A Conceptual Model for Product Family Testing

Antti Tevanlinna and Raine Kauppinen

University of Helsinki
Department of Computer Science
P. O. Box 26 (Gustaf Hällströmin katu 2b)
FIN-00014 UNIVERSITY OF HELSINKI
{Antti.Tevanlinna, Raine.Kauppinen}@cs.helsinki.fi

Abstract. The promoters of product family engineering promise decreased implementation costs and time to market. With the assumption that also testing can benefit from a product family orientation, testing methods and strategies suited to the approach have been proposed. A question of how the special characteristics of product family engineering and its methods should be leveraged is at the heart of product family testing and has been addressed in the ESAPS, CAFÉ and Families projects.

In this contribution, we present a conceptual model of product family testing. We base our proposition on the research results of three components needed for realization of product family testing: general testing theory, usable methods to design and test product families, and overall product family engineering process.

We recognize the main tasks in product family testing. Current reference models for product family engineering include testing, but there is little indication of the tasks performed during testing. On the other hand, the developed methods and processes are narrowly scoped to a specific testing objective. There is a gap to be filled from the objective of cost-effective quality assurance to definition of multi-site, multi-team, and multi-product asynchronous development.

We discuss the testing needs, benefits, and drawbacks related to the different tasks of the overall process. We also present the main work product flows in a product family and the relation of different tasks. Our model serves as a basis for relating the high level objectives of product family engineering to the detailed methodologies required to execute the everyday work, and it can be used as a starting point for definition of testing processes in the product family engineering context. It can also be used in test process improvement and as a tool for relating new and old product family testing knowledge.

We relate the current product family testing results with the proposed model. Our results suggest that many significant aspects of product family testing have not been publicly described. We hope to raise awareness of the problems to be solved.

1 Introduction

A *software product family* is a set of software products that share common features. The commonality across members of a family is useful in the design and implementation of products. It is possible to develop products as a family of applications instead of considering the applications as separate. The commonality of the products can be managed conceptually [1,2], technically [1,3,4], and with defined processes [5]. Thus, a possibility of large-scale reuse is presented.

Product family testing is a multi-faceted problem that has connections to regression testing, testing of incomplete programs, and efficient use of reusable test assets. The product family members can be seen as variants of each other, which makes regression testing inviting. The products are created based on a semi-finished skeleton of a family, and therefore we would like to be sure that the common skeleton is free of errors. Finally, because of common and extensible assets on every abstraction level from requirements to implementation, we are interested in creating reusable test asset patterns that can be cost-effectively implemented in product family wide testing.

We recognize areas not yet studied in product family testing. Current reference models for product family engineering include testing, but there is little indication of the tasks performed during testing. On the other hand, the developed methods and processes are narrowly scoped to a specific testing objective. There is a gap to be filled from the objective of cost-effective quality assurance to definition of multi-site, -team, and -product asynchronous development, which is inherent to many companies specializing in product families. It is difficult to grasp the overall picture of product family testing. This has slowed down the development of process and methodology results in the field as well as it has made their evaluation difficult. We claim our model fills the gap and is a starting point to define product family testing processes and methodologies.

In this contribution, we propose a conceptual model for product family testing and discuss a testing process that can be used to execute testing according to the conceptual model. We base our proposition on the research results of three components needed for realization of product family testing: general testing theory, usable methods to design and test product families, and the overall product family engineering process.

We relate the current product family testing results with the proposed model. Our results suggest that significant aspects of product family testing have not been publicly described. We hope that this paper will raise awareness of the problems to be solved, and that it can used as an entry point for future product family testing.

The rest of this paper is organized as follows. Sections 2 and 3 discuss testing in the product family context and present the high level model for product family testing. Section 4 discusses the effects of different product family testing strategies and the optimization of test processes. In Section 5, test process implementation is discussed. Finally, problems to be solved in product family testing are highlighted in Section 7, and Section 8 summarizes and concludes this paper.

We refer to the guidelines of design science [6] and illustrate the contribution of our research. This work constitutes a design contribution based on the following:

– Design as an artefact
  The conceptual model presented in the paper is the main artefact produced. It identifies the essential process-related components that must be addressed in product family testing. The process implementation is a derived artefact. It is an instantiation of the model providing a full high-level product family testing process.
– Problem relevance
  The conceptual model describes the context of product family testing process which is not discussed in current literature. The current understanding of the overall product family testing process is very limited. The high-level understanding is required to provide detailed testing processes that are effective.
– Design evaluation
  The conceptual model is evaluated through applying it to define a sound product family testing process and to evaluate other work in product family testing. Our evaluation is thus descriptive.
– Research contributions
  The conceptual model and the process implementation provide the first description of a holistic product family testing process. The contribution lies within the foundations of product family testing.
– Research rigor
  The conceptual model is extracted from former work in product family testing and general testing theory. The evaluation is performed against other product family testing research.
– Design as a search
  The conceptual model recognizes the main issues and strategies in product family testing. These sub-artifacts can be used in designing product family testing processes and to evaluate other product family testing research. We posit the conceptual model is satisficing as the model is usable for both the derivation of a process and the evaluation of other product family testing process.

2 Testing in product family context

Testing-related results include process frameworks for the product family approach. Frameworks by the ESAPS and CAFÉ projects [7,8], SEI [9], and ideas of reusable testing assets and their management [10] are examples of recent research.

Essential product family engineering activities are shown in Figure 1. The division in domain and application engineering is captured in *core asset development* and *product development* activities. Product development is a consumer of *reusable assets* produced in core asset development. There, final products are created by leveraging the reusable assets. Feedback from product development to core asset development is essential as is the overall management of product family engineering.

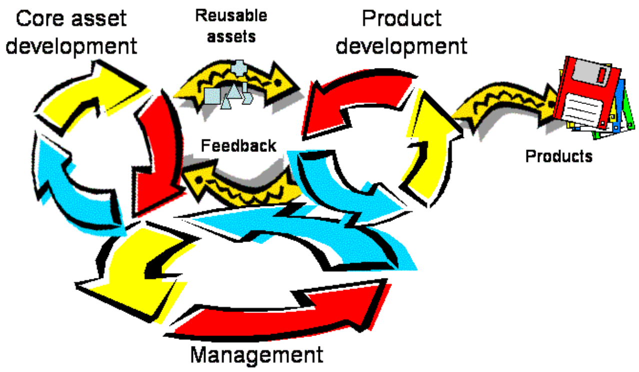


Fig. 1. Essential product family engineering activities [5].

Product family testing requires careful planning and a well-defined process [11], and testing should be incorporated with the realization of software in every phase of a process to produce high-quality products. Effective fault recovery is especially important when software components are being constantly reused, as is the case with the product families. Defects pose a strategic risk especially in domain engineering as a defect in a reused asset is copied to every product using the asset [9]. However, testing does not integrate smoothly into the division of domain and application engineering tasks in the product family approach. Many participants in the CAFÉ project [7] have recognized the difficulties of efficient division of both process tasks and test assets between domain and application engineering [12,13][1]. Integration of the V-model of testing and the product family engineering reference models is an example of this. The proposed ESAPS, CAFÉ and SEI frameworks do not fully integrate the testing levels of the V-model into the development phases although testing is supposed to be performed according to the levels of the model. The division of tasks and responsibilities seems to be somewhat easier in product assembly than in testing of created products. In practice, the integration of the testing processes to the general product family engineering process has proved to be the most problematic issue.

The V-model of testing is problematic in the context of product families. The V-model is not incremental or parallel, issues which are key principles of

[1] Product family testing has been studied extensively in ESAPS, CAFÉ, and Families. Unfortunately many of the results are not publicly available and therefore those results cannot be cited. That work includes, among others, results by Erwin Engelsma and Kari Känsälä.

product family engineering processes. The model has also received criticism for the lack of feedback to test and software design. Especially the crucial issue of design for testability cannot be modeled with the V-model. In CAFÉ the V-model was augmented to provide feedback and to model both incremental and parallel processes. It is currently unclear how the testing process of each of the testing phases (unit, integration etc.) is integrated with the overall product family engineering process.

We can characterize the main problems in product family testing from two viewpoints. First, domain assets are tested and a *domain tester's problem* arises. A domain tester looks toward application testing trying to minimize the testing needed. For the tester, it is difficult to deliver software that works correctly in all the possible usage scenarios, all of which have not yet been specified. Then in application engineering, the created products are tested, and the *application tester's problem* arises. An application tester looks at domain testing and testing of other applications trying to figure out how testing can be made adequate for the particular product under test. It is hard to decide how much we can rely on testing of the family performed during its lifespan and what amount of testing is required in application testing.

A domain tester will, for example, usually find complete integration and product testing according to the V-model infeasible because the implementation of variations is not yet specified. Testing the combinations of components leads to exponential growth of tested configurations [10,14].

To give an example of the application tester's problem, let us suppose that the fifth product in a product family is being implemented. It is likely that most of the assets have already been tested in the other products using the assets and in domain engineering phase, but how much can the tester rely on the prior testing of the reused assets? Trusting blindly on the reliability of a reused asset can lead to failures in products and sometimes serious consequences [15].

There are four strategies to model product family testing and to integrate testing into product family engineering:

1. *Testing product by product* relies on application engineering to deliver well-tested products. This kind of approach was used by Nokia when implementing a family of mobile browsers [16]. The approach is not product family oriented and it mitigates the benefits of reuse, but it guarantees the quality of the products. The testing process of the approach is product-specific keeping management simple.
2. *Incremental testing of product families* is an approach that exploits the commonalities of the products. In this approach, the first product is fully tested and all following products are tested using regression testing techniques. This approach can also be refined to test features of a family so that a complete test cycle is applied according to the levels of the V-model for every feature in the family.
3. *Reusable asset instantiation* provides an approach that resembles the implementation strategy of product family engineering. In the approach, test assets are created as extensively as possible in domain engineering. Variation

in products and their testing needs is anticipated by creating for example document templates and abstract test cases. In application engineering, a full testing process according to the levels of the V-model is instantiated. The concrete test assets are used as is and the abstract assets are extended or refined to test the whole product in application engineering [17].

4. *Division of responsibilities* is based on dividing the testing on the levels of the V-model between the different engineering units. One division could be to have common components unit tested in domain engineering. When an application engineering unit assembles a product, it is responsible for component integration, system and acceptance testing.

The approaches are illustrated graphically by examining the activities in application engineering in Figure 2. The approaches of testing product by product and reusable asset instantiation fall into the first category where the V-model is fully instantiated for every product. Incremental testing relies on testing of the previous products. Only new features are tested according to the levels of the V-model. The last category depicts the division of responsibilities approach. In this example, unit testing and some integration testing has been performed in domain engineering before the first product is implemented. The rest of integration testing and testing on high levels of the V-model are performed in application engineering. If there are some product-specific assets that have not been tested in domain engineering (or possibly in other products), then the assets are tested on all the levels of the V-model in application engineering.

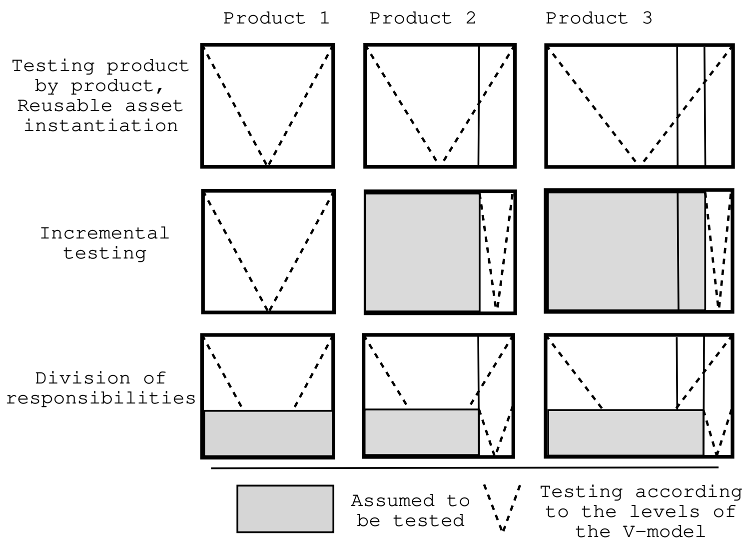


Fig. 2. The relation of the integration strategies.

The presented methods are not exclusive in use. It is more likely that they are mixed and matched to test the right aspects of the right assets when it is cost-effective. Generally, incremental testing is appealing when a product family contains well-defined decoupled features. Incremental testing is not appealing when several variable cross-cutting features, such as performance or reliability, exists. Testing of such features can seldom be confined nicely into increments. Similar advantages and disadvantages can be found from every strategy.

Most of the current knowledge of traditional testing is still relevant. Product family can be tested applying object-oriented testing methods. For example, the test pattern collection by Binder [18] will provide a solid ground for product family testing when the context is understood. The same that applies to implementation will apply to testing: the product family approach creates new methodology, but the low-level methods, such as object-orientation or state-based testing, stay the same.

3 A conceptual model for product family testing

We propose a *conceptual model for product family testing* (Figure 3). The model is based on the division between *domain and application testing*, and *division of testing responsibilities between them*. In both domain and application testing, *incremental testing* is performed. *Product-specific testing* is required to test the unique variance in products. *Reusable test assets* flow from domain to application testing and *feedback* and *opportunistic reuse* flow from application to domain testing. Also *synchronization and traceability* have to be maintained and are related to both domain and application testing. *Organization* and *tools* span over the whole model. Concerns of *process and management* are formed from the activities required to realize and maintain testing according to the elements of the model. We discuss the elements of the model in the following paragraphs.

Division of responsibilities is the basis for testing in the whole product family scope. Testing responsibilities are divided between the different test stages and organizations, for example, based on the levels of the V-model. Domain engineering could be responsible for unit testing and application engineering responsible for integration, system and acceptance testing [19]. Trying to do as much testing as possible in domain engineering will ultimately lead to a need of reference products or other types of example applications, which may prove costly to discard, and a balance must therefore be reached. In application engineering testing has to be complemented with product-specific testing when the product contains features neither included in the family or in other products.

Testing a product family differs from traditional testing by the inherent presence of reuse. When testing a single product it is directly possible to design and implement concrete test assets that contain test plans, test suites, and test reports. In the product family approach, testing can be based on generating, managing and reusing test assets that can be applicable to several products to reduce the costs. In a way, the same modifications that have been done to the product realization assets, such as architecture and components, should be

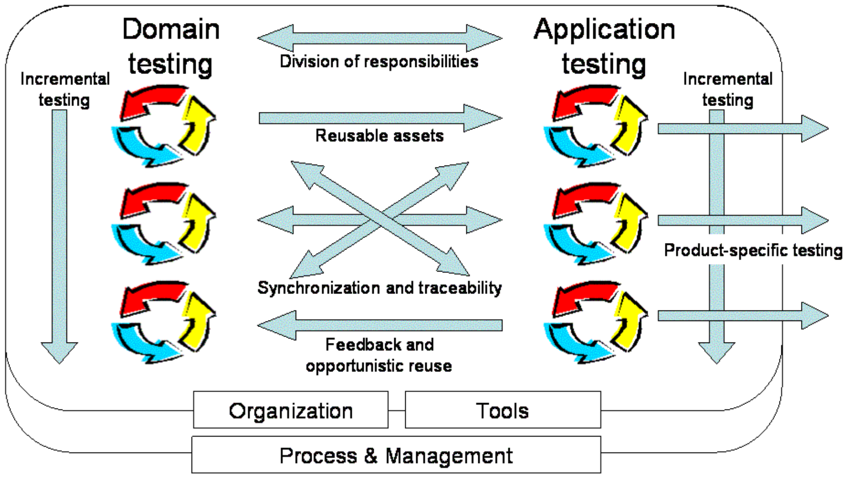


Fig. 3. A conceptual model for product family testing.

done to test assets [10,17]. Several approaches that create test cases or specify coverage criteria including product-specific variability have already been proposed [20,21,22,23].

Based on the previous we define domain testing and application testing. Our definition is broader and more specific than the definition adopted by the Families project (Appendix ABC). Domain testing encompasses the activities related to the development of test assets and test execution performed in the domain engineering assuring the quality of the product family assets and supporting application testing. Application testing encompasses the selection and instantiation of test assets of the family, the production of additional product-specific tests, and the execution of testing performed in application engineering to assure the quality of the product produced.

Feedback is needed from application testing when reusable test assets are used. It is impossible to fully assess the quality of the reusable test assets in domain engineering. This is because of the assets must be instantiated and used to test a product before their real quality can be determined. Feedback from application engineering is therefore essential to both reusable test asset production and maintenance. A similar flow is formed from opportunistic reuse of test assets produced in application testing. Provided that the application test assets are reusable in the family context, the assets can be promoted to be a part of the family asset base.

Synchronization of test processes and of both product realization and test asset lifecycles is important in the product family context. When testing responsibilities are divided between teams, parallelism is possible. This however adds to the complexity of test process management. Fluent checkpoints and handouts of tested assets between teams are essential to avoid slack, missed deadlines, and duplicated work when waiting for other testing teams. Clear information about test statuses and histories of a product family asset are needed in software production as well as in testing.

Management of testing becomes overly difficult if the testers cannot tell the difference between test assets that belong to the domain and assets that belong to a product. Related to synchronization, the inherent commonality and variability of software in product families require traceability thorough the product family from requirements to implementation and test assets. One way to create traceability between implementation and test assets is to have similar structures for both [10]. Traceability in a product family is also helpful in testing and it can be used to select the appropriate test assets for every situation during testing. This requires advanced implementation and test asset repositories.

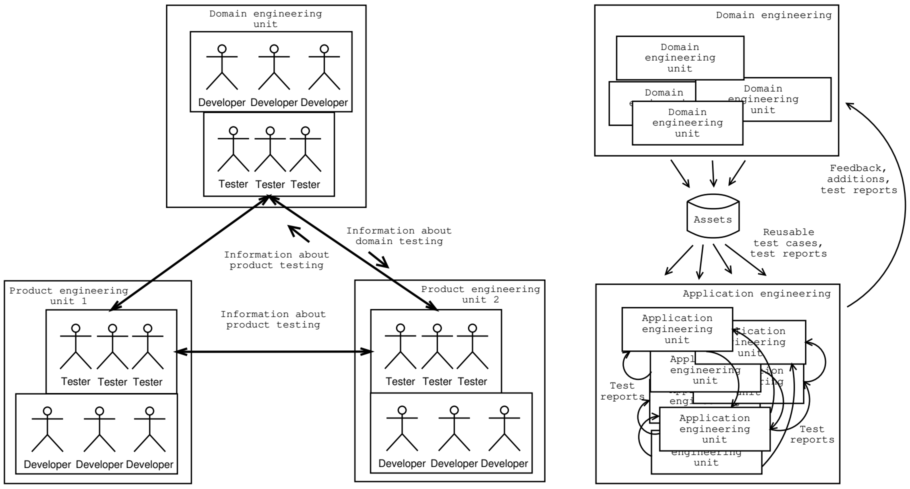
Incremental testing is a part of practical testing work in the product family context, where prior testing of assets is leveraged. If a common function has worked correctly in a former product, it is probable that it will still work correctly in a later product. The approach used is to exploit the commonality of the products using regression testing techniques [24,25]. Incremental testing is applied in both domain and application engineering by common software assets or by its features respectively.

The structure and size of an organization is another aspect that should be taken into account when defining product family testing processes. Figure 4(a) represents an example structure with a testing team in every application engineering unit and in the domain engineering unit. The product testing teams need to be aware of what is tested in the other products and what is tested in domain engineering. The domain testing team needs to be aware of what is tested in the product testing teams.

In this example, to exploit the testing performed in domain engineering and other products, communication is needed to avoid overlaps and omissions in the coverage of testing. When the product family grows, the amount of information exchanged will be large as in Figure 4(b). If a three-layer organization of product population [26], multiple product families, and applications as described by Bosch [1] is introduced, the communication is again multiplied. Generally, the Brook's law of communication applies here. Lack of coordination is also emerging. By default, there is no all-knowing party to tell what has been tested and what still needs to be tested.

An important factor in product family testing is automated testing and the use of testing tools. This decreases the effort of reusing test assets and simplifies the management of complex testing processes. Also, the extra cost of automating tests is justified because the common tests may be extensively reused.

Processes and management have special importance in product family testing. Large product families, or even product populations, require a huge amount of effort to build. In practice, large product families are built by distributed organizations located in different places, sometimes even in different countries. Incrementation, division of responsibilities, and reusing test assets require carefully defined processes and management in this context. In our model, pro-



(a) a small organization

(b) a larger organization

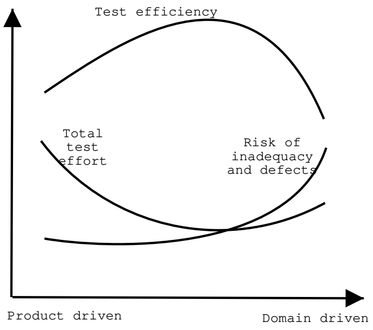
Fig. 4. Examples of organizations and asset flows in them.

cesses and management include the orchestration needed to realize and maintain the core elements.

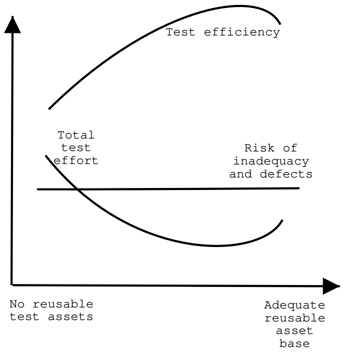
4 Optimizing product family testing

In traditional testing of a single product, we compromise, among other, between effort spent and remaining defects. Product family testing introduces new compromises: division of test responsibilities between domain and application engineering, effort put into developing reusable test assets, and amount of incremental testing of evolving functionality included in the family. Compromises these factors lead to effects in effort, risk, and test efficiency shown in Figure 5. Effort is the total effort spent in all testing performed on the family scale. Risk of defects in products is caused by unsafe testing methods. Counter-proportional to both, test efficiency raises when of the product family testing produces high quality with smaller effort.

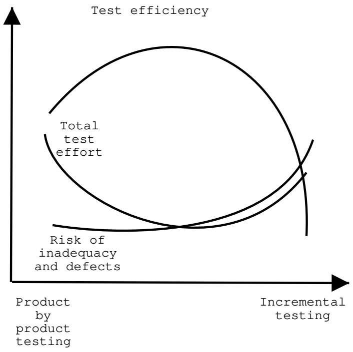
Testing performed solely in application engineering does not differ from traditional testing (Figure 5(a)). If the test process finds faults effectively, test adequacy is good with little risk. The total cost is high since a testing process is needlessly repeated for all the common assets and features. Theoretically, it is cost-effective to shift testing towards the domain side. This corresponds to the processes in construction phases, where we build common requirements, architectures and realization assets. Similar cost-efficiency can be achieved by testing



(a) effect of product and domain driven testing



(b) effect of reusable assets



(c) effect of incremental testing

Fig. 5. Effects of product family testing strategies.

the common assets once as they are built only once in implementation. Moving testing towards domain engineering lowers the overall costs at first because duplicate work is removed. The total costs will eventually rise because of creating overly general test assets and trying to cover all the possible variation in the family. The risk of inadequacy will rise when moving toward domain driven testing as functionality in certain applications may not be tested although it contains defects.

For example, in a framework-based approach to implement a product family, the application framework is the core of the product family and should be well-tested before any applications are derived from it. Ideally the testing of the applications derived from the framework should be based on the testing of the framework [27]. Integrating the framework first will also simplify the explosion of tested configurations caused by the variability [14].

The current view of product family testing is often product-oriented meaning that domain assets are tested through products using them. For example, reusable test assets must be instantiated to test a product before the assets can be leveraged. Testers trying to test domain assets before releasing them to application engineering must rely on traditional testing techniques.

In practice, it is impossible to define all relevant test cases for a reusable product realization asset because parts of its structure or logic are yet to be defined. Trying to develop all possible test cases in domain engineering is as hard as trying to capture all the aspects of a target domain in domain engineering. There are bound to be domains and organizations where this is not possible. For instance, in Nokia Mobile Phones testing is performed predominantly in domain engineering because variation between the different models is only moderate [28]. On the other hand, in Nokia Networks much more testing must be performed in the application side as product variation is large due to heterogeneous hardware.

Reusable test assets (Figure 5(b)) lower application engineering effort when executing a test process. Capturing the commonality of the test processes in application engineering efforts and producing the required common artifacts only once removes duplicate work. Therefore, the total effort decreases to a certain point the more reusable test assets are leveraged. Like the realization assets, some test assets must be made partial and abstract when they are developed in domain engineering. An extra price for reusability and abstractness must therefore be paid in test asset design and implementation. Eventually, the total cost will start to increase because of trying to support needs of an overly wide range of product-specific testing. Like in product family design and implementation, the test assets will become too abstract, they will not fulfill the real needs of application engineering, and they will become difficult to use.

Isolated from the effects of other tradeoffs, reusable assets have no impact on the adequacy provided that testing is product by product testing. A repeatable test process provides the same adequacy for the products. Then, the reusable test assets save costs, but do not increase the test adequacy, which is defined by the test process. The test objectives are defined by the process, not by the assets available. It can however be argued that reusable assets raise quality as more testing can be performed with the same budget.

Current methodology of reusable test asset instantiation can lower the cost of testing one product significantly. Reusable asset instantiation cannot be the only optimum in terms of cost if a complete testing process is assumed to be realized for the whole product. For the reused assets, this is repetitive and becomes inefficient quite soon.

In a practical world, the reused assets are not retested in every product. Some safety-critical domains require complete retesting for every application, but for most cases selective regression testing strategies can be used. Again, relying on the prior testing of the assets and features in an application removes duplicate work and cost are lowered (Figure 5(c)). Incremental testing has boundaries defined by the cost of the used test case selection strategies and their safety; that is, how much effort is put into selecting the relevant tests and how many of the introduced errors the strategies discover. Generally, the cost of analysis increases and the safety decreases the more duplicate effort put into retesting of already tested functionality is removed in application engineering. It is sometimes even hazardous to think that excellent modularity and rigorous variability management would lead to a situation where errors exists only assets created in the future once the common assets are tested. This is not true in general case as a defect can be resident in many configurations and manifest itself only in one.

Based on these considerations it is possible to find a balance between the use of different strategies in product family testing. Typically the optimum should reside closer to domain driven incremental testing supporting application engineering with reusable test assets.

5 Process implementation

Our conceptual model is at a high abstraction level. It is not, however, as general as the high-level models of SEI and CAFÉ. In this section, we provide a connection between our model and detailed strategies by discussing a high-level product family testing process. We do not prescribe the low-level testing aspects, such as testing methodologies or coverage criteria, in the various testing tasks. The detailed integration of other considerations is left to the user of our model and this process.

Resembling the overall product family engineering process (See Section 2), the testing process implementation is divided into three process areas: domain testing, application testing, and test management. For every process area, we propose the essential test process activities needed to implement testing according to our model.

We have modeled product family testing as a process that delivers tested products based on requirements and design. We have factored out requirements engineering and different kinds of design from our model by representing their results as external inputs. Domain engineering has an input of domain requirements and design and an output of reusable assets. Application engineering takes requirements, design, and reusable assets as input and produces the products. Activities relating to the synchronization and cooperation of the different tasks are allocated to the management process area. Finally, a family asset repository represents the accumulated assets of the organization.

Besides a testing cycle for every product and the use of reusable assets, incremental processes with feedback and synchronization between tasks and different organizations are important. This is included in our model as explicitly modeled exchange of test reports. When tests are executed, test reports indicating the relevant information are submitted to the asset repository. The gathered history data is exploited in both domain and application testing to facilitate testing targeted specifically to the relevant aspects.

Coordination is needed between the many different tasks in product family testing. This calls for test management including rigorous planning and analysis of testing [13]. The most important task in product family test management is the selection of the testing strategies described in Section 2 and overseeing that the testing follows the prescribed test plan. Test management handles the following tasks:

1. **Process definition and improvement.** The distributed and multi-team effort requires thorough planning. Institutionalize a testing process covering the tasks performed and keep it up to date.
2. **Analysis variability and allocation of testing responsibilities.** The division of testing responsibilities aids in avoiding overlaps and gaps in test coverage. Variability in the products of the family must be recognized as basis for the division. Allocate the testing responsibilities based on the variability and the used test reference model. This division can be made e.g. according to the levels of the V-model and the used technical variability realization mechanisms.
3. **Anticipation and division of testing of increments.** New increments in the functionality can be thoroughly tested when they are used for the first time in application engineering or by using reference products in domain engineering. Create a plan to divide the testing of the increments between the product and domain test cycles.
4. **Reusable test asset scoping.** The reusable test assets need defined boundaries to maintain their cost-efficiency. Guide the creation of reusable test assets by establishing a scope.
5. **Progress monitoring.** Product family testing is a multi-organization process that needs careful guidance. Monitor progress in testing with respect to the test plans.
6. **Test assets maintenance.** Any testing performed in application engineering may be relevant in the whole family scale. Also, application testing suffers if reusable assets have poor quality. Promote relevant test assets to the family and require feedback from application testing to facilitate adequate domain driven testing and the use of reusable test assets.

Domain engineering produces reusable software assets to be leveraged in application engineering. With effort in domain engineering, we try to minimize the total cost of the applications by lowering the effort to build a product based on the family. As already discussed, the assets built in domain engineering should be extensively tested in various expected configurations. Optimally, test assets should be reusable so that they can be complemented and leveraged also in application engineering (Figure 6).

Note that we separate domain test specification and implementation from test execution in domain engineering (Figure 6). This emphasizes that a complete testing cycle including test execution must be included in domain engineering.

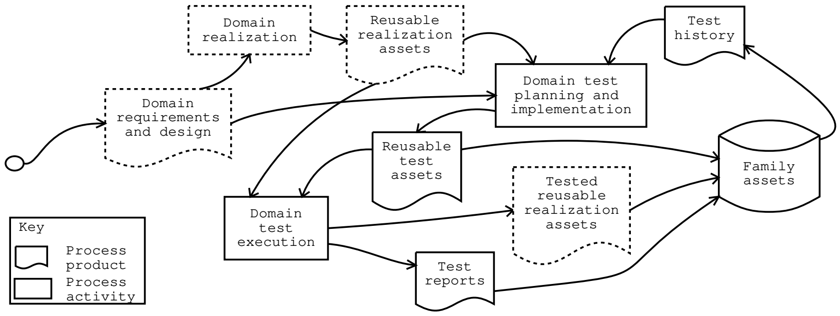


Fig. 6. The test-related activities in domain engineering.

The tasks in domain testing are as follows.

1. **Design and implementation of test assets to test the structure and functionality of the family.** Domain engineering produces a structure to partially implement functionality included in the family. Design and implement test assets to test the common software while leaving open variation points to include product-specific testing aspects.
2. **Test execution using the produced assets in a representative set of configurations of the family.** Reusable assets are to be tested thoroughly to minimize the need of application testing. Execute the available executable test cases created in the previous task in a representative but minimized set of configurations. The configurations should cover the anticipated structure of future products using, e.g., combinatorial approaches [29] or operational profiles [30]. Also, past product configurations may be used here as test beds for new assets.
3. **Test execution of the produced increments in a representative set of reference products of the family.** New functionality produced in domain engineering can be tested using reference and work products. Need for application testing is decreased when increments are tested in domain engineering. Analyze the testing performed to deduce what new functionality requires testing and test the selected functionality.
4. **Test analysis and asset maintenance.** Quality of the reusable test assets is determined when they are used in concrete testing. Analyze the testing performed in domain and application engineering. Analyze coverages, usability of the assets and level of reuse. Upkeep the assets over time to facilitate adequate domain driven testing and to better support of application testing.

Application engineering produces products using the assets created in domain engineering. In many cases additional assets must be created to accomplish product-specific functionality. The last line of defense against faulty products is application testing, where the realized product is tested. Application testing in product family engineering is based on instantiating reusable test assets, using regression testing techniques to leverage testing of other products, and finally complementing the product testing with the required product-specific testing (Figure 7).

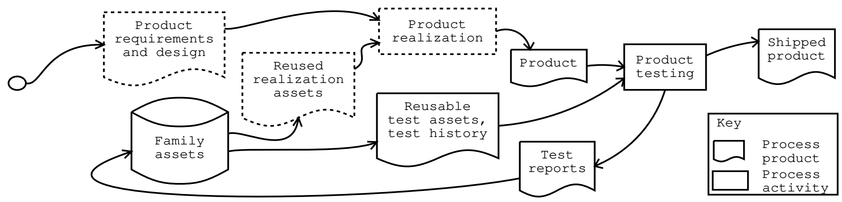


Fig. 7. The test-related activities in application engineering.

The tasks in the implementation of product testing are:

1. **Selection of the increments in the family tested in the product effort.** Product families contain applications that are variants of each other containing shared features and functions. If the functionality is not product-specific, it has already been tested in the previous products and there may not be a reason to test it again. Analyze the past testing to deduce the functionality needing testing.
2. **Selection of test assets to test the structure and functionality of the product.** Select the needed reusable assets to test the functionality analyzed in previous point.
3. **Analysis of adequacy, and design and implementation of additional test assets.** Based on test plans, test history, and coverage measures gained during test execution (see next task) decide if product testing is adequate both structurally and functionally. If the testing is judged inadequate, produce additional product-specific test assets to achieve application-specific testing objectives.
4. **Execution of product testing.** Execute the selected tests. In product family context, test planning and determination of adequacy are substantially more difficult than in single-product development. Thus iteration of steps 3 and 4 is beneficial.

A rarely discussed phase in product family testing could be called product "pre-testing". Before initiation of a product realization cycle in application engineering, the reused domain assets can be tested to fulfill product-specific requirements, for example variable performance requirements. It is also possible to partially integration test the assets now that the configuration is fixed. This testing serves two purposes. First, it is a way of moving testing to an earlier phase in the overall process. Second, it proves the usability of the reused components, which is important when components not build in-house are involved [31,32]. Assets produced in domain engineering are black boxes for application engineering teams in larger organizations. At least the test history of the assets should be consulted and the associated automated tests should be executed. Insight into the quality of the assets should be gained before investing considerable amount of effort into trying to use the assets.

6 Application of the conceptual model

We apply the conceptual model of product family testing to analyze the testing-related contributions of this book in relation to the overall product family testing process. The purpose is to demonstrate the model rather than evaluating the merits of the contributions analyzed.

The analyzed contributions are:

– Kamsties et. al. (Chapter XYZ) who describe a scenario-based system and integration test derivation for product families.
– Nebut et. al (Chapter ZYX) who present how product-specific testing is accomplished automatically from common requirements modeled as use cases.

The approaches analyzed use a very similar process to generate concrete testing logic and the target of the testing.

In domain engineering, a set of descriptions of the system level functionality of the product family is created in scenarios and use cases respectively. The descriptions contain the possible variation space of the product family as variation points in an action sequence of the scenario or pre- and postconditions of the use cases. The process by Kamsties et. al. proceeds on to create interaction sequences between the actors and the system still retaining the variability.

In application engineering, the variation of the product is resolved and the generic test assets are pruned accordingly. The interactions or constraints related to variants not present are removed. Products can now be tested using the concrete testing logic produced. Both of the methods also address the verification of variant absence by defining tests to show that.

Most significantly the methods are related to *reusable assets*. Both methods embody a full sequence of actions needed to produce half-ready and reusable test assets in domain engineering to lower the cost of testing a product.

Kamsties et. al. discuss *product by product testing* of an application that contain variations not considered by the domain model. They propose to create complementary scenarios and additional entries to domain test assets to accommodate the product-specific testing. It is not discussed in detail how additional testware is propagated to the family scale to achieve *opportunistic reuse* product-specific test assets. A partial reuse mechanism exists in moving from general to product-specific testing. If a match of variation binding exists in previous products, the former product-specific testware is leveraged. However, the reuse is quite coarse-grained as no partial matches are reused. Nebut et. al. have not devised any reuse mechanisms, but it can be argued that the need of reuse is smaller due the extensive automation. Finally because the both methods are technical rather than process-oriented, *feedback* from application engineering is not discussed.

The evaluated methods provide rigorous management of variability and are therefore excellent grounds for *incremental testing* both in the domain and application engineering side. The both chapters state the need of verifying all functions of each product, and do not view incremental testing as part of the method.

Kamsties et. al. provide a rigorous mechanisms for *traceability* from requirements to all test assets and backwards. This traceability is highly useful in change management and planning for testing both in domain and application testing. Nebut et. al. provide traceability by automation; meaning that: test cases can be automatically created from requirements. Both kinds of traceability have also potential in incremental testing. Again, neither of the methods discusses the process-related *synchronization*.

The *division of responsibilities* is implicit in both of the methods. The methods assume that testing is performed - the test cases are executed - in application engineering. The possibilities of test execution in domain engineering is not discussed.

The testing *organization* is two-fold in the method by Kamsties et. al.. The roles and responsibilities of domain and application engineering activities are clear and defined. The method by Nebut et. al. is more geared towards closed product families, where all variation can defined and modeled in domain engineering. Although not discussed explicitly, it can be assumed that all test design is done in domain engineering. The responsibility of application engineering is to use tools to generate test cases for a product from the common behavioral test patterns and to execute the testing.

The methods identify the tools needed to perform testing. Nebut et. al. propose to use UML-models annotated with additional first order logic constraints. The models are elaborated to sequence diagrams in design, while in testing test objectives, test scenarios, and behavioral test patterns are generated with a tool. Finally, the TGV/UMLAUT tools are used to generate runnable test cases. Kamsties et. al. propose a more general approach in terms of tools used. The required tools are a UML-modeling tool that is used to model the system scenarios and a hierarchical repository for persisting the created assets so that the created traceability is not lost.

With regard to *process and management* the both methods provide a partial solution. The chapters provide detailed information how to arrange the system testing in a product family organization. However as we have seen in this section, the methods focus on technical details. It therefore takes much additional analysis to be able to introduce testing according to the methods and incorporate it with other testing required. The both methods promise automation, reuse, and cost-efficiency, but without knowledge of the context this is not guaranteed.

To sum up this evaluation we conclude that the methods adhere to their scope: creating reusable test assets. We have argued earlier that overall product family testing process is an elusive target. This evaluation has fortified our argument as it still is quite hard to grasp the overall process based on these methods, because of the missing links to other testing of a product family. This however is not the fault of the methods but future directions for the research community.

7 Problems and future directions in product family testing

While product families are a part of everyday business in many companies and successes have been reported, little is still known about testing practices in the industry. Product families provide strategic headway, and it is understandable that advances are closely guarded from becoming common knowledge. For the science community, this leads to a lack of empirical data. We need more both qualitative and quantitative information about product family testing in practice.

On the qualitative side of our understanding of product family engineering, we need more case studies of hands-on work in testing product families. The set of currently published knowledge is very limited and not specific to testing [16,19]. The text space devoted to testing is accordingly small. There is little information about the practical use of current methodologies. We have very little knowledge of the real effects of current state-of-the-art product family testing. Also, the current product family testing methodologies have not been compared to each other.

Quantitative data is required to test the theories and methodologies. Currently there is no such empirical data and therefore no indication of how effort is spent testing product families. Claims about the increase of quality in products, like [33,19], have been made, but that is usually credited to the improvements in the implementation phase. Overall, there is no product family specific quantitative information about testing.

Testing methodologies are most often validated by empirical data gathered in testing of a well-known reference system. There are very few publicly available product families that can be used for this purpose, and, in product family testing research, we are missing the reference systems. Currently, the created methodologies are validated with proprietary systems, which hinders the evaluation of the methodologies. Also, the research community is open only who have access to the systems.

There are numerous mature testing tools available, but product family testing lacks efficient tool support. The problem with testing tools is that few of them directly support product family testing. For example, there are separate tools that provide entire testing environments for kinds of testing such as the *JUnit framework* [34], the *STA Framework* [35], and *TestDirector* [36]. There is a need for product family specific testing tools that should on one hand help to manage the reusable testing assets and on the other hand automate the execution of tests and the analysis of their results, and ultimately integrate the whole process of product family testing [27].

At the moment, we have only ideas how development of product families can support testing. A product family specific research question is to study how product family architectures and variability modeling can support both domain and application testing. In product family design and implementation, a gap between the goal (reuse) and the lower level methods (object orientation etc.) was bridged by reference architectures. Thus, we arrive at the testability [37] of product families.

One possible implication of product family engineering is the reduction of the importance of testing. As thorough and cost-effective testing is problematic especially in domain engineering other quality assurance techniques may gain more emphasis. Especially when the faults should be discovered early, as is the case in product family engineering, the use of inspections, walkthroughs, and other similar static testing methods is inviting.

8 Conclusion

Compared to its significance, testing has received little attention despite the problems rising from the application of system family engineering. A product family engineering approach requires a carefully planned testing process that can be easily adapted and used for product families in different application domains. Unfortunate for the practitioners, testing product families is a challenging task. A gap between the overall product family engineering processes and the assumed testing processes utilizing the practical testing methods is evident, because it is not clear how the processes and test assets are aligned to test a product family cost-effectively, at the same time preserving quality.

The conceptual model for product family testing presented in this paper is a formulation of the general theory for product family testing. The essential concepts of division of testing responsibilities, incremental testing, reusable test assets, feedback, synchronization, traceability, organization, tools, process, and management form the core issues that must be tackled in a testing process. The concepts have special significance in product family testing and they should be addressed when product family projects are initiated.

The influence of the different product family testing strategies was conjectured to give insight into the effects of currently known product family testing strategies. Supporting test process definition, the main process areas and tasks were recognized. Overall, this contribution describes our view of how product family testing can be conceptualized.

The research community has only recently started to focus on the importance of product family testing theory and methodology. In the near future, we expect to see good results on theory, process, and methodology in the field. Product family testing is a large-scale effort and therefore studying it is often time-consuming. Most of the produced methods have not been verified. There is a clear lack of published case studies describing the experiences gained in product family testing. Several companies, such as Philips, Nokia and Siemens, have extensive experience in product family engineering, but case studies about testing product families have not been published. The gained results are usually in company-specific internal documents.

Acknowledgments The authors thank Professor Jukka Paakki, Professor Inkeri Verkamo, and co-researcher Juha Taina for the comments regarding this work. This work was funded by ITEA-project Families.

References

1. Bosch, J.: Design and Use of Software Architectures. Addison-Wesley (2000)
2. Jazayeri, M., Ran, A., van der Linden, F.: Software Architecture for Product Families: Principles and Practices. Addison-Wesley (2000)
3. Fayad, M.E., Hamu, D.S., Brugali, D.: Enterprise frameworks characteristics, criteria, and challanges. Communications of the ACM **43** (2000) 39–46
4. Szyperski, C.A.: Component Software. Addison Wesley (1998)
5. Northrop, L.M.: SEI's software product line tenets. IEEE Software **19** (2002) 32–40
6. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Quarterly **28** (2004) 75–105
7. van der Linden, F.: Software product families in Europe: The Esaps & Café Projects. IEEE Software **19** (2002) 41–49
8. van der Linden, F., Bosch, J., Kamsties, E., Känsälä, K., Obbink, H.: Software product family evaluation. In: Proceedings of The Third Software Product Line Conference SPLC 2004, to Appear. (2004)
9. Northrop, L.M.: A framework for software product line practice – version 4.1. Technical report, Software Engineering Institute, Carnegie Mellon University (2003) URL: http://www.sei.cmu.edu/plp/framework.html [November 11, 2003].
10. McGregor, J.D.: Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, Carnegie Mellon University (2001)
11. McGregor, J.D.: A Practical Guide to Testing Object-Oriented Software. Addison Wesley (2001)
12. Kolb, R., Muthig, D.: Challenges in testing software product lines. In: Proceedings of CONQUEST'03. (2003) 103–113
13. Tevanlinna, A., Taina, J., Kauppinen, R.: Product family testing – a survey. ACM SIGSOFT Software Engineering Notes **29** (2004)
14. Muccini, H., van der Hoek, A.: Towards testing product line architectures. In: Electronic Notes in Theoretical Computer Science. Volume 82., Elsevier (2003)
15. Weyuker, E.J.: Testing component-based software: A cautionary tale. IEEE Software **15** (1998) 54–59
16. Jaaksi, A.: Developing mobile browsers in a product line. IEEE Software **19** (2002) 73–80
17. McGregor, J.D.: Structuring Test Assets in a Product Line Effort. In: Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications. (2001) 89–92
18. Binder, R.V.: Testing Object-Oriented Systems. Wiley (2000)
19. Clements, P., Cohen, S., Donohoe, P., Northrop, L.M.: Control channel toolkit: A software product line case study. Technical Report CMU/SEI-2001-TR-030, Software Engineering Institute, Carnegie Mellon University (2003)
20. Kauppinen, R.: Testing framework-based software product lines. Master's thesis, University of Helsinki, Department of Computer Science (2003)

21. Bertolino, A., Gnesi, S.: Use case-based testing of product lines. In: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering. (2003) 355–358
22. Kamsties, E., Pohl, K., Reis, S., Reuys, A.: Testing variabilities in use case models. In van der Linden, F., ed.: Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5). LNCS 3014, Siena, Italy, Springer Verlag (2003) 6–18
23. Le Traon, Y., Nebut, C., Jézéquel, J.M., Pickin, S.: Automated requirements-based generation of test cases for product families. In: Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE 03). (2003) 50–59
24. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. IEEE Transactions on Software Engineering **22** (1996) 529–551
25. Onoma, A.K., Tsai, W.T., Poonawala, M., Suganuma, H.: Regressing testing in an industrial environment: progress is attained by looking backward. Communications of the ACM **41** (1998) 81–86
26. van Ommering, R.: Beyond product families: Building a product population? In: Proceedings of the 3rd international workshop on the development and evolution of software architectures of product families. (2001)
27. Kauppinen, R., Taina, J.: Rita environment for testing framework-based software product lines. In: Proceedings of the Eighth Symposium on Programming Languages and Software Tools. (2003) 58–69
28. Young, Q.: Addressing the synchronization issue in product family test planning & preparation in practice. In: Proceedings of the International Workshop on Product Line Engineering The Early Steps: Planning, Modeling, and Managing (PLEES'03). (2003) 29–34
29. Williams, A.W., Probert, R.L.: A measure for component interaction test coverage. In: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2001). (2001) 304–311
30. Musa, J.D.: Operational profiles in software reliability engineering. IEEE Software **10** (1993) 14–32
31. Comella-Dorda, S., Dean, J.C., Morris, E., Oberndorf, P.: A process for COTS software product evaluation. In: COTS-Based Software Systems, First International Conference, ICCBSS 2002, Proceedings. (2002) 86–96
32. Liu, A., Gorton, I.: Accelerating cots middleware acquisition: The i-mate process. IEEE Software **20** (2003) 72–79
33. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Series in Software Engineering. Addison Wesley (1998)
34. Gamma, E., Beck, K.: Junit: A cook's tour. Java Report **4** (1999) 27–38
35. Rankin, C.: The software testing automation framework. IBM Systems Journal **41** (2002) 126–139
36. Mercury Interactive: Mercury Interactive TestDirector. (2003) URL: http://www-svca.mercuryinteractive.com/products/testdirector/ [May 22, 2003].
37. Binder, R.V.: Design for testability in object-oriented systems. Communications of the ACM **37** (1994) 87–101