2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises

Formalising the Requirements of an E-Voting Software Product Line using Event-B

Abderrahim Ait Wakrime*, J. Paul Gibson† and Jean-Luc Raffy†
*Institut de Recherche Technologique Railenium, F-59300, Famars, France.
†Telecom Sud Paris, SAMOVAR UMR 5157 CNRS Research Laboratory, Evry, France

***Abstract*—A Software Product Line (SPL) is a tool/method used to generate a family of program/system variants for a specific domain, and to support a more efficient software development of future products within the same domain. A Feature Model (FM) is a popular graphical/textual representation used in SPL requirements specification; it is used to capture commonality and variability information existing in an SPL as a set of inter-related and configurable features. A concrete model of an SPL instance is obtained by binding the variation information in the FM with a configuration that meets a specific set of feature requirements. Since configuration decisions are taken prior to instantiation, invalid configurations should be detected/avoided before design begins. This paper addresses the problem of the verification of the correctness (validity) of FM instances and FM configuration during requirements modelling. It proposes a requirements model based on Event-B contexts, allowing us to check the correctness of a given configuration, before starting the correct-by-construction design and implementation process, based on refinement.**

***Keywords*-Software Product Lines, Feature Model Configuration, Formal Modelling, Event-B, E-Voting**

I. Introduction

A Software Product Line (SPL) is [1]:*"... a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."* A Feature Model (FM) is a language used to model the commonality and variability between product variants. Several approaches have been proposed for combining features, where each combination corresponds to a configuration or a program in a SPL [2], [3]. Such approaches verify only that a specific instance respects the configuration rules specified by the feature model, so that the system can be configured. They do not verify that the features, as configured, are compatible (coherent). The use of a formal specification of feature models would facilitate the verification of both types of properties - valid configuration and coherent configuration.

Once a configuration is verified as correct, then we must start the development (design and implementation). A SPL generally incorporates a general architecture which all such instances must be built upon. Specific instances will have different combinations of features, each of which must be added incrementally to the system. Adding a feature's behaviour can be thought of as a system refinement, and so we chose to use a formal specification language - Event-B [4] - that supports refinement-based development. Our approach is based on modelling using Event-B contexts. These contexts are used to model three inter-related types of entities: domain specific concepts, product line features, and product line instantiation.

The structure of the paper is as follows. Section 2 puts our contribution into context and reviews relevant related work. Section 3 provides an overview of our generic formal SPL approach. Section 4 illustrates how our general approach can be applied to the problem of e-voting. Section 5 provides more details concerning the formal verification of the instantiation process, using a concrete e-voting system as an example. It also comments on the notion of feature interaction, and how such interactions can be detected during different stages of the formal development. The 6th and final section concludes the paper, and comments on future extensions to the research.

II. Related Work

In this section, we will focus on the related works of existing approaches of analysis of feature models (FMs) using formal methods and the use of formal methods during e-Voting service construction. In [5] product line models are represented as constraint programs in order to specify configuration requirements and provide support for the product configuration activity. In [6] constraint logic programming is also used to translate and formalize extended FMs over finite domains in order to analyze the different relationships. In another work [7], an approach to modelling and analyzing SPL variant feature diagrams using first-order logic is presented. This formalization is based on logical expressions that can be built by modelling variants and their dependencies by using propositional connectives. In [8] the authors adopt a description logics-based FM, including the feature class and the constraints of the features. Some constraints rules are introduced to verify the consistency and completeness of FM instances.

An alternative approach, proposed in [9], applies an artificial intelligence planning technique to automatically select suitable features that satisfy both the stakeholders preferences and constraints especially with regard to non-functional properties. In addition, the research in [10] presents a model checking technique for SPsL, precisely modelling the FM with non-Boolean features (numeric attributes) and multi-features. Petri nets [11] can be used to model a FM configuration. This approach, based on workflow Petri nets, allows a formal operational model for staged configuration that makes explicit causal dependencies among feature selections. Petri nets are

978-1-5386-6916-7/18/$31.00 ©2018 IEEE
DOI 10.1109/WETICE.2018.00022

also integrated as an established formalism for modelling systems with a high degree of variability in an SPL [12]. To achieve this, a Petri net extension, named Feature Nets, is used to provide modelling dynamic SPLs. Variability is often modelled using transition systems. In [13], [14], modal transition systems are used to model the behaviour in product families, in order to define and derive a valid product behaviour starting from product family behaviour.

Alternative approaches are used that provide a precise and rigorous formal interpretation of the feature diagrams. A binary-search based approach to FM verification is presented in [15] in order to detect deficiencies in FMs. Similar analysis are possible in [16] using an efficient technique for synthesis of models from respectively CNF and DNF formulas. In another work [17], automated analyses of FMs is adopted by translating models to propositional logic and using satisfiability (SAT) solvers.

With respect to our validation case-study, formal methods have been previously used for e-Voting; for example, first-order logic is used for the analysis and development of voting schemes to provide a formal specification and verification [18]. In [19] a formal symbolic definition of election verifiability, based on π-calculus, to precisely identify which parts of a voting system need to be trusted for verifiability. In [20], formal techniques are exploited to build technical solutions for electronic governance in order to specify desirable functionality, build an implementation model and verify that the implementation satisfies the specification. In [21], a formal specification is presented using the Z language for the Single Transferable Vote form of election. This specification exemplifies a functional decomposition style supporting the validation of requirements.

Generally, the purpose of FMs modelling and verification is to detect deficiencies in FMs, in order to avoid having these deficiencies enter into the process of product development. In our work, we provide an Event-B formal approach to check the consistency between a FM and its configurations to manage variability in e-voting services as well as the scalability of automatically generating and validating configurations.

III. General Approach

Our approach aims to formally specify a SPL based on FMs at a high level of abstraction using the Event-B method based on a sequence of correct-by-construction refinements. This formal specification provides a way to verify correctness of such properties of FM configuration instances with respect to product validity.

Our approach is structured as the following sequence of activities: The user specifies their requirements through configuration of the feature tree model. The validity of the configuration is checked by the Event-B tool set and the user can animate the high-level abstract behaviour. Once they validate the specified requirements using the Rodin theorem prover that supports the generation of Proof Obligations belonging to Event-B models, design and implementation are done through refinement. The whole process focuses on the client trusting that their requirements are properly specified, and that correctness of implementation is guaranteed because the requirements are formally specified. Fig. 1 depicts the overall approach.

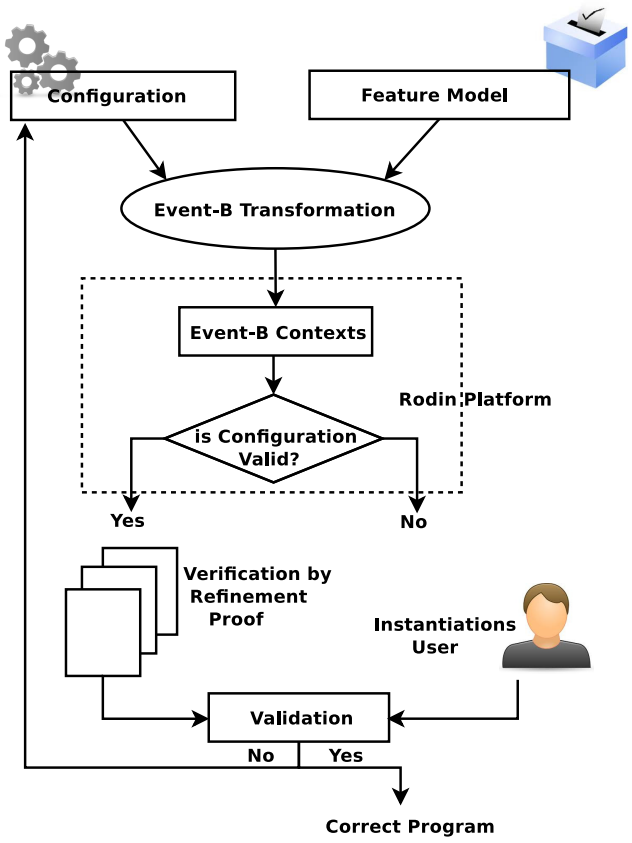

Fig. 1. Feature Models Transformation and Verification Approach.

It should be noted that this paper reports on only the requirements engineering phase of devlopment, where we model the chosen features using only the Event-B contexts. The next phase of development (refinement-driven design) combines the context specifications with Event-B machines. An example of this *correct-by-construction* design approach can be found in [22].

IV. E-Voting Feature Model

Before presenting our e-voting FM, we need to recall some basic concepts and definitions.

A. E-Voting Model

A FM definition was first introduced by [23] in the FODA report in 1990. A FM represents the information of all possible products of a SPL regarding features and relationships between them. A FM is a graphical/textual representation that is widely used in SPL engineering. The FMs are presented as trees in

which each node is a feature and each edge can have five possible values:

- And: all sub-features must be selected.
- Alternative: only one subfeature can be selected.
- Or: one or more can be selected.
- Mandatory: features that required.
- Optional: features that are optional.

A FM may also have constraints that cannot be easily expressed hierarchically and graphically. These constraints are named cross-tree constraints. Cross-tree constraints are propositional formulae using features as variables. A feature can be concrete or abstract. A concrete feature is a terminal feature (a leaf. of the tree). Contrastingly, an abstract feature is a non terminal and compound feature. Fig. 2 depicts a simplified FM to better understand the relationships between its different entities.

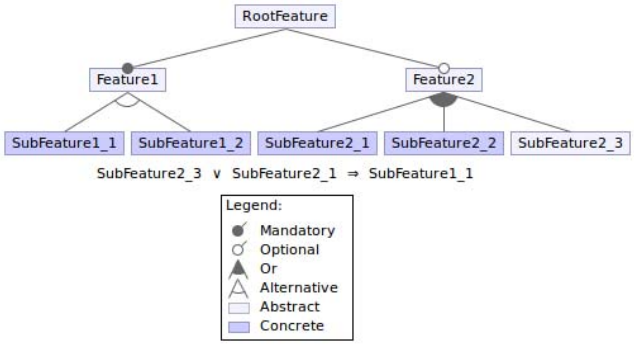

Fig. 2. A sample Feature Model and its Legend.

An extract of our e-voting FM (tree) can be seen in Fig. 3. All the concepts which appear in the tree have been formally specified in an e-voting domain ontology. This ontology has been validated by a domain expert. A government that wishes to procure a particular instance of a voting system specifies a valid instance by removing branches from the SPL FM.

A FM configuration is a set of concrete features. A configuration is valid if the selection of all features contained in the configuration and the deselection of all other concrete features is allowed by the FM. If a concrete feature is selected, its parent must also be selected. If a parent is selected, all the *Mandatory* sub-features must be selected, one subfeature in each of its *Alternative* groups must be selected and at least one of its sub-features in each of its *Or* groups must be selected. It is necesssary that each valid configuration satisfies propositional constraints. For example in Fig. 2, a valid FM configuration is: $\{SubFeature1\_1, SubFeature2\_1, SubFeature2\_2\}$.

B. Transformation E-Voting Model using Event-B

In this section, we introduce our Event-B formalism of the FM and its configuration correctness. The consistency of the model is ensured by formal proofs, see section IV-C.

The Event-B model, defined in a static Context, is used primarily to fix the definitions and formalizations of the concepts in the FM. It also defines a static part where all the relevant properties and hypotheses are formalised. This model is generated automatically from the graphical feature model specification.

The properties considered in our approach formalize the FM, that contains a sets of relationships between a parent feature and its child features. The FM nodes are represented by a *FEATURE* set that it is divided into three subsets of the nodes and they are classified as follows: *Root*, *Node*. A *Root* represents the initial and single parent which has no parent and identifies the SPL. The *Node* type is the node that has a child and one parent. These three nodes types are used to define a part of tree structure of FM (`axm1`, `axm2` and `axm3` of Listing 1).

The relationships between a feature and its sub-features are represented using four constants named operators: *And*, *Or*, *Xor* and *Null*. The *And* operator is used to mention that all sub-features are mandatory. When the *Or* operator is bound to a node, this node has at least one sub-feature. When the *Xor* operator is bound to a node, that represents the alternative between sub-features. Howeover, the *Null* operator indicates that the feature has no sub-features i.e. this feature is a leaf. The successors of a given feature are specified by the $SuccOf$ relation (`axm7` of Listing 1).

CONTEXT $TreeSpec$
SETS $FEATURE$
CONSTANTS $Root\ Node\ SuccOf\ And\ Or\ Xor\ Null\ Mandatory\ SuccOfTransChildrenOf$
`axm1`: $finite(FEATURE)$
`axm2`: $partition(FEATURE, \{Root\}, Node)$
`axm3`: $partition(FEATURE, And, Or, Xor, Null)$
`axm4`: $partition(Node, And, Or, Xor, Null)$
`axm5`: $Mandatory \subseteq FEATURE$
`axm6`: $Root \in Mandatory$
`axm7`: $SuccOf^{-1} \in Node \twoheadrightarrow FEATURE \setminus Null$
`axm8`: $finite(SuccOf)$
`axm9`: $SuccOf[And] \subseteq Mandatory$
`axm10`: $\langle theorem \rangle\ SuccOf[Xor] \cap Mandatory = \varnothing$
`axm11`: $\langle theorem \rangle\ SuccOf[And \cap Mandatory] \subseteq Mandatory$
`axm12`: $SuccOfTrans^{-1} = SuccOf^{-1} \cup (SuccOf^{-1}; SuccOfTrans^{-1})$
`axm13`: $\langle theorem \rangle\ (FEATURE \lhd id) \cup SuccOfTrans^{-1} = \varnothing$
`axm14`: $\forall s.(s \subseteq SuccOf[s] \Rightarrow s = \varnothing)$

Listing 1. An Event-B Context for describing a FM structure: a Context $TreeSpec$.

The structure of a FM is represented by a graph without cycle i.e. a tree. For that, we explicitly express the property that verifies if a FM is exactly a tree in which any two features are connected by exactly one path. Then, we define a relation *SuccOfTrans* to check that a FM does not contain cycles (`axm12`, `axm13` and `axm14` of Listing 1).

As mentioned before, the different relationships of a FM are formalized by a set of operators as: *And*, *Or*, *Xor* and *Null* (Listing 2). The *And* operator is used when it is bound to a node. The children of a given feature are obtained using the $ChildrenOf$ relation which is defined in Event-B

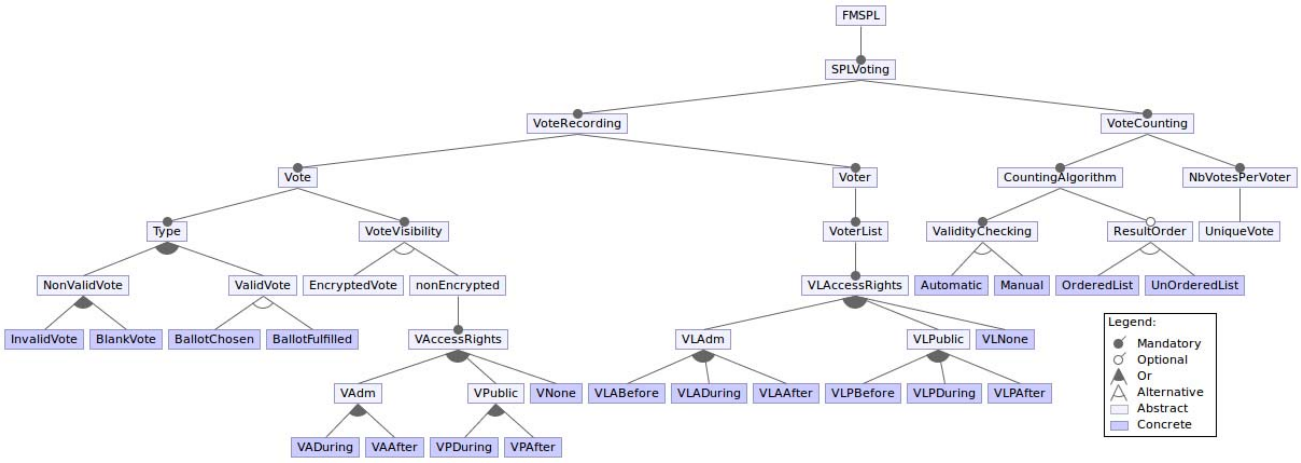

Fig. 3. Feature Model extract for e-voting Product Line.

context (axm15 of Listing 2). The two relations $SuccOf$ and $ChildrenOf$ are used at the operators *And*, *Or* and *Xor* level. The operator *And* allows one to select all children that are also the successors (axm16 of Listing 2). When the operator *Or* is bound to a feature, this feature has at least one child and all of its children are also successors of that feature (axm17 and axm18 of Listing 2). When the operator *Xor* is bound to a feature, this feature has at least and at most a child which is also one of its successors (axm19 and axm20 of Listing 1.2).

```
...
axm15: ChildrenOf ⊆ SuccOf
axm16: ⟨theorem⟩ (And ◁ ChildrenOf ⊂ And ◁
       SuccOf) ∧ (And ◁ SuccOf ⊂ And ◁
       ChildrenOf)
axm17: ⟨theorem⟩ Or ◁ ChildrenOf ⊆ Or ◁ SuccOf
axm18: ⟨theorem⟩ Or ⊆ dom(ChildrenOf)
axm19: ⟨theorem⟩ Xor ◁ ChildrenOf ⊆ Xor ◁ SuccOf
axm20: ⟨theorem⟩ ¬(Xor ◁ ChildrenOf ∈ Xor →
       FEATURE)
axm21: Null ◁ ChildrenOf = ∅
axm22: ⟨theorem⟩ ∀x·(ChildrenOf[{x}] = SuccOf[{x}])
       ∧x ∈ Xor ⇒ card(SuccOf[{x}]) = 1
axm23: ⟨theorem⟩ SuccOf ▷ Mandatory ⊆ ChildrenOf
axm24: ⟨theorem⟩ ran(Or ◁ SuccOf) ∩ Mandatory ⊆
       ran(Or ◁ ChildrenOf)
END
```

Listing 2. A part of Event-B Context for describing a FM operators: a Context $TreeSpec$.

C. Verifying E-Voting Configuration

This section describes the verification and validation of our development. In the verification step, we have the static properties of the system that can be formally verified. We verify the static properties, which are expressed in terms of Contexts, using formal proofs (proof obligations). In order to check a given instance that represents a FM configuration, we use the Context $TreeSpec$ that defines the types and structure FM (Listing 1 and Listing 2). The general e-voting FM is transformed into an Event-B context to include it in the verification process of each instance i.e. configuration. For space reasons,we represent only a part of the complete generic e-voting FM in Fig. 4, and how it is transformed to an Event-B Context (Listing 3) named $FMVoting$. In addition, each configuration to be verified can be translated to a specific Event-B Context. We use the Rodin tool that supports the application of the Event-B formal method, providing core functionality for syntactic analysis and proof-based verification of Event-B models [24].

```
CONTEXT FMVoting
SETS FEATURE
CONSTANTS Vote VoteVisibility EncryptedVote nonEncrypted
          VAccessRights VAdm VPublic VNone
          VADuring VAAfter VPDuring VPAfter
axm1: FEATURE = {Vote, VoteVisibility,
      EncryptedVote, nonEncrypted, VAccessRights,
      VAdm, VPublic, VNone, VADuring, VAAfter,
      VPDuring, VPAfter}
axm2: {Root} = {Vote}
axm3: Node = {VoteVisibility, EncryptedVote,
      nonEncrypted, VAccessRights, VAdm, VPublic,
      VNone, VADuring, VAAfter, VPDuring,
      VPAfter}
axm4: And = {Vote}
axm5: Or = {VAccessRights, VAdm, VPublic}
axm6: Xor = {VoteVisibility}
axm7: Null = {VADuring, VAAfter, VPDuring,
      VPAfter}
axm8: Mandatory = {VoteVisibility, VAccessRightsy}
axm9: SuccOf[{Vote}] = {VoteVisibility}
axm10:SuccOf[{VoteVisibility}] = {EncryptedVote,
      nonEncrypted}
axm11:SuccOf[{nonEncrypted}] = {VAccessRights}
axm12:SuccOf[{VAccessRights}] = {VAdm, VPublic,
      VNone}
axm13:SuccOf[{VAdm}] = {VADuring, VAAfter}
axm14:SuccOf[{VPublic}] = {VADuring, VPAfter}
END
```

Listing 3. An Event-B Context for describing a FM of e-voting.

```
CONTEXT FMVotingConfOK
SETS
```

CONSTANTS
axm1: $ChildrenOf[\{Vote\}] = \{VoteVisibility\}$
axm2: $ChildrenOf[\{VoteVisibility\}] = \{nonEncrypted\}$
axm3: $ChildrenOf[\{nonEncrypted\}] = \{VAccessRights\}$
axm4: $ChildrenOf[\{VAccessRights\}] = \{VAdm\}$
axm5: $ChildrenOf[\{VAdm\}] = \{VADuring, VAAfter\}$
END

Listing 4. An Event-B Context for describing a e-voting *FMVotingConfOK* configuration.

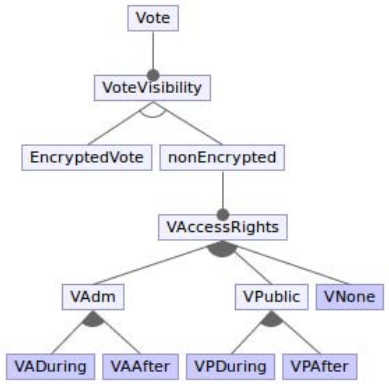

Fig. 4. Feature Model extract for e-voting Product Line to model.

An example of the proofs, we established, concerns the e-voting configuration $FMVotingConfOK$ represented in Listing 4. We have to prove the correctness of this instance using the Rodin platform with its automatic prover. For this, the proof obligations were automatically discharged without interactions to help the provers to find the right rules, as shown in Fig. 5. Furthermore, concerning the e-voting configuration $FMVotingConfKO$ represented in Listing 5, proof obligations are discharged automatically (see the proving perspective Rodin shown in Fig. 6).

CONTEXT $FMVotingConfKO$
SETS
CONSTANTS
axm1: $ChildrenOf[\{Vote\}] = \{VoteVisibility\}$
axm2: $ChildrenOf[\{VoteVisibility\}] = \{nonEncrypted, EncryptedVote\}$
axm3: $ChildrenOf[\{nonEncrypted\}] = \{VAccessRights\}$
axm4: $ChildrenOf[\{VAccessRights\}] = \{VAdm\}$
axm5: $ChildrenOf[\{VAdm\}] = \{VADuring, VAAfter\}$
END

Listing 5. An Event-B Context for describing a e-voting *FMVotingConfKO* configuration.

V. Formal Specification and Integration of Features

In this section we describe the problem of feature interactions, and the utility of a formal approach (as presented in this paper) for detecting possible interactions as early as possible in the development process; and to avoid future interactions by updating the formal FM appropriately.

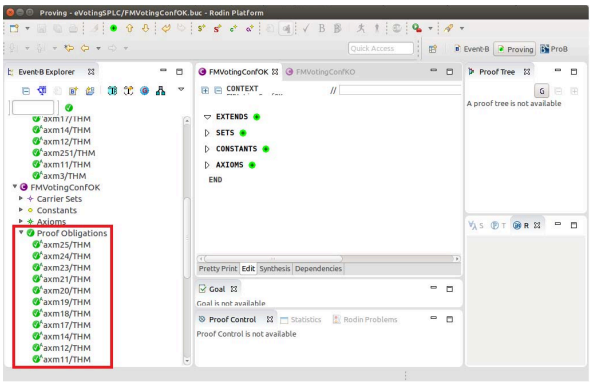

Fig. 5. Proving *FMVotingConfOK* configuration in Rodin.

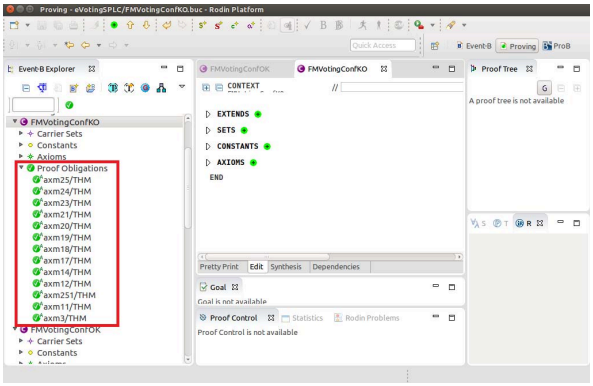

Fig. 6. Proving *FMVotingConfKO* configuration in Rodin.

A. Feature Requirements

The feature requirements are represented as Event-B Contexts using sets and relations from the Event-B domain model. Each abstract feature is represented by a Context that has a behavior. The mismatch between FM configurations and program variants is primarily caused by abstract features. In this context, the interactions between abstract features can be realized, if they share constants or/and sets. When two abstract features interact in a coherent manner, we can classify these abstract features as being *Friends* when they can always work together in a coherent manner, or *Politicians* when they must co-operate in a particular way in order for their behaviours to be coherent. Otherwise, if they cannot interact cohernetly, then we can classify them as *Enemies* [25]. We give an example of these different interactions levels.

Example 1. *Let F_1 and F_2 be two abstract features represented by two different Contexts and $nbreCandidates$ a constant that belongs to both Contexts. When $nbreCandidates \in \{1,2,3\}$ in the F_1 and $nbreCandidates \in \{1,2,3\}$ in the F_2, the two abstract features F_1 and F_2 are Friends. When $nbreCandidates \in \{1,2,3\}$ in the F_1 and $nbreCandidates \in \{3,4,5\}$ in the F_2, the two abstract features F_1 and F_2 are Politicians as they can agree on a value*

of 3. On the other hand, When $nbreCandidates \in \{1, 2, 3\}$ *in the* F_1 *and* $nbreCandidates \in \{4, 5\}$ *in the* F_2*, the two abstract features* F_1 *and* F_2 *are Enemies.*

The formal event-B models allow us to automatically identify friendly feature requirements. In this case, each feature can be developed independently, and its correctness guaranteed by refinement. Irrespective of the design decisions taken during the refinement process, friend machines are guaranteed to inter-operate correctly when they are composed. In contrast, when we detect two abstract feature requirements as *Enemies*, it is mandatory to change the FM by adding a constraint that prohibits the use of these two features, or by manual modification of the FM structure. The 3rd case is the most challenging: when feature requirements are *Politicians* we must be careful when we develop each of them in parallel. Although each formal refienement guarantees the correctness of the individual feature's behaviour, it risks breaking the coherency with the other feature(s). We currently have no automated technique for managing the complexity that arises when we have to develop a SPL instance in which there are a large number of *Politician* features.

B. Analyses of Interactions

As mentioned above, each abstract feature has a single Event-B Context that describes its behavior. This feature can have an potential incoherent interactions with other feature during refinement process. For example, the abstract feature $VoteCounting$, in the Fig. 4, allows to calculate the number of votes for each candidate in the elections. For that, we consider the presidential elections second round with two candidates and the result will be a sorted list generated automatically, that is represented with the configuration $\{VoteCounting, CountingAlgorithm, ValidityChecking,$-$Automatic\}$. Listing 6 represented the Event-B Context of this configuration.

In this Context, the *Urn* set is part of the domain model common to all e-voting systems. A *Vote* set (as can be recorded in an urn) is part of the domain model common to all e-voting systems. In the 2nd round of the election this is the first of two candidate options is represented by *OPTION1* constant. *OPTION2* constant represents the second of two candidate options in the 2nd round of the election. An empty urn is part of the domain model common to all e-voting systems is defined by *EMPTY_URN*. A valid vote, *ValidVote* constant, is part of the domain model common to all e-voting systems. but the invalid one is represented by *InvalidVote* constant. The function permitting the addition of a vote to an urn is described by *addVoteToUrn*. The result of counting the votes in the urn is stored in *result* constant. *count* constant represents the function which calculates the result of counting the votes in the urn. In the 2nd round of the election this is the candidate option having the most votes recorded in the urn. when the count is carried out (or an equality if no unique winner exists), *elected* constant record the candidate in question. *countVotesInUrnForOption* constant is used as function to calculate the number of votes for a specific option. For the exceptional case when there is no clear winner after the count i.e. we have an equality, this case is representd by *TIE* constant.

Now, we will describe the internal behavior of our valid configuration. Votes can be valid or invalid, but not both, this constraint is part of the domain model common to all e-voting systems, `axm1`. In the 2nd round of the election, only 2 candidates correspond to a valid vote, `axm2`. At the beginning of the election we need an urn with no votes recorded, `axm3`. The function permitting the addition of a vote to an urn, `axm4`. The result of counting the votes in the urn, provides a function mapping valid votes to their final count, `axm5`. `Axm6` describes the function which calculates the result of counting the votes in the urn. In the 2nd round of the election this is the candidate option having the most votes recorded in the urn. when the count is carried out (or an equality if no unique winner exists), `axm7`. `Axm8` defines an utility function to calculate the number of votes for a specific option. When there are no valid votes in an empty urn, it is represented by `axm9`. Taking a single vote from the urn and incrementing the count for the valid candidate option associated to it (if the vote is for the specified candidate option), this behavior is defined using `axm10`. `Axm11` is used to remove a single invalid vote from the urn, and do not count it. `Axm12` is exploited to take a single vote from the urn and do not increment the count as the candidate option is not the one we are lokking for. A utility function to calculate the number of votes for a specific option is defined by `axm13`. `Axm14`, `axm15` and `axm16` are used, respectively, to describe if option1 wins, option2 wins and if we have equality between the two options.

CONTEXT $presidentialElectionSecondRound$
SETS $Urn\ Vote$
CONSTANTS $OPTION1\ OPTION2\ EMPTY\_URN\ ValidVote$
$InvalidVote\ addVoteToUrn\ result\ count$
$elected\ countVotesInUrnForOption\ TIE$

`axm1:` $partition(Vote, ValidVote, InvalidVote)$
`axm2:` $ValidVote = \{OPTION1, OPTION2\}$
`axm3:` $EMPTY\_URN \in Urn$
`axm4:` $addVoteToUrn \in Vote \times Urn \rightarrow Urn$
`axm5:` $result \in ValidVote \rightarrow \mathbb{N}$
`axm6:` $count \in Urn \rightarrow result$
`axm7:` $elected \in result \rightarrow \{OPTION1, OPTION2, TIE\}$
`axm8:` $countVotesInUrnForOption \in Urn \times ValidVote \rightarrow \mathbb{N}$
`axm9:` $\forall option \cdot option \in ValidVote \Rightarrow countVotesInUrnForOption(EMPTY\_URN \mapsto option) = 0$
`axm10:` $\forall option, urn \cdot option \in ValidVote \wedge urn \in Urn \Rightarrow countVotesInUrnForOption(addVoteToUrn(option \mapsto urn) \mapsto option) = 1 + countVotesInUrnForOption(urn \mapsto option)$
`axm11:` $\forall option, urn \cdot option \in InvalidVote \wedge urn \in Urn \Rightarrow countVotesInUrnForOption(addVoteToUrn(option \mapsto urn) \mapsto option) = countVotesInUrnForOption(urn \mapsto option)$
`axm12:` $\forall option1, option2, urn \cdot option1 \in ValidVote \wedge option2 \in ValidVote \wedge urn \in Urn \wedge option1 \neq option2 \Rightarrow countVotesInUrnForOption(addVoteToUrn(option1 \mapsto urn) \mapsto option2) = countVotesInUrnForOption(urn \mapsto option2)$
`axm13:` $\forall opt, urn, c \cdot opt \in ValidVote \wedge c \in \mathbb{N} \wedge urn \in Urn \Rightarrow countVotesInUrnForOption(urn \mapsto opt) = c$

axm14: $\forall option1, option2, urn \cdot\ option1 \in ValidVote \wedge option2 \in ValidVote \wedge urn \in Urn \wedge (countVotesInUrnForOption(urn \mapsto option1) > countVotesInUrnForOption(urn \mapsto option2)) \Rightarrow elected(count(urn)) = option1$

axm15: $\forall option1, option2, urn \cdot\ option1 \in ValidVote \wedge option2 \in ValidVote \wedge urn \in Urn \wedge (countVotesInUrnForOption(urn \mapsto option1) < countVotesInUrnForOption(urn \mapsto option2)) \Rightarrow elected(count(urn)) = option2$

axm16: $\forall option1, option2, urn \cdot\ option1 \in ValidVote \wedge option2 \in ValidVote \wedge urn \in Urn \wedge (countVotesInUrnForOption(urn \mapsto option1) = countVotesInUrnForOption(urn \mapsto option2)) \Rightarrow elected(count(urn)) = TIE$

END

Listing 6. An Event-B Context for vote counting of presidential elections.

This Event-B Context of the Listing 6, shows explicitly the interactions between three abstract features namely: $VoteCounting$, $CountingAlgorithm$ and $ValidityChecking$. These features are *Politicians*, the combined requirements can be met in a coherent manner but we have to be careful during refinement that we do not introduce inconsistency. It is beyond the scope of this paper to detail the way in which the Event-B context requirements models are integrated into an Event-B machine, ready for refinement-driven design.

VI. Conclusion and Future Work

In this paper, we have presented a formal approach to verifying the correctness of a FM configuration based on variability management and modelling using the Event-B method. We have also exploited requirements engineering and domain analysis of e-voting to propose a general approach, from design to implementation of SPLs, by strengthening and underpinning the correct-by-construction process.

In future work, we plan to extend our formal approach to make it more dynamic. In other words, to permit changes to the generic SPL and specific configurations during execution of the system. The challenge is to use the dynamic (machine) part of Event-B, in order to validate the insertion/deletion/replacement of a concrete/abstract feature.

References

[1] P. Clements and L. Northrop, *Software product lines*. Addison-Wesley,, 2002.

[2] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman, "Sat-based analysis of large real-world feature models is easy," in *Proceedings of the 19th International Conference on Software Product Line*. ACM, 2015, pp. 91–100.

[3] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake, "Analysis strategies for software product lines," *School of Computer Science, University of Magdeburg, Tech. Rep. FIN-004-2012*, 2012.

[4] J.-R. Abrial, *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[5] R. Mazo, C. Salinesi, O. Djebbi, D. Diaz, and A. Lora-Michiels, "Constraints: The heart of domain and application engineering in the product lines engineering strategy," *International Journal of Information System Modeling and Design*, vol. 3, no. 2, p. 50, 2012.

[6] A. S. Karataş, H. Oğuztüzün, and A. Doğru, "From extended feature models to constraint logic programming," *Science of Computer Programming*, vol. 78, no. 12, pp. 2295–2312, 2013.

[7] S. Ripon, K. Azad, S. J. Hossain, and M. Hassan, "Modeling and analysis of product-line variants," in *Proceedings of the 16th International Software Product Line Conference-Volume 2*. ACM, 2012, pp. 26–31.

[8] G. Shen, Z. Huang, C. Tian, Q. Ge, and W. Zhang, "Feature modeling and verification based on description logics." in *SEKE*, 2012, pp. 422–425.

[9] S. Soltani, M. Asadi, D. Gašević, M. Hatala, and E. Bagheri, "Automated planning for feature model configuration based on functional and non-functional requirements," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 2012, pp. 56–65.

[10] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay, "Beyond boolean product-line model checking: dealing with feature attributes and multi-features," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 472–481.

[11] S. Mennicke, M. Lochau, J. Schroeter, and T. Winkelmann, "Automated verification of feature model configuration processes based on workflow petri nets," in *Proceedings of the 18th International Software Product Line Conference-Volume 1*. ACM, 2014, pp. 62–71.

[12] R. Muschevici, J. Proença, and D. Clarke, "Feature nets: behavioural modelling of software product lines," *Software & Systems Modeling*, vol. 15, no. 4, pp. 1181–1206, 2016.

[13] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti, "Modelling and analysing variability in product families: model checking of modal transition systems with variability constraints," *Journal of Logical and Algebraic Methods in Programming*, vol. 85, no. 2, pp. 287–315, 2016.

[14] K. Lauenroth, K. Pohl, and S. Toehning, "Model checking of domain artifacts in product line engineering," in *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 2009, pp. 269–280.

[15] W. Zhang, H. Zhao, and H. Mei, "Binary-search based verification of feature models," in *International Conference on Software Reuse*. Springer, 2011, pp. 4–19.

[16] N. Andersen, K. Czarnecki, S. She, and A. Wkasowski, "Efficient synthesis of feature models," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 2012, pp. 106–115.

[17] M. Mendonca, A. Wkasowski, and K. Czarnecki, "Sat-based analysis of feature models is easy," in *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University, 2009, pp. 231–240.

[18] B. Beckert, R. Goré, C. Schürmann, T. Bormer, and J. Wang, "Verifying voting schemes," *Journal of Information Security and Applications*, vol. 19, no. 2, pp. 115–129, 2014.

[19] S. Kremer, M. Ryan, and B. Smyth, "Election verifiability in electronic voting protocols," in *European Symposium on Research in Computer Security*. Springer, 2010, pp. 389–404.

[20] J. Davies, T. Janowski, A. Ojo, and A. Shukla, "Technological foundations of electronic governance," in *Proceedings of the 1st international conference on Theory and practice of electronic governance*. ACM, 2007, pp. 5–11.

[21] M. R. Poppleton, "The single transferable voting system: Functional decomposition in formal specification," in *Proceedings of the 1st Irish conference on Formal Methods*. British Computer Society, 1997, pp. 132–149.

[22] J. P. Gibson, S. Kherroubi, and D. Méry, "Applying a dependency mechanism for voting protocol models using event-b," in *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, ser. Lecture Notes in Computer Science, A. Bouajjani and A. Silva, Eds., vol. 10321. Springer, 2017, pp. 124–138. [Online]. Available: https://doi.org/10.1007/978-3-319-60225-7_9

[23] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," DTIC Document, Tech. Rep., 1990.

[24] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in event-b," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 12, no. 6, pp. 447–466, 2010.

[25] G. Hamilton, J. Gibson, and D. Méry, "Composing fair objects," in *International Conference on Software Engineering Applied to Networking and Parallel/Distributed Computing (SNPD '00)*, Fouchal and Lee, Eds., Reims, France, May 2000, pp. 225–233.