Linking Feature Models to Code Artifacts using Executable Acceptance Tests[1]

Yaser Ghanam and Frank Maurer

Department of Computer Science
2500 University Dr. NW, Calgary
Alberta, Canada T2N 1N4

{yghanam, fmaurer}@ucalgary.ca

Abstract. A feature model is a representation of the requirements in a given system abstracted at the feature level. Linking conceptual requirements in feature models to actual implementation artifacts provides for many advantages such as increased program comprehension, implementation completeness assessment, impact analysis, and reuse opportunities. However, in practice, as systems evolve, traceability links between the model and the code artifacts may become broken or outdated. In this paper, we contribute an approach to provide traceability links in a way that ensures consistency between the feature model and the code artifacts, enables the evolution of variability in the feature model, and supports the product derivation process. We do that by using executable acceptance tests as a direct traceability link between feature models and code artifacts. We evaluate our approach and present a brief overview of the tool support we provide.

Keywords: agile product line engineering, feature models, traceability, variability evolution, executable acceptance tests.

1 Introduction

Feature modelling has become an essential aspect of software engineering in general and software product line engineering (SPLE) in particular. A feature model is a representation of the requirements in a given system abstracted at the feature level [30]. A feature can be broadly defined as a chunk of functionality that delivers value to the end user. In SPLE, feature models represent a hierarchy of features and sub-features in a product line and include information about variability in the product line and constraints of feature selection.

Linking conceptual requirements in feature models to actual implementation artifacts provides for advantages such as increased program comprehension, implementation completeness assessment, impact analysis, and reuse opportunities [2]. Nevertheless, traceability is a non-trivial problem. Berg et al. [3] analyzed

[1] This research is supported by iCore – Alberta Innovates Technology Futures.

traceability between the problem space (i.e. the model) and the solution space (i.e. the development artifacts) in a software product line context. The results suggested that the feature model provided an excellent visualization means at individual levels of abstraction. However, it did not improve the traceability between artifacts across development spaces. Furthermore, in practice, as the product line evolves, traceability relationships between the model and the code artifacts may become broken or outdated [29]. This happens either because changes in the model are not completely and consistently realized in the code artifacts; or because changes due to continuous development and maintenance of the code artifacts are not reflected back in the model. This problem is not unique to SPLE. In fact, outdated traceability between requirement specifications and other development artifacts has always been an issue in software engineering [13, 6].

Traceability links provided by some commercial tools (e.g. DOORS [7]) mitigate this issue, but leave some other problems unsolved. For example, say feature A and feature B are independent features in the product line. During the maintenance of feature A, the developer introduced a change that unintentionally caused a technical conflict between feature A and feature B. Although the tool will maintain the traceability links between each piece of code and the correspondent feature, it cannot, uncover the newly introduced conflict in order to reflect it back in the model.

In this paper, we propose the use of executable acceptance tests as a direct traceability link between feature models and code artifacts. In the next subsection, we give an overview of executable acceptance tests and their characteristics.

1.1 Executable Acceptance Tests

Requirement specifications – in its traditional format – exist in a number of documents and are written in a natural language. The correctness of the behaviour of a system is determined against these specifications using test cases or scenarios. On the other hand, executable specifications are written in a semi-formal language that aims to reduce ambiguities and inconsistencies. Executable specifications take various formats ranging from very formal [11] to English-like [22]. The English-like ones are often called scenario tests [17], story tests [19], or acceptance tests [26]. They are usually used in organizations where Agile Software Development [20] is practiced. These names highlight the role of these artifacts as:

1. Cohesive documentation of the specifications of a given feature.
2. Accurate, high-level validity tests: by being executable, these specifications can be run (executed) against the system directly in order to test the correctness of its behaviour.

Throughout this paper, we will use the general term executable acceptance test (EAT) to refer to the English-like specifications that can play the two roles above. In this paper, we present an idea on how EATs can be used as a traceability link between feature models and code artifacts. Fig. 1 shows an example of an EAT. If the behaviour of the system matches the expected one as specified in the EAT, the test passes. Otherwise, the test fails indicating either a technical problem in the code, or a business problem in understanding the specifications of the system. To link the EAT to actual production code, a thin layer of test code – called fixture – is used. EATs are usually executed using tools like FIT [10] and GreenPepper [14].

Home owner is notified after two failed attempts				
Start	Screen.Login			
Enter	Name	John	PIN	1234
Check	Info is valid	False		
Enter	Name	John	PIN	4321
Check	Info is valid	False		
Check	Owner is notified			

Fig. 1. Example of an EAT

1.2 Traceability from EATs to Code Artifacts

The fundamental basis of our approach is that EATs natively provide the necessary links to code artifacts. The reason why acceptance tests can be executed against the system is that they are linked to a thin layer of test code, and from there to actual production code. Fig. 2 shows an example of this traceability. At the first layer, only one row of a row-fixture EAT is shown for simplicity. This row is linked – by a test automation framework (e.g. FIT) – to a method in the test code called *addResidentWithPIN(...)*. This method in turns uses the *addResident(...)* method in the production code, specifically in the *HomeResidentsList* class. When the test is executed, an attempt to add a resident with the given parameters will be made. In this scenario, if the attempt is not successful – for a variety of reasons such as the PIN being too short or too long – the EAT will fail. Otherwise, it will pass. Usually, a suite of EATs is executed rather than a single EAT. Moreover, with appropriate test coverage, tools generate reports stating which methods where involved in the execution process of a certain EAT. Later in the paper, we will discuss how this traceability is useful in linking features models with the code artifacts.

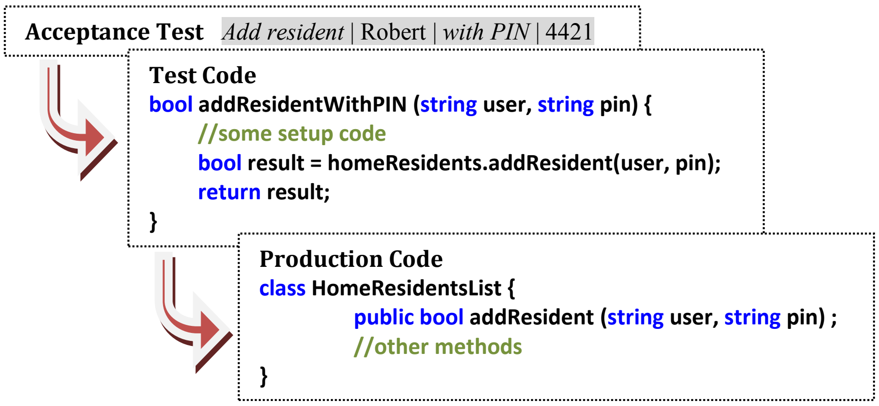
The rest of this paper is structured as follows. Section 2 is a review of relevant literature. Section 3 presents the proposed approach. Section 4 elaborates on the positive implications of the approach. Section 5 is an evaluation of our approach in comparison to other traditional approaches. Finally, we conclude in Section 6.

2 Literature Review

There is a large body of research on feature modeling in software engineering in general, and SPLE in particular. FODA [18] was one of the earliest techniques off which many other techniques were based (e.g. [16] and [8]). In our work, we use feature trees as described in traditional modeling techniques such as FODA, but the generality of our work is not affected by that choice.

Efforts to study traceability links between feature models and other development artifacts include the one by Filho et al. [15] in which they proposed the integration of feature models with the UML meta-model to facilitate the instantiation process. Another effort was the one by Ramesh et al. [28] in which use cases (representing requirements) were linked to design artifacts and from there to code artifacts. To group requirements at a more meaningful and comprehendible level of abstraction, Riebisch [29] suggested the use of feature models as an intermediate element between use cases and other artifacts. The main issue with this approach is that in real settings a massive effort is required to establish and maintain the traceability links due to the informal descriptions of the requirements – which made automation impossible [25]. To solve the language informality issue, new techniques were proposed. For example, Antoniol et al. [2] proposed an information retrieval method to link flat requirements to code artifacts. The caveat of the approach is that it is based on the hypothesis that programmers use names for program items (e.g. classes, methods, variables) that are also found in the text documents. There is also the issue of managing and maintaining the established traceability links. In a panel report, Huang [15] discusses the state-of-the-practice in traceability techniques. The report asserts that requirement trace matrices (RTMs) are often maintained either manually or using a management tool; and the amount of effort needed to keep these links up-to-date is enormous. Commercial tools are available to support traceability. CaliberRM [4], DOORS [7] and other tools are used to manage and visualize traceability links. However, these links have to be established manually, and the tools do not address issues specific to feature models such as variability in requirement. Some software product line tools like pure::variants [27] provide add-ins to allow requirement models in traditional management tools to be remodeled as feature models.

Our contribution in this paper is novel because we link feature models to specifications that are executable. We also show in the sections to follow how this linkage provides advantages specific to feature models and software product lines.


Acceptance Test *Add resident* | Robert | *with PIN* | 4421

Test Code

```
bool addResidentWithPIN (string user, string pin) {
      //some setup code
      bool result = homeResidents.addResident(user, pin);
      return result;
}
```

Production Code

```
class HomeResidentsList {
            public bool addResident (string user, string pin) ;
            //other methods
}
```

Fig. 2. Traceability through EATs

3 Using Feature Models with EATs

We propose extending feature models by including EATs as concrete descriptors of features at the lowest level of the feature tree. EATs should be associated with features that originally would be considered leaf nodes in the tree as shown in Fig. 3.

For instance, the feature "Access by PIN" is associated with three EATs. These EATs describe scenarios that need to be satisfied in the implementation of this specific feature. [2]

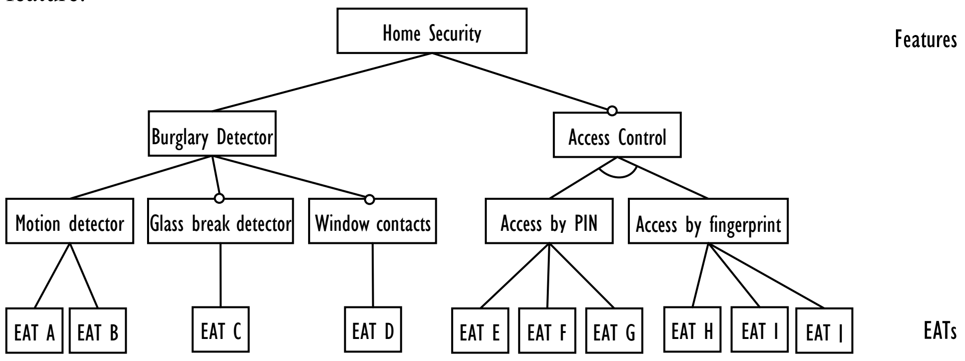


Fig. 3. The proposed extension to feature models

Linking between an EAT node in the model and the actual specification happens by associating a test unit to the EAT node. An EAT node can link to a test table, a test page, or a test suite. We intentionally do not put any constraints on the granularity of the test unit to leave it flexible for various contexts. Nevertheless, a single test table may be insufficient given that usually more than one table is needed to specify some behaviour. This makes a single table less cohesive than desired. On the other hand, a test suite may be too large because it involves more than one feature creating dependencies between test units. Therefore, we suggest the use of a test page as a usual test unit that provides reasonable cohesion and independence. Depending on the testing tools, test pages can take various formats such as html files or excel sheets.

3.1 Linking Features to EATs

Following the earlier definition of a feature as a chunk of functionality that delivers value to the end user, one EAT generally is not sufficient to represent a feature in a system. In practice, a group of EATs represent the different scenarios or stories expected in a given feature in a system. This implies that in order to somehow link features in a feature model to EATs, one-to-one relationships are not practical. Rather, each feature in the feature model should be linked to one or more EATs (Fig. 4). The "Access by PIN" feature is specified using three EATs. In order for the behaviour of this feature to be deemed correct, all three EATs should pass. Moreover, in some cases, a single EAT can be at a level high enough to cut across a number of features in the system. Consider, for example, a high-level EAT such as "Owner entering premises" as in Fig. 4. Say in order for the scenario specified in this EAT to pass,

[2] This is a simple example of a feature model. All features are mandatory unless there is a white circle indicating their optionality. For instance, the "Access Control" feature is optional. Grouping features (or sub-features) with an arch indicates that these features are alternatives. That is, only one feature can be selected from the group. If more than one feature can be selected from a group, a multiplicity constraint of the form [min..max] will be included.

more than one feature should be involved (i.e. EAT X cuts across a number of features). This implies that a many-to-many relationship is needed in order to accurately represent the relationship between EATs and features in a feature model.

Linking features to EATs has consequences. For one, the selection of a feature in the product derivation phase automatically implies the inclusion of all its EATs. Secondly, EATs shall inherit all the dependencies and constraints originally imposed on their parent nodes. For example, according to the model in Fig. 4, the two features "Access by PIN" and "Access by fingerprint" are mutually exclusive. This implies that the groups: {EAT E, EAT F, EAT G} and {EAT H, EAT I, EAT J} are mutually exclusive too. The importance of explicating these consequences will be discussed later in the paper.

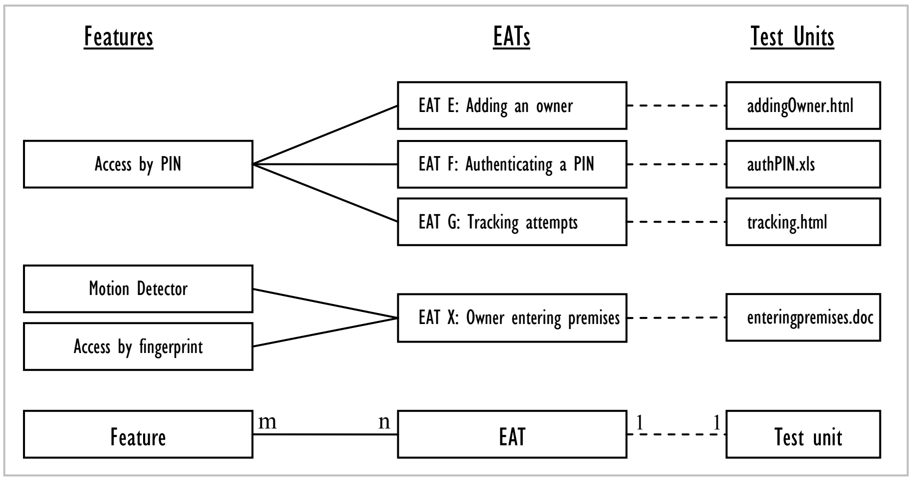


Fig. 4. Relationships between features, EATs, and test units.

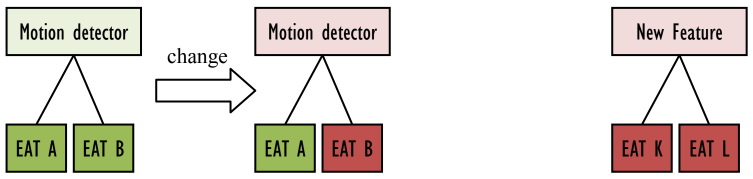
4 Implications of Using EATs as Traceability Links

In the previous sections, we discussed how features in the feature model can be linked to EATs in order to provide traceability links between the feature model and the code artifacts. This section analyzes the implications of using EATs by highlighting three main ways through which EATs provide significant contribution to feature models.

4.1 Consistency between the Feature Model and the Code Artifacts

EATs provide a means to ensure that the problem space (i.e. the specifications), and the solution space (i.e. the implementation) are consistent. This consistency is due to the fact that these specifications can be executed against the implementation, and the result of their execution gives an unambiguous insight of whether or not the intended requirements currently exist in the system. In our approach, we provide a link between feature models and EATs in order to inherit this important property. Within this context, we realize two key advantages of our approach:

Continuous Two-way Feedback. Maintaining a practice where every feature in the feature model has to be associated with some EATs is valuable. Changes due to continuous development and maintenance of the code artifacts are reflected back in the model, because – at any point of time – the EATs are either in a passing state (visualized as green) or a failing state (visualized as red). For instance, Fig. 5 shows how a change in the code (e.g. bug fix) caused EAT B to fail – also causing the "Motion Detector" to be denoted as incomplete. The opposite direction of feedback occurs when introducing a new feature to the model. The accompanied EATs will initially be in a failing state indicating that the feature is not implemented yet.



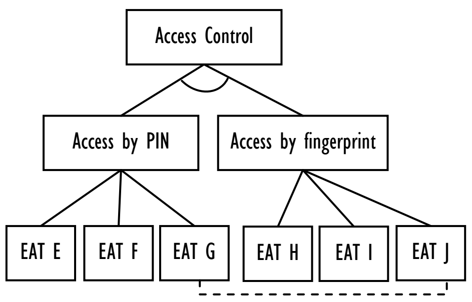
(a) a change in the code caused an EAT to fail providing immediate feedback in the feature model.

(b) when adding a feature to the model, initially EATs fail indicating incomplete implementation.

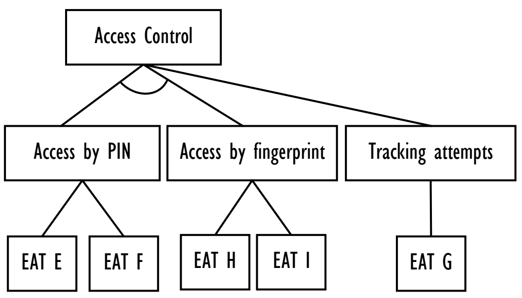
Fig. 5. Continuous two-way feedback

Exploiting Hidden Variability Concerns. Using EATs helps in revealing unwanted feature interactions that otherwise might be hidden. It also supports the realization of common aspects of features. We illustrate these points further by going through a number of scenarios.

Scenario 1: In some cases, the same EAT can be used as part of the specifications of two different features. If the features are originally mutually exclusive, and the same EAT passes in both, then this EAT is agnostic to the source of variation in the features. This means that the specifications in this EAT are part of the common portion of the parent node, which exploits a commonality aspect that was not originally apparent. Fig. 6 shows that because EAT G and EAT J are the same (we use a dashed line to denote this – it is also possible to give them the same name), we can abstract the commonality as a mandatory sub-feature under "Access Control".
Scenario 2: Using EATs allow finding unwanted feature interactions. EATs for independent features may pass when the features are selected separately; but fail when selected together. This is indicative of an unwanted feature interaction. This conflict is either a problem in the implementation and should be resolved, or an unavoidable real conflict that should then be reflected in the model as an "excludes" dependency or using a multiplicity constraint.
Scenario 3: Some EATs for independent features fail when these features are selected separately, but when selected together, they pass. This is indicative of a dependency between the features. It can be either due to unnecessary coupling in the implementation itself that should be resolved, or due to a necessary "requires" dependency that should then be reflected in the model.



(a) EAT G and EAT J are the same, and they both pass in mutually exclusive features.



(b) exploiting the commonality aspect as a mandatory sub-feature.

Fig. 6. Abstracting the commonality as a mandatory sub-feature

4.2 Supporting the Evolution of Variability in the Extended Feature Model

Using EATs as a basis for evolving variability in the feature model is rewarding in a number of ways. Consider the following scenarios:

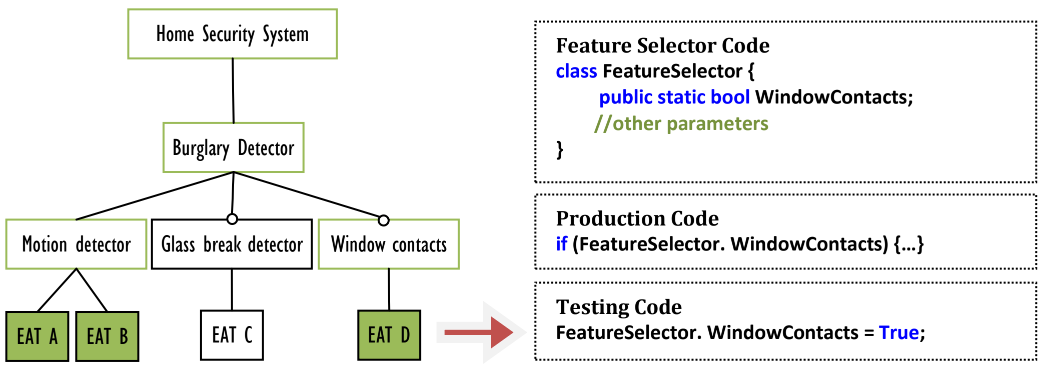
Scenario 1: A new feature or sub-feature is added to the feature model. In case the newly added feature causes EATs of other features that were originally passing to fail, this is a sign that a new conflict was introduced by the new feature. Without the direct feedback of failing tests, it is less likely for this conflict to be immediately exposed.
Scenario 2: An existing feature or sub-feature is removed from the feature model. If this feature was originally related to other features, then all dependencies are to be resolved before removing the feature safely. However, in case there was a hidden (unexploited) dependency between this feature and other features, removing this feature and its corresponding code might have a destructive effect on the other features. The fastest way to discover such effects is by looking for EATs that started to fail only after removing the feature.
Scenario 3: A new variant is to be added to a group of variants under a given feature. For developers, using EATs provides guidance on where and how this new variant should be accommodated in the system. For example, suppose we want to add a new alternative "Access by Magnet Card" under "Access Control". First of all, we may be able to reuse the EATs of the other sibling alternatives and tweak them to reflect the requirements of the new alternative. And because EATs are traceable to code artifacts, we can look at the implementation of the sibling alternatives in order to have a better comprehension on where in the code we should incorporate the new variant, and how it should be handled. With appropriate tool support, we can also automate the process of adding a variant by using the sibling nodes as templates, and directing the developer to the exact place in the code base where the new logic should be added [12]. This is particularly important for legacy systems with poor or outdated design documentation or for development environments where design documentation might not be available at all.

Scenario 4: Abstracting a variability aspect to the common layer. Say an EAT is used as part of the specifications of two mutually exclusive features, and this EAT passes in both. This means that the specifications in this EAT can be abstract to become part of the common layer of the parent node (as a mandatory sub-feature – this was discussed in the previous section).

4.3 Deriving Products using the Extended Feature Model

In a software product line context, feature models are used to select features and variants that constitute a product instance. The selection process should take into consideration the constraints and dependencies between features and variants, as conveyed in the feature model. Nowadays, tool support is available to make this process easier, faster and less error-prone. Once the features and configurations have been selected, an instance is derived that has the required feature composition and configuration. This section discusses the beneficial roles EATs can play in the product derivation process (aka. product instantiation process).

Selecting Configurations. During the derivation process, we usually need to set certain parameters (e.g. compiler directives, configuration classes) in order to select certain configurations for the product instance at hand. We can rely on EATs to automatically set up these parameters. This can be done because for an EAT to pass (independently of other EATs), it needs to set the correct parameter before it can execute the production code. When we finish the selection process of features in the feature model, we can run all the EATs that are relevant to the current selection. Given that all EATs have passed for the current selection, this means that all parameters in the system have been set properly, and the system is now ready to produce the right instance (Fig. 7). Another role of EATs in this context can be described as "*configuration by example*." That is, EATs provide a good starting point for the developers to learn how to configure a certain feature



```
class FeatureSelector {
    public static bool WindowContacts;
    //other parameters
}
```

```
if (FeatureSelector. WindowContacts) {…}
```

```
FeatureSelector. WindowContacts = True;
```

Fig. 7. Using EATs to select configurations

Extracting Required Artifacts. In some derivation techniques, a subset of code artifacts are extracted from a common base according to which features in the feature model were selected. EATs can play an important role in supporting this process. After the selection process of features in the feature model, we can run all the EATs that are relevant to the current selection as shown in Fig. 8 (CU refers to code unit and TU refers to test unit). Static code analysis can provide details on which code artifacts are needed to produce the desired instance by computing the transitive closure of all calls in the fixture classes used in the EATs of the instance.

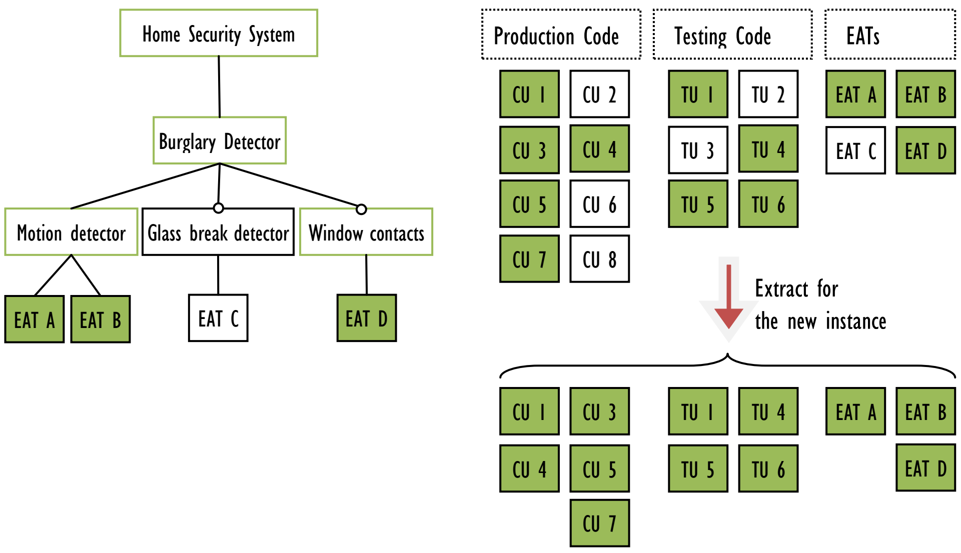


Fig. 8. Using EATs coverage reports to extract artifacts

4.4 Tool Support

In order to realize the benefits we discussed in the previous sub-sections, we built a tool that supports traceability links between the feature model and code artifacts via EATs[3]. To avoid reinventing the wheel, an open-source modeling tool was chosen in order to be extended. We used Feature Model DSL as the basis (available online [1]). The tool provides a feature modeling toolbox integrated in the Visual Studio environment. It includes a visual designer to create and modify models. It also provides a configuration window that allows the creation of configurations based on the feature model. We extended the tool in two ways, namely: allow the linkage between features and EATs, and define a course of action to complete the derivation process of individual instances after the configuration process. The remaining of this section will explain the currently available features.

The user can represent features and the relationships between them following the typical feature modeling notation. In our extension of the tool, the leaves of the feature tree can be linked to EATs as shown in Fig. 9.

[3] We thank Felix Riegger for his contribution in providing the tool support.

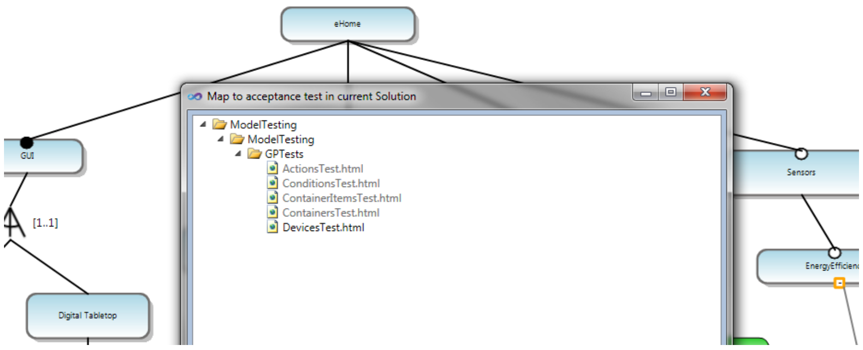


Fig. 9. The leaves of the feature tree can be linked to EATs

The tool also allows the user to run EATs directly from the feature model as shown in Fig. 10. Nodes that have passing tests are coloured in green and those with failing tests are coloured in red. After feature selection, the tool checks the constraints to ensure the validity of the selected subset of features, and it runs only those EATs that are relevant to a given instance. This is shown in Fig. 11. The extended version of the tool will be made available online in Spring 2010.

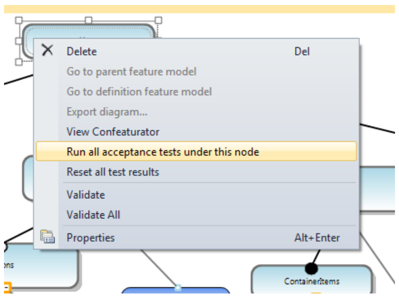


Fig. 10. The tool allows the user to run ATs directly from the variability model

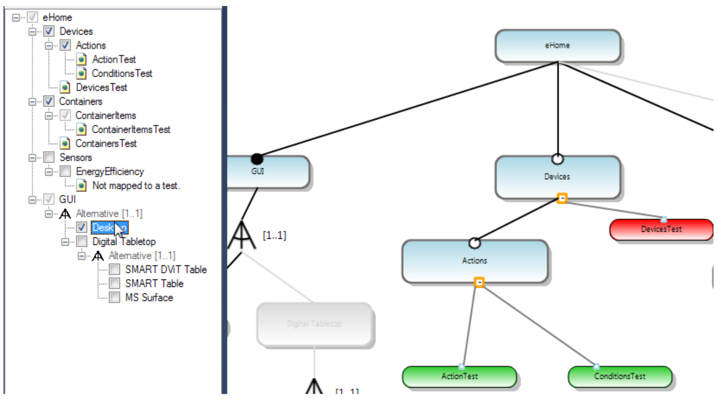


Fig. 11. The tool runs only those EATs that are relevant to a given instance

5 Evaluation

In this paper, we proposed the use of EATs to link feature models to code artifacts. This section presents an evaluation of the proposed approach. We evaluate the approach in two different ways. First, we compare our approach with traditional requirement traceability approaches and other approaches that involve feature models, as discussed in the literature review section. Then, we use the running example presented throughout this paper to list the limitations of the approach (the advantages of the approach were already discussed in the previous section). Using a running example for validation and evaluation purposes has been a well accepted technique in the community [31, 24, 5].

Table 1 lists a number of criteria against which we conduct our comparison. The criteria are based on guidelines obtained from the literature such as [29] and [2]. The system evolution criterion describes how traceability links are affected with the evolution of a system such as adding or removing requirements. In the case of feature model approaches, we are more concerned with the evolution of variability such as adding or removing variation points and variants. We use the program comprehension criterion to describe the ability of the developers to form a mental model of the variability definition as described in the feature tree as well as the realization of that variability at the code level. This evaluation is limited by the subjectivity arising from the criteria being considered. We intend to conduct a more thorough evaluation to collect empirical evidence of the feasibility and usefulness of the proposed approach.

Having illustrated the advantages of our approach in comparison to other traditional approaches, we think there is a raft of issues that need to be addressed. For one, we cannot currently predict how scalable our approach is – especially when dealing with a large number of variation points and variants. This problem is inherited from the scalability issues associated with feature modeling in general. Furthermore, despite the fact that EATs provide an elegant way to specify functional requirements in software systems, they have not yet been widely used in specifying non-functional attributes such as usability and security (other non-functional attributes like performance can be specified and executed as described in [21]). For feature models that contain variability due to non-functional aspects, our approach may not be sufficient. Moreover, the most common practices involving EATs focus on code artifacts much more than other development artifacts. For organizations that consider design artifacts, for instance, to be essential, the adoption of our approach may result in these artifacts becoming rapidly outdated - mainly because from a developer's perspective there will be no need to maintain them anymore. However, the organization can solve this problem by requiring that some EATs be used as placeholders to associate important information such as links to design documents, standards or data files [23]. Another critical point that may be a real challenge in some organizations is the commitment and discipline needed to provide sufficient EAT coverage of all features in the system in a sustainable manner. Adopting test-driven development practices is one way to deal with this issue.

It is also important to point out that contrary to the initial impression that this approach may lead to architectural drift, the approach may actually improve adherence to the architecture. This is because of the transparency and traceability between the model artifacts and the code artifacts, which provide the developers with a holistic and consistent understanding of the product line. This, however, is still an open issue to investigate in the near future.

Table 1. Comparison between the different approaches of traceability

	Traditional Requirement Traceability	*Traceability through Feature Models*	*Traceability through Feature Models and EAT*
Number of links	Very large, because every requirement is linked to relevant design and code artifacts.	Somewhat large, because every feature is linked to relevant design and code artifacts.	Fairly small, because every feature is only linked to the EATs files specifying that feature.
Quality of links over time	Links become broken or/and outdated without appropriate manual revisions and updates.	Links become broken or/and outdated without appropriate manual revisions and updates.	Links stay consistent and up-to-date because of the immediate feedback on broken or outdated links.
System evolution	Not supported efficiently. If a requirement is added or removed, links have to be re-established.	Not supported efficiently. If a new variation is added or removed, links have to be re-established. Also, there are no automatic checks for new hidden conflicts in the feature model.	Full support. Only links for the added or removed features or variations need to be handled. Failing EATs indicate newly introduced conflicts.
Impact analysis	Provides information on the artifacts that can be potentially impacted by a change. No details on the actual impact.	Provides information on the artifacts that can be potentially impacted by a change. No details on the actual impact.	Provides information on the artifacts that are *actually* impacted by a change, and provides immediate feedback on the *actual* impact of that change.
Program comprehension	Improved over systems with no traceability. But requires an effort for developers to link requirements with code tasks (reading RTMs is not simple). Also, given that variability is not modelled explicitly, handling each type of variation in code is not straightforward.	Reasonable, because requirements are conceptualized at a more comprehendible level of abstraction (i.e. features), and variability is modelled explicitly.	Good, because features are linked directly to code artifacts, and hence variants can be traced to code easily. Also, developers get instant feedback on changes to the code.

6 Conclusion & Future Work

The significance of establishing good traceability links cannot be overstated as evident in the literature and in practical contexts. We presented an approach to link feature models to code artifacts using executable acceptance tests. This paper contributed an approach to provide traceability links in a way that:

- ensures consistency between the feature model and the code artifacts,
- enables the evolution of variability in the feature model, and
- supports the product derivation process.

The valuable implications of these three characteristics were illustrated in detail, and the approach was compared to traditional approaches to highlight its strengths. In spite of the limitations our approach has, we think this is a first yet significant step towards a framework to adopt efficient traceability practices in software product line organizations. For future work, we need to conduct a more comprehensive evaluation of this approach in an industrial setting. We also would like to continue working on a complete tool support for creating, managing, refactoring, and linking EATs within the context of feature models.

References

1. André, F., http://featuremodeldsl.codeplex.com/. *Feature Model DSL Homepage*, 2009. Accessed February 10, 2010.
2. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E., Recovering traceability links between code and documentation, *IEEE Transactions on Software Engineering*, vol.28, no.10, pp. 970- 983, Oct 2002.
3. Berg, K., Bishop, J., and Muthig, D., "Tracing Software Product Line Variability — From Problem to Solution Space," presented at *2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, White River, South Africa, 2005.
4. CaliberRM, http://www.borland.com/us/products/caliber/index.html, accessed March 1, 2010.
5. Cho, H., Lee, K., and Kang, K. C. 2008. Feature Relation and Dependency Management: An Aspect-Oriented Approach. *Proceedings of the 2008 12th international Software Product Line Conference* (2008). IEEE Computer Society, Washington, DC, 3-11.
6. Cleland-Huang, J., Zemont, G., and Lukasik, W. 2004. A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability. In *Proceedings of the Requirements Engineering Conference, 12th IEEE international*(September 06 - 10, 2004). RE. IEEE Computer Society, Washington, DC, 230-239.
7. DOORS, http://www-01.ibm.com/software/awdtools/doors/, accessed March 1, 2010.
8. Fey, D., Fajta, R., and Boros, A., Feature Modeling: A Meta-Model to Enhance Usability and Usefulness. In *Software Product Lines (SPLC2)*: Spr*inger, 2002, pp. 198-216.*
9. Filho, I.M., Oliveira, T.C., Lucena, C.J.P., 2002. A proposal for the incorporation of the features model into the UML language. Proceedings *of the 4th International Conference on Enterprise Information Systems* (ICEIS2002), Ciudad Real, Spain.
10. FIT, http://fit.c2.com, accessed Nov, accessed March 1, 2010.

11. Fuchs, N.E.: Specifications are (Preferably) Executable. IEE/BCS Software Engineering Journal 7(5) (1992) 323–334.
12. Ghanam, Y., and Maurer, F., Extreme Product Line Engineering – Refactoring for Variability: A Test-Driven Approach. *The 11th International Conference on Agile Processes and eXtreme Programming* (XP 2010), Trondheim, Norway, 2010.
13. Gotel, O., and Finkelstein, A., "An Analysis of the Requirements Traceability Problem," *1st International Conference on Requirements Eng.,* 1994, pp. 94-101.
14. GreenPepper, http://www.greenpeppersoftware.com, accessed March 1, 2010.
15. Huang, J. C. "Just enough requirement traceability", *Proceedings of the 30th Annual International Computer Software and Applications*, Chicago, September 2006,pp. 41– 42.
16. K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, FORM: A feature-oriented reuse method with domain specific reference architectures, *Annals of Software Engineering,* vol. 5, pp. 143-168, 1998.
17. Kaner, C. "Cem Kaner on Scenario Testing: The Power of 'What-If…' and Nine Ways to Fuel Your Imagination", *Better Software*, *5(5)*:16–22, 2003.
18. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A., Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990
19. Kerievsky, J. "Storytesting", http://industrialxp.org/storytesting.html, accessed March 1, 2010.
20. Manifesto for Agile Software Development, available at: http://www.agilemanifesto.org/, accessed May 13, 2010.
21. Marchetto, A., http://selab.fbk.eu/swat/slide/2_Fitnesse.ppt, accessed March 10, 2010.
22. Melnik, G., Maurer, F., and Chiasson, M., "Executable Acceptance Tests for Communicating Business Requirements: Customer Perspective," *Proc. Agile 2006 Conf.,* IEEE CS Press, 2006, pp. 35–46.
23. Park, S.S., Maurer, F.: The benefits and challenges of executable acceptance testing. In: APOS 2008: Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral, pp. 19–22 (2008).
24. Parra, C., Blanc, X., Duchien, L.: Context Awareness for Dynamic Service-Oriented Product Lines. *Proceedings of 13th International Software Product Line Conference* (SPLC), San Francisco, CA, USA (2009).
25. Pashov, I., Feature Based Method for Supporting Architecture Refactoring and Maintenance of Long-Life Software Systems. *PhD Thesis*, Technical University Ilmenau, 2004.
26. Perry, W. *Effective Methods for Software Testing, 2/e*, John Wiley & Sons: New York, NY, 2000.
27. Pure::Systems, http://www.pure-systems.com/DOORS.102+M54a708de802.0.html, accessed March 1, 2010.
28. Ramesh, B., Jarke, M., Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*, vol. 27, Issue 1 (January 2001) pp. 58 – 93.
29. Riebisch, M., Supporting Evolutionary Development by Feature Models and Traceability Links. *Proceedings of the 11th IEEE international Conference and Workshop on Engineering of Computer-Based Systems* (May 24 - 27, 2004). ECBS. IEEE Computer Society, Washington, DC, 370.
30. Riebisch, M., Towards a more precise definition of feature models. Position Paper, in: M. Riebisch, J.O. Coplien, D, Streitferdt (Eds.), *Modelling Variability for Object-Oriented Product Lines*, 2003.
31. Tun, T. T., Boucher, Q., Classen, A., Hubaux, A., Heymans, P. (2009) Relating Requirements and Feature Configurations: A Systematic Approach, *International Software Product Line Conference* (SPLC 2009).