desirable. To make automated product derivation possible one important ingredient is that the solution domain should be formalized. This should allow the formal description of the variability and the formal description of the choices. Such a formalization should be presented to the application engineer in a way that gives insight in the functioning of the system. To give insight in the structure and functioning of a software system at a high-level of abstraction, an architectural description is often used. Lately, so-called ADL's (Architectural Description Languages) have been introduced for allowing clear architectural descriptions with well-defined semantics. It is the contribution of this paper to provide an ADL extension called AVDL (Architectural Variability Description Language), which allows formalizing variability at the architectural level. Relevant aspects of AVDL are described in this paper, and the promising results of applying AVDL to two industrial based examples are presented.

1 Introduction

Product families are a means for large-scale reuse of software, consisting of a reusable common software architecture and reusable components [BOSCH00][CLEME02]. They allow distributing the investment of building large complex systems over customers and are therefore widely used in industry [NORTH02]. In order to serve various customers, product families have a large "in-built" variability, e.g. different variants of components that can be used and possible variations in their implementation. This large variability may however again lead to problems in efficiently realizing specific products for specific customers, thus hampering the efficiency gain of large-scale reuse. One reason is that there may be so much variability that knowing which variant(s) to fill in for realizing which feature in

[1] This work was performed in the context of an IST programme, project ConIPF: Configuration of Industrial Product Families

the product is difficult to find out. Another reason is that a chosen variant may be difficult to implement.

We assert therefore that in order to make product derivation in the context of software product families more efficient, support for the application engineer is needed. In the European project ConIPF[1] such product support is developed.

ConIPF product derivation support is based on Knowledge based configuration as known in AI [STUMP97]. The knowledge firstly encompasses problem space variability, i.e., the possible choices the application engineer has to describe the product in terms of the features that the product must deliver. Here a feature is understood to describe "prominent or distinctive user-visible aspects, quality or characteristics of a software system or systems" [KANG90]. The knowledge also encompasses solution space variability, including e.g., possible variants of components. It furthermore encompasses how choices in the problem space are mapped to the solution space. Thus, using this knowledge, support can be given to the application engineer to correctly configure in the problem space and map this to a solution configuration. The knowledge finally encompasses how a solution space configuration can be mapped to a final realization, in code extensions or parameterization of artifacts.

Thus in the proposed product derivation support, both the formalized description of the problem space and of the solution space and of the mapping between the two is needed. In this paper we will focus on the formal description of the variability in the solution space, i.e., the description of possible solutions in the context of a software product family. One important requirement for such a description is that a mapping must be possible between a solution configuration to real implementation artifacts, such as files, settings etc. One other requirement is that both the formal description of the possible solutions and the configuration should make sense to the application engineer and give insight in the functioning of the system.

Typically engineers describe and develop their solution at relatively separate abstraction levels or aspects, e.g., the highest abstraction level of components cooperating in a certain structure, which is called the architecture. The next abstraction level is (in an object-oriented approach) the used classes for realizing such components. Other abstraction levels may be the code level for realizing certain code/ methods within classes. Furthermore, the run-time parameterization of components and objects maybe another aspect layer of the solution formalization and modeling. The main question to be answered in this paper is how to formalize possible solutions and the corresponding configuration at the *architectural* level.

In current literature much attention has been given to modeling architecture. A need for a modeling notation for architectures has been established that has the following properties: [CLBBG02][MEDVI00]:

- Clear and simple for communicating the high-level functioning of a system with stakeholders,
- Useful for estimating certain quality attributes, such as performance, availability, maintainability,
- Clear semantics, we note that this becomes especially relevant when it is necessary to map an architecture configuration to an implementation,

From this need so-called architectural description languages have come up [MEDVI00]. Currently, some of these notations are also input to the UML2.0 standardization effort. Mainly, these notations include components, connectors, interfaces as will be detailed later on in this paper.

However, these approaches in general are not directed towards describing variability of possible products in the context of a product family. In order to hold-on to the positive aspects as mentioned above, an extension to ADL formalisms for describing the architectural variability in the context of a product family is needed. It is the contribution of this paper to provide an ADL extension called AVDL (Architectural Variability Description Language) for formalizing the solution space of a product family at the architectural level of abstraction.

The remainder of this paper is structured as follows:

In Section 2 we introduce (abstracted) examples obtained through our industrial partners. In Section 3 we describe the main requirements for formalizing architectural variability, which includes our basic assumption that we use the main elements from an ADL. In Section 4 we introduce different aspects and possibilities of AVDL. In Section 5 we give results of a preliminary validation of AVDL by discussing how the examples described in Section 2 are solved using the approach. In Section 5 we discuss related work and in Section 6 we conclude.

2 Examples

This work has been tested preliminarily on basis of two examples of architectural variability as obtained from Robert Bosch GMBH Germany and Thales Naval Nederland The Netherlands. At Bosch we especially examine software for cars and trucks, so-called automotive systems. We have used a conceivable (but not realized) example that came up during our discussions with them. A Combat Management System (CMS) originating from Thales serves as a basis for our second case. Such a CMS contains subsystems that control sensor (e.g. radars) and actuators (e.g. guns) and interacts with human operators; also contained are the required connections between those subsystems. These cases also serve as basis for the examples used in this paper. However, these examples have been transformed in order to remove business critical information and simplified to focus at essentials.

Our examples correspond with two different kinds of variability that can occur:

1. Variability w.r.t variants of components and their cooperation.

   Description: In an architecture the high-level cooperation of components is described. In these cooperation components have different functions. A component serving a certain function can have several variants. Certain variants of one functional component may also depend on certain variants of an other functional component.

   Example: In the Bosch example, a motor control component and a vehicle control component cooperate. Three variants of motor control are of interest for this example: Gasoline, Diesel and Turbo-Diesel. For vehicle control, two relevant variants are vehicle control with ambient air pressure measurement and vehicle control without ambient air pressure measurement. When the motor is a Turbo Diesel, the motor control needs ambient air pressure measurement information which it should get from the vehicle control. The other motor controls do not need this information. We want to be able to describe this variability.

2. Variability w.r.t. the sub-architecture of a used component

   Description: In an architecture model also the internal architecture of a component can be of interest. Also in this internal architecture variability can occur.

   Example: In the Thales example a track management system keeps track of the position of an incoming (possibly dangerous) projectile or airplane. Within that component, two variants of a tracking system can be chosen and two variants of data logging system.

In the two examples there is furthermore a difference in the way the communication between the components is realized. At Bosch, the communication between components is based on direct point-to-point messages. At Thales, components communicate through a so-called bus, on which a publisher – subscriber pattern has been realized: Components can publish information, while other components can subscribe to specific information.

3 Requirements for formalizing architectural variability

As already described in Section 1, AVDL extends on the modeling of architectures as given by typical ADL's. An architecture modeled in an ADL has the following advantages:

- Ease of communication with stakeholders about functioning,
- Estimation of quality attributes,
- Well-defined semantics

We note that a well-defined semantics is especially relevant since in product derivation support it must be possible to generate a realization from a specific configuration.

In the following, we shall define a set of requirements that AVDL must comply to:

1. **Domain independence**. We require that the language is domain independent meaning that the same language can be used in different kinds of product families. For example it should be able to handle the differences in the way communication is realized in the two examples.
2. **Clear Graphical Notation.** A graphical notation for architectural variability is required that can easily be communicated to stakeholders.
3. **Location of Variability.** It must be explicit where variability is in the architecture. We call this a *variation point*. For example it must be expressible where in the architecture variants such as Gasoline, Diesel and Turbo Diesel motor control can be chosen.
4. **Different kinds of variation points.** According to [BABA01], there are different kinds of variation points that must be supported:
   a. Optional: There exists exactly one component that could be included in the product.
   b. Alternative: There exist multiple components and one of them must be included.

   c. Set of alternatives: There exist multiple components and at least one of them must be included.
   d. Optional alternative: There exist multiple components and one of them could be included.
   e. Optional set of alternatives: There exist multiple components and a collection of them could be included.
5. **Explicit Possible Choices.** It must be expressible what the alternatives are. For instance we want to express that the developer has the choice between the Gasoline, Diesel and Turbo Diesel motor control at a certain variation point.
6. **Dependencies between components.** Certain components depend on each other to function (e.g., the turbo Diesel component depends on the motor control with pressure measurement to function). Certain components may not be able to function together. Such dependencies between components must be expressible.
7. **Expressing a Choice.** It must be possible to be able to express the choices of the product developer, so the language must not only allow expressing the variability of the architecture but expressing that choice in the architecture. In our example, next to expressing the variability in terms of the variants Gasoline, Diesel, and Turbo Diesel, it must be possible to represent a specific choice, e.g. Gasoline.
8. **Backtracking.** When a choice has been made, this must not mean that the variability is no longer there: the choice can still be reconsidered, see however the requirement on binding times in point 9 which gives certain restrictions on this. Thus, variability in the choice of components such as Gasoline, Diesel, and Turbo Diesel is not removed when the choice for Gasoline is made, the developer can still reconsider and choose Diesel. These two aspects: variability and the current choice are together part of the same architecture description. Thus, the way variability is modeled and choices are represented should take into account that backtracking must remain possible.
9. **Taking into account Binding Times.** Binding times must be taken into account in variability modeling. In literature, the development process of choosing variants within product derivation consists of various steps: Requirements Collection, Architectural Design, Detailed Design, Implementation, Compilation, Linking, Start-up, Run-time [GURP]. When we focus on variability in architecture, in the context of formalized product derivation support process, at the following points in time (so-called binding times) architectural choices can be made: architectural design[2], compile time, link-time, run-time. Typically at architectural design functional choices between components are made (as in our example, the choice between Gasoline, Diesel and Turbo Diesel) while at compile time choices between implementations of components can be made, realized for example via a compilation switch. Run-time, certain components may still be added to a system, e.g., by

[2] Note that normally architectural design would mean here making the full design of the architecture. Within product derivation support for a product family, only a limited set of possibilities are left.

downloading a component to be used in a browser. Summarizing, it must be taken into account that in a certain phases of the derivation process certain variability can no longer be offered since the derivation phase is past the (latest) binding time of the corresponding variation point.

4 Modeling Architectural Variability in AVDL

In this section we will introduce the main elements of AVDL. Each subsection will contain one small example and in this way introduces a certain topic.

4.1 Basic Elements

In Fig. 1 an example of a simple architecture description is given, describing the possible cooperation of a component X and a component Y over a publish-subscribe connection. This is an example without variability.

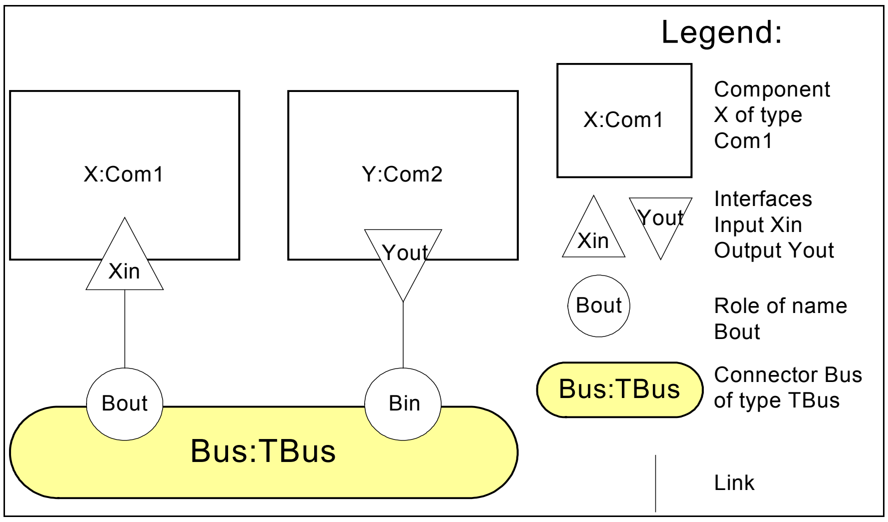

Fig. 1. A simple architecture (no variability). Output interface Yout of component Y is linked to role Bin, input interface Xin of component X is linked to Bout of connector Bus. Given that (not shown) Bin and Bout are the publish and subscribe role of Bus respectively, Y and X can communicate in this way.

In this description variability is not yet used and thus the main elements in the figure are those that are common to most ADL's [MEDVI00]: Component, connector, interface, role, link. As in other ADL's the meaning of this is the following, see for details [MEDVI00]:

Component: A component is the main element of computation.

Interface: A component can have both provided and required interfaces.

Connector: A connector describes the explicit medium of communication between components, it makes explicit that two components can cooperate.

Link and role: The interfaces of components can be linked to a role of a connector. A connector can have different kinds of roles, e.g. a role on which input of the communication is expected or one on which output of the communication is expected.

In contrast to some ADL's for example [OMMER02], we enforce that the communication between components is realized through a connector. As a result of this, the well-defined semantics of the architectural description can be upheld in the usage within different domains, as required by requirement 1. For different kinds of connectors (such as bus or point-to-point message passing) the cooperation between components as modeled through the connector and the links to that connector gets a different meaning and realization.

One other element we adopt from most ADL's is that a component itself can again have an architecture (possibly again with variability) with subcomponents. The relation between a component and its subcomponents does not need to be modeled with connectors. A link between an interface of the (outer-)component and an interface of a subcomponent has a default mapping semantics. See further Section 4.4 where components with an inner variable architecture are shown.

4.2 Variation points

In this subsection we explain some elements of how variability is modeled and used, see Fig. 2. In Fig. 2, the principle of an "alternative" variation point is modeled, see Section 3 requirement 4.

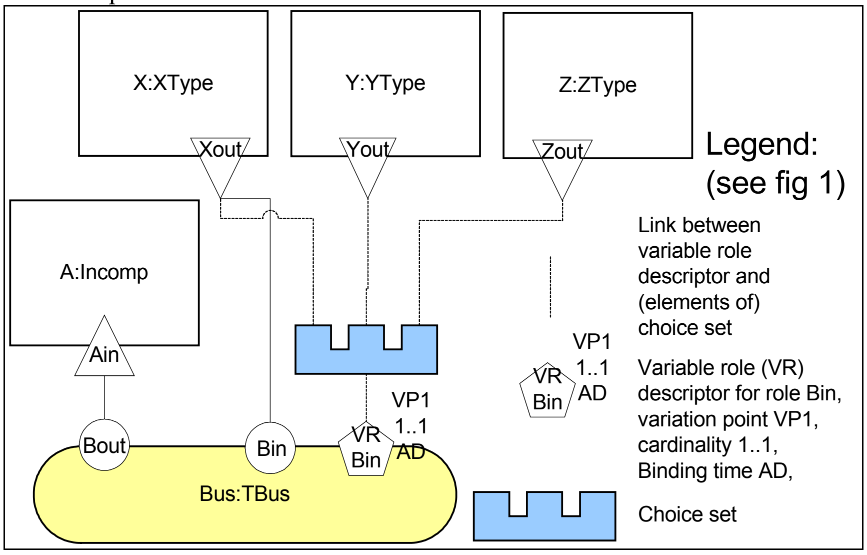
X:XType
Y:YType
Z:ZType
Xout
Yout
Zout
Legend:
(see fig 1)
A:Incomp
Ain
Link between variable role descriptor and (elements of) choice set
VP1
1..1
AD
VR
Bin
Variable role (VR) descriptor for role Bin, variation point VP1, cardinality 1..1, Binding time AD,
Bout
Bin
VR
Bin
VP1
1..1
AD
Bus:TBus
Choice set

Fig. 2. Definition of a variable role Bin for variation point VP1, with choice set (alternatives) X, Y and Z. In the corresponding role Bin a choice is made: it is linked to X.

Different aspects of variation points as discussed in Section 3 are modeled in Fig 2:

- The location of the variation point see requirement 3. The variation point is modeled by means of a variable role descriptor on a connector. This describes variability with respect to the number and linking of roles of a connector,
- The cardinality, the minimum and maximum number of components that must be chosen: In this case the cardinality is 1..1, one component must be chosen,

The choice set (alternatives) from which must be chosen in this case X, Y and Z.

For an optional alternative variation point, the cardinality would be 0..1 with a choice set of more than one. For an optional variation point, the cardinality is 0..1, with a choice set of only one component. In this way all kinds of variation points mentioned in requirement 4 can be modeled.

In Fig.2 also shows that a variable role descriptor has a binding time. This indicates when a variant must be bound, and thus in which phases the variability is applicable. The variable role descriptor VR-Bin models that the variability is only available at the binding time AD which stands for Architectural Description. Other possible abbreviations are C (for Compile-time), L (for Link-time), SU (for Startup), R (For run-time). Binding time modeling corresponds to requirement 9.

In Fig. 2 both the variability and the specific choice are represented. The role Bin contains a link to X, thus representing the choice of X for this variation point. Through this mechanism the following is achieved:

- Both the variability and the choice are represented, see requirements 5 and 7,
- Backtracking is possible: the link of Bin can be rearranged, as long as the development hasn't progressed to the next step in the process, see requirement 8.

We note explicitly that variation points as discussed here present variability with respect to the linking of roles.

4.3 Variation points with two or more Variable Role Descriptors

In Fig. 3 a variation point with two variable role descriptors is represented. This situation can be described in the following way: A component chosen from the set {X, Y, Z} must not only be linked to the connector Bus, but also to the connector CBus, possibly in order to be indirectly connected (not shown) to some user interface component.

We see in Fig. 3 that there are two variable role descriptors VR-Bin at Bus and VR-Bout at CBus both connected to the same variation point. When a choice is made at one role, e.g, to link X to the role Bin at connector Bus, a role Bout at CBus must be linked to the same component X. Since only one role can be chosen independently, we make the distinction between independent and dependent variable role descriptors. Thus for a role corresponding to the independent variable role descriptor VR-Bin at the connector Bus a choice between X, Y and Z can indeed be made while for a role corresponding to VR-Bout at CBus the linked component depends on that choice.

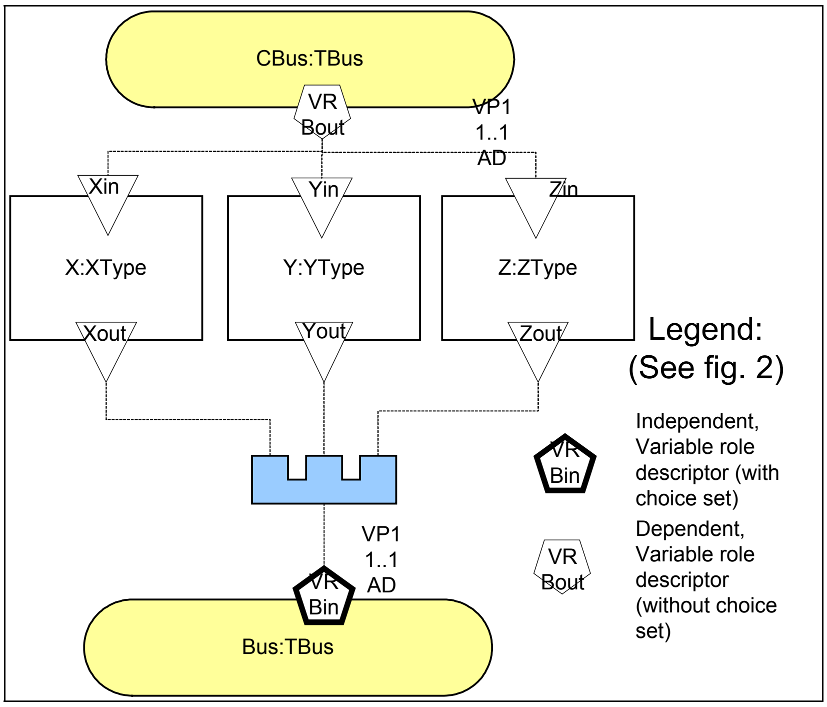

Fig. 3. Both Bin and Bout correspond to the same variation point VP1. The choice in Bout of CBus depends on the choice in Bin.

4.4 Variability in used subcomponents

As described in Section 4.1, we allow components with an inner architecture. In this inner architecture variability is again possible.

Fig. 4 shows how variability within components is modeled. So-called variable interface descriptors are introduced here. A variable interface descriptor describes how an interface of the outer component can be linked in different ways to interfaces of subcomponents. The variable interface descriptor of Sin is independent. It has the choice set {O, P}, therefore either O or P can be linked to Sin. The variable interface descriptor of SoI is dependent, so that when O is chosen at Sin, O will also be linked to SoI.

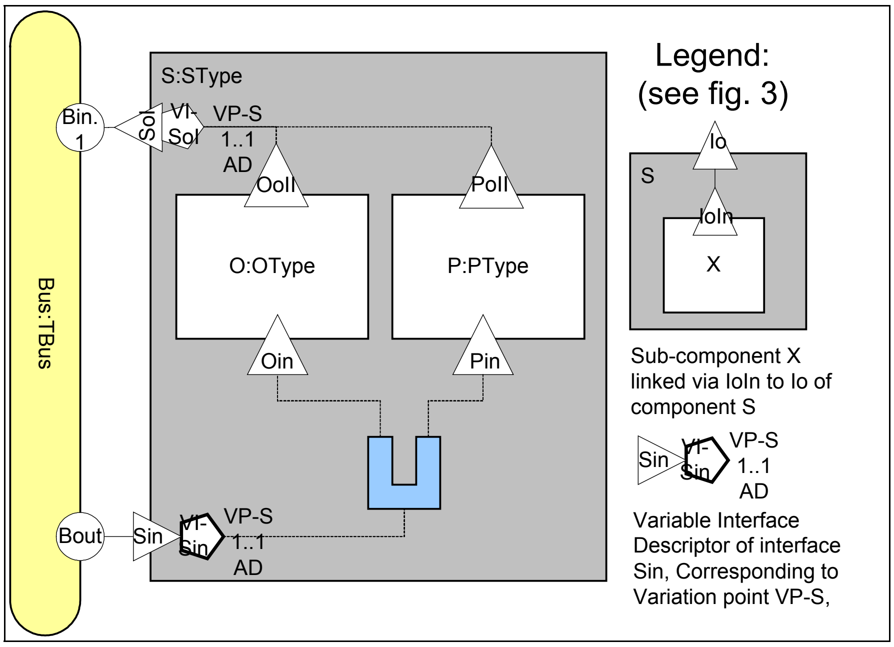

Fig. 4. A component with variability within. The Grey box S is the outer component. Within S, at the interface Sin there is one independent variable interface descriptor, the developer can choose at architectural description time between O and P. The variable interface descriptor of SoI is dependent.

5. Validation

Given the modeling elements described in Section 4 we can now model the variability for the examples in Section 2.
Fig. 5. shows a (variable) architecture modeled in AVDL with both variability and specific choices for the Bosch example. Salient aspects are:

- Turbo Diesel and car control (pressure measurement) are not explicitly linked through some kind of dependency (the dependency relationship is possible in AVDL, but not explained in Section 4), however the required – provided interface check connects the two components: when the Turbo Diesel component is selected which requires a pressure measurement interface, there must be somewhere a component that provides the pressure measurement interface. In this case only the car control (pressure measurement) component provides that interface.

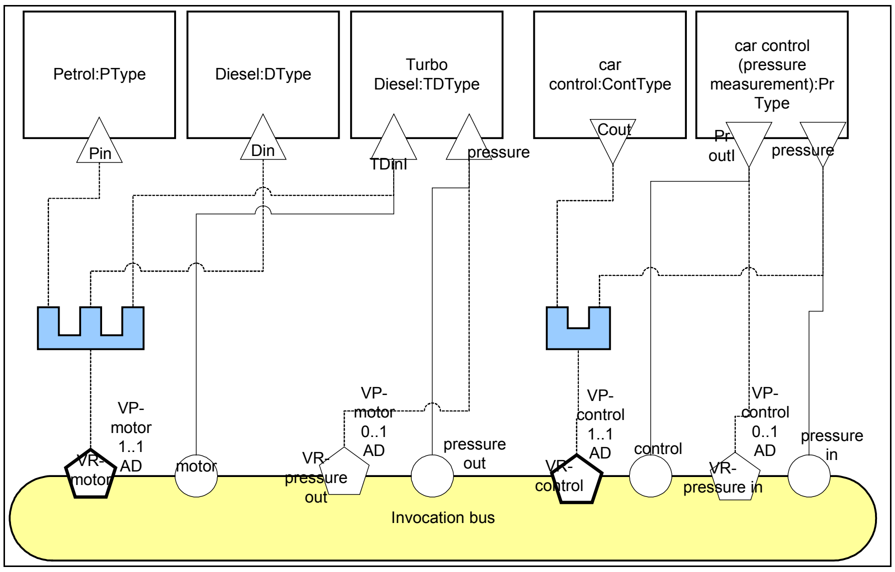

Fig. 5. Variability and valid choices for the simplified Bosch example.

- The "VR-pressure out" variable role descriptor at the invocation bus is dependent of the "VR-motor" role descriptor and describes an optional role "pressure out". Both correspond to the variation point "VP-motor". When a Turbo Diesel component is selected, the pressure out role is automatically linked to that component. The analogy is true for "VR-pressure in" and "VR-control" variable role descriptors for the variation point "VP-control".

Fig. 6 shows a part of the Thales example. In this figure no new aspects are introduced with respect to Section 4.4. For each of the two variation points (abbreviated with VP-TS for Tracking System and VP-DL for Data Logging System) there are one independent and two dependent variable interface descriptors.

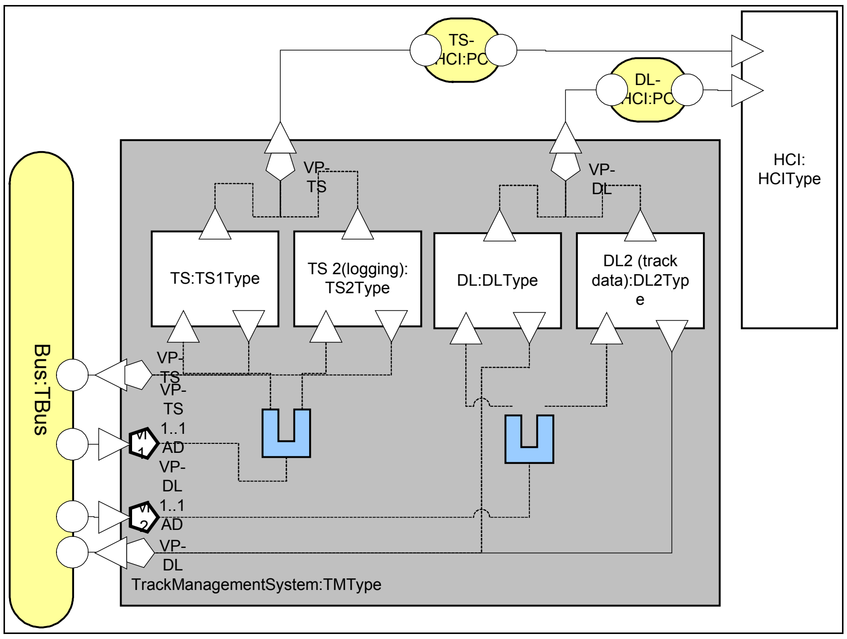

Fig. 6. Variability Modeled for the simplified Thales Example

5.2 Remarks

For the Bosch example, it is useful if the option for pressure measurement is not defined as a separate variant of the motor control but as a variation point with respect to the interface of the component. This is a useful extension of AVDL.

For the Thales example the provided modeling elements seem to be sufficient.

5 Related work

Quite some work is and has been done on handling large-scale solution domain variability in product families through the formalization of this variability, possibly mapping it to feature variability. Some of this work, e.g. [TRYGG95], [DEURS02], [GUENT92] represents variability in a way (more or less) independent of the software domain. Especially the general representation of [GUENT92] is of interest for us in order to map AVDL to this in order to get automatic support for configuration, derivation of correct products etc. However, we believe that a representation of both the variability and the configuration that gives insight to the functioning of the system is essential for supporting the derivation of complex products.

With respect to representation of variability that gives insight in the functioning of the system, some work is based on UML [CLAUS01], [MUHTI02]. While better possibilities for architectural description are introduced in version 2.0 of UML, so far UML has not been viewed as optimal for describing architecture, and **therefore** the work for describing variability through UML is better suited for describing variability at the level of detailed design.

ADL's are seen as better for describing architecture [CLBBG02]. Of interest for describing architectural variability is the work on KOALA [OMMER02] and the work on Mae [HOEK01]. An important disadvantage of Koala which makes it unsuitable for our purposes is that it is domain dependent: Links between components have a default semantics, therefore disallowing porting Koala to other domains. Mae is directed at supporting evolution through versioning, but otherwise in Mae variability is not explicitly modeled and therefore also insufficient for our purposes.

Our work makes use of the ideas of Bachmann & Bass [BABA01]. We furthermore subscribe to the viewpoint of Asikainen et.al. [ASIKA03] that the underlying aims of ADL modeling and configuration modeling are not totally similar and for that reason decided to introduce AVDL which could make this bridge.

6 Conclusion

In order to handle the large-scale variability in product families, automated product derivation support is needed. To make automated product derivation possible one important ingredient is that the solution domain should be formalized, both allowing the formal description of the variability (the possible solutions) and the formal description of the choices, the configuration. We assert that in order to support the application engineer such a formalization should be presented to the developer in a way that gives insight of the functioning of the system. It is for example of importance for the application engineer to be able to view and describe his/her solution at the architectural level, in terms of components and relationships between components. ADL's provide means for describing systems in this way and ideas of ADL's are being introduced in the coming version of UML, UML2.0. Currently however, most ADL's don't provide specific support for describing architectural variability in the context of product families. On basis of ADL's we have therefore introduced AVDL (Architectural Variability Description Language). One of the salient aspects of AVDL is that it allows describing the fixed part of the architecture, it's variability and the choices in one model. If one would separate these aspects, it would either not be clear to the developer where in the architecture certain choices can be made, and which choices (s)he made corresponding to which variability where in the architecture.

We have done a first test on AVDL, by using its language elements on one industrial based (Bosch) and one industrial example (Thales). This validation is preliminary since the examples are still not full-fledged, they don't cover the full product families, which is still to be done. The first results are however promising.

We have formalized AVDL on basis of X-ADL [XADL] which stands for "eXtensible Architectural Description Language", we plan to publish about this. In the near future we intend to study modeling of variation points as independent entities. We furthermore plan to map AVDL to a logic-based language for knowledge based configuration as described by [GUENT92]. We also intend to submit constructs defined in this paper for new versions of UML.

References

[ASIKA03] T. Asikainen, T. Soininen, and T. Männistö. Towards Managing Variability using Software Product Family Architecture Models and Product Configurators. In *Proc. of Software Variability Management Workshop*, pages 84–93, Groningen, The Netherlands, February 13-14 2003.

[BABA01] F. Bachmann, L. Bass. Managing Variability in Software Architectures. *ACM SIGSOFT Software Engineering Notes, Proceedings of the 2001 symposium on Software reusability*: putting software reuse in context, May 2001Volume 26 Issue 3

[BOSCH00] J. Bosch. Design & Use of Software Architectures: Adopting and Evolving a Product Line Approach. Addison-Wesley, May 2000.

[CLAUS01] M. Clauss. Generic Modeling using UML Extensions for Variability. In *DSVL 2001 (OOPSLA Workshop on Domain Specific Visual Languages)*. Jyvaskylae University Printing House, Jyvaskylae, Finland, 2001.

[CLEME02] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[CLBBG02] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures*. Addison-Wesley, 2002.

[DEURS02] A. van Deursen, M. de Jonge, T. Kuipers. Feature-Based Product Line Instantiation using Source-Level Packages, *Proceedings of SPLC2*, Springer-Verlag 2002

[GUENT92] A. Guenter, R. Cunis, Flexible Control in Expert Systems for Construction Tasks, *Journal Applied Intelligence*, 2(4): 369-385, 1992.

[HOEK01] A. vd. Hoek, M. Mikic-Rakic, R. Roshandel, N. Medvidovic. Taming Architectural Evolution, *Proceedings of ESEC/FSE*, ACM 2001

[KANG90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented Domain Analysis (FODA) Feasibility Study. *Technical Report CMU/SEI-90-TR-021*, 1990.

[MEDVI00] N. Medvidovic, R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages, *IEEE Transactions on Software Engineering*, 26(1): 70-93, Jan. 2000.

[MUHTI02] D. Muhtig, C. Atkinson. Model-Driven Product Line Architectures. *Proceedings of SPLC2*, Springer-Verlag 2002.

[NORTH02] L. Northrop. SEI Software Product Line Tenets. *IEEE Software*, 19(4), July/August 2002.

[OMMER02] R. van Ommering. Building Product Populations with Software Components. *Proceedings of ICSE'02*, ACM 2002

[STUMP97] M. Stumptner. An Overview of Knowledge-based Configuration. *AI Communications*, 10(2):111–126, 1997.

[TRYGG95] E. Tryggeseth, B. Gulla, and R. Conradi. Modelling Systems with Variability using the PROTEUS Configuration Language. In J. Estublier, editor, *Software Configuration Management: Selected Papers SCM-4 and SCM-5*. Springer-Verlag, Seattle, WA, USA, April 1995.

[XADL] XADL homepage. http://www.isr.uci.edu/projects/xarchuci/