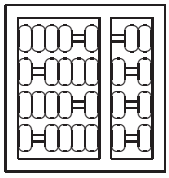

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Verifying Architectural Variabilities in Software Fault Tolerance Techniques

Patrick H. S. Brito *Rogério de Lemos*
Cecília M. F. Rubira

Technical Report - IC-09-10 - Relatório Técnico

March - 2009 - Março

Verifying Architectural Variabilities in Software Fault Tolerance Techniques

Patrick H. S. Brito[1*] Rogério de Lemos[2]
Cecília M. F. Rubira[1]

[1] University of Campinas, Brazil
{pbrito, cmrubira}@ic.unicamp.br

[2] University of Coimbra, Portugal
rdelemos@dei.uc.pt

Abstract

This technical report considers the representation of different software fault tolerance techniques as a product line architecture (PLA) for promoting the reuse of software artefacts, such as formal specifications and verification. The proposed PLA enables to specify a series of closely related applications in terms of a single architecture, which is obtained by identifying variation points associated with design decisions regarding software fault tolerance. These decisions are used to choose the appropriate technique depending on the features selected for the instance, e.g, the number of redundant resources, or the type of adjudicator. The proposed approach also comprises the formalisation of the PLA, using B-Method and CSP, for systematising the verification of fault-tolerant software systems at the architectural level. The properties verified cover two complementary contexts: the selection of the correct architectural variabilities for instantiating the PLA, and also the properties of the chosen fault tolerance techniques.

1 Introduction

Software systems that can cause risks for human lives or great financial losses should be made fault-tolerant. Software fault tolerance is an inherent aspect of strongly-structured systems [18], i.e., systems in which the structuring of redundancy is part of the actual system, thus restricting the impact of faults. Also, because fault tolerance has a global system scope, it should be related to both architectural elements (components and connectors) and architectural configurations which implement the rules by which they interact. However, the incorporation of fault tolerance into systems normally increases their complexity, making their analysis more difficult. In order to support this analysis, many development approaches suggest the use of automatic verification techniques, which constitute an important way to remove faults at specification time. Although its importance, the verification activity is considered resource consuming and can expensive the software cost. One of the possible ways to to reduce the verification effort is to maximize the reuse of formal artefacts during the verification process.

*Supported by Fapesp/Brazil, grant 06/02116-2 and CAPES/Brazil, grant 0722-07-3.

Currently, many efforts are being spent in order to achieve a bigger level of reuse. One of the main approaches present in the literature for promoting the reuse of software artefacts is called software product lines (SPL). A SPL is an approach that systematises the reuse of software artefacts through the exploration of commonalities and variabilities among similar instances [3]. One of the main artefacts in the contexts of a SPL is the product line architecture (PLA), which explicitly represents the commonalities and variabilities of architectural elements and their configurations. The commonalities are reused for instantiating the specific instances, while the variabilities are resolved through design decisions related to the choices at the PLA.

This technical report considers the representation of different software fault tolerance techniques into a PLA for promoting the reuse of software artefacts. In particular, the variabilities of the PLA capturing different software fault tolerance techniques, such as, recovery blocks and N-version programming [13], should be associated with design decisions that define, for instance, the number of redundant resources available and the different kinds of adjudicators. For systematising the verification of fault-tolerant software systems using PLAs, and increase the reuse of formal artefacts related to the verification process, we have adopted B-Method and CSP. The adoption of B-Method and CSP was mainly motivated by the explicit separation between the structural and behavioural specifications. We claim that this specification facilitates the comprehension of the formal model, as well as its adjustment for running extra verification. Moreover, the existence of a compliant model checker tool (ProB [14]) was also considered. The properties verified cover two complementary contexts: the selection of the correct architectural variabilities, and also the properties of the chosen fault tolerance techniques.

The contributions of this technical report are twofold: a PLA for software fault tolerance techniques, and the support for verifying the PLA, as well as the respective instances created from it. The rest of this technical report is organised as follows. Section 2 presents some background regarding the concepts considered in this technical report. Section 3 contextualises the proposed approach with some related work. Section 4 presents the product line architecture specified for the software fault tolerance techniques. Section 5 describes how the architectural verification is done using the ProB model checker [14]. Section 6 summarises the evaluation of the proposed approach Finally, Section 7 evaluates the the overall approach and provides some concluding remarks and future directions of research.

2 Background

This section presents some background concepts regarding product line architectures, the software fault tolerance techniques used in this technical report, and formal methods.

2.1 Product Line Architectures

Software product line is a systematic software reuse approach that promotes the generation of specific products from a set of core assets for a given domain, exploiting the commonalities and variabilities among these products [3]. Feature modeling is one of the most accepted ways to represent commonalities and variabilities at the requirements phase. A

feature is a system property that is relevant to some stakeholder and it is used to capture commonalities and variabilities amongst systems in a product line [8]. Based on a hierarchical structure, a *feature model* represents the commonalities among all products of a product line as mandatory features, while variabilities among products are represented as variable features, which is also called *variants*. Variable features largely fall into three categories: (i) *optional*, which may or may not be present in a product; (ii) *alternative*, which indicates a set of features, from which only one must be present in a product; and (iii) *multiple features*, which represents a set of features, from which at least one must be present in a product. Besides the structural relationships between features, the feature model can also represent additional constraints between features. These constraints indicate which feature combinations are valid to generate a product in a product line. Some examples of constraints are *mutually dependency*, when a feature requires another, and *mutually exclusion*, when a feature excludes another.

A key factor for successfully implementing an architectural product line approach is to structure commonalities and variabilities into a *product line architecture* (PLA) in terms of variable architectural elements, and their respective interfaces, which are associated to variants. In PLAs, software variability can be reached by delaying certain architectural design decisions, which are described through variation points. A *variation point* is the place at the software architecture where a design decision can be made. These variation points should reflect the variants of the feature model, which affect design alternatives associated to the variation points [10].

For generating a specific product, it is necessary to instantiate the PLA considering the design choices associated with particular variants. In order to support these choices, a decision model should be constructed in order to relate the possible choices of variants in the feature model, to high-level decisions of the software architecture in the form of variation points. In other words, the *decision model* documents the decisions that need to be made at the context of a PLA, and which are related to the variants of the feature model. In other words, it provides support for tuning the software architecture according to the requirements of the system [3].

2.2 Software Fault Tolerance

Fault tolerance is the ability of a system to continue its normal operation despite the presence of faults [2]. For implementing techniques of software fault tolerance, it is normally necessary to structure the use of multiple versions of software (or diversity), for preventing that design faults in one version cause system failures [5]. There are three main techniques for implementing software fault tolerance using design diversity: recovery blocks, N-version programming, and N-self-checking programming. Each technique can be realised by a *reference architecture*, which provides a proven template solution for a particular domain, as well as a common vocabulary to facilitate the communication between the stakeholders [12]. The *recovery blocks* (RB) technique combines the basics of the checkpoint and restart approach with multiple versions of a software component such that a different version is tried after an error is detected [18]. Checkpoints are created before a version is executed for providing an operational state for recovering after a version fails. A reference architecture

for tolerating a single fault using the *recovery blocks* (RB) technique is shown in Figure 1. The Switch is responsible for choosing a proper Alternate to execute the service, in case an error is detected by the AcceptanceTest. The data integrity between two executions is guaranteed by the Checkpoint element.

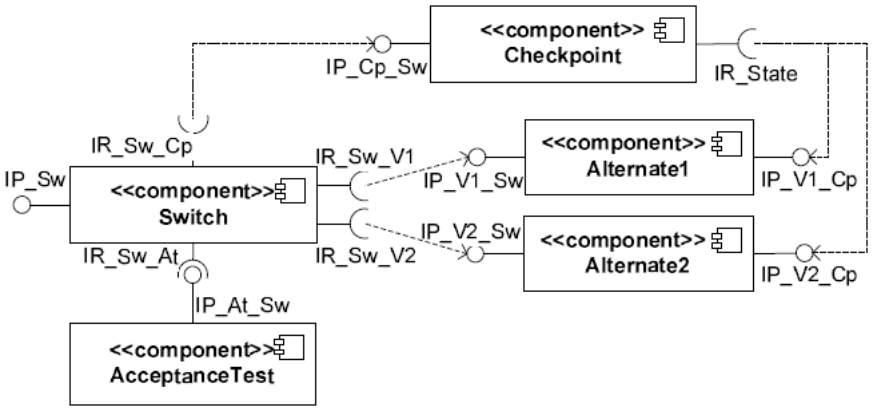

Figure 1: Reference Architectures for Recovery Blocks Technique

N-version programming (NVP) is a multi-version technique in which all the versions are designed to satisfy the same specification. The outcome is obtained using compensation by comparing the outputs of the versions through majority voting [5]. Figure 2 presents a reference architecture for tolerating a single fault using the *N-version programming* (NVP) technique. In this architecture, the Adjudicator connector is responsible to receive the results of all the three versions of components and then judge if there is a reliable result based on majority election.

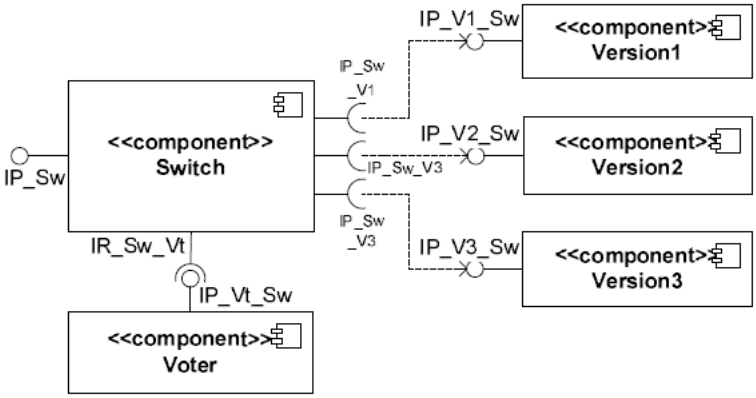

Figure 2: Reference Architectures for N-Version Programming Technique

Finally, *N-self-checking programming* (NSCP) consists on the use of multiple software versions, where each version is able to detect its own errors [13]. This detection is usually performed either using acceptance test, or by comparison. Since all versions are executed

in parallel, it is also necessary to have multiple adjudicators. Using acceptance test, each version has an individual acceptance tester for detecting errors. Using detection by comparison, it is necessary a comparator for each pair of distinct versions, in order to judge if the result is correct. A reference architecture for tolerating a single fault using the *N-self-checking programming* (NSCP) technique with comparison is shown in Figure 3. In this case all the Versions are executed in parallel, and the Switch is responsible for switching the result in case an error is detected by the Comparators, which are responsible for comparing two-by-two the results provided by the Versions[1].

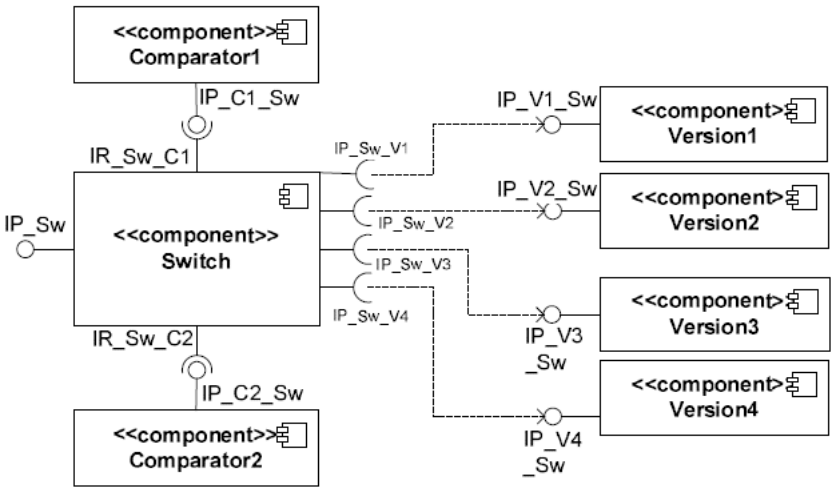

Figure 3: Reference Architectures for N-Self-Checking Programming Technique

Analysing the structure for implementing different fault tolerance techniques, their reference architectures present similarities regarding its elements. Examples of common architectural elements are: (i) version(s), used for executing the functionalities of the application; (ii) adjudicator(s), which are responsible for detecting errors in the versions' result; and (iii) switcher, used for switching the result in the occurrence of an error and for synchronising the parallel execution, when appropriate. A product line architecture for software fault tolerance techniques is presented in Section 4.

2.3 Formal Verification

Regarding the formal representation, Architecture Description Languages (ADLs) have the specific purpose of formally representing software architectures; however, these languages usually lack on support for representing specific aspects of the system. Examples of ADL's limitations concerns the representation of the state of the architectural elements, identification of the operations into the interfaces, specification of architectural scenarios, etc. For overcoming the ADLs' limitations, it is necessary to use a formal language that makes it possible to represent types in an explicit way, in order to distinguish the different states of

[1] Although [13] justifies the usage of the term 'variant', this technical report uses 'version' to avoid conflict with the notion of variant from SPL.

an architectural element. Moreover, for representing the chaining of error propagation and masking, the formal notation should also support the specification of scenarios involving the architectural elements.

B-Method is a general-purpose formal language based on set theory for specifying and verifying system models with explicit representation of the state, and a modular representation through the concept of refinement [1]. *Refinement* allows us to build a model gradually by making it more precise. The modularisation of the development facilitates the design and improves the scalability of the verification because it is conducted incrementally. Once the refinement is formally guaranteed, the properties verified at the abstract level are reused at the refined level, hence do not need to be re-verified.

A limitation of the B-Method is its inability to easily restrict the correct order for executing operations. Communicating Sequential Process (CSP) [7] is a process algebra that allows an easy representation of execution sequences, and if combined with B-Method, it compensates the aforementioned limitation [14]. As a combined solution, ProB [14] is a model checker that uses B-Method and CSP in a complementary way. In ProB, a CSP specification can be used to restrict the sequence of B-Method operations that are executed.

3 Related Work

Several contributions have proposed techniques for verifying software product lines. Most of the existing solutions focus on the verification of the feature model, intending to check the consistency of the choices used for instantiating specific products [6, 16, 17]. Examples of these violations are dependencies between features, such as the existence of mutually exclusive features. These works are complementary to ours, since we focus on the verification of the product line architectures, dealing with specific aspects of software fault tolerance. The work of Blundell et al. [6] is more similar to ours in what concerns the existence of two phases of verification; first, the verification of individual elements; and second, the verification of the elements interaction. But instead of dealing with variabilities at the feature model, we explicitly consider the variation points of the product line architecture, which reflect design decisions at the software architecture. These decisions are centred on aspects regarding different techniques for software fault tolerance.

The work of Auerswald et al. [4] has the same goal as ours in what concerns the specification of product lines for fault-tolerant systems. In that work, the authors propose a PLA with variabilities related to patterns for implementing software fault tolerance. The proposed PLA presents two types of components: *channel*, which is responsible for executing the functionalities and might be replicated; and *controller*, which is responsible for requesting services to the channels and for choosing a proper result to be returned. In contrast, the PLA proposed in our work is related to design decisions associated with software fault tolerance techniques, presenting a more detailed design with elements that play specific roles, such as adjudicator, switch, checkpoint, and the versions executed. Moreover, the solution proposed in this technical report also covers the formal representation and verification of both the PLA and the respective products generated from it. The focus of the verification approach is to separate commonalities and variabilities in order to improve the reuse of

formal artefacts, as well as the scalability of the verification.

Finally, Lutz and Gannod [15] describe experiences with tool-assisted architectural analysis of a mission-critical software product line. The authors use model checking for facilitating the identification of points of interaction in the software architecture that need to be better analysed using complementary informal analysis approaches. However, since they do not represent variation points, it is difficult to reuse the formal artefacts over different instances of the same PLA, which is also addressed by our approach. Moreover, beyond the verification of the consistency of the variants' selection, our approach also verifies specific properties regarding techniques of software fault tolerance.

4 A PLA for Software Fault Tolerance Techniques

4.1 Feature Model of Software Fault Tolerance Techniques

A feature model for the software fault tolerance techniques presented in Section 2.2 is showed in Figure 4, based on the representation proposed by Ferber et al. [9]. This feature model was derived from the work of Laprie et al. [13], which compare the main techniques of software fault tolerance. This feature model was used as the basic requirements for designing the PLA. In the figure, the root feature, SFTArchitecture, is composed of six mandatory features. The Error-processing technique feature captures a set of alternative features related to the different ways that can be employed for detecting errors. The Execution scheme feature represents the two possible ways for executing the system components: either sequential or in parallel. The Number Variants for tolerating f sequential faults represents some characteristics regarding the resources necessary for tolerating "f" faults. The Suspention of service delivery during error processing feature indicates if the error recovery technique suspend or not the execution when an error is detected. In case the execution is suspended (the Suspention feature), it is also necessary to define what is the purpose of the suspention: either for re-executing the service, or only for switching to another result. The Judgement on result feature presents how the acceptance test should be performed, either with an absolute criteria (involving the result of only one executor), or a relative criteria (involving the results of more than one executor). Finally, the Consistency of input data feature presents how the consistency of data is achieved, either implicitly through backward error recovery, or explicitly through specialised algorithms of data consistency. The lines between features represent constraints, according to the legend present in the figure. As it can be seen in Figure 4, each set of alternative features represents a variant (V1 to V7).

4.2 An Architecture for Software Fault Tolerance Techniques

The product line architecture (PLA) was designed in three steps: (i) design a software architecture for each software fault tolerance technique; (ii) identify commonalities and variabilities between the software architectures; and (iii) design the PLA based on the commonalities and variabilities previously identified. Figure 5 presents a PLA that realises the feature model of Figure 4. The AdjudicatorN is the architectural element responsible for detecting errors in the results, in order to improve the system reliability. Depending on

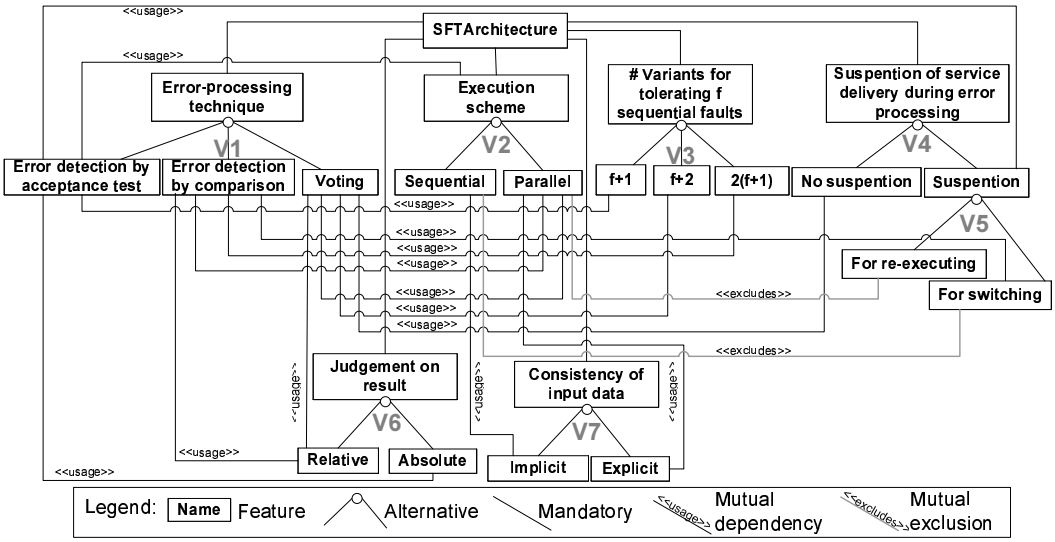

Figure 4: Feature Model of Software Fault Tolerance

the choices regarding the adopted technique, it is possible to have many adjudicators. The Switch is an architectural element responsible for reconfiguring the architecture when an error is detected. The VersionN architectural element is responsible for executing the functionalities of the application. The number of versions depends on the adopted technique, as well as the number of faults to be tolerated. Finally, the Checkpoint is an architectural element responsible for providing support for roll-back the system state, when it is appropriate. The activities of the Checkpoint includes the storage of error-free states of the Versions, as well as the restoration of the system state in case of error. The communication between the Checkpoint and the Versions is conducted through the IR_State interface. The seven variants presented in the feature model (Figure 4) were mapped into design decisions represented as variation points. These variation points, which are represented in Figure 5 as dashed grey polygons, are used to generate the decision model that is necessary during the instantiation of the PLA for generating specific products.

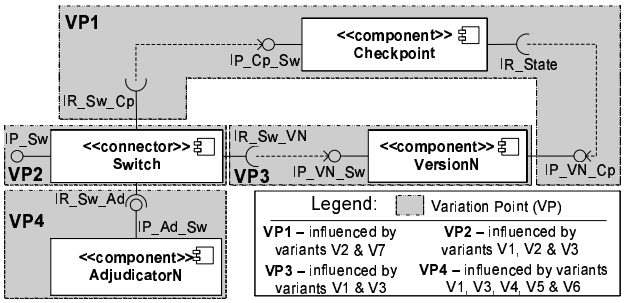

Figure 5: PLA for Software Fault Tolerance

Table 1 shows the decision model of the PLA presented in Figure 5. As it can be seen,

the goal of this artefact is to list all the combination of variabilities that do not violate the constraints specified in the feature model (Figure 4). This information is very useful for identifying the impact of the variant choices in the PLA, supporting the developer in the instantiation of a specific product with no violation of the system requirements. Each line of Table 1 represents a scenario of instantiation, containing the selection of features, and the respective impact in the software architecture. Columns V1 to V7 of Table 1 present the possible decisions in each variant of the feature model. Column STF Technique show the technique of fault tolerance that is used in each scenario. Columns VP1 to VP4 present the impact of the combination of decisions in the software architecture. Due to space restrictions, we have used an abbreviated version of the features' names to refer to their choices in the variants. For exemplifying one scenario of use, the first scenario of Table 1 shows that in case of this set of choices, the recovery block technique should be applied. These choices are: (i) acceptance test for the error processing technique (VP1); (ii) sequential execution (VP2); (iii) availability of 'f+1' versions (VP3); (iv) re-execution of the service in case of an error (VP4 and VP5); (v) absolute judgement of the result; and (vi) implicit consistency of the input data (using roll-back). The impact of these choices in the PLA is also presented in Table 1: (i) use of the Checkpoint facility (VP1); (ii) the Switch should reconfigure the architecture and use the Checkpoint when necessary; (iii) the architecture should have 'f+1' Versions elements; and (iv) the architecture should have only one adjudicator, which implements acceptance test.

Table 1: Decision Model of the PLA for Software Fault Tolerance

Variants of the Feature Model								Variation Points of the PLA			
V1	**V2**	**V3**	**V4**	**V5**	**V6**	**V7**	**SFT Technique**	**VP1**	**VP2**	**VP3**	**VP4**
AccpTst	Seq	f+1	Susp	Re-Exec	Abs	Impl	Recovery Blocks	Used	*switch* between different versions & roll-back controller	f+1 Versions	one Adjudicator: *acceptance test*
AccpTst	Par	f+1	Susp	Switch	Abs	Expl	N-self-checking Programming with acceptance test	Not used	*synchronisation* & *switch* between different versions	f+1 Versions	f+1 Adjudicators: *acceptance test*
Comp	Par	2(f+1)	Susp	Switch	Rel	Expl	N-self-checking Programming with comparison	Not used	*synchronisation* & *switch* between different versions	2(f+1) Versions	f+1 Adjudicators: *comparison*
Voting	Par	f+2	NoSusp	—	Rel	Expl	N-version Programming	Not use	*synchronisation*	f+2 Versions	one Adjudicator: *voting*

5 Verifying Product Line Architectures

5.1 Representing Variability in Software Architectures

For the representation of the PLA, shown in Figure 5, the B-Method is used for specifying the structure of the PLA, while CSP is employed for specifying its behaviour. Regarding the structural specification, we have defined a hierarchy of B-Method machines, shown in Figure 6, which explicitly separates the commonalities and variabilities of the PLA. Following ProB notation [14], the rectangles represent B-Method machines, and the arrows represent relationships between them. The CSP specifications are represented by dashed ellipses. The featureType and plaTypes machines contain setsthat store the data types necessary for representing the feature model and the PLA, respectively. The featureModel machine `uses` the data of featureTypes to represent the feature model presented in Figure 4 in terms of its variants, and the relations among features. The pla machine `uses` the data of plaTypes, and the information of the feature model (featureModel) to structure the PLA presented in Figure 5 in terms of its architectural configuration, explicit exception channels, and the respective variation points. The `uses` relationship means that the featureModel and pla machines can refer, respectively, to the shared sets of featureTypes and plaTypes for defining its invariants. Finally, each of the other four machines (recoveryBlocks, nvp, nscpAt and nscpComp) have their behavioural scenarios specified in CSP.

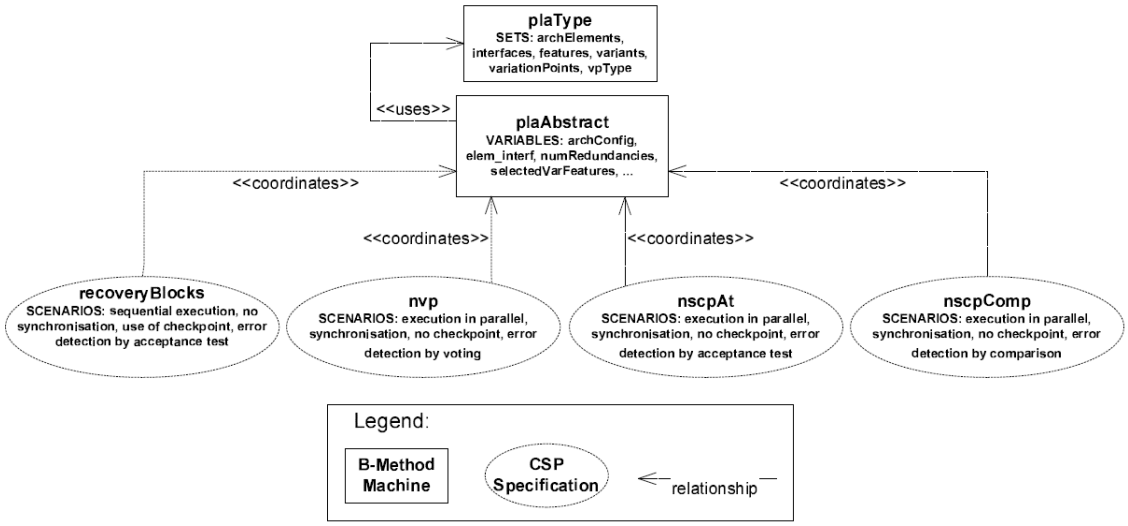

Figure 6: Hierarchy of B-Method Machines and CSP Specification

The behavioural specification of the PLA is obtained by restricting the B-Method machines with CSP specifications related to them. In the context of the pla machine, the CSP specification defines the behaviour associated with the reference architectures. For example, in the case of the recoveryBlocks machine, the CSP specification states the following restrictions: (i) only a single alternate can be executed in each time (sequential execution); and (ii) before a retry through a different alternate the system state has to be restored

using the checkpoint facility.

5.2 Properties of Interest

We have defined two complementary categories of properties to be verified: (i) decision checking, which checks the consistency between the variants of the feature model and the variation points of the PLA, according to the decision model presented in Table 1; and (ii) behavioural checking that verifies the consistency between the behavioural specification of a product against the behaviour of the reference architecture.

The *decision checking* consists of two complementary checks: (i) feature consistency, which verifies the integrity of the selected features when compared with the rules defined in the feature model (see Figure 4); and (ii) structural consistency, which verifies the integrity of the software architecture concerning the selected features. An example of a violation of the feature consistency would be the selection of "Voting" as the error processing technique, and "f+1" as the number of redundant versions, which is forbidden according to the feature model (Figure 4). An example of a violation of the structural consistency would be the existence of the Checkpoint component in a software architecture when the execution scheme should be parallel.

The objective of *behavioural checking* is to assess the consistency of the behavioural specification of a product according to the behaviour of a reference architecture that implements a specific software fault tolerance technique. The behaviour of each architectural element is assessed according to their specified role in the reference architecture. Examples of roles that are verified for the N-self checking programming technique are: (i) the Versions should be executed in parallel; (ii) the Switch should synchronise the returns from the different Versions, request the error detection to an Adjudicator, and switch the result in case of error; and (iii) the Adjudicators should detect errors in the values returned by the Versions.

6 Evaluation

One of the claims we have for the proposed approach is that since it explicitly separate type definition, structural variabilities and behavioural variability, the model is easier to evolve in two different ways. First, a common change done in pla (see Figure 6) is automatically reflected in all the fault tolerance techniques which are represented on it. Second, it is easy to define new CSP specifications at the behavioural level, which reuse the structural model to deal with other techniques of software fault tolerance.

In order to assess the evolvability of the proposed PLA towards other reference architectures, we have considered two software fault tolerance techniques: (i) *distributed recovery blocks* (DRB), which tolerates both software and hardware faults through the distribution of RBs in different locations [11]; and (ii) *consensus recovery blocks* (CRB), which combines the use of NVP and RB techniques [19]. To illustrate the evolution of PLAs, we will discuss about the adaptation of the PLA presented in Figure 5 for also supporting DRB. The adaptation of the PLA for CRB was done in a similar way. For considering the reference

architecture of the DRB technique, which is presented in Figure 7, first of all it was necessary to update the feature model presented in Figure 4. The new feature model permits the selection of two different execution schemes at the same time, as well as, two ways for suspending the execution of service delivery during error processing. For this, variants V1 and V2 had to be changed to multiple selection, instead of alternative selection. In addition, the changes in the feature model had to be reflected in the variation points of the PLA and in the respective decision model presented in Table 1. Basically, the decision model had to add another possible decision: the selection of acceptance test as error processing techniques and "f+1" Alternates, which should be instantiated twice, in different physical locations. Finally, since each RB is executed sequentially and the two distributed locations executes in parallel, the execution scheme is considered both sequential and in parallel at the same time. Finally, the PLA also needed to be extended, in order to support many Checkpoints.

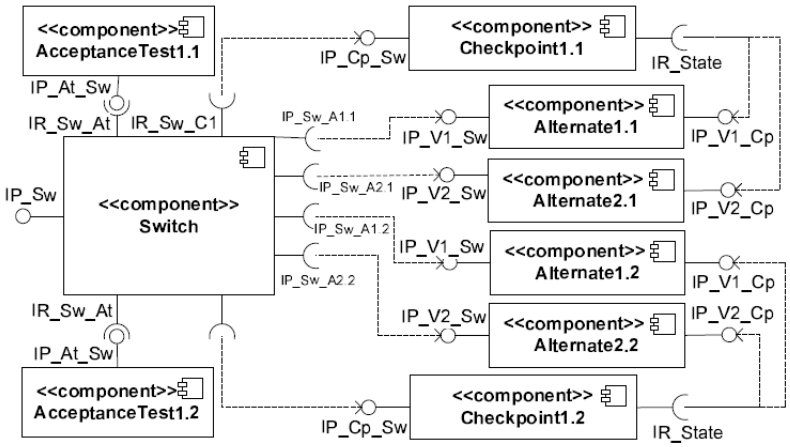

Figure 7: Reference Architectures for Distributed Recovery Blocks Technique

To evolve the formal model of the PLA (see Figure 6), it was necessary three punctual changes. First, the featureModel machine was updated for reflecting the changes in the feature model. Second, the pla machine was updated in order to modify the variation points and decision model of the software architecture. Third, it was necessary to define a new CSP specification, called drb. This CSP specifies the scenarios according to the expected behaviour of the DRB technique, which is: (i) parallel execution of primary alternate of location 1 (Alternate1.1), and the secondary alternate of location 2 (Alternate2.2); (ii) if Alternate1.1 fail, the switch should verify the result of Alternate2.2; (iii) if both fail, it is necessary to execute the roll-back in both physical locations, and (iv) repeat the steps for the Alternate1.2 and Alternate2.1.

7 Conclusions and Future Work

This technical report has presented a rigorous development approach for the formal specification and verification of a product line architecture for fault tolerance techniques. This

approach considers the representation of different software fault tolerance techniques into a PLA for promoting the reuse of software artefacts. The proposed approach combines the use of B-Method and CSP for promoting the representation of architectural elements, variants and variation points, as well as, the specification of scenarios according to the characteristic of each fault tolerance technique.

The claim being made in this technical report is that the explicit representation of architectural variabilities in B-Method reduces the effort for verifying PLAs through the reuse of formal artefacts. Due to space restrictions, we were not able to present in great detail the proposed verification approach, as well as its benefits and limitations. In the following, in order to complement the material presented so far, we summarise the main benefits and limitations of the proposed approach.

The main benefits of the verification approach concerns three important issues: (i) the reuse of formal artefacts; (ii) the scalability of the verification; and (iii) the coverage of the verified properties. The explicit separation between the specification of the commonalities and variabilities makes it possible to reuse the formal specification of the common part, and the properties associated with these. In that case, the properties verified in the formal model of the commonalities do not need to be re-verified in the refined models, which represents the variabilities according to each software fault tolerance technique. The existence of two levels of verification (commonalities and variabilities) also improves the scalability of the solution because it considers the system in parts, instead of considering it as a whole. Finally, the properties of interest cover both the verification of the PLA and the verification of products' architecture instantiated using it. The PLA is verified regarding the selection of the correct architectural variabilities, according to the restrictions of the feature model it should be consistent with. The individual instanes of the PLA are also verified through specific properties, which should be instantiated based on the properties of the reference architecture of the specific software fault tolerance technique that is used by the application.

The current limitation of the proposed approach is the focus on the design of software fault tolerance, instead of covering all the development cycle, from the requirements, passing to the analysis, design and implementation. In this way, the proposed approach documents the main architectural decisions regarding software fault tolerance and systematises the architectural design according to the decisions. In order to complement the proposed approach with support for the analysis phase, first of all it is necessary to specify a feature model which is comprehended by the client. Since we focus on the design, the variants represented in the feature model of Section 4 does not reflect real requirements of the instantiated product in the point of view of the clients. For example, instead of choosing an error-processing technique (variant V1 of Figure 4) and the execution scheme (variant V2 of Figure 4), it would be easier for the client to say if the application has critical requirements of real-time, as well as the fault model of the application. Finally, it would be necessary to define associations from the decisions at the client-level to design decisions at the software architecture.

References

[1] Jean-Raymond Abrial, Matthew K. O. Lee, Dave Neilson, P. N. Scharbach, and Ib Sorensen. The b-method. In *Proc. of the 4th International Symposium of VDM Europe on Formal Software Development (VDM '91)*, volume 2, pages 398–405, 1991.

[2] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall, 1981.

[3] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based Product Line Engineering with UML.* Addison-Wesley, 2002.

[4] Marko Auerswald, Martin Herrmann, Stefan Kowalewski, and Vincent Schulte-Coerne. Reliability-oriented product line engineering of embedded systems. In *Proc of the 4th International Workshop on Software Product-Family Engineering (PFE'01), LNCS 2290*, pages 83–100, London, UK, 2002.

[5] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. on Software Engineering*, 11(12):1491–1501, 1985.

[6] Colin Blundell, Kathi Fisler, Shriram Krishnamurthi, and Pascal Van Hentenryck. Parameterized interfaces for open system verification of product lines. *Automated Software Engineering (ASE'04)*, 0:258–267, 2004.

[7] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[8] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process Improvement and Practice, John Wiley & Sons*, 10(2):143–169, 2005.

[9] Stefan Ferber, Jrgen Haag, and Juha Savolainen. Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In *Proc. of the Second International Software Product Lines Conference (SPLC), LNCS 2379*, pages 37–60, 2002.

[10] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[11] K. H. Kim and Howard O. Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Trans. on Compututers*, 38(5):626–636, 1989.

[12] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] Jean-Claude Laprie, Jean Arlat, Christian Béounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *IEEE Computer*, 23(7):39–51, 1990.

[14] M. Leuschel and Michael J. Butler. Prob: A model checker for b. In *Proc. of Interfational Conference on Formal Methods (FME'2003), LNCS 2805*, pages 855–874. Pisa, Italy, 2004.

[15] Robyn R. Lutz and Gerald C. Gannod. Analysis of a software product line architecture: an experience report. *Journal of Systems and Software (JSS)*, 66(3):253–267, 2003.

[16] Prasanna Padmanabhan and Robyn R. Lutz. Tool-supported verification of product line requirements. *Automated Software Engineering (ASE'05)*, 12(4):447–465, 2005.

[17] Michael Poppleton. Towards feature-oriented specification and development with event-b. In Peter Sawyer, Barbara Paech, and Patrick Heymans, editors, *13th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 07), LNCS 4542*, pages 367–381, 2007.

[18] B. Randell. System structure for software fault tolerance. In *Proc. of the International Conference on Reliable software*, pages 437–449, 1975.

[19] R.K. Scott, J.W. Gault, and D.F. McAllister. The consensus recovery block. In *Proc. of Total Systems Reliability Symposium*, pages 74–85, December 1983.