2013 39th Euromicro Conference Series on Software Engineering and Advanced Applications

Towards Component-based Domain Engineering

Asmaa Alayed, Kung-Kiu Lau, Petr Štěpán and Cuong Tran
School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
Email: aalayed,kung-kiu,pstepan,ctran@cs.manchester.ac.uk

***Abstract*—Domain engineering is the first phase of product line engineering. The key artefact created by domain engineering for product engineering is a reference architecture for all possible systems or products in the domain. In existing domain engineering techniques, there are no well-defined methods for constructing reference architectures. Existing domain engineering tools mostly stop short of constructing reference architectures. In this paper, we show how a component-based approach to domain engineering can provide a remedy.**

I. Introduction

Domain Engineering is the first phase of Product Lines Engineering [10]; the second phase being Product Engineering. Domain Engineering is concerned with analysing and modelling domain requirements as well as implementing the results. The latter feed into Product Engineering, which is concerned with constructing actual products or systems in the domain. In this paper, we focus on Domain Engineering only. For terminology, we follow [17], [16], [19], [11].

Domain Engineering starts with domain *analysis*. Domain analysis results in domain models such as a *feature model*, which defines feature composition and variability in products, and a *functional model*, which defines the behaviour of all possible products (containing all possible variations of features).

Domain analysis is followed by domain *design*. In domain design, the feature model and the functional model together are used to create a *reference architecture* for all possible systems or products in the domain. The reference architecture contains variation points (as defined in the feature model) where different features may be selected for a specific product. It should also provide a blueprint for generating all possible products (containing all possible variations of features) whose behaviour is defined by the functional model.

The reference architecture is thus the key artefact for Product Engineering. However, constructing a reference architecture is not straightforward, and in existing domain engineering techniques, there are no well-defined methods for doing it. Existing domain engineering tools [30] mostly stop short of constructing reference architectures.

In this paper, we show how a component-based approach to Domain Engineering can provide a remedy. In particular, we show how the functional model can be defined in a component-based manner, using a suitable component model, as an architecture that contains all the variation points defined in the feature model. Such an architecture is of course a reference architecture, and so by defining the functional model in this way we automatically get a reference architecture as well.

II. Domain Engineering

The *feature model* represents the mandatory and optional features of a product, together with any dependencies between optional features. Fig. 1 shows the feature model for vehicle

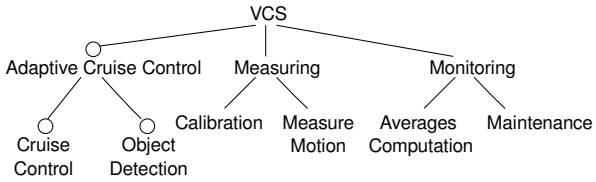

Fig. 1. Feature model for vehicle control systems.

control systems (VCS). Circles denote optional features, and composition rules define dependencies between these features. In a VCS product, the optional feature Object Detection can only be present if (the optional feature) Cruise Control is also present.

The *functional model* describes the *data* (or *structural*) *model* and the *control model*. The functional model specifies functional structure and behaviour in terms of a *data model* and a *control model* respectively. The data model describes data flow in the domain, starting from data I/O to the domain, as a hierarchical decomposition of functionality in the domain where each node is a *data flow diagram* (DFD). Within a DFD (Fig. 2), functions are specified as *data transformations* (DTs) interconnected by data flows. The decomposition process of

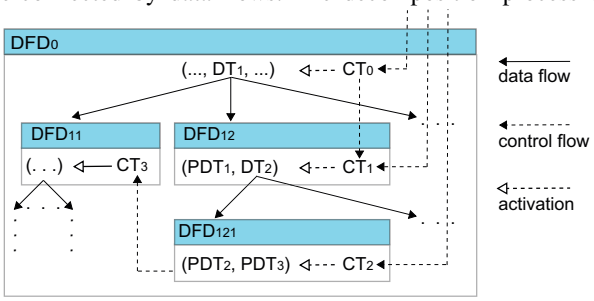

Fig. 2. Hierarchical decomposition of DFDs.

data transformations ends with *primitive* data transformations (PDTs). Each primitive data transformation is associated with a specification for data access or general data processing.

Data transformations are activated by control transformations (CTs). A control transformation receives input control

978-0-7695-5091-6/13 $26.00 © 2013 IEEE
DOI 10.1109/SEAA.2013.44

signals, processes or transforms them, outputs them, and activates data transformations in the corresponding DFD. For example, in Fig. 2, CT_2 activates the primitive data transformations PDT_2 and PDT_3.

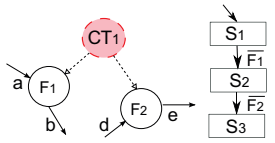

Fig. 3. A simple functional model.

To complete the definition of a functional model, a state transition diagram (STD) needs to be given for each control transformation and its associated data transformations. Fig. 3 shows a simple functional model, which defines the sequential invocation of two PDTs (F_1 and F_2). The STD specifies that CT_1 activates first F_1 then F_2.

Once the feature and functional models have been defined, the next step is to construct a reference architecture for all possible products in the domain. This is difficult for existing domain engineering techniques, because feature and functional models do not readily translate to architectures.

The feature and functional models described above follow feature-oriented domain analysis methods, the best-known of which is FODA [19]. In these methods, data models define structured data transformations, i.e. functions, that correspond to features in the feature model. However, data models do not capture feature variability.

Control models define activations of functions corresponding to features by means of state transition diagrams. Such diagrams do not readily translate into architectures. Consequently, creating a reference architecture becomes a separate task: the task of using the feature and functional models as a specification and coming up with product architectures with all possible feature variations.

III. Related Work

According to the survey in [1], domain engineering based on feature-oriented analysis (as opposed to object-oriented analysis) is best suited to component-oriented Product Engineering. The survey includes an analysis of how well these methods support the construction of a reference architecture, given the feature and functional models. These methods [19], [23], [20], [13], [15] define processes and guidelines for modelling and designing architectures. They do not derive reference architectures directly from the feature and functional models, and do not construct products (and variants) as executable systems.

Another survey [14] also concludes that in existing domain engineering approaches, it is not clear how the reference architecture should be built.

Other feature-oriented approaches take a programming approach, i.e. they define features as programming language constructs and define architectures by putting features together. In generative programming, for example, an architecture is represented as nested templates (which contain features) based on GenVoca grammar [6], and code for different products can be generated from instances of such templates.

Feature-oriented software development (FOSD) [2] uses object-oriented languages with special classes that define features, and develop object-oriented systems by putting feature objects together.

Component-based product line engineering approaches include, for instance, Koala [31], [29] and KobrA [4], [5]. Koala is a component model which is an architecture description language [28], [25]. Components have interfaces and may contain other components. A configuration is a special component that has no interface and is not part of another component. Koala uses a configuration as a specific product in a family of products (constructed from components). However, Koala does not define variation points, so it cannot define reference architectures. Koalish [3] adds variation points to Koala, but it also does not define reference architectures; rather it is used for configuring products directly from components.

KobrA uses object-oriented analysis, so according to [1] it would not be as good as feature-oriented approaches. In KobrA, components are UML components as defined in [9]. (These are different from UML2.0 [26] components. The former are just classes/objects whereas the latter are architectural units.) Their behaviour and interactions are defined by sequence diagrams. Products are aggregations or compositions of objects, and their structure is modelled by class diagrams. However, these UML diagrams do not directly define products or architectures. Neither do they define explicit functional models as in feature-oriented domain analysis. In KobrA, variability is defined by a decision model with links to the UML modelling diagrams. Nonetheless, to define reference architectures explicitly would require another task.

IV. A Component-based Approach

All the feature and functional models (and parts thereof) in Section II are defined according to FODA. In this section, we present a component-based approach to domain engineering which enables us to define the functional model in a component-based manner (i.e. using a component model), which directly yields a reference architecture, unlike FODA. We use VCS as a running example.

A component-based approach has an underlying component model. We use the X-MAN component model.

A. The X-MAN Component Model

In the X-MAN component model [22], [21], [18] (Fig. 4), there are two basic entities: (i) *components*, and (ii) *composition connectors*.

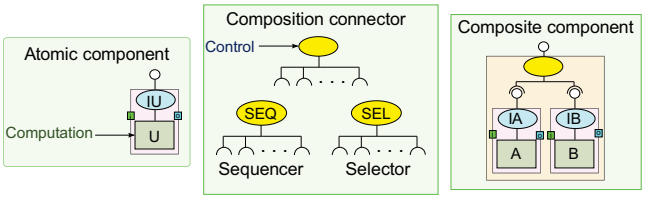

Fig. 4. The X-MAN component model.

Components are *encapsulated* units of behaviour: a component defines computation (methods) which have no external

dependencies on other components, i.e. a component does not call methods in other components.

A component's behaviour is exposed through its interface as *provided services* (lollipops in Fig. 4). Composition connectors invoke components' provided services, and initiate and coordinate such invocations and their results; as such, composition connectors are composition mechanisms for components.

Components can be atomic or composite. An atomic component consists of an invocation connector (*IU*) linked to a computation unit (*U*). The computation unit contains an implementation of some behaviour (methods) in a chosen programming language. The computation unit's computation does not call the methods of another computation unit (of another component). The invocation connector exposes the methods of the computation unit via the provided services in the component's interface and allows them to be invoked by composition connectors.

Composition connectors are control structures. They are used to compose (atomic or composite) components into composite components. In a composite component, sub-components do not call each other. Instead, the composition connector coordinates the sub-components' execution.

Two basic composition connectors are *Sequencer* (*SEQ*) and *Selector* (*SEL*). *Sequencer* provides sequencing, whilst *Selector* offers branching. For instance, the composite component in Fig. 4 could be a bank system, where the sequencer *SEQ* first calls an ATM (component A) to get customer inputs and then calls the customer's bank branch (component B) to handle the inputs.

In addition, there are special connectors that are not used for composition, but for adapting single components. These connectors are called *adaptors*; examples are *Loop* and *Guard*. *Loop* allows for iteration[1] (over a single component) and *Guard* offers gating.

The execution semantics for X-MAN is control-driven. Component execution is initiated and coordinated by composition connectors. Composition connectors initiate control to invoke components. At each component, control then triggers read and write operations on *data channels* (not shown in Fig. 4, but can be seen in Fig. 5) associated with the component in order to supply the component's inputs from other components and data stores, and distribute the outputs from the component.

B. Functional Model using X-MAN

The basic idea of using X-MAN to define the functional model is that in X-MAN an architecture actually defines a behaviour which is the result of computations (on data) triggered by control flow. Thus an X-MAN architecture defines all the elements of the functional model: data model (data transformations: data and functions), control model (control transformations) as well as the associated STDs. In other words, using X-MAN to define a functional model gives us not only a functional model, but also an architecture containing all the features in the feature model (though not the variation points therein).

1) Primitive Data Transformations: An atomic X-MAN component defines a primitive data transformation (PDT). More precisely, the methods of a component define functions (data transformations), and the data channels of a component define data flow in/out of the component. Fig. 5 shows two X-MAN components that define 2 PDTs connected by data flow. In component X, data values a and b are on its input data ports, and data value c is on its output data port. In component Y, data value c is on its input data port, and data value d is on its output data port. Data value c is passed from component X to component Y. These two components and their data flow define the two PDTs P_X and P_Y in Fig. 5.

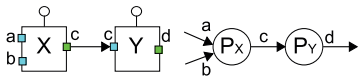

Fig. 5. X-MAN components define PDTs.

2) Control Transformations: A composition connector in X-MAN defines control flow between the composed components. The behaviour of the composite can be described by an STD. The composition connector therefore defines a control transformation on the data transformations (functions) corresponding to the components.

For example, the sequencer composition connector passes control sequentially among the composed components. It therefore defines a control transformation that activates or invokes a sequence of data transformations, with a simple 'linear' STD. This is shown in Fig. 6, where n (*atomic*) components are composed by a sequencer.[2] In the composite, the behaviour starts in the initial state (S_1) when the sequencer starts control flow, invokes component 1 (function F_1), transits to the next state (S_2), invokes component 2, ..., transits to the next state (S_n), invokes component n (function F_n), transits to the final state (S_{n+1}). This composite thus defines the same behaviour defined by the control transformation CT_1 and the associated STD in Fig. 6.

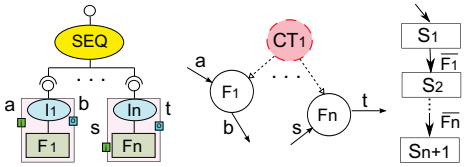

Fig. 6. Control transformation defined by sequencer.

Similarly, a selector composition connector also defines a control transformation. A selector invokes a selected component, therefore it defines a control transformation that activates a selected data transformation, with a simple 'branching' STD. This is shown in Fig. 7, where a selector composes n (*atomic*) components. In the composite, the behaviour starts at the initial state (S_1). The next transition is one of n alternative transitions depending on the evaluation of the condition c to a range of values $v_1 \ldots v_n$. If c evaluates to v_i then component i is invoked (function F_i) and the next state is the final state (S_{i+1}). This composite thus defines the same behaviour

[1]Finite iteration only, except for adaptors at the top-level of a system.

[2]For simplicity we omit the data channels.

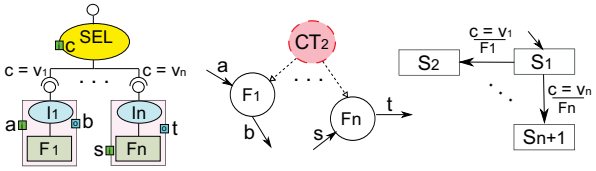

Fig. 7. Control transformation defined by selector.

defined by the control transformation CT_2 and the associated STD in Fig. 7.

Adaptors in X-MAN also define control transformations, on single data transformations. Fig. 8 shows an adaptor (AD)

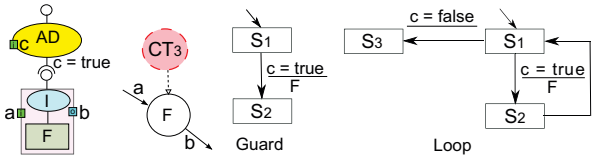

Fig. 8. Control transformation defined by an adaptor.

applied to a single (*atomic*) component. If the adaptor is a guard then it invokes the component (function F) if the condition (c) is true. If the adaptor is a loop then it repeatedly checks the condition (c) and invokes the component (function F) if the condition is true; it terminates when the condition is false. Both a guard and a loop adaptor define the same control transformation (CT_3) but they have different STDs as shown in Fig. 8.

3) Non-primitive Data Transformations: Non-primitive data transformations contain hierarchical levels of decomposition, with control transformations activating data transformations at different levels (as shown in Fig. 2). At the bottom level of decomposition, data transformations are finally decomposed into primitive data transformations (PDTs). In the last section, we have shown how X-MAN composition connectors define control transformations for activating PDTs. Now we show how composition connectors also define control transformations for activating non-primitive data transformations.

The key thing to observe is that in X-MAN, a composite component has exactly the same outward appearance, i.e. interface, as an atomic component. This can be seen in Fig. 4. As far as composition connectors are concerned, it makes no difference whether the components they compose are atomic or composite.

Thus we could easily extend our discussion of X-MAN composition connectors as control transformations to non-primitive data transformations. For example, the composite

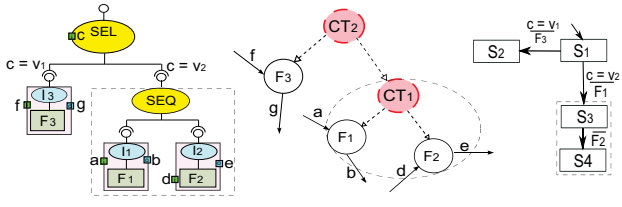

Fig. 9. Composing composite components.

in Fig. 9 is the result of a selector composing an atomic component with a composite component. The latter, the corresponding non-primitive data transformation, and the corresponding compound state are all indicated by an enclosure of dashed lines. The control transformation CT_2 activates the non-primitive data transformation that contains the control transformation CT_1 that in turn activates the PDTs F_1 and F_2. This example thus shows how composition connectors define control transformations that activate non-primitive data transformations and do so in a hierarchical manner (as depicted in Fig. 2).

4) The VCS Example: Fig. 10 shows the top level of the X-MAN functional model for the VCS domain, created by using a modelling tool we have implemented for X-MAN.[3] The system level loop adaptor (Sys_Loop) and the composition

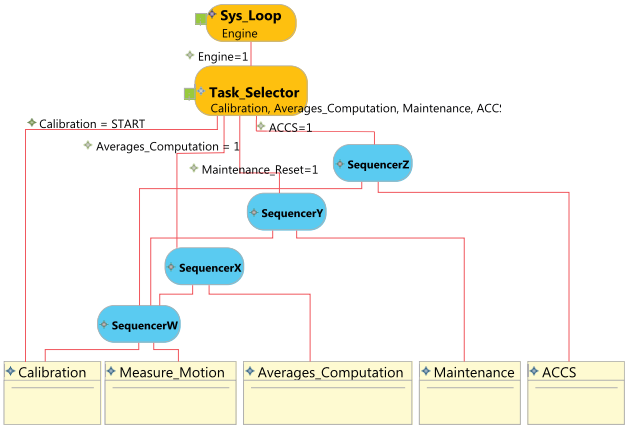

Fig. 10. X-MAN functional model for VCS.

connectors (a selector (Task_Selector) and four sequencers drawn as rounded rectangles) combine control transformations and their STDs.

The five components at the bottom of the architecture are composite components. As an example, the MeasureMotion component is shown in Fig. 11. It is a composition of three atomic components by a sequencer connector. Fig. 11 also shows how our modelling tool depicts features of X-MAN that have not been shown in other diagrams so far: (i) services are shown as (purple) round rectangles; (ii) data channels are arrows; (iii) data stores – shared places storing inputs and outputs of services – are cylinders; and (iv) external entities – a special kind of data stores for sensor readings and actuator commands – are represented as triangles.

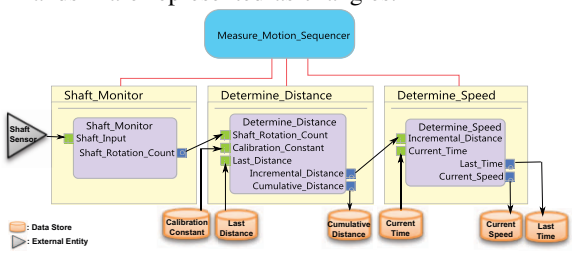

Fig. 11. The Measure_Motion composite component.

[3]For clarity, we omit data channels and services in Fig. 10.

C. Reference Architecture using X-MAN

In most approaches to functional modelling (see Sections II-III), including FODA, elements of the functional model do not readily translate to elements of a software architecture; creating a reference architecture is usually a separate step that follows the creation of the feature and functional models, and that requires substantial human knowledge, experience, ingenuity and effort. By contrast, a functional model based on X-MAN yields a reference architecture immediately if it also incorporates *variation points* as defined in the feature model. Variation points define places where optional features may or may not be present in a product.

To define variation points explicitly in the functional model would require a meta-level extension to X-MAN to represent variant behaviours. Nonetheless, we can define variability via the absence or presence of components in an X-MAN architecture.

1) Defining Variability: In general, a functional model specifies the behaviour of all the features in the domain. An individual product, however, impleme nts only some subset of the features; its behaviour is therefore specified by some subset of the functional model. In the case of X-MAN, since the functional model is an architecture, an individual product also has an architecture that is some subset of the functional model.

To be able to derive a product architecture, we need a way of mapping the product's features to a subset of the functional model; in particular we must establish links between parts of the functional model and features from the feature model. The reference architecture is just the functional model with these links.

Functional models in our approach have the structure of an X-MAN system architecture; they are trees in which leaves are formed by atomic components and every inner node corresponds to a composition connector (forming a corresponding composite component). Likewise, feature models form trees in which features can comprise sub-features (inner nodes) or be on their own (leaves). To establish the mapping between a functional model and a feature model, we use the fact that both models are trees: we associate leaves in the feature tree – features – with the leaves in the functional model tree – atomic components implementing the features' behaviour (see Fig. 12). We rely on an assumption (similar to the one FODA makes in creating its functional models) that it is possible for each atomic component to identify a feature or features whose behaviour it implements. One feature can be implemented by more than one atomic component; and conversely, an atomic component can implement the behaviour of multiple features.

To establish links between atomic components and features, we annotate each atomic component with the feature or features whose behaviour it represents, as soon as we have modelled the corresponding primitive data transformation.

Having manually defined the correspondence between the leaves of the functional and feature models, we can use composition mechanisms defined in both models to automatically propagate the correspondence to the whole trees.

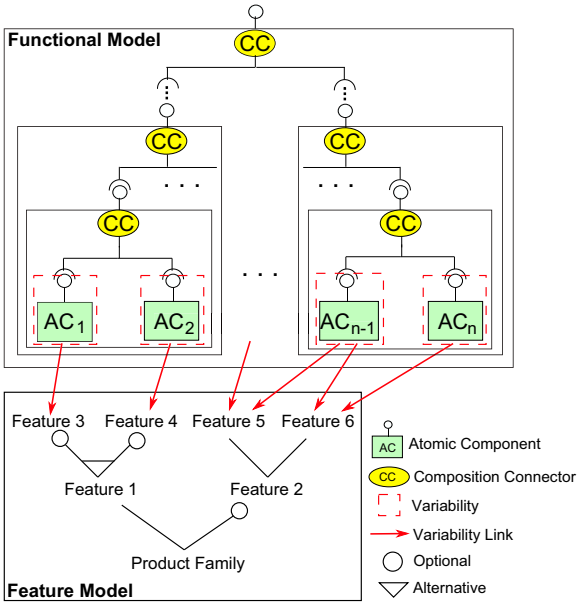

Fig. 12. Creating a reference architecture.

In a functional model, a composite component implements the union of the features of all the children of its top-level composition connector. For example, in Fig. 12, the composite component comprising AC_1 and AC_2 implements features 3 and 4.

In a feature model, the variation links to a super-feature comprise variation links to all its sub-features. For instance, Feature 2 in Fig. 12 is the target of the four variation links going to its sub-features 5 and 6. The mapping can be computed as part of the product derivation process described in Section IV-D.

2) The VCS Example: Fig. 13 shows the reference architecture for the VCS domain. It illustrates the fact that a reference

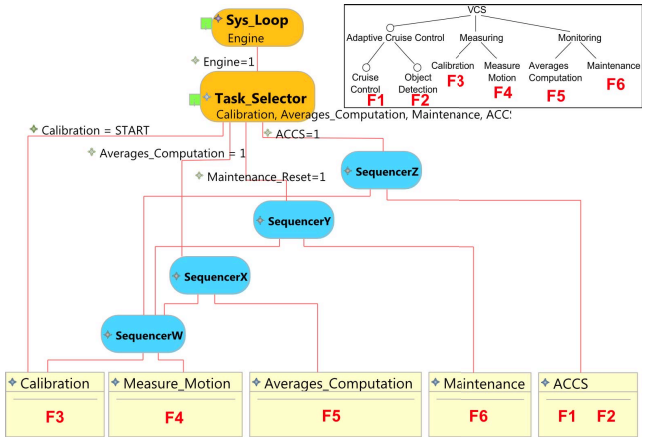

Fig. 13. X-MAN reference architecture for VCS.

architecture in our approach corresponds to a functional model linked to a feature model. There is no one-to-one mapping between features and components. Annotations for composite components can be generated automatically from those for atomic components; however, atomic components have to be

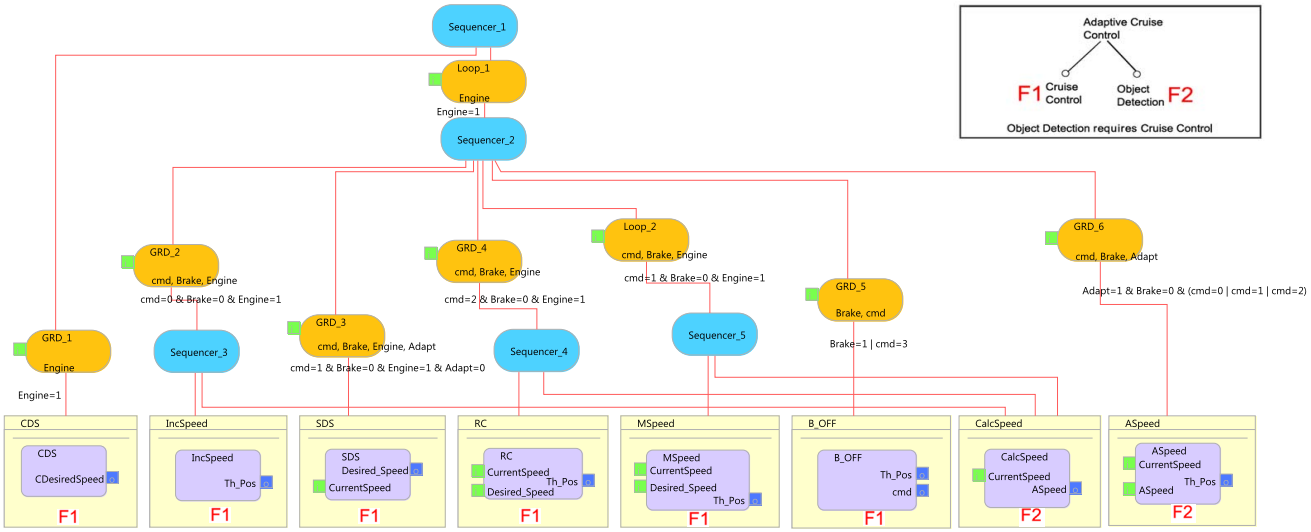

Fig. 14. X-MAN reference architecture for ACCS.

annotated manually. For instance, in Fig. 14, which shows the reference architecture for Adaptive Cruise Control (ACCS), all the sub-components of the ACCS component that implements the behaviour of Adaptive Cruising are atomic and have to be annotated manually.

Fig. 14 shows that not every feature can be encapsulated as a separate composite component: components implementing the Cruise Control feature (F1) form one composite component together with atomic components implementing the Object Detection feature (F2). It would be unreasonable to expect the opposite. After all, related features in the feature model may not have any behavioural relations, whereas components in X-MAN are composed precisely because their behaviours need to be composed into the desired behaviour of the resulting composite component.

D. Deriving Product Variants

A product variant derivation takes two inputs: the reference architecture and a valid selection of (the product's) features from the feature model; and produces an architecture that implements the behaviour of the product with the selected features. To derive each product variant, it is necessary to disentangle its architecture from the reference architecture. In our approach, product variant derivation consists of three steps:

1) mark all atomic components linked with the selected features (to be included in the product architecture);
2) trace all dependencies, both control and data flow, between marked components and unmarked components;
3) extract architecture for product variant from reference architecture according to all the above dependencies.

Step 1 ensures that the product architecture implements only the behaviour of the selected features. It uses the links defined in the reference architecture between atomic components and the features they implement.

Step 2 allows us to construct control flow and data flow for the product architecture from the reference architecture. It identifies all flows of data and control that will not be present in the product architecture because their targets are components implementing features that are absent in the product.

Step 3 extracts the product architecture that contains the atomic components marked in Step 1, and defines control flow and data flow derived from the reference architecture by excluding dependences identified in Step 2.

The extraction of control flow from the reference architecture amounts to deriving a composition connector tree. Its structure may differ from the composition structure of the reference architecture since some coordination behaviour is not needed in the product, but the coordination behaviour related to the marked components must be preserved. The fixed set of basic composition connectors in X-MAN enables the derivation to be guided by the following set of rules.

Because no coordination behaviour is added during the product derivation (it is defined in the reference architecture), a control connector from the reference architecture can be replaced by its children in the product architecture, its arity can decrease, or it can be omitted altogether. In general there are two cases.

- A connector in the reference architecture does not have enough parameters present in the product architecture (i.e. no parameters for an adapter and less than two parameters for a composition connector) and cannot be therefore copied as is. If such connector has one parameter left, then the parameter (an atomic or composite component) takes the place of the connector in the composition tree of the product. Here, the designer has to ensure that such a change is valid, e.g., by comparing pre-conditions of the connector and the parameter. If the connector in the reference architecture has no parameters left, the arity of its parent connector in the product architecture must be decreased according to the following rule.
- A composition connector in the reference architecture has some parameters missing in the product architecture, but at least two of its parameters are present. The composition

connector only needs to have its arity decreased. If it is a sequencer, its present parameters may need re-numbering. If it is a selector, the present parameters may need reformulating in terms of their selection conditions. Again, the designer needs to make sure that such changes are valid.

1) The VCS Example: The feature model for the VCS (Fig. 1) allows the three possible products to be derived from the VCS reference architecture (Fig. 13):

Product	Features
Product 1	F1, F2, F3, F4, F5, F6
Product 2	F3, F4, F5, F6
Product 3	F1, F3, F4, F5, F6

The architecture of Product 1 corresponds directly to the reference architecture (Fig. 13). In general, it may not always happen that a product corresponds to the whole reference architecture: the main role of a reference architecture is to aggregate the behaviour of all the features of the domain; it may not define a sensible system behaviour.

Product 2 does not contain the ACCS feature. In Step 1 of the product derivation, all atomic components but those realising Cruise Control (F1) and Object Detection (F2) from the ACCS composite component (Fig. 14) have been marked for inclusion in the product architecture. In Step 2, all data and control flows to the ACCS have been identified, and excluded in the extraction of the product architecture in Step 3. The derivation of the connector tree of the product resulted in excluding the Sequencer_Z connector and decreasing the arity of the Task_Selector connector in the product architecture (Fig. 15).

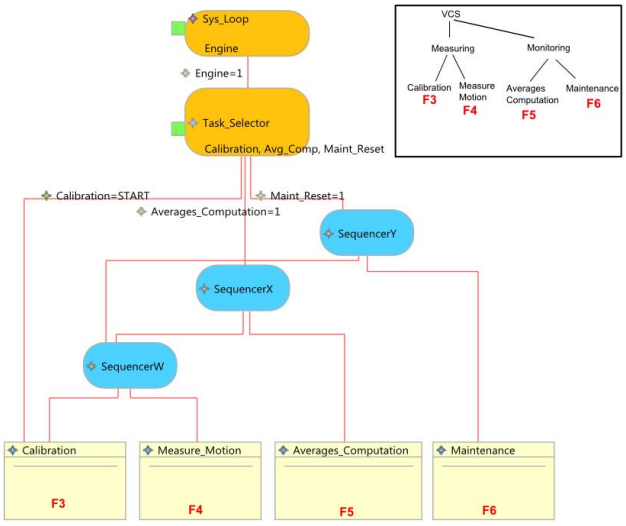

Fig. 15. Top level of Product 2 architecture.

Product 3 contains the Cruise Control feature (F1), but does not have the Object Detection capability (F2). As a result, the top level of the product architecture is the same as the one in Fig. 10. The difference is in the composition of the ACCS composite component. In Step 1 of the product derivation, atomic components linked to the Object Detection feature (F2) are not marked for inclusion (Fig. 14). In Step 2, control flow and data flow dependencies of these components are identified. In Step 3, the above mentioned dependencies are removed; therefore, the ACCS composition connector tree in the product does not contain GRD_6 and the parameters of Sequencer_2, Sequencer_3, Sequencer_4 and Sequencer_5 have been renumbered.[4]

V. Discussion and Conclusion

We have presented an approach to domain engineering that we believe potentially extends FODA. Our approach constructs the functional model as an executable architecture, and thus enabling the reference architecture to be defined as such a functional model with variability as defined in the feature model. Furthermore, our reference architecture enables all product variants to be extracted from it; and these products are all executable. By contrast, FODA (and FORM) does not construct explicit reference architectures; neither does it construct explicit product variants. According to surveys on domain engineering such as [1] and [14], other domain engineering methods have similar shortcomings. Therefore we believe our approach is a potential advance on the state-of-the-art in domain engineering [8], [12], [7].

Using the X-MAN component model to construct a functional model has a couple of advantages. Firstly such a functional model is executable and this means that we can verify the reference architecture against the domain requirements earlier in the analysis phase, and this would reduce the cost of development.

Secondly, X-MAN construction is strictly hierarchical, and this means increased scalability in software design. In other words, the reference architecture will have the ability to include large numbers of components and interactions among them during the analysis and design phase.

The high level of encapsulation in X-MAN model offers other advantages, compared with other component-based product line engineering, e.g. KobrA and Koala. One advantage is that it is a recognised factor in modelling high quality reference architectures, see e.g. [24].

Additionally, it helps to mitigate the problem of unintended feature interactions [27], [2]. These occur when two features behaving correctly on their own exhibit incorrect behaviour once combined in a product, due to certain implementation interferences (e.g. sharing some data). They are difficult to discover, especially in approaches with components with many (even undocumented) interdependencies. In X-MAN, the more restricted communication between components provides a check against unintended feature interactions.

Our domain engineering approach also addresses the issue of cross-cutting features, which is another problem that arises in product line engineering. Unlike aspects, which deal with cross-cutting features by definition, the traditional black-box components are ill-suited to representing such features [2],

[4]For lack of space we do not show the final form of ACCS in Product 3.

because they are monolithic blocks without variability. In our domain engineering approach, cross-cutting features can be realised by fine-grained components instantiated across the composition hierarchy of the reference architecture. During a product derivation according to the product's features, composite components composing these cross-cutting features in the reference architecture may or may not compose them in product architecture, according to the variability defined in the reference architecture.

We have evaluated our approach qualitatively. We showed (as summarised in Section II) that there exists a mapping between basic constructs of X-MAN and FODA functional models; this demonstrates that our method is as expressive as FODA-based domain engineering approaches.

As the focus of this paper is on domain engineering, we did not validate the run-time behaviour of the derived products; nonetheless we validated their design by informal design reviews in our research group. Also, the product derivation algorithm needs more empirical evidence to establish its validity. As future work, we have to validate our approach formally. Such validation has to be done against existing approaches, in particular FODA. This may not be straightforward, since current approaches, including FODA, have no formalisation.

We have implemented a modelling tool, used in this paper, for domain engineering. The tool was developed using the Eclipse Modelling Framework (and other related frameworks). As a result, all the designs produced by the tool conform to the meta-model of X-MAN, which guarantees their structural validity. For product line engineering, we need to link this tool to the X-MAN tool described in [18], [21] that allows execution, simulation and testing. The latter is also implemented using Model-driven Engineering. This should facilitate linking the two tools, via model exchange.

Finally, we plan to extend X-MAN to allow explicit representation of variation points in the reference architecture by augmenting the existing meta-model. Such a reference architecture explicitly defines all possible product architectures, as opposed to product architectures extracted by the derivation procedure described in this paper.

REFERENCES

[1] GMV Aerospace and Defence S.A. Domain Engineering Methodologies Survey. A CORDET (Component Oriented Development Technology) report, 2007.

[2] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[3] T. Asikainen, T. Soininen, and T. Männistö. A Koala-based approach for modelling and deploying configurable software product families. In *Software Product-Family Engineering, 5th Int. Workshop*, LNCS 3014, pages 225–249. Springer, 2003.

[4] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2001.

[5] C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development: the KobrA approach. In *Proc. 1st Conf. on Software Product Lines*, pages 289–309. Kluwer, 2000.

[6] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Soft. Eng. Methodology*, 1(4):355–398, 1992.

[7] T. Berger, R. Rublack, D. Nair, Joanne M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A survey of variability modeling in industrial practice. VaMoS '13, pages 7:1–7:8. ACM, 2013.

[8] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the systems software domain. Technical report, University of Waterloo, 2012.

[9] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. The Component Software Series. Addison-Wesley, 2000.

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

[11] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000.

[12] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. VaMoS '12, pages 173–182. ACM, 2012.

[13] J.-M. DeBaud, O. Flege, and P. Knauber. PuLSE-DSSA – a method for the development of software reference architectures. In *Proc. 3rd Int. Workshop on Software Architecture*, pages 25–28. ACM, 1998.

[14] X. Ferré and S. Vegas. An evaluation of domain analysis methods. In *4th CAiSE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, 1999.

[15] W. Frakes, R. Prieto-Diaz, and C. Fox. DARE: Domain analysis and reuse environment. *Annals of Software Engineering*, 5:125–141, January 1998.

[16] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley Longman, 1993.

[17] D.J. Hatley and I.A. Pirbhai. *Strategies for Real-time System Specification*. Dorset House, 1987.

[18] N. He, D. Kroening, T. Wahl, K.-K. Lau, F. Taweel, C. Tran, P. Rümmer, and S. Sharma. Component-based design and verification in X-MAN. In *Proc. Embedded Real Time Software and Systems*, 2012.

[19] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, 1990.

[20] K.C. Kang, S. Kim, J. Lee, K. Kim, G.J. Kim, and E. Shin. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[21] K.-K. Lau and C. Tran. X-MAN: An MDE tool for component-based system development. In *Proc. 38th EUROMICRO Conf. on Soft. Eng. and Advanced Applications*, pages 158–165. IEEE, 2012.

[22] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In G.T. Heineman *et al.*, editor, *Proc. 8th Int. Symp. on Component-based Software Engineering, LNCS 3489*, pages 90–106. Springer-Verlag, 2005.

[23] K. Lee, K.C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Proc. 7th Int. Conf. on Software Reuse*, pages 62–77. Springer-Verlag, 2002.

[24] C. Maga and N. Jazdi. A survey on determining factors for modeling reference architectures. *Int. Conf. on Object-Oriented Programming, Languages, Systems, and Applications (OOPSLA)*, 2009.

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

[26] OMG. OMG Unified Modeling Language Specification, November 2007. http://www.omg.org/cgi-bin/doc?formal/07-11-01.pdf.

[27] E. Pulvermüller, A. Speck, J. O. Coplien, M. D'Hondt, and W. De-Meuter. Position paper: Feature interaction in composed systems. In *Proc. ECOOP*, pages 1–6, 2001.

[28] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[29] R. van Ommering. Building product populations with software components. In *Proc. 24th ICSE*, pages 255–265. ACM, 2002.

[30] R. van Ommering and J. Bosch. Widening the scope of software product lines – from variation to composition. In *Proc. 2nd Int. Conf. on Software Product Lines*, pages 328–347. Springer-Verlag, 2002.

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