Autom Softw Eng (2011) 18: 77–114
DOI 10.1007/s10515-010-0076-6

The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study

Deepak Dhungana · Paul Grünbacher · Rick Rabiser

Received: 7 May 2010 / Accepted: 1 November 2010 / Published online: 17 November 2010

Abstract The variability of a product line is typically defined in models. However, many existing variability modeling approaches are rigid and don't allow sufficient domain-specific adaptations. We have thus been developing a flexible and extensible approach for defining product line variability models. Its main purposes are to guide stakeholders through product derivation and to automatically generate product configurations. Our approach is supported by the DOPLER (**D**ecision-**O**riented **P**roduct **L**ine **E**ngineering for effective **R**euse) meta-tool that allows modelers to specify the types of reusable assets, their attributes, and dependencies for their specific system and context. The aim of this paper is to investigate the suitability of our approach for different domains. More specifically, we explored two research questions regarding the implementation of variability and the utility of DOPLER for variability modeling in different domains. We conducted a multiple case study consisting of four cases in the domains of industrial automation systems and business software. In each of these case studies we analyzed variability implementation techniques. Experts from our industry partners then developed domain-specific meta-models, tool extensions, and variability models for their product lines using DOPLER. The four cases demonstrate the flexibility of the DOPLER approach and the extensibility and adaptability of the supporting meta tool.

D. Dhungana (✉)
Christian Doppler Laboratory for Software Engineering Integration for Flexible Automation Systems, Vienna University of Technology, Vienna, Austria
e-mail: deepak.dhungana@tuwien.ac.at

P. Grünbacher
Institute Systems Engineering and Automation, Johannes Kepler University Linz, Linz, Austria
e-mail: paul.gruenbacher@jku.at

R. Rabiser
Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University Linz, Linz, Austria
e-mail: rabiser@ase.jku.at

Keywords Product line engineering · Decision models · Meta-tools

1 Introduction and motivation

Software product lines aim at increasing the degree of reuse in software engineering. A software product line has been defined as "*a set of software-intensive systems sharing a common, managed set of features that satisfy the needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" (Clements and Northrop 2001). It has been demonstrated that software product line engineering (PLE) is a successful approach in many business environments (van der Linden et al. 2007; Clements and Northrop 2001; Pohl et al. 2005).

Many software product lines today are developed and maintained using model-based approaches. Models are used to define and communicate the often tacit knowledge regarding the variability of systems and to support the derivation of new products. Defining the variability of a product line involves modeling the problem space (i.e., the variability of the product line's features and capabilities), the solution space (i.e., the architecture and the components of the technical solution), and mappings between problem and solution space (i.e., traceability links between model elements in both spaces). More specifically, modeling the *problem space* requires a language for expressing the variability of stakeholder requirements. Examples are configuration decisions or features available for selection during product derivation. The *solution space* consists of diverse reusable assets representing, e.g., the architecture, code, test suite, or documentation. Due to the complexity of real-world systems and the heterogeneous technologies used for implementing variability, solution space models need to be managed at different levels of abstraction and across arbitrary development artifacts (Berg et al. 2005). The *mappings* between the problem space and the solution space are important when configuring and assembling a product based on customers' requirements. Establishing traceability between the two spaces is a prerequisite for automation.

Numerous approaches have been proposed for modeling product lines, including feature-oriented modeling languages (Czarnecki and Eisenecker 2000; Kang et al. 1990), decision-oriented approaches (Campbell et al. 1990; Schmid and John 2004), UML-based variability modeling approaches (Gomaa and Shin 2002; Gomaa 2005), architecture modeling languages (Dashofy et al. 2002), or orthogonal approaches (Pohl et al. 2005). However, despite many success stories (Steger et al. 2004; Thiel and Hein 2002; Estublier and Vega 2005; Verlage and Kiesgen 2005) there are still several obstacles inhibiting the widespread adoption of PLE. For instance, many existing variability modeling tools are rigid and only allow minimal domain-specific adaptations. It thus remains challenging for organizations to adapt available methods and techniques to their particular development context.

A key goal of developing our DOPLER approach was therefore to make it extensible and customizable to different domains. While earlier publications introduced the modeling approach (Dhungana et al. 2007a, 2010b) and tool support (Dhungana et al. 2007b) the focus of this paper is on investigating the utility and suitability of our approach in different practical settings. It has been pointed out by several

researchers that the research community lacks reports about industrial experiences with variability modeling approaches (Berger et al. 2010). Our aim was thus to investigate how variability is represented in different industrial domains and how well our approach meets the expectations of modelers in these environments. We describe the DoplerVML variability modeling approach (Sect. 2) and the DOPLER meta-tool (Sect. 3). We then report a multiple case study comprising four cases from different domains and development environments (Sects. 4–5). We discuss results, benefits and limitations and report lessons learned (Sect. 6). We also present related work (Sect. 7) and round out the paper with conclusions (Sect. 8).

2 Decision-oriented variability modeling with DoplerVML

DoplerVML is a modeling language for defining product lines. It supports modeling the problem space using decision models and defining the solution space using asset models which represent arbitrary types of reusable assets. Figure 1 depicts the high-level meta-model of our approach. The key modeling elements are *decisions* for representing a problem space view on the product line's variability as well as *assets* for defining an abstract view of the solution space in the degree of detail needed for subsequent product derivation. Decisions and assets are linked with *inclusion conditions* that define traceability between the problem space and the solution space.

The ultimate goal of PLE is to turn out products (Clements and Northrop 2001). DoplerVML thus emphasizes product derivation by using decision models that specify the available customization options from the perspective of users. Based on the decision values set by a user, the assets required for composing the product are automatically determined and product configurations can be generated. Unlike other approaches to variability modeling the main purpose of DoplerVML is not the documentation and analysis of the domain. This means that DoplerVML models often do not describe all available features and their dependencies (which is e.g., often done in feature models for the purpose of analyses as we will also discuss in Sect. 7). Instead, DoplerVML models focus on just the decisions that are needed to generate customer-specific solutions. This means that no decisions are defined for parts of the product line that are not variable.

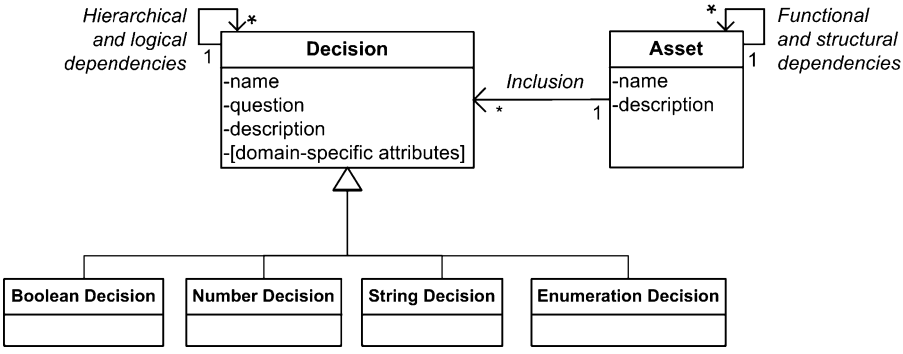

Fig. 1 The core meta-model of DoplerVML. Modelers define domain-specific meta-models based on these basic types, relationships, and attributes

This section provides a thorough description of the key modeling elements of the DoplerVML language[1] illustrated with simple examples from a component-based product line of our industry partner Siemens VAI.

2.1 Decisions

Decisions describe the variation points in a product line and define the set of choices available at a certain point in time when deriving a product. Taking a decision involves judging the merits of multiple options and then selecting a course of action among the available alternatives to reflect customers' requirements.

A decision is specified by a unique *name* and its decision *type*. Every decision corresponds to a decision variable which is comparable to a typed variable in programming languages. A *validity condition* defines the set of allowed values (with respect to the decision type and additional user-defined constraints). Further decision *attributes* can be defined to provide additional information for the user. Decisions are typically not isolated and depend on each other. For instance, taking a decision can lead to new decisions and decisions can also be constrained by already taken decisions. For this purpose, DoplerVML distinguishes hierarchical and logical dependencies (see Fig. 1). A *decision model* consists of a set of decisions and their dependencies.

The **Decision Type** defines the range of values which can be assigned to a decision. The three predefined basic decision types in our modeling language are Boolean, String and Number. We also provide an Enumeration type which has been introduced to simplify the modeling process. In the example in Fig. 2 the decision model contains the Boolean decisions UI, TT3D, and TimeLine3D as well as the number decision NumStrands. The decision FeedingMode is defined by an enumeration type describing the available feeding modes (Top, Bottom, Horizontal, Vertical).

Decision Attributes are annotations on decisions for capturing information for the modeler as well as the users taking decisions. For instance, a description allows to further document the meaning of a decision. A question defines the text that is presented to the user when enacting the decision model during product derivation. The decision FeedingMode, for example, is presented to the user using the question "Which feeding modes shall be supported?"

The **Validity Condition** restricts the value range determined by the basic decision type (which is often too broad). The validity condition of a decision can be seen as a post condition which has to be fulfilled after a user takes a decision and before assigning a value to the decision variable. Validity conditions can be arbitrarily complex Boolean expressions. An example of a validity condition specified for the decision NumStrands is NumStrands>0 && NumStrands<=6.

The **Visibility Condition** specifies when a particular decision becomes relevant to the user. The visibility condition thus defines hierarchical dependencies between decisions. Figure 2 shows the visibility condition UI==TRUE specified for the decision TT3D. A user will not be asked about 3D visualization if no user interface is to be deployed. Visibility conditions also define the order of taking decisions. If there is a visibility condition associated with a decision, the user has to first take the decisions

[1] The formal semantics of the DoplerVML modeling language are described in Dhungana et al. (2010b).

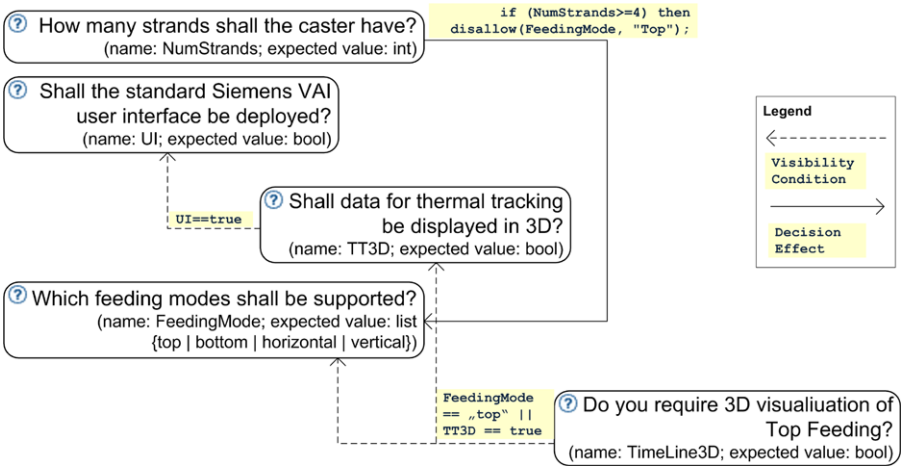

Fig. 2 Simplified example of a DoplerVML decision model of a Siemens VAI product line

appearing in the visibility condition. This possibility of hierarchically arranging decisions allows building models with few high-level decisions and lower-level decisions presented to the user later only if necessary.

Decision Effects specify logical dependencies between decisions. For example, the decision about the number of strands has an effect on the options available for the decision concerning the available feeding modes. Logical dependencies are described using conditions that are checked after a decision has been taken and actions that need to be executed depending on the condition. Decision effects are modeled using a set of rules with pre-defined actions such as setting values of other decisions. They are specified in the form: if ⟨condition⟩ then ⟨action⟩ where condition is a Boolean expression defined using decision variables and action is a function affecting decision variables. A rule is activated or triggered when its condition evaluates to TRUE.

We present a few examples of rules for the actions *assert*, *bind*, *and update*: We use the *Assert* action for dependencies among decisions, where certain conditions always need to hold. For example, a constraint in the form (v1==n1) implies (v2==n2) could be specified using the rule if (v1==n1) then assert (v2==n2) or simply assert (!(v1==n1)||(v2==n2)). *Assert* does not change the value of the variables but only makes sure that the condition holds. Whenever there is a need to change the values of the decision variables we use the *Bind* action, e.g., if (v1 == n1) then setValue(v2, n2). setValue is an example of a binding action (the actual syntax and semantics of the actions are defined by the language selected for specifying decision dependencies for the domain of interest). In general, a binding action is comparable to a constraint in constraint satisfaction problems (CSPs) where a condition implies a binding. In contrast to the assertion action, binding actions change the actual value of the decisions. The binding and assertion rules may be combined arbitrarily. The semantics of rules used for assertion and binding is identical to constraints specified using Boolean expressions in CSPs. We use the *Update* action if we want to manipulate not only the values but also different attributes of decisions. For instance, depending on the value of one decision, the validity condition of another decision might change. Such an up-

date action could also be used to change the specification of the model at runtime if desired.

2.2 Assets

Assets are used to represent solution space artifacts available in a product line. Examples are software components, test cases, or documentation fragments. For a concrete domain, asset types are specified together with their attributes and possible dependencies. For instance, an asset type Component might have an attribute File of type URL pointing to its actual definition and a requires dependency to asset type Property being an initialization parameter. The flexibility of asset models is achieved by defining a domain-specific asset meta-model that addresses domain-specific concepts based on the generic core DoplerVML meta-model depicted in Fig. 1.

An *asset model* is a collection of the assets that describes the solution space at the level of abstraction that allows subsequent derivation of products. The domain-specific meta-model defines the granularity of the reusable building blocks. While for certain product lines variability is realized at the coarse-grain level of components it might be necessary to consider even individual lines of code in other cases. It is critical to understand this granularity when defining the asset types.

An **Asset Type** is a refinement of the generic element Asset defined in the core meta-model. For instance, the three asset types Component, Resource, and Property are used to define a component-based product line of our industry partner Siemens VAI. The structure and organization of the solution space is specific for the domain and industrial context at hand. In particular, the definition of asset types for a specific domain depends on the granularity needed for subsequent automation. Building such a model thus requires knowledge about the domain and the organization's implementation practices as we will demonstrate in the case studies.

Asset Attributes are used to define additional properties of assets. For instance, in Fig. 3 the component asset CastHMI has four attributes and the property asset Feeding Mode has five attributes.

Asset Dependencies define relationships between assets. Structural dependencies are used to specify the physical organization of the assets. This can for example mean the way how assets are packaged or divided into sub-systems. Structural dependencies are represented with relationship links like *consists of*, *contributes to*, *is predecessor of*, or *is successor of*. Functional dependencies specify relationships stemming from the underlying implementation of a system. They describe the logical organization of the assets and can be represented with relationship links like *requires* or *excludes*.

Inclusion Conditions link assets to decisions. They describe the context and situation when a particular asset is required in the desired product. One inclusion condition can refer to several decision values. E.g., the inclusion condition of component Caster might expect one decision to be FALSE and another to be TRUE. In other words, assets are "aware" of the decisions as they influence their selection for a product while decisions are "unaware" of the assets realizing them.

Assets are also included if required by other assets. For example, the property Casting Lines is part of the final product if the component Caster is included due to the

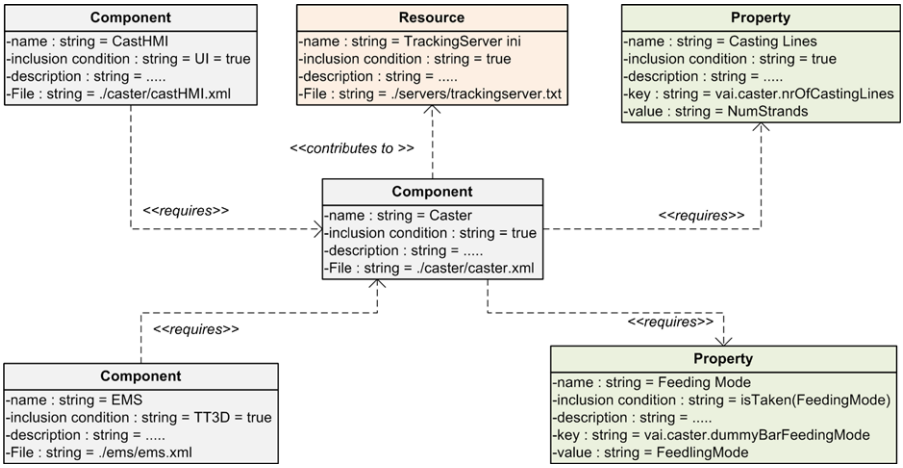

Fig. 3 A partial asset model depicting a set of available assets, their attribute values, and relationships among them. The inclusion conditions refer to the decisions defined in Fig. 2

requires dependency between the two assets. Modeling functional asset dependencies significantly reduces the number of inclusion conditions needed. Only few assets will have direct links to the decisions. More often assets will be included in a final product because of technical dependencies as shown in the aforementioned example.

3 The DOPLER meta-tool for defining and enacting variability models

Product line practices vary largely in different organizations and domains due to different types of reusable assets, architectural styles, programming languages, development tools, etc. Unfortunately, many existing variability modeling tools are rigid and only allow minimal domain-specific adaptations. Our goal was thus to develop configurable and extensible tool support based on DoplerVML that can be tailored to different domains (Grünbacher et al. 2009). We have thus developed an extensible meta-tool for defining and enacting DoplerVML models as part of our Eclipse-based DOPLER tool suite. DOPLER allows creating a meta-model to define the asset types, attributes and dependencies. A domain-specific variability model editor is then automatically provided for a defined meta-model. Modelers and end users can "enact" the variability models meaning that the tools allow users to take decisions based on the models and are capable of determining the required assets of a product. The tool also offers a number of extension points. This allows organizations to easily add new capabilities like model verification tools or generators by exploiting the tool's plug-in architecture.

3.1 Defining meta-models

The core meta-model from Fig. 1 can be refined for a specific domain using DOPLER's meta-model editor (cf. Fig. 4). The base type Asset provides the three default attributes Name, Description, and IncludedIf. Further attributes can be added by the

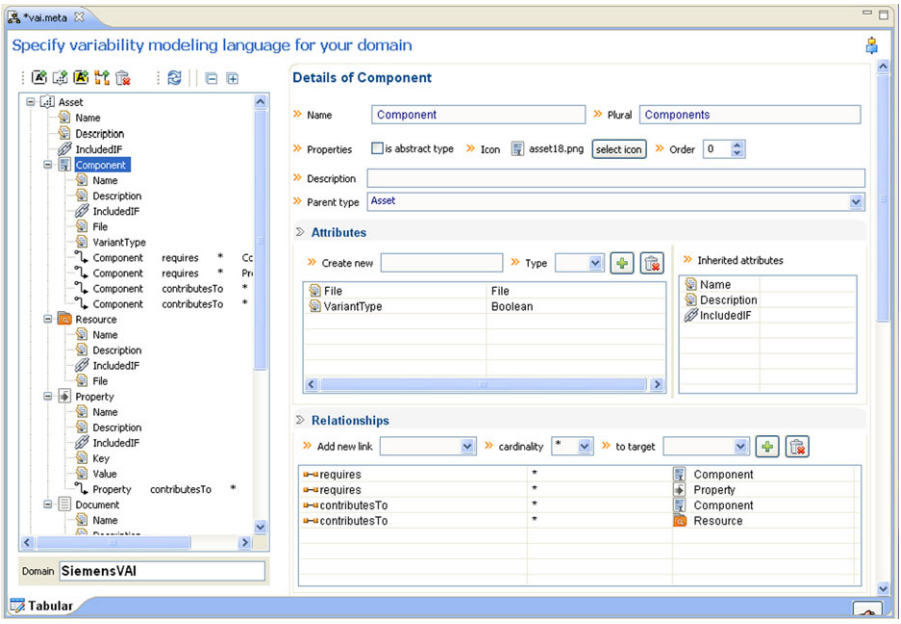

Fig. 4 DOPLER meta-model editor

modeler to take into account domain-specific characteristics. For instance, a modeler needs to define a File attribute for the asset types Component and Resource, as well as key and value attributes for the asset type Property to prepare the tool for the model shown in Fig. 3. DOPLER currently implements the default attribute types Boolean, Expression, File, List, Number, String, Paragraph, and URL. The tool currently implements eight basic types of relationships between assets. We support the generic types parent, child, inclusion, exclusion, implementation, abstraction, predecessor, and successor. When defining a new relationship in the meta-model (e.g., Component requires Component) the modelers select one of these pre-defined types to determine its semantics. For example, there may be several instances of the generic parent relationship such as *contained in*, *contributes to*, *is part of*, *is constituent of* in a DoplerVML meta-model. The list of assets to be included in a derived product is determined based on the types of links in the asset model.

The meta-model editor in Fig. 4 depicts a new asset type Component defined as the subtype of the default type Asset. It inherits the default attributes Name, Description, IncludedIf and defines the two new attributes File and VariantType. One can also see the different relationships between the type Component and other asset types, e.g., *requires* * Property and *contributesTo* * Resource.

3.2 Editing decisions and assets

DOPLER's variability model editor allows creating variability models conforming to a specific meta-model. For instance, Fig. 5 shows the variability model editor that is automatically provided by the tool suite based on the meta-model depicted in Fig. 4.

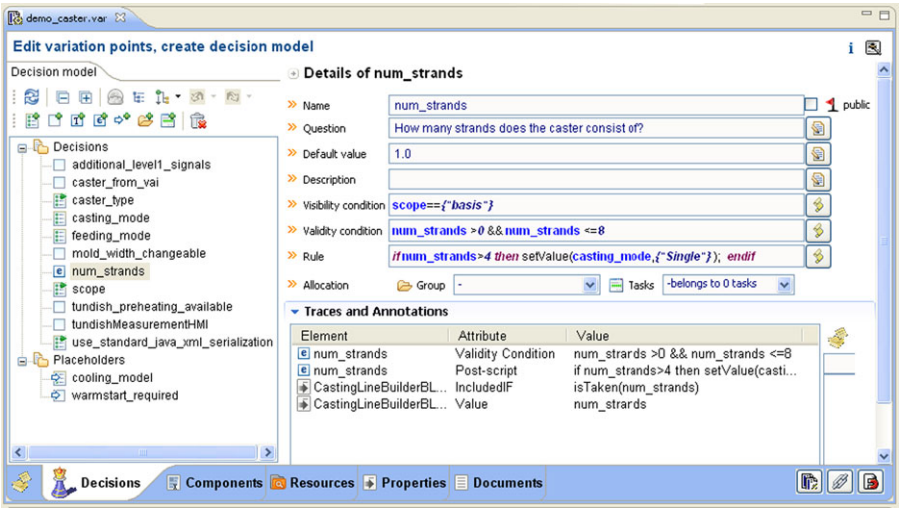

Fig. 5 DOPLER variability model editor

The editor follows the concept of master-tree (e.g., elements shown in a tree comparable to the file system visualization in the Windows Explorer) and elements-details (for a selected element, details can be edited).

The Decision panel allows defining the decision model. The tool supports four basic types of decisions (Dhungana et al. 2010b): Boolean decisions are used to represent yes/no questions. Besides the states TRUE and FALSE the state undefined is used to distinguish between the answers "yes/no" and "not yet decided". An example is the UI decision shown in Fig. 2. Number decisions are used mostly for parameter values, where the user decides on a numerical value. An example is the NumStrands decision shown in Fig. 2. String decisions are frequently needed as input to domain-specific generators. Enumeration decisions are used whenever different alternatives for one variation point need to be modeled. An example is the FeedingMode decision shown in Fig. 2. The Decision panel allows defining the decision attributes needed by the ConfigurationWizard tool (see Sect. 3.4) to communicate decisions to the end user, i.e., the Question asked to the user and a Description used to clarify the meaning of a decision.

The variability model editor also provides a separate panel for each asset type defined in the meta-model to define assets, their attributes, and relationships. For instance, in Fig. 5 panels are provided for the asset types Component, Resource, Property, and Document.

3.3 Support for defining dependencies

Decision attributes such as visibility condition, validity condition, and decision effects rely on a formal language. Our language shows high syntactic resemblance to Java and supports standard Boolean and arithmetic operators to build expressions. It provides several actions to query the value of decisions and build more complex

expressions. These actions can be used to specify logical dependencies among decisions. The language provides basic actions for querying and manipulating the values of decisions: setValue(d, p) assigns the value p to decision d. isTaken(d) is used to query whether a decision has already been taken by the user or whether its value is still undefined. reset(d) is used to retract a taken decision. Retracting a decision also resets all its implications.

Our current implementation is based on JBoss Drools,[2] an open-source, object-oriented production rule engine. JBoss has become a popular business logic framework, used by Java developers to create complex rule-based applications. JBoss Drools is a forward chaining rule engine with the knowledge encoded in IF-THEN rules. In DOPLER the rules are the decision effects modeled as decision attributes and the facts are the decisions taken by the user. The system examines all the rule conditions and determines a subset of the rules whose conditions are satisfied based on the working memory. The Drools pattern matching is based on the Rete (Forgy and Shepard 1987) algorithm, which evaluates a declarative predicate against a changing set of rules in real time. When a rule is fired, any actions specified in its THEN clause are carried out. These actions can modify the working memory, the rule-base itself, or do just about anything else the system programmer decides to include. This loop of firing rules and performing actions continues until one of two conditions is met: there are no more rules whose conditions are satisfied or a rule is fired whose action specifies the program should terminate. Figure 5 depicts the editor for defining the dependencies among decisions. For this purpose, we have defined a simple language on top of JBoss Drools allowing a user to define the basic functions explained above. All expressions in DoplerVML are translated into the corresponding representation in Drools.

3.4 Enacting variability models

It is important to enact and test decision models during the modeling process. This is achieved by the *ConfigurationWizard* tool which presents the decisions defined in a variability model (see Fig. 6). The order of taking decisions is partly specified by the visibility conditions. Based on the answers given by the user, the set of other available or relevant questions is dynamically calculated and presented to the user. The tool has been developed with a focus on usability for end users (Rabiser 2008; Rabiser et al. 2007a, 2007b) to support product derivation. It triggers domain-specific generators to create products based on the concrete values of the decisions. The ConfigurationWizard hides the complexity of variability models by presenting only the effects of the rules which are executed in the background.

4 Case study design and planning

Conducting empirical research in PLE is difficult (Sinnema and Deelstra 2007) as companies are typically reluctant to provide access to data about their product lines.

[2] http://www.jboss.com/products/rules/.

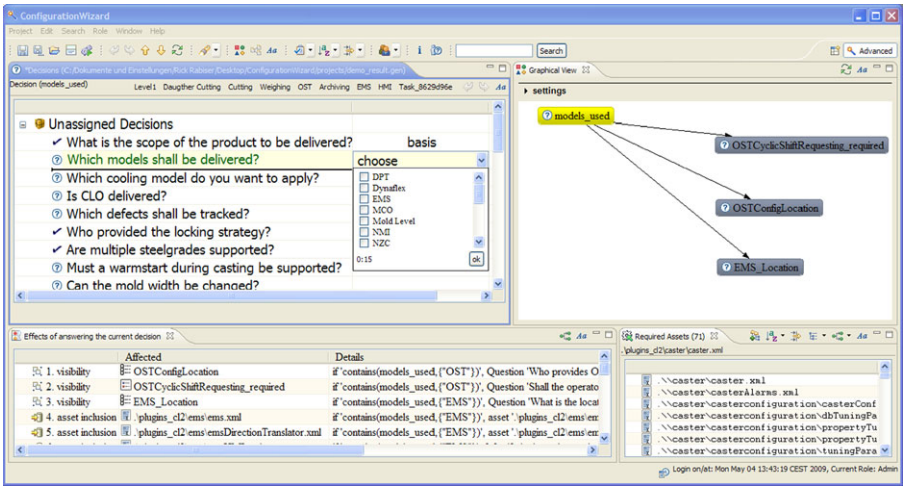

Fig. 6 The DOPLER ConfigurationWizard lets users interactively take decisions to derive and configure products based on DoplerVML models (Grünbacher et al. 2009)

Controlled experiments are often infeasible in PLE due to the long time span and the lack of data points caused by highly uncontrollable conditions and circumstances. For instance, product line models need to be observed over a long time period to unveil their real benefits. This is infeasible in many practical settings including the collaborations with our industry partners.

The overall goal of the evaluation was to demonstrate the flexibility of the tool-supported DOPLER approach in different domains and for different product line technologies. Case studies are suitable to evaluate approaches that cannot be evaluated merely through analytic research methods. In particular, in our evaluation we wanted to involve modelers from different domains and with different background and experience. We wanted to understand how well our approach can be applied in different development environments. We thus report on a multiple case study (Yin 2003) about the replicated application of DOPLER in diverse fields. Each of the four cases is treated as a single case study and each case's conclusions are then used as information contributing to the whole study. We describe the case studies based on existing schemes of conducting and reporting case studies (Runeson and Höst 2009; Robson 2002).

The **objective** of the multiple case study has been *to evaluate the tool-supported DOPLER approach to variability modeling with respect to flexibility*. We investigated different variability implementation mechanisms in diverse domains and demonstrated the flexibility of our modeling language and tools. Following Runeson and Höst (2009) we classify our research as partly exploratory and partly descriptive, as we focus on finding out what is happening and then portray the situation of applying our approach and tools to change the current state of practice. Our conclusions are based on qualitative data and modelers' feedback collected in different domains. More specifically, we investigated two **research questions:**

Table 1 Summary of main characteristics of the four selected cases

	Case 1	Case 2	Case 3	Case 4
System	Continuous casting steel plant automation system	Industrial automation systems	Maintenance and setup system for steel plant automation	Customer relationship management software
Size	1.6 MLOC	200 Function Blocks	200 kLOC	890 kLOC
Product line implementation	Spring Framework, Java	IEC 61499 Function Blocks, Java/C++	Eclipse Plug-ins, Java	.NET Libraries, C#
Purpose of modeling	Automated derivation of customer-specific products	Automated runtime reconfiguration	Automated creation of customer-specific systems	Automated role- and task-based selection of features at runtime
Duration of case study	4 years	2.5 years	1.5 years	1.5 years

RQ1 *How is variability represented in the different domains?* Berger et al. (2010) have emphasized the need for reports about real-world experiences and challenges in representing and dealing with variability. Similarly, we wanted to get insights on variability implementation practices used in different domains to assess the flexibility of our approach. We chose different domains to cover a broad range of technologies and variability implementation practices.

RQ2 *Is DOPLER flexible enough to support variability modeling in the different domains?* Based on the identified specifics of implementing variability in different domains our aim was to involve domain experts in modeling the variability of the selected case study systems to understand if the approach is flexible enough from the perspective of modelers in different domains. The flexibility not only refers to the modeling language but also to the extensibility of the DOPLER tools (e.g., whether the interfaces provided are regarded as sufficient to meet the specific needs in different domains). We thus also explore the tool extensions required in the four environments and investigate how the DoplerVML modeling engine can be used as a backend in other tools using its API.

Although our research initially focused on a system of our industry partner Siemens VAI we managed to attract the attention of other companies and researchers. This allowed us to test and evaluate our tool-supported approach in different domains and contexts. The **selected cases** cover different types of product lines (e.g., automation software, software tools), domains (e.g., business software vs. industrial automation) and implementation techniques (e.g., design time vs. runtime binding). A summary of the main characteristics of the cases is presented in Table 1.

Case 1—Steel Plant Automation Software. This case study was carried out in cooperation with Siemens VAI Metals Technologies, the world leader in engineering and

plant-building for the iron, steel, and aluminum industries. The company has developed and maintains the CC-L2 (Continuous Casting Level 2) product line of steel plant continuous casting automation software. About 40 software engineers are involved in developing and maintaining this system. The size of the software is about 1.6 million lines of code (mainly Java). Our case study focuses on the variability of the CC-L2 software (Dhungana et al. 2010a) which is responsible for process supervision, optimization, material tracking, etc. Understanding variability is relevant as the steel plant automation software is frequently configured to match the needs of different customers.

Case 2—IEC 61499 Industrial Automation Systems. This case study was carried out in cooperation with FH Oberösterreich Research. We investigated the usefulness of our tools and techniques for modeling product lines in the domain of industrial automation systems (IAS) (Froschauer et al. 2008). Such systems are usually based on a distributed architecture consisting of multiple physically and/or logically distributed components. More and more IAS are based on the emerging standard IEC 61499 which provides a component-based framework for automation systems and standardizes the use of function blocks in distributed industrial process measurement and control systems. Our case study investigated the variability in function block based systems conforming to the standard IEC 61499. Variability is relevant when such industrial automation systems are reconfigured at runtime. We thus deal with runtime variability of installed function blocks that can be exploited to automate runtime reconfiguration.

Case 3—Eclipse-based Maintenance and Setup Tool (MSS). This case study was carried out in cooperation with Siemens VAI Metals Technologies (Grünbacher et al. 2009). The MSS supports developers and operation staff of Siemens VAI customers to fine-tune deployment parameters of the CC-L2 process automation system on site. Some parts of MSS are also embedded in IDEs used by Siemens VAI developers. For instance, MSS provides graphical editors to maintain configuration files. MSS comprises 100 Eclipse plug-ins organized into 20 Eclipse features that can be combined flexibly and customized through diverse parameters. The size of the code base is about 200 kLOC. MSS is adapted to 20+ customer environments per year. While the CC-L2 software in case 1 is the actual system automating the industrial process, the MSS is configured and deployed as a separate but related product to customers operating industrial plants.

Case 4—.NET-based Business Software. This case study was carried out in cooperation with BMD Systemhaus GmbH, a medium-sized company offering enterprise software products to 18.400 customers and 45.000 active users mainly in Austria, Germany, and Hungary. BMD Software is a comprehensive suite of enterprise applications for customer relationship management, accounting, payroll, enterprise resource planning, as well as production planning and control. BMD's target market is diversified, ranging from small tax counselors to medium-sized auditing firms or large corporations. Customized products are an essential part of BMD's marketing strategy to address the needs of those markets. Our case study investigates the variability of distinct user-visible features of the business applications represented by plug-ins (Rabiser et al. 2009). Variability is relevant in such systems as the business processes like accounting, customer relationship management, or production

Table 2 Data collection methods and their use in the case studies

Case	Unstructured interviews	Workshops	Tutorials and joint modeling sessions	Observations and think aloud protocols	Archival data	Questionnaires
Case 1: Steel Plant Automation Software	×	×	×	×	×	×
Case 2: IEC 61499 Industrial Automation Systems	×		×	×	×	×
Case 3: Eclipse-based Maintenance and Setup Tool (MSS)	×		×	×	×	×
Case 4: .NET-based Business Software	×		×		×	×

planning require user interfaces specific to the company's needs and roles of the users.

4.1 Data collection methods and sources

The data collected in the case studies is primarily qualitative in nature. We used several data sources to limit the effects of interpreting a single data source only. For example, we collected data with the engineers and modelers in informal interviews and using questionnaires. We also analyzed code and documents to confirm their statements. Table 2 shows an overview of the data collection methods used in the four studies. We could not use the same data collection methods uniformly for all case studies due to the different roles of modelers involved and the availability of data sources.

Unstructured Interviews. We interacted with developers in unstructured interviews to understand the variability of the systems and the implementation techniques used. In particular we addressed the following questions: How do you perceive the variability of the system? How is variability reflected in the software system? How is variability implemented? Which approaches and tools are currently used to manage variability in the system?

Workshops. We conducted several workshops to understand the variability in different domains and involved experts from the respective areas. We followed a collaborative process to structure the workshops and to elicit variability from the involved stakeholders (Rabiser et al. 2008).

Tutorials and joint modeling sessions. It was essential to train and assist the engineers who used our tools for variability modeling. We assisted them in using the tools in joint modeling sessions and provided tutorials on decision-oriented variability modeling.

Observations and think aloud protocols. We observed selected engineers customizing a product for a customer to understand how the engineers currently deal with variability in their system. While they were using their tools, we asked them to speak out what they intended to do. In particular, we asked them to describe loudly the configuration steps necessary to realize a specific customer requirement from the specification. Typically, engineers made multiple manual changes in different artifacts and at different levels of granularity which helped us to understand the variability mechanisms of the systems. We documented the engineers' intentions and the actions they performed to later analyze the tools' capabilities and limitations.
Archival data. Analyzing existing data was one of the major sources of information in our case studies. We analyzed the software architecture, requirements documents, technical specifications, and data collected during workshops (flip charts, notes, audio recordings, and meeting minutes). As understanding variability was often only possible at a very detailed technical level (e.g., finding different implementations of the same interface) the help of domain experts was sometimes necessary. Furthermore, we analyzed the meta-models and models which were created in the different studies to analyze the level of granularity chosen by the modelers for variability modeling. This was an important source of information to refine and improve our approach.
Questionnaires. We used questionnaires post-hoc to get qualitative feedback about DOPLER from the modelers of the different case studies. More specifically, the questionnaires contained the following questions: How do you assess the overall suitability of DOPLER for modeling the variability of your system? What made it easy to model the variability of your system using DOPLER? What made it difficult to model the variability of your system using DOPLER? What are specific aspects you were unable to model using DOPLER?

4.2 Case study phases

Each case study was carried out in four stages:

I *Variability Analysis.* We began with an analysis of the domain together with the experts from the different domains. More specifically, we wanted to understand how variability is managed in the different domains. This included a short description of the variability implementation practices used by engineers.
II *Refinement of meta-model.* Before using DoplerVML for modeling variability the modelers created a domain-specific meta-model (cf. Sect. 3) defining the asset types and their attributes and dependencies based on the knowledge gathered in phase I.
III *Tool customization.* Together with the domain experts the researchers developed tool extensions to deal with the specifics of the different domains, e.g., to automate the analysis of legacy code or to implement consistency checking in models and from models to code. This was done in close collaboration, sometimes even using pair programming techniques.
IV *Variability modeling.* The domain experts developed variability models in individual sessions and joint modeling workshops guided by the researchers. The resulting models were then tested in product derivation and/or used to automate

runtime reconfiguration. While the variability analysis step focused on finding where and how variations occur in concrete artifacts the variability modeling step explored the representation of the variations in the product line models.

4.3 Data analysis and reporting

We analyzed the data collected from different sources together with the engineers from the different case studies. Such an approach helps us to reduce biased conclusions. We also analyzed the appreciations, suggestions for improvement, and other comments from the engineers because the chain of evidence to derive conclusions about the answers to our research questions is hidden in such statements. Because of the primarily qualitative nature of the data collected we do not present any statistical analyses. However, we provide details about the size and complexity of the developed models.

For *variability analysis* we give examples of the data collected and the process of analyzing the data. For the *refinement of the meta-model* we describe the domain-specific results of the variability analysis process. Regarding *tool customization* we present examples of tool extensions developed to deal with the specifics of the different domains. For *variability modeling* we describe characteristics of the models and their use (e.g., for generating configurations) and report size metrics like numbers of decisions and assets modeled.

5 Case study results

We report results from the four case studies following the stages I–IV described above.

5.1 Case 1: Steel Plant Automation Software (CC-L2)

CC-L2 is an industrial automation system for process supervision, optimization, and material tracking of the continuous casting process in steel plants. The modelers involved in the case study were lead software architects from the CC-L2 platform team.

5.1.1 Variability analysis

In CC-L2 variability is managed at different levels of abstraction. Fine grained configuration is handled using configuration parameters. For instance, this approach is used for changing colors, UI appearance, or for defining parameters like speed, amount, and formats. Similarly, different computational models and optimization components can be parameterized. The next level of variability deals with components which can either be included or excluded when composing a customer-specific system. Another level of variability is introduced by combining related components into groups that run on Java virtual machines. A customer-specific CC-L2 software system is built using the selected components (running in logical groups of processes) and (newly developed) customer-specific extensions. Within this context our case study focused on modeling component-level variability, associated configuration parameters and process-level variability.

5.1.2 Refinement of the meta-model

We identified different types of core assets in various workshops with engineers and sales experts of Siemens VAI and refined the DOPLER core meta-model according to their needs: Components represent Spring XML files which represent software components realized by a set of Java Beans. It turned out to be sufficient to only model the components and their dependencies but to exclude the Java beans as they were not needed in the model for automating component-level composition. Properties represent configuration parameters for components and can range from simple properties to lists, maps, and hash tables. Properties are treated as pairs defined in the two additional attributes key and value. The value attribute is an expression. This means that arbitrary decisions can be used (and combined) for setting the value of properties. Resources represent legacy hard- or software elements of the CC-L2 system. Resources are also used to model process level variability by modeling the lists of components belonging to a particular process.

The meta-model defines several dependencies: the functional dependency requires allows to express if a software component relies on one or more other components to function properly. Information about the deployment structure of the system is modeled using the relationship contributesTo. A simple example is a component contributing to a sub-system (i.e., modeled as a Resource asset) it belongs to. Both relationships define which related assets to include in a derived product.

5.1.3 Tool customization

We built several extensions to DOPLER during this case study which used DOPLER's API to initialize, manipulate, or check variability models. For example, we created a Spring Importer for mining the variability in existing component descriptions files (Dhungana et al. 2010a). The importer parses Spring XML files describing components and their dependencies and creates initial asset models. A second tool extension checks the consistency of variability models with the underlying Spring files (Vierhauser et al. 2010). The tool uses existing models and artifacts to find inconsistencies: whenever models are changed, existing architectural elements are used as a reference for comparison. Whenever architectural elements are changed, the existing models serve as a lookup table for establishing consistency. For example, whenever a new variant is introduced by changing the variability model, the tool ensures that there exists an artifact with the same name and structure (together with dependencies to other artifacts). Similarly, when a new component is added to the architecture, the tool automatically looks for its existence in the variability models.

5.1.4 Variability modeling

The lead architect created models for several subsystems of the CC-L2 software. The developed models vary in size and complexity. Overall, 462 components, 121 configuration properties, and 95 decisions were defined. These models are currently utilized and extended in a pilot study at Siemens VAI. Using the Spring Importer extension allowed to automatically create initial asset models and revealing some of the

technical variability of the CC-L2 system automatically in a bottom-up manner. For instance, the tool recognizes variants of Spring components based on the analysis of implemented interfaces. Based on these initial models the modelers manually added additional information required for deploying the assets.

The decision model required for instantiating CC-L2 products was populated from several sources: some decisions were automatically generated by the Spring Importer. More decisions were found following a top-down approach through brainstorming sessions in moderated workshops and by analyzing specifications and contracts of earlier projects (Rabiser et al. 2008). We experienced that such workshops result in rather high-level decisions which needed to be refined later (e.g., will the VAI-Q Quality Control system be part of the final system?). Further decisions were then modeled by individual developers and engineers at Siemens VAI. The identification of decisions by individual stakeholders was done at the level of subsystem (e.g., a decision related to the secondary cooling system Dynacs "Is adjusting the nozzle spray width controlled by the cooling model?").

Figure 7 depicts an excerpt of a variability model for the CC-L2 system at Siemens VAI. It consists of four decisions of three types (boolean, number, and user-defined enumeration). Two types of dependencies among the decisions are depicted: the decision TT3D is visible to the user only if the question about UI is answered with yes (TRUE); one of the options from the decision FeedingMode is deactivated whenever the caster has more than 3 strands (decision NumStrands). The figure also depicts 6 assets of three different types (Component, Property and Resource). The assets have dependencies among each other, e.g., the component Caster *requires* two other assets, *is required by* two assets and *contributes to* another asset. The assets are linked to the decisions through the expressions in their inclusion conditions, which refer to different decisions, e.g., the component CastHMI is included, if the decision UI is set to TRUE.

5.2 Case 2: Variability of IEC 61499 Industrial Automation Systems

This case study investigates the variability of industrial automation systems (IAS) based on function block networks. Understanding variability is highly relevant in this domain as such systems need to be frequently reconfigured at runtime. An IAS domain expert developed variability models for an energy management system and a bottle sorting plant.

5.2.1 Variability analysis

The most significant difference between IAS and common software products is their distributed and hardware-related nature. IAS consist of numerous complex sensor units cooperating with actuators to perform measurement and control tasks. Developing IAS based on IEC 61499 means defining function block networks that define the different dependencies among function blocks. Creating applications means introducing new function block types and appropriately wiring their function block instances. An automation application consists of one or multiple function block networks with interconnected function block instances. At runtime different function blocks can be

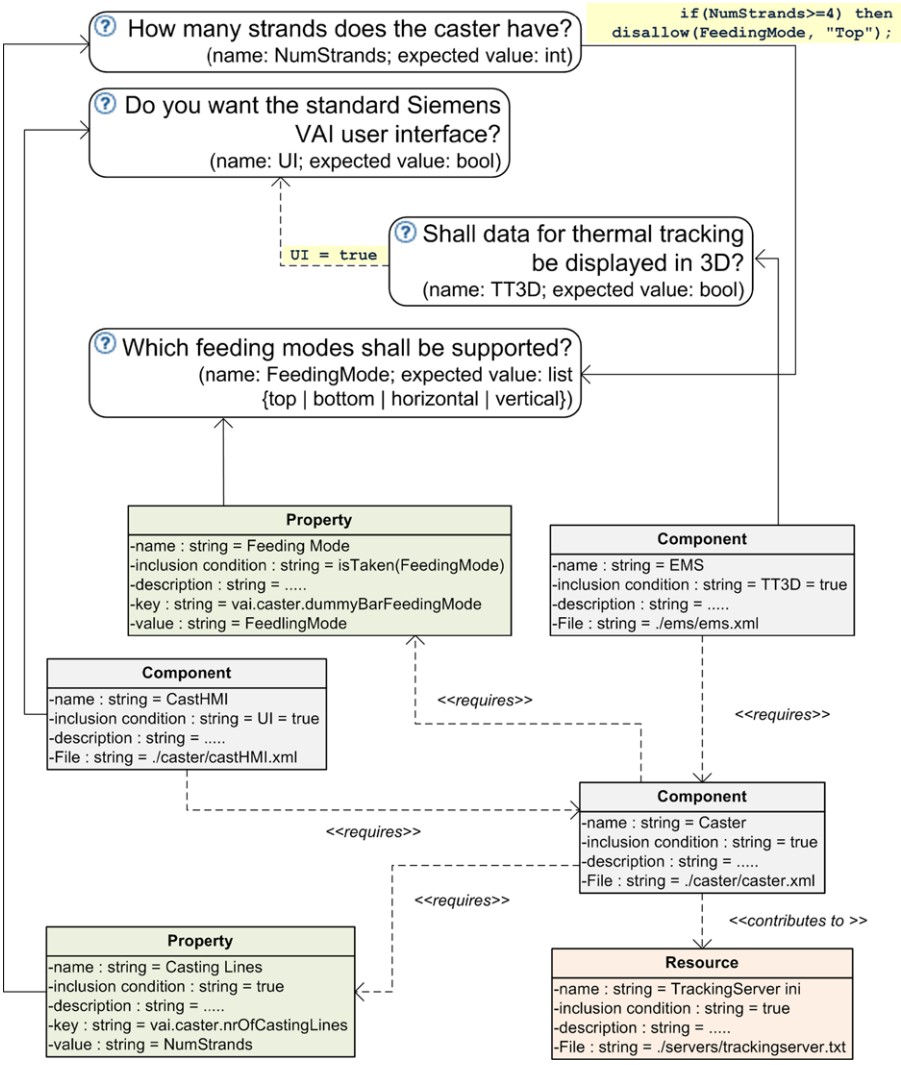

Fig. 7 A partial variability model of the CC-L2 system at Siemens VAI

instantiated and wired as required. This is where variability models help—they can be used to automate the manual, error-prone approach of instantiating and (re-)wiring the function blocks.

Hardware variability is reflected in the set of available hardware components. Depending on the devices available (e.g., conveyor, robot-grippers, sensors, etc.) and their specific attributes (e.g., width of conveyor, maximum span of robot grippers, types and numbers of sensors, etc.) an IAS can produce a wide range of products. The products that need to be manufactured determine the hardware devices used to build the manufacturing system. In this environment decisions represent the characteris-

tics of the products. There can also exist complex dependencies among the hardware components, for example, the speed of the conveyor must match the capabilities of the robot. The variability of the hardware has a strong impact on *software variability*, which is reflected by the set of software controllers used to drive the hardware components. The software is determined by the available hardware, however, to drive the same set of selected hardware components, one can deploy different variants of the controller software. The case study concentrated on modeling the software variability of IAS.

5.2.2 Refinement of the meta-model

The modeler created an IAS-specific meta-model which has been used to define features and devices needed in specific application domains in automation and control including transporting and sorting systems. The requirement to reconfigure IAS at runtime means that our models need to differentiate between design-time model elements and runtime instances. This allows an engineer to generate an executable application by taking decisions that automatically lead to the selection of applications at runtime (Froschauer et al. 2008, 2009).

DoplerVML does not support multiple instances of the same asset. In this case study we thus needed support for creating asset instances by updating the meta-model accordingly. *Design-time elements* capture basic knowledge about the target application domain and specific constraints and can be compared to elements in domain-specific meta-models (Dhungana et al. 2007a). The design-time elements reflect the domain component instances of previously defined domain component types (e.g., MovePart as a domain-level representation of components for moving parts in IAS. *Runtime elements* reflect the desired deployment platform and available runtime component types, which define the execution environment for runtime components (e.g., FBMoveSlow, FBMoveFast as a runtime instance of MovePart). Using these decisions the engineer can answer questions, such as "How fast would you like to move parts?", and the required components are included into the deployed application at runtime.

5.2.3 Tool customization

In this case study DOPLER acts as a variability modeling component in a larger tool suite called ControlKing. ControlKing is based on the open source IEC 61499 framework 4DIAC[3] and integrates DOPLER's variability modeling capabilities with 4DIAC's capabilities for IEC 61499.

ControlKing manages the *domain variability model* specifying the execution infrastructure for IAS and the types of assets to be executed on the platform. It also manages the *runtime variability model* defining function block instances of running IEC 61499 applications. Variability models created with ControlKing are manipulated at runtime and updated using DOPLER's model API. The model execution

[3] http://www.fordiac.org.

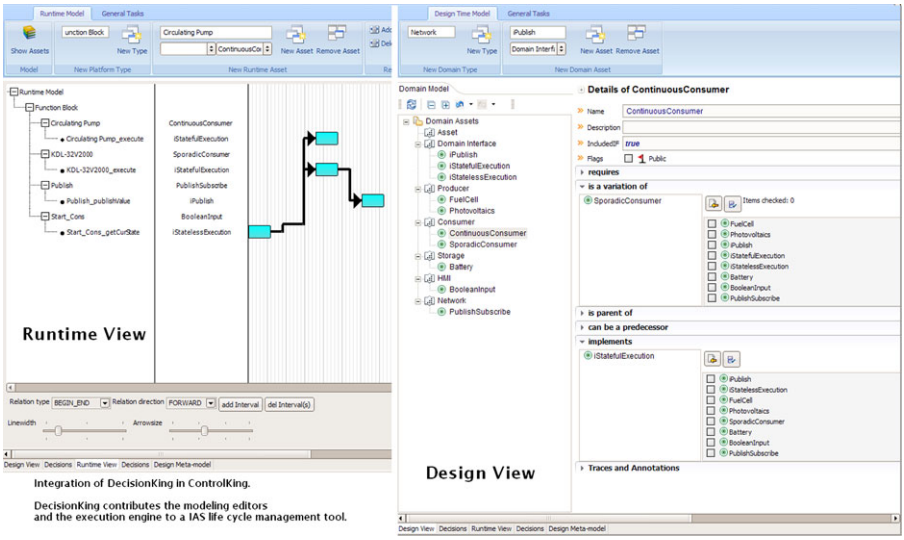

Fig. 8 ControlKing: a tool for managing the life-cycle of IAS components, showing the variability modeling editor components contributed by DOPLER (part of the design view depicted in the *right half* of the figure)

engine of DOPLER is an integral part of ControlKing and enables end-users to reconfigure IAS applications by taking decisions as they emerge at runtime. Figure 8 depicts the design-time and runtime variability model editors in ControlKing. The design time variability model editor is contributed by DOPLER which demonstrates its flexibility for being used as an off-the-shelf variability modeling component.

5.2.4 Variability modeling

The domain expert developed variability models for two applications, an energy management system and a bottle sorting plant. Both systems are implemented using IEC 61499 function blocks.

The main goal of the energy management application is to manage and control the distribution of electrical energy from producers to consumers following specific rules, such as priorities of consumers or environmental constraints for producers. The energy management application on the system layer was modeled by defining domain-level components, runtime components (firmware and user components) and decisions. The resulting domain variability model contains 9 design component types (e.g., Energy Producer, Energy Consumer), 14 design components (e.g., Battery, Wind Turbine), 3 runtime component types (e.g., ResourceInstance), and 23 components (e.g., Honda4kw, Banner90Ah). The application example demonstrates the feasibility of generating applications using the information from the runtime variability model. This is a big step forward as so far the wiring of components was possible only manually at the level of function blocks.

The second example is a bottle sorting plant, a simple application from the domain of discrete transportation systems. The system consists of a button-based user interface, two conveyor belts, a bottle selector, an infrared gate, a pick & place unit, a color sensor and a switchblade. Although the bottle sorting plant consists of only eight components, several variations exist regarding the placement of each component and their order of execution, such as the position of the color sensor, which may be mounted either at the pick-up position or at the drop-down position of the pick&place unit. A component variant may also arise from the use of different types of color sensors. The resulting domain variability model contains 8 design component types (e.g., Ultra Sonic Sensor), 19 design components (e.g., BottleDetector, Conveyor), 3 runtime component types (e.g., Function Block instance), and 22 components (e.g., BottleSensor 1, MoveBottle2).

ControlKing was used to import the existing applications and to add decisions on top of the function block networks. This was mostly done manually but could be supported with tools that suggest new decisions whenever two or more implementations of certain interfaces are detected. Variability models were used to keep track of running applications. This was done by refreshing the model after each change in the application at runtime. Whenever the engineer planned to make new changes, she no longer had to adjust the function block network manually—this was done automatically by ControlKing. The approach turned out to be flexible enough to model manufacturing processes and/or complex machines with cooperating devices as also pointed out in the post-hoc questionnaire.

Figure 9 depicts an excerpt of a variability model of a bottle sorting automation system based on IEC 61499 function blocks. It consists of three decisions (Move, Speed, Direction). The decisions Speed and Direction become visible to the user after the decision Move is answered with TRUE. The model excerpt also displays 5 assets of type *FunctionBlockType* and *FunctionBlockInstance*. The relationship *is instance of* between the two asset types is of type *inclusion*. This means that whenever a function block instance is part of the final product the corresponding function block type will also automatically be included.

5.3 Case 3: Eclipse-based Maintenance and Setup Tool (MSS)

The Eclipse-based MSS tool of Siemens VAI Metals Technologies supports developers and operation staff of Siemens VAI customers to fine-tune deployment parameters of the CC-L2 process automation system on site (Grünbacher et al. 2009). A lead software architect developed a variability model of the MSS.

5.3.1 Variability analysis

The MSS is based on the Eclipse platform and variability is achieved through a plug-in architecture. An Eclipse plug-in represents a unit of functionality which can be developed and deployed separately. Plug-ins can declare named extension points and an arbitrary number of extensions to extension points declared in other plug-ins. An extension point can provide an API that is implemented by the plug-ins contributing to the extension point. During startup the Eclipse Platform Runtime discovers

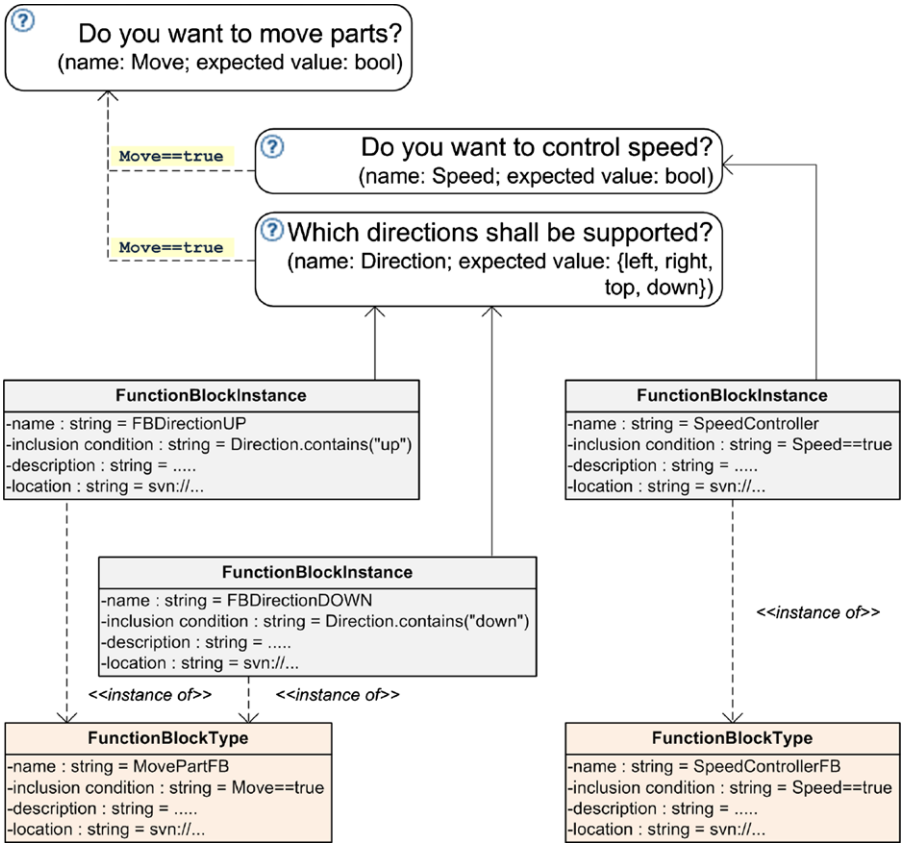

Fig. 9 A partial variability model of a bottle sorting automation system based on IEC 61499 function blocks

the available plug-ins and builds an in-memory plug-in registry. This allows adding, replacing, or deleting plug-ins even at runtime.

Understanding the variability of the MSS is critical, as the system needs to be customized and deployed 20+ times per year. The MSS comprises 100 Eclipse plug-ins (which can contain hundreds of Java classes) organized into 20 Eclipse features that can be combined flexibly and customized through diverse parameters. The customization needs to take into account different usage scenarios such as debugging, analyses, tests, and simulations. Furthermore, the MSS is frequently customized to different roles: certain MSS editions are used by domain experts such as metallurgists while other editions are mainly targeted at software engineers. Manual product derivation of the MSS at Siemens VAI can be error-prone and time-consuming. Only few developers can perform this task manually which makes short development cycles difficult. For instance, during plant startup at a customer's site, the MSS installations must be updated regularly and their integrity must be ensured during re-configuration. This is especially hard under time pressure or when remote communication with headquarters is impossible.

5.3.2 Meta-model refinement

The core meta-model was adapted so that Eclipse plug-in architectures could be modeled. The following concrete asset types were defined: An *Eclipse Feature* is used to model predefined packages of plug-ins for deployment via relations to the asset type Plug-in. License, copyright, version, and location are key attributes of Eclipse Features. A *Plug-in* represents Eclipse plug-ins—the building blocks of the software tools. Attributes are the plug-in id, version, and location. Plug-ins can require each other and can provide and implement Extension Points. An *Extension Point* represents the extensibility provided or implemented by plug-ins. A *Setting* represents a simple key-value pair. The value attribute takes the value of a decision as taken by the user. The file containing a setting can also be specified. A Setting can contribute to a plug-in meaning that the parameter is used to change the behavior of the plug-in, e.g., the layout of editors. A *Deployment Parameter* allows defining further parameters needed for the deployment of a plug-in, i.e., the size of the plug-in archive file and a list of native libraries that may be required for deployment.

5.3.3 Tool customization

No special tool extensions were required for modeling the variability of the MSS. We however developed a new application generator which can build an Eclipse update site based on the required constituent plug-ins of the MSS selected by taking decisions in product derivation (Grünbacher et al. 2009).

5.3.4 Variability modeling

In a first iteration the domain expert modeled the assets representing the MSS's architecture and captured 20 Eclipse features and 100 plug-ins together with their dependencies. In a second iteration he built a decision model expressing the variability for later deriving different variants of the MSS. Three top-level decisions allow a user to choose a certain edition (developer or metallurgist edition), to decide whether the quality management system *VAIQ* should be included, and to decide whether the MSS should also be used to supervise or to customize a CC-L2 system. Further decisions allow customizing additional tools such as the *Speedexpert* and *DYNACS 3D* by supporting domain experts to adapt Siemens VAI's mathematical plant models. In the third iteration the modeler defined inclusion conditions to link the Eclipse Features and Plug-ins with the decisions. Depending on the selected edition different features and plug-ins are selected that constitute the developer or metallurgist edition of the MSS. The quickest way to deploy the tool suite is to just select the edition. Only one decision needs to be answered in this case ("MSS Edition?"). Further decisions select additional features/plug-ins.

Figure 10 depicts an excerpt of a variability model of the maintenance and setup system at Siemens VAI. Edition is a top level decision. The other two decisions SpeedExpert and VAIQ become visible to the user after selecting an edition.

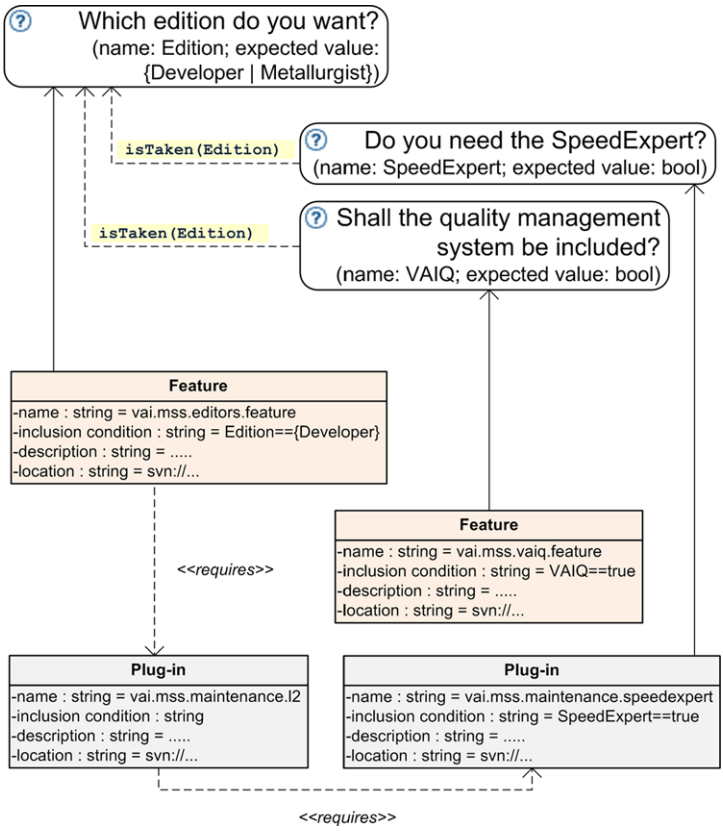

Fig. 10 A partial variability model of the MSS system at Siemens VAI

5.4 Case 4: Variability of .NET Plug-in-based Business Software at BMD

This case study was carried out in cooperation with BMD Systemhaus GmbH for their enterprise application for customer relationship management, accounting, payroll, enterprise resource planning, as well as production planning and control. At the supplier level, BMD has structured the software into seven solutions which can be individually licensed and composed into five main products covering major markets. For example, BMD-Consult is optimized for chartered accountants while BMD-Commerce is targeted at corporations. BMD's software has a total size of 4 million LOC. In our case study we focus on the BMDCRM subsystem which has a total size of about 890 KLOC (Rabiser et al. 2009). A researcher (other than the authors) collaborating with BMD was in charge of creating the variability model.

5.4.1 Variability analysis

BMD's Enterprise Resource Planning product supports product customization at different levels through different technically not fully related mechanisms. All binaries are shipped for each product regardless of the licensed features. An individual license

key is used to activate licensed features. Unlicensed features are blocked, i.e., corresponding widgets in the user interface are disabled or hidden. At the customer level, configuration is accomplished in a similar fashion through permissions. A customer can build individual feature subsets for different departments by revoking permissions for unneeded features. Features for which a user lacks privileges can thus be hidden. At the end-user level, the permission mechanism can also be applied to individual user accounts. A user account can be granted permissions to individual feature sets. However, since in practice this is typically done also by administrators, end-users have only limited ways to personalize their application.

As a preparatory step the monolithic legacy software had to be decomposed into a small core system and a set of pluggable extensions. Each extension was defined to contain a single user-visible feature which can be integrated with the core system using plug-in techniques. We identified artifacts such as source code and resources related to a feature to decompose features to the granularity of plug-ins. We then re-engineered the individual components to allow their use within the plug-in platform and developed a core application such that plug-ins can be integrated. The decomposition resulted in a plug-in solution comprising 20 specific plug-ins and 28 components for the core system.

5.4.2 Refinement of the meta-model

The modeler identified assets at three levels of granularity and defined the following asset types in the refined meta-model: *Plug-ins* represent single user-visible features that can be integrated individually into the application. *Packages* combine tightly related features that are commonly used together. *Solutions* combine packages into a solution. The following dependencies among the asset types were also modeled. A solution *contains* a package (for instance, the solution BMDCRM contains the packages Label Printing, Standard Letter, Organizer, and Docs). A package *contains* a plug-in (for example, the package Docs contains the plug-ins Archive, Scanning, Retrieval, and Import). A plug-in can *require* another plug-in (e.g., the plug-in Scanning providing document scanning features functionally requires the plug-in Archive providing document archiving features; this means that Archive can be used without Scanning but Scanning requires Archive).

5.4.3 Tool customization

We integrated DOPLER with Plux.NET, a platform which allows adaptation of .NET systems at runtime. In particular, we developed a Java to .NET connector to provide automated reconfiguration facilities at runtime (Wolfinger et al. 2008).

5.4.4 Variability modeling

A researcher modeled possible adaptations as customization decisions at different levels. In the model decisions at higher levels constitute abstractions of decisions at lower levels. The relations between assets and decisions are described as inclusion conditions that determine the set of required solutions, packages, and plug-ins to be

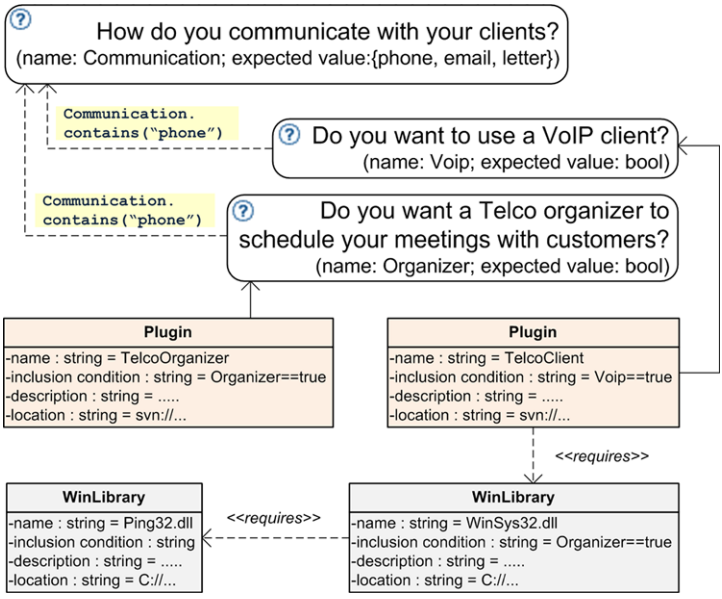

Fig. 11 A partial variability model of the CRM tool at BMD

deployed. Overall, for BMDCRM, the variability model developed in the feasibility study contains 7 decisions, 20 plug-ins, 4 packages, and 1 solution (cf. Sect. 5.4.2).

Figure 11 depicts an excerpt of a variability model of the CRM tool at BMD. It consists of three decisions (Communication, Voip, Organizer). The decisions Voip and Organizer are visible to the user if the value phone is selected for the decision Communication. The assets of type *Plug-in* are associated with the decisions, whereas the assets of type *WinLibrary* are included in the final product because of the *requires* relationships.

6 Discussion of results

The DOPLER meta-tool was used for modeling the variability of several product lines. In each of the four cases, domain experts analyzed the domain assets and developed a domain-specific meta-model based on our core meta-model (cf. Fig. 1). This process took between several weeks and several months, as it required a number of workshops and the analysis of existing documents. The modelers also developed domain-specific tool extensions (importers, consistency checkers, generators, etc.) to support the automated composition of components, initialization and maintenance of the models, or re-configuration of components at runtime.

Our approach and tools benefited significantly from being continuously evaluated by engineers modeling the systems. Users intensively experimented with the tools after they recognized their potential benefits. Their feedback helped us improving the tools and validating the modeling concepts. User acceptance was highly influenced by our ability to enact the models. It was regarded important by the users involved in

Table 3 Summary of results for research question 1

	Case 1	Case 2	Case 3	Case 4
Modeling granularity	Components based on Spring, Java properties	Components based on Function blocks	Components based on Eclipse Plugins	Components based on .NET Assemblies
Plattform, middleware	Spring, Java	4DIAC, IEC 61499	Java, Equinox	.NET
Variability mechanism	Selection and parameterization of components	Rewiring and runtime reconfiguration of function blocks	Selection and parameterization of plugins	Selection and on-the-fly activation of .NET libraries

modeling to immediately understand the purpose of the models for the configuration of the systems.

RQ1: How is variability represented in the different domains? Related to the first research question, we collected data during the variability analysis phase. The results summarized in Table 3 reflect the peculiarities in each domain such as technologies and variability mechanisms.

Variability is represented at different levels of granularity (e.g., replacing an entire component vs. changing the value of a single initialization parameter). A key challenge is to find the appropriate granularity of the assets during the analysis phase. This was however not always straightforward. Our tools allow choosing an arbitrary level of granularity. As a result some users sometimes tended to define a very detailed meta-model in the beginning before realizing that a fine-grained model was not necessary to achieve the purpose of generating configurations (and would have also increased the maintenance effort). Finding a trade-off between modeling details and the maintenance effort (Dhungana et al. 2010a) was only possible by discussing the modeling options with engineers and iteratively experimenting with smaller parts of the case study systems.

While in our case studies the granularity was typically rather high there were also cases of mixing different granularities in one model (e.g., components and properties in case 1). DoplerVML is flexible regarding the necessary granularity although the experiences learned in the case studies indicate that modeling the assets at a higher level of granularity is a success factor. Our approach also makes use of templates when fine-grained assets need to be added or removed from a base artifact as the result of the decisions. This is made possible by adding generators using the extensible architecture of the tool suite.

RQ2: Is DOPLER flexible enough to model the variability in the different domains? Related to the second question, we collected data during the meta-model adaptation, tool customization and modeling phases (cf. Table 4). In each case, we were able to adapt the tool as required by extensions and meta-modeling capabilities. The modelers successfully used the customized tool for variability modeling. The models were used for configuring products both at design time and run-time.

The case studies show that the flexibility of the tools and of the modeling language are success-critical to deal with specifics of different domains and develop-

Table 4 Summary of results for research question 2

	Case 1	Case 2	Case 3	Case 4
Meta-model refinements	3 asset types, 2 dependency types	2 asset types, 3 dependency types	5 asset types, 3 dependency types	1 asset type, 2 dependency types
Tool extensions	Spring Importer, Consistency checker, Configuration generator and simulator plug-in	ControlKing, (DOPLER is used as an engine)	Eclipse update site generator	Java to .NET connector
Model complexity	6 models, average size: 16 decisions and 97 assets	17 models, average size: 10 decisions and 65 assets	2 models, average size: 15 decisions and 40 assets	1 model, size: 16 decisions and 32 assets

ment contexts. We found that the flexibility of the tools was sufficient to adapt it to the domains in three of the cases. However, we also experienced difficulties: Our modeling approach was initially designed to model variability of static configurations of assets and we did not consider modeling runtime instances of assets and runtime constraints. In the IAS case study, we thus had to complement our approach with the ability to integrate a platform meta-model and a runtime asset model into a runtime variability model. This required a major effort including the development of a new tool front-end for end-users.

The numbers reported cannot directly be compared between case studies due to the different granularity of assets. Generally, 100 decisions is quite a large number due to the possibility to reuse and combine decisions and their values throughout models (e.g., in inclusion conditions of assets). The number of assets depends on what system is modeled. For example, in case of CC-L2, a component asset represents a Spring component description which describes a Java Bean. In case of BMDCRM, a plug-in asset can represent a number of classes collected in a .NET assembly.

As the four cases show users were able to extract variability manifested in many different kinds of artifacts (e.g., documents, software components, test cases, configuration parameters) and different mechanisms supported by different programming languages, architectural styles, design patterns, etc. The multiple case study provides evidence that DoplerVML is flexible enough to deal with the diverse implementation practices we encountered.

The utility of an extensible technique however not only lies in the extensibility but also in its reusable functionality. Our approach supports both properties of an extensible technique. The tool is extensible because new plug-ins can be easily added, e.g., domain-specific product generators, model builders and importers. The tool is however also reusable because the DoplerVML modeling engine can be used as a backend in other tools using its API. For example, it was used in case 2 as the backend engine of the IAS variability modeling tool ControlKing.

6.1 User feedback

Using questionnaires we collected qualitative feedback from modelers post-hoc about the DOPLER approach.

How do you assess the overall suitability of the DOPLER approach for modeling the variability of your system? We received positive feedback on this issue. Regarding the first question case 2 modeler reported that the meta-modeling features "... enabled easy modeling of different types of IAS" while at the same time arguing that the "modeler should have IAS process know-how to create good meta-models". Case 4 modeler was in charge of implementing a user interface with role-specific views. Depending on the role, the application is composed from a different set of plug-ins. The modeler reported that "... the variability model that relates decisions (roles) and assets (plug-ins) with inclusion conditions was a perfect match for our plug-in based architecture and role-specific interface."

What made it easy to model the variability of your system using DOPLER? The feedback of case 2 modeler was that "... industry's needs have been taken into account" when developing the approach. A positive experience of case 2 modeler was that "... the key advantage of DOPLER is to have a small set of easily understandable modeling elements" for creating different meta-models. Two modelers emphasize the importance of understanding the base of reusable assets. Case 3 modeler explained that with "... a good manufacturing process definition the modeling may be done straight forward." Similarly, case 4 modeler emphasized the need to structure the reusable elements before starting to model: "Our plug-in-based architecture where every feature is implemented by a separate plug-in component made it easy to model assets and decisions".

What made it difficult to model the variability of your system using DOPLER? Two modelers raised the issue of understanding the system before modeling. E.g., case 1 modeler said that "The difficulty lies in the complex system itself, understand it and map the system to a model. Tooling is then secondary". Similarly case 3 modeler reported that "The main advantage of having a reduced set of simple modeling elements sometimes makes it hard to get the big picture how to combine these elements to a good meta-model. This problem mainly arises if the modeler has not enough information about the IAS which shall be modeled". Case 3 modeler also stressed the need for an iterative modeling approach when saying that "... some meta-modeling elements or relationships may turn out to be inappropriate during the later IAS modeling, therefore the modeling process often requires more iterations". Another challenge was highlighted by case 4 modeler who said that "... The phrasing of the questions for decisions required serious reflection, because the questions must translate the features offered by the plug-ins into problem space language which is easily understandable by the user."

Please list specific aspects you were unable to model using DOPLER (if any). Case 1 modeler says that "DOPLER works great for component oriented software (component included or not included), but the approach has its problems with physical items that can occur several times." This issue was also raised in case study 2 when developing an asset instance model.

6.2 Further lessons learned

Carefully phrase customization questions. An interesting issue when using a decision-oriented approach to variability modeling is the definition of proper customization questions. Engineers tend to formulate such questions based on the available technical assets (e.g., software components). It is however more advisable to phrase the questions according to the tasks of end-users who do not need (and often do not want) to know technical details. It is thus preferable to use abstractions in the language of the users such that they can more easily understand the implications of decisions they take. For example, a user should not decide whether a particular component shall be included but should rather select capabilities needed for her tasks.

Text-based notations for decision models are sufficient. DOPLER does currently not offer a graphical representation (as e.g., feature-oriented or orthogonal modeling approaches do). However, when studying how variability is represented in the different cases we noticed that domain experts do typically use tables, spreadsheets, and other text-based notations to describe variability. Also, as the post-hoc feedback shows, none of the modelers raised the lack of a graphical notation as an issue. Modelers using the text-based editor did not express interest in graphical modeling support. The tree- and table-based representation in the tool was regarded as sufficient.

Automate model creation where possible. An important lesson learned is that automation is critical when creating models of large product lines as e.g., shown in case 1. The effort required upfront to develop such model generators is definitely worthwhile to accelerate model definition and maintenance.

6.3 Limitations

Product generators still need to be developed manually. The approach to create a high-level asset model of the solution space worked well in the four case studies. However, while DOPLER helps in determining the assets required in a derived product, the actual deployment is still done by manually written generators capable of creating a working configuration. For instance, the generators use templates to add or remove fine-grained assets from base artifacts. Although their development was pretty straightforward in the four case studies we have studied the structure of the generators with the goal of developing a domain-specific language that allows to generate product generators from a high-level specification.

Runtime binding issues. From an abstract point of view the representation of variability using decisions is not relevant to variability binding time. However, when defining models in case study 2 (Variability of IEC 61499 Applications) we discovered a number of challenges that could not be foreseen in the design-time binding cases. For instance, when dealing with runtime variability, one has to consider technological bridges between variability modeling tools and the application's runtime environment, the runtime instances of assets that can change the variability models on the fly, as well as scalability issues due to the large number of asset instances. We managed to address these problems after some changes to the API of our tools. However, to fully support dynamic adaptation more research is needed. For example, using DoplerVML models only allows to address the anticipated and foreseen changes to a system (which was sufficient in the case) but lacks support for unanticipated changes.

6.4 Threats to validity

As any empirical research, our multiple case study exhibits a number of threats to validity. A threat to *construct validity* is the potential bias caused by the systems selected for the multiple case study. However, our case study involves several large software systems developed by multiple people and with different implementation languages. We also chose cases from different domains that are representative for other systems in industry. There are also threats to *internal validity* meaning that the results might have been influenced by our treatment. For instance, the duration of the case studies was quite different which might have influenced the level of detail and granularity of variability modeling. Also, we had no influence on the selection of the subjects as they were nominated by our industry partners. However, they were all experienced software engineers and responsible for the systems they were modeling. There is also the risk of maturation as the processes within the different domains possibly changed during our case study. Also, the increasing maturity of our own approach during the case study setup played an important role in the outcome. Regarding *conclusion validity*, there is a threat that the results are not based on statistical relationships but rather on qualitative data. Clearly, a sample of four cases cannot prove the flexibility of our approach. However, as companies are typically reluctant to provide access to data about their product lines and because of the lack of data points in this field of research we believe that our multiple case provides adequate evidence given these constraints. With respect to *external validity* (can we generalize the results?) we selected four real-world, large systems to represent realistic application contexts. However, the different cases all apply a component-based development approach. This means that our approach might not work as well in other environments (e.g., monolithic systems). However, component-based architectures are typically seen as a prerequisite for moving towards product line engineering (Clements and Northrop 2001) and we thus believe that the selected systems are representative.

7 Related work

Our discussion of related research focuses on the areas of feature models, decision models, architecture description languages, meta-tools, and situational method engineering.

Feature modeling. Feature modeling is currently the most widely used approach for modeling variability. In general, a feature model captures user-visible characteristics and aspects of a product line, such as functional features of individual products as well as software quality attributes of both the product line and the individual products to provide an overview of a system's capabilities. Literature on product line engineering also shows that it is the most intensively researched method for variability modeling. Starting from FODA (Feature-Oriented Domain Analysis (Kang et al. 1990)), the feature-oriented view of product lines has already gone far beyond variability modeling and system documentation. Today numerous variants of feature-based variability modeling tools and techniques (Asikainen et al. 2006; Czarnecki et al. 2005, 2006;

Czarnecki and Pietroszek 2006) are available and several authors have proposed different formal interpretations of feature models (Batory 2005; Schobbens et al. 2007; Heymans et al. 2007; Schobbens et al. 2006).

The most important difference between feature models and DoplerVML models is the perspective from which the models are built. DoplerVML emphasizes product derivation by using decision models that specify the available customization options from the perspective of a user. This perspective leads to a number of distinguishing characteristics of our approach: (i) The strong end user perspective means that DoplerVML models allow defining configuration aspects that turned out to be important when working with domain experts. For instance, DoplerVML offers constructs such as visibility conditions, decision effects and inclusion conditions that go beyond the extensions proposed by expressive features models and that are needed to develop usable product derivation tools. When considering the data types used in our decision models it is tempting to compare DoplerVML with expressive feature models, that allow more than just boolean choices, e.g., work by (Czarnecki et al. 2006). However, our argument about the distinction between feature models and decision models is not grounded on data types but rather the semantics of the models. (ii) DoplerVML aims at a clearer separation of the problem space and the solution space compared to feature models that do not have this explicit distinction. The decision model thus can be seen as a "configuration interface" to a complex system while the asset model describes the technical realization of the product line at the level of granularity necessary for composing customized products. The use of domain-specific asset meta-models also eases the development of product generators that can take advantage of different artifact types. (iii) Finally, feature models are very suitable for getting an overview about a system's functionality during domain engineering. Due to their different purpose, DoplerVML models, however, do not necessarily describe all the available features. They focus on just the decisions that are needed to generate a customer-specific solutions. While features in a feature model can be defined as mandatory, a modeler in DoplerVML would need to set the inclusion conditions of mandatory assets to TRUE. There would be no decision visible to the customer in this case.

Decision models. Decision modeling in product lines was initially introduced as a part of the Synthesis Project by Campbell et al. (Consortium 1991; Campbell et al. 1990) where decisions were defined as "*actions which can be taken by application engineers to resolve the variations for a work product of a system in the domain*" (Campbell et al. 1990). Many other researchers like Forster et al. (2008), Schmid et al. (Schmid and John 2004), or Mansell et al. (Mansell and Sellier 2004) have been proposing different variations of decision models. Typically these approaches are defined rather informally. We thus believe that DoplerVML goes beyond these approaches due to its features for enacting models and the degree of tool support.

Architecture description languages. DoplerVML provides support to model the solution space via customizable asset models. This is related to the idea of architecture description languages (ADLs). ADLs are formal notations for describing software systems and lie at the conceptual intersection between requirements, programming, and modeling languages. While many general purpose modeling languages

usually focus on modeling the internal structure of components, ADLs usually describe the interplay of components. A number of ADLs have been proposed for modeling architectures, both within a particular domain and as general-purpose architecture modeling languages, e.g., Darwin (Magee et al. 1995), Aesop (Garlan et al. 1994). A systematic survey (Medvidovic and Taylor 2000) of architecture description languages reveals that most ADLs share a set of fundamental modeling constructs and concepts, including components, connectors, interfaces, and architectural configurations. ADLs have also been extended for modeling product line reference architectures. Examples of such approaches are Koala (van Ommering et al. 2000) and xADL (Dashofy et al. 2001). While ADLs aim at describing systems to allow further analyses and simulation the aim of DoplerVML is to describe systems only at the level of detail required for deriving customized products.

Meta-tools. Numerous tools have been proposed that support customization by generating domain-specific tools based on a specification. For example, MetaEdit+ (Tolvanen and Rossi 2003) is a tool for designing a modeling language, its concepts, rules, notations, and generators. Pounamu (Zhu et al. 2007) is a meta-tool for the specification and generation of multiple-view visual tools. The tool permits the rapid specification of visual notational elements, the underlying information model, visual editors, the relationships between notational and model elements, and the elements' behavior. IMP (Charles et al. 2007) is an IDE meta-tooling platform that aims to reduce the burden of IDE development in Eclipse. The approach supports the customization of IDE appearance and behavior and aims at reusing code during IDE development. Similar ideas for generating domain-specific tools were also proposed by Grundy et al. (2006). The authors present meta-tools capable of generating domain-specific visual language editors from high-level tool specifications. While DOPLER is similar to these meta-models its focus is more on variability. Consequently, decisions are a first-class citizen in the core DOPLER meta-model as opposed to other meta-tools.

Situational method engineering. Our approach is also related to the Situational Method Engineering (SME), an approach aiming for controlled flexibility, i.e., achieving a balance between rigid general-purpose methods and ad-hoc, flexible development (Harmsen and Brinkkemper 1995). Method Engineering is the discipline to study engineering techniques for constructing, assessing, evaluating and managing methods for developing Information Systems Development Methods. Situational method engineering adds variability to the discipline of method engineering, by enabling methods to be adaptable to concrete domains or application scenarios. SME is enabled by various tools and techniques, the use of meta-tools is a prominent approach in this context. Furthermore "method fragments", or "method chunks" are a part of SME research, as they enable building more complex, tailored methods based on small process steps (Henderson-Sellers et al. 2007). This can be compared to our work, as in our approach the primary elements of modeling variability are the same in each domain. They are, however, composed differently as required in the different application scenarios.

8 Summary and conclusions

In this paper, we presented the DoplerVML variability modeling approach that is based on decision models and emphasizes the product derivation perspective of product line engineering. The approach is supported by the DOPLER meta-tool for defining and executing variability models. DoplerVML allows developers to systematically describe the variability of arbitrary domain-specific assets and their dependencies. Instead of solving the variability modeling problem for one particular domain, we developed a meta-approach, which can be configured to the specifics of different domains as required.

Our research has been guided by an analysis of different industry partners' software systems. The need for a flexible variability modeling approach was confirmed when investigating how variability management is handled in the four cases with different programming languages, modeling notations, architectural styles, and implementation practices. It became evident that variability has to be understood at different levels (e.g., requirements, architecture, or implementation) and for diverse domain-specific artifacts.

The multiple case study presented in this paper demonstrated the need for flexibility in product line modeling tools. The use of a meta-tool helped us adapting our modeling approach by providing domain-specific meta-models and the use of a plug-in architecture enabled us to extend our tool suite to deal with domain-specific tools, processes and artifacts in use.

Acknowledgements This work has been supported by the Christian Doppler Forschungsgesellschaft, Austria and Siemens VAI Metals Technologies. We also want to thank BMD Systemhaus GmbH and FH Oberösterreich Research. The authors would like to thank Daniela Lettner for the careful review of the manuscript.

References

Asikainen, T., Männistö, T., Soininen, T.: A unified conceptual foundation for feature modelling. In: 10th International Software Product Line Conference (SPLC 2006), Baltimore, MD, USA, pp. 31–40. IEEE Computer Society, Los Alamitos (2006)

Batory, D.: Feature models, grammars, and propositional formulas. In: 9th International Software Product Line Conference (SPLC 2005), Rennes, France. LNCS, vol. 3714, pp. 7–20. Springer, Berlin/Heidelberg (2005)

Berg, K., Bishop, J., Muthig, D.: Tracing software product line variability: from problem to solution space. In: SAICSIT '05: Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries, pp. 182–191. South African Institute for Computer Scientists and Information Technologists, Pretoria (2005)

Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K.: Variability modeling in the real: a perspective from the operating systems domain. In: ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20–24, 2010, pp. 73–82. ACM, New York (2010)

Campbell, G.H., Faulk, S.R., Weiss, D.M.: Introduction to synthesis. Tech. rep. Software Productivity Consortium, Herndon, VA, USA (1990)

Charles, P., Fuhrer, R.M., Stanley, M., Sutton, J.: IMP: A meta-tooling platform for creating language-specific IDEs in Eclipse. In: Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering, New York, NY, USA, pp. 485–488. ACM, New York (2007)

Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley, Reading (2001)
Consortium, S.P.: Synthesis guidebook. Tech. rep., SPC-91122-MC. Software Productivity Consortium, Herndon, Virginia (1991)
Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Techniques, and Applications. Addison-Wesley, Reading (2000)
Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, New York, NY, USA, pp. 211–220. ACM, New York (2006)
Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Softw. Process Improv. Pract. **10**(1), 7–29 (2005)
Czarnecki, K., Kim, C.H.P., Kalleberg, K.T.: Feature models are views on ontologies. In: SPLC, pp. 41–51 (2006)
Dashofy, E., van der Hoek, A., Taylor, R.: A highly-extensible, xml-based architecture description language. In: Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Amsterdam, The Netherlands, pp. 103–112. IEEE Computer Society, Los Alamitos (2001)
Dashofy, EM, van der Hoek, A., Taylor, R.N.: An infrastructure for the rapid development of xml-based architecture description languages. In: ICSE, pp. 266–276 (2002)
Dhungana, D., Grünbacher, P., Rabiser, R.: Domain-specific adaptations of product line variability modeling. In: Ralyté, J., Brinkkemper, S., Henderson-Sellers, B. (eds.) Situational Method Engineering: Fundamentals and Experiences, Proceedings of the IFIP WG 8.1 Working Conference, Geneva, Switzerland. IFIP, vol. 244, pp. 238–251. Springer, Berlin (2007a)
Dhungana, D., Rabiser, R., Grünbacher, P., Neumayer, T.: Integrated tool support for software product line engineering. In: ASE, pp. 533–534 (2007b)
Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T.: Structuring the modeling space and supporting evolution in software product line engineering. J. Syst. Softw. **83**(7), 1108–1122 (2010a)
Dhungana, D., Heymans, P., Rabiser, R.: A formal semantics for decision-oriented variability modeling with DOPLER. In: Proc. 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010), ICB-Research Report No. 37, pp. 29–35. University of Duisburg Essen, Linz (2010b)
Estublier, J., Vega, G.: Reuse and variability in large software applications. In: 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal, pp. 316–325. ACM Press, New York (2005)
Forgy, C.L., Shepard, S.J.: Rete: a fast match algorithm. AI Expert **2**(1), 34–40 (1987)
Forster, T., Muthig, D., Pech, D.: Understanding decision models: visualization and complexity reduction of software variability. In: Heymans, P., Kang, K., Metzger, A., Pohl, K. (eds.) Second International Workshop on Variability Modeling of Software-Intensive Systems, ICB Research Report, Essen, Germany, vol. 22, pp. 111–119 (2008)
Froschauer, R., Dhungana, D., Grünbacher, P.: Managing the life-cycle of industrial automation systems with product line variability models. In: 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Parma, Italy, pp. 35–42 (2008)
Froschauer, R., Zoitl, A., Grünbacher, P.: Development and adaptation of IEC 61499 automation and control applications with runtime variability models. In: Proceedings 7th IEEE Int'l Conference on Industrial Informatics (INDIN 2009), Cardiff, UK, pp. 905–910 (2009)
Garlan, D., Allen, R., Ockerbloom, J.: Exploiting style in architectural design environments. In: Foundations of Software Engineering, pp. 175–188 (1994)
Gomaa, H.: Designing Software Product Lines with UML. Addison-Wesley, Reading (2005)
Gomaa, H., Shin, M.E.: Multiple-view meta-modeling of software product lines. In: ICECCS '02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems, Washington, DC, USA, p. 238. IEEE Computer Society, Los Alamitos (2002)
Grünbacher, P., Rabiser, R., Dhungana, D., Lehofer, M.: Model-based customization and deployment of eclipse-based tools: Industrial experiences. In: 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), Auckland, New Zealand, pp. 247–256 (2009)
Grundy, J., Hosking, J., Zhu, N., Liu, N.: Generating domain-specific visual language editors from high-level tool specifications. In: 21st IEEE International Conference on Automated Software Engineering (ASE'06), Tokyo, Japan, pp. 25–36. IEEE, New York (2006)
Harmsen, F., Brinkkemper, S.: Design and implementation of a method base management system for a situational case environment. In: Asia Pacific Software Engineering Conference, APSEC, pp. 430–438 (1995)

Henderson-Sellers, B., Gonzalez-Perez, C., Ralyté, J.: Situational method engineering: fragments or chunks. In: International Conference on Advanced Information Systems Engineering (CAiSE Forum) (2007)

Heymans, P., Schobbens, P., Trigaux, J., Matulevicius, R., Classen, A., Bontemps, Y.: Towards a comparative evaluation of feature diagram languages. In: Software and Services Variability Management Workshop—Concepts, Models and Tools, HUT-SoberIT-A3, Helsinki, Finland, pp. 1–16 (2007)

Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., Technical Report CMU/SEI-90TR-21. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA (1990)

Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Proceedings of the 5th European Software Engineering Conference. Lecture Notes In Computer Science, vol. 989, pp. 137–153. Springer, London (1995)

Mansell, J., Sellier, D.: Decision model and flexible component definition based on xml technology. In: Lecture Notes in Computer Science: Software Product-Family Engineering 5th International Workshop, PFE 2003, pp. 466–472. Springer, Berlin/Heidelberg (2004)

Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000)

Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (2005)

Rabiser, R.: Flexible and user-centered visualization support for product derivation. In: ViSPLE 2008—2nd International Workshop on Visualisation in Software Product Line Engineering, held in conjunction with the 12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland, pp. 323–328 (2008)

Rabiser, R., Dhungana, D., Grünbacher, P.: Tool support for product derivation in large-scale product lines: a wizard-based approach. In: ViSPLE 2007—1st International Workshop on Visualisation in Software Product Line Engineering, held in conjunction with the 11th International Software Product Line Conference (SPLC 2007), Kyoto, Japan, pp. 119–124 (2007a)

Rabiser, R., Grünbacher, P., Dhungana, D.: Supporting product derivation by adapting and augmenting variability models. In: 11th International Software Product Line Conference (SPLC 2007), Kyoto, Japan, pp. 141–150 (2007b)

Rabiser, R., Dhungana, D., Grünbacher, P., Burgstaller, B.: Value-based elicitation of product line variability: an experience report. In: Second International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2008), Essen, Germany, pp. 73–79 (2008)

Rabiser, R., Wolfinger, R., Grünbacher, P.: Three-level customization of software products using a product line approach. In: 42nd Hawaii International Conference on System Sciences, Big Island, Hawaii, USA, pp. 1–10 (2009)

Robson, C.: Real World Research, 2nd edn. Blackwell, Oxford (2002)

Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**(2), 131–161 (2009)

Schmid, K., John, I.: A customizable approach to full-life cycle variability management. J. Sci. Comput. Program. **53**(3), 259–284 (2004). Special Issue on Variability Management

Schobbens, P., Heymans, P., Trigaux, J., Bontemps, Y.: Generic semantics of feature diagrams. Int. J. Comput. Telecommun. Netw. **51**(2), 456–479 (2007)

Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: a survey and a formal semantics. In: 14th IEEE International Conference on Requirements Engineering, Minneapolis/St.Paul, Minnesota, pp. 136–145 (2006)

Sinnema, M., Deelstra, S.: Classifying variability modeling techniques. J. Inf. Softw. Technol. **49**, 717–739 (2007)

Steger, M., Tischer, C., Boss, B., Müller, A., Pertler, O., Stolz, W., Ferber, S.: Introducing pla at Bosch gasoline systems: experiences and practices. In: Nord, R. (ed.) Third Software Product Line Conference (SPLC 2004), Boston, MA, USA, pp. 34–50. Springer, Berlin/Heidelberg/Boston (2004)

Thiel, S., Hein, A.: Modeling and using product line variability in automotive systems. IEEE Softw. **19**(4), 66–72 (2002)

Tolvanen, J.P., Rossi, M.: MetaEdit+: Defining and using domain-specific modeling languages and code generators. In: Conference on Object Oriented Programming Systems Languages and Applications (OOPLSA'03), Anaheim, CA, USA, pp. 92–93. ACM Press, New York (2003)

van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action—The Best Industrial Practice in Product Line Engineering. Springer, Berlin/Heidelberg (2007)

van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The koala component model for consumer electronics software. Computer **33**(3), 78–85 (2000)

Verlage, M., Kiesgen, T.: Five years of product line engineering in a small company. In: 27th International Conference on Software Engineering (ICSE 2005), St. Louis, MO, USA, pp. 534–543. IEEE Computer Society, Los Alamitos (2005)

Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., Heider, W.: Flexible and scalable consistency checking on product line variability models. In: 25th IEEE/ACM International Conference on Automated Software Engineering, pp. 63–72 (2010)

Wolfinger, R., Reiter, S., Dhungana, D., Grünbacher, P., Prähofer, H.: Supporting runtime system adaptation through product line engineering and plug-in techniques. In: 7th IEEE International Conference on Composition-Based Software Systems, Madrid, Spain, pp. 21–30. IEEE Computer Society, Los Alamitos (2008)

Yin, R.: Case Study Research: design and methods, 3rd edn. Sage, Thousand Oaks (2003)

Zhu, N., Grundy, J., Hosking, J.: Pounamu: a meta-tool for exploratory domain-specific visual language tool development. J. Syst. Softw. **80**(8), 1390–1407 (2007)