A product line is a set of products that together address a particular market segment. Product lines are, of course, nothing new in manufacturing. Examples are Airbus, Ford, Dell, and even McDonald's. Each of these companies exploits commonality in different ways. Boeing, for example, developed the 757 and 767 kinds of transport in tandem, and the parts lists for these very two different aircraft overlap by about 60%, achieving significant economies of production and maintenance. But software product lines based on inter-product commonality are a relatively new concept that is rapidly emerging as a viable and important software development paradigm. Product flexibility is the anthem of the software marketplace, and product lines fulfill the promise of tailor-made systems built specifically for the needs of particular customers or customer groups. A product line succeeds because the commonalities shared by the software products can be exploited to achieve economies of production. The products are built from common assets in a prescribed way. This chapter throws light on the review of the literature on SPL, configuration management of SPL, variability aware design patterns for SPL configuration management and product derivation, and the use of ontologies in improving SPL configuration management and product derivation.

2.1 SOFTWARE PRODUCT LINE

Liebig et al. [20] analyzed around 40 open-source C projects. In all the pre-processor based SPLs, different aspects of projects were investigated. For instance, the influence of variability, complexity, granularity, and types of extensions to core functionalities were considered. In the process metrics to measure extensions, granularity, complexity, and variability were derived. The focus was on metrics such as Lines of Code (LOC) representing size of software, Number of Features Constants (NOFC) representing configuration dynamics of SPL, Lines of Feature Code (LOF)
representing code pertaining to feature expressions, Scattering Degree (SD) representing feature constant occurrence, Tangling Degree (TD) representing count of different feature constants, Average Nesting Depth (AND) representing average nesting depth, Granularity (GRAN) representing fine grained nature, and Type (TYPE) representing number of extension occurrences of particular type. The work of Hunsen et al. [21] also focused on pre-processor based SPL exploring variability. Dealstra et al. [22] made a very useful study on the derivation of product from software product line. The study revealed that it is an expensive and time-consuming activity and hence investigation led to the analysis of many product lines and product derivation activity. A framework was presented that can be used to simplify product derivation. Houston et al. [23] focused on the risk management in software industry which can be extended to SPL. The architectural analysis provided by Choi and Scacchi [24] can help in analyzing SPL. Similar kind of study was made in [25, 26, 27, 28] to investigate crucial process activities in SPL architecture while Juristo et al. [29] gave importance to usability and impact on the design of software. Alves et al. [30] made a comprehensive review of requirements engineering required by SPL. In the similar fashion, Snook et al. [31] focused on using requirements of the product line for failure management while Bai et al. [32] built a failure prediction model based on Markov Bayesian network model. Effort estimation based on neural network model in [33] was found to be useful in SPL.

Engstorm and Runeson [34] focused on testing of SPL. They opined that testing SPL is the area which needs further research which is evident in their review. In [52] a virtual fault injection framework was proposed for testing SPL to make it reliability-aware. Hanssen et al. [9] and Chow and Cao [35] focused on SPL with agile engineering. Since agile software process model is the modern approach used in software development, it is best used with the product line as well. These researchers used an industrial case study to demonstrate the SPL Dynamics with agile. Noor et al. [1] did similar kind of research with a collaborative approach for product line planning which proved to be useful, usable and practical and Vlaanderen et al. [36] worked on SCRUM principles to software product management. In [37] Christoph et al. focused on agile portfolio management. Ajila and Kaba [38] focused on the evolution of SPL and its support mechanisms. They identified change management
mechanisms and concluded that change is inevitable in SPL and the variability-aware SPL process evolution is required. Huang, Yee and Mak [39] built a web-based application for automatic change management. Moreover, product families can be leveraged by improving standards in artifact development thus making an SPL robust.

Bellifemine et al. [40] opined that JADE tool can be used to build software applications which are distributed in nature. This can be used for building complex SPL using agent-based approach. Towards taking the software development process to students, Pfahl et al. [41] built a CBT module that could help students to understand project management. Similar kind of work was done in [42]. Software maintainability aspects explored in [43] are useful for SPL as well. Scientific product line engineering and scientific computing can be leveraged with the concepts of SPL [44]. The approach of Medvidovic et al. [45] in integrating process methodologies can help in improving the quality of SPL. Security aspect can be incorporated in SPL as the product derivation is based on quality factors as well. Lean measurement was introduced by Peterson and Wohlin [46] for software process improvement which results in a change in thinking process and culture in SPL.

A good review of software product line evolution is found in the work of Pussinen [47]. There are evolution metrics such as usability of core assets, quality of assets and opportunities for future assets. Other metrics found in the review includes software change control capability, parameterized modifiability, and modification complexity. Ali et al. [48] made a review of economic models for SPL. The economic models were described in terms of adoption and life cycle. Other factors considered for comparison include the type of analysis, economic function, underlying model, viewpoint, scenario, market attribute, cost factors and risk adjustment. Tichy [14] made a review of SCM for better understanding.

Measuring different aspects of software engineering can improve productivity. Kaner et al. [49] proposed software metrics and evaluated them. The metrics explored include correlation, consistency, tracking, predictability, discriminative power, and reliability. There are some direct measurements that can provide details of software. They include the length of the source, duration, the number of defects identified, and
time spent by the programmer. Instruments used for measuring are counting, matching, comparing, and timing. Similar kind of research was made by Mills [50].

Tawhid and Petriu [51] explored the concept of model transformation in SPL through product model derivation. The product model derivation is the process of mapping between features present in the feature model to the actual implementations in the SPL. Trinidad et al. [52] also explored feature models and mapped them to component models in order to build SPL dynamically. The feature model they used shows potential states in the product. Feature models refer to the features in the software product line while the component models include the artifacts. By mapping these two models it is possible to have dynamic SPLs. They explored the feature model for TV SPL. The TV platform contains layers, user interface, and effects. There are many kinds of features in the feature model. They include a mandatory feature, optional feature, set-relation, requires and excludes. Schuster [53] proposed a method known as pattern-based SPL which includes role modeling. Feature model is exploited in order to implement a solution for better SPL.

2.2 PRINCIPLES OF SOFTWARE PRODUCT LINES AND PROCESS VARIANTS

Emerging technologies and new trends in the application development in the wake of distributed computing technologies like web services revolutionized the way software is developed and used. To withstand ever changing needs of businesses, the present solution is software product line (SPL). SPL infrastructure has domain engineering and application engineering. Sengupta et al. [54] explored reasons especially business reasons for the development of distributed technologies and their usage in the real world. Bayer et al. [55] explored generic product line engineering life cycle. Products derived in the product line are associated with both domain and application. The domain engineering includes domain analysis, domain design, and domain implementation. The application engineering includes application analysis, application design and application implementation. Schmid and Geppert [56] found that there were many issues pertaining to the product line. They include variability
modeling, product line management, organizational issues, and transition approach. Variability refers to changes to be incorporated, product line management refers to planning and strategies, while organizational issues refer to the organizational structure which needs to support the SPL paradigm. These issues are related to software product line. Variability modeling plays a vital role in the success of product line and its configuration. The product line management refers to set of activities used to manage product line. The product line includes set of products and each product is made up of certain artifacts. The artifact includes core and custom assets. Identifying reusable components in the product line is thus given high importance. The best support product line can be achieved by considering all issues and handling them with care.

Pohl et al. [57] focused on the documentation of product lines and analyzed benefits of documenting variability management. Variability management can have following activities.

- Identifying commonalities and variability at the time of domain engineering.
- Building domain artifacts with variability.
- Identifying bindings for variability as part of application engineering.
- Obtaining final individual applications by making use of variability among domain artifacts.

Nyman [58] focused on interface theories and product lines in order to achieve model transition systems. The model transition is nothing but converting a model to another model. Filho et al. [59] explored product line that is model-driven and multidimensional. They focused on identifying challenges pertaining to leveraging variability in SPL, besides providing solutions and comparing with existing variability models. Their case study includes multiple products, multiple phases, multiple views, and multiple levels of configuration. They concluded that there should be integration between variability level and domain-specific level. They focused on product lines that include multiple dimensions and model-driven approaches for leveraging variability. The different dimensions considered in the system development include generation, consistency, and traceability. The overview of their proposed approach includes customer operational need analysis, system need analysis, system
architecture, logical architecture, physical architecture, and integration of components.

Buchmann et al. [60] proposed a model-driven approach for SPL configuration management in order to improve configuration management of SPL. Similarly Guana and Correal [61] followed a quality attribute-driven approach for SPL CM. They employed model-driven strategy in order to achieve good results. Schaefer et al. [62] explored variability – aware design patterns for exploiting reusability further in SPL and improve configuration management.

2.3 SPL CONFIGURATION MANAGEMENT APPROACHES

SPL configuration management needs to be cautious when compared to traditional configuration management. This section reviews the literature on this. Chao and Chen [63] opined that the characteristic data of a product goes to product configuration. Felfering et al. [64] stated that SPL configuration management is challenging as it faces much complexity and dynamism. Song and Kusiak [65] followed data mining approach for configuration management of software. They focused on mining customer behavior in historical data and then decided to product configurations and management. With extracted business intelligence they could derive products with the quality needed by customers. Tseng et al. [66] proposed Case-Based Reasoning (CBR) for product configuration management and customizations. White et al. [67] focused on SPL configuration management. Especially, they proposed an approach for automatic diagnosis of feature model configurations in SPL that are part of overall configurations. Their solution is useful for discovering configuration errors in SPL configuration management systems. The integration of project management and system analysis tools as described in [68] can be used to improve configuration management. Szykman et al. [69] advocated the usage of Information Technology (IT) for improving software configuration management and advocated the use of Computer-Aided Software Engineering (CASE). Mc Gregor [70] introduced Strategic Organization methodology that can be used to configure and manage systems. The methodology includes plan for change.

Kon et al. [71] explored dynamic configuration management for multimedia systems. These systems are distributed in nature and that needs efficient configuration
management due to its complexity and size. The focus was on modeling variability using Mat lab software. Automotive domain was chosen for the purpose. They also evaluated variability concepts. A controlled experiment was conducted with various programmers who are involved in the variability modeling. The variability concepts include break function and break pressure in the automotive domain. Their research hypotheses focused on the models that yielded smallest models, best encapsulation of concepts, concepts versus well-structured models. They made both objective and subjective assessments on the proposed approach. Their results revealed that data modeling has promising advantages in the case of modeling variability. There are many configuration management approaches as explored by Pussinen [47]. They are component-based product population approach and Kobra approach. The latter has many integration schemes which are known as framework change integration, application change integration, feedforward integration, and feedback change integration. Configuration management includes visualization of traces. The visualization can be done with evolution graphs, topology diagrams, and dependency tracking. Hong et al. [72] explored configuration management approach that is based on components. Since component technology forced the developers to adapt reusability this is given high importance in the software industry.

Leung and Lui [73] defined the modern testing process as tool-driven process, and focused on software variability and its related testing dynamics. There are certain issues in the testing process. For instance, software cloning can cause wastage of time in testing. The researchers explored on historical testing records in order to improve the process of testing. Testing integrated product families and discovery inconsistencies and discrepancies if any was the intent. To achieve this framework was proposed and the framework does not consider internal functionalities of software systems. Test case portfolio in the form of an algorithm was proposed and also discussed resource allocation and project management. The usage of historical records improved the quality of testing but it does not eliminate all issues.

Hachemi and Ahmed-Nacer[74] discussed software patterns. A software pattern is nothing but a proven solution to a frequently occurring design issue. Once the pattern is used, the underlying issue gets resolved. Thus software patterns play a
vital role in developing software that improves maintainability. There are relationships between software patterns. These relationships also play a vital role in identifying the components that work together. There are different relationships between patterns such as uses, refines, generalizes, specializes, used by, required to, works for, and other relationships. First of all two patterns are taken and they are analyzed. Stop words are removed. Then stemming takes place. Inclusion and TF/IDF are carried out. And finally, it is possible to extract relationships between two patterns.

Ayora et al. [75] proposed a framework for supporting systemic evaluation of variability support in software. Especially their focus was on process-aware information systems. There has been increasing adaptation of process-aware software systems in the real world. In this context, it is essential to have a good understanding of business process variability. The researchers proposed a framework to assess and compare software process or business process variability. They conducted a review of the literature and found that process variability is important in any information systems. The variability gives rise to different possibilities in the real world applications. It also helps process engineers implement process variability in order to improve the quality of software development. A process model can have different aspects such as optional perspective, functional perspective, organizational perspective, temporal perspective, behavioral perspective, and informational perspective. There is a relationship among these processes. Behavioral perspective deals with control edge, and control connector. The informational model deals with activity and process model. The functional perspective includes automatic and complex activities. Temporal perspective is associated with events. Organizational perspective deals with roles and operational perspective deals with operations.

Frauenberger and Stockman [16] explored design patterns that can be used in SPL. They proposed a paco framework and explored its methods. It includes pattern mining, application, and feedback. Pattern mining includes localize, describe, extend context, new version, and rate. The application includes localize, explore and apply. The feedback includes extending context, new version, and rate. They focused on the usage of design patterns or expressing things in the form of design patterns. When
they are expressed in the form of design patterns it is possible to reuse in the software development process as much as possible.

Mylärniemi et al. [76] focused on performance variability in software product lines. As the customer needs are diversified, it is required to have performance variability among products in SPL. The architecture used for software product line needs to have provision for handling commonality and variability. The variability related to the customer needs can trigger differentiation in the product line and finally leads to deriving a new product. In fact, variability manifests in various levels and forms. Therefore the realization of variability and efficiently managing variability is the cornerstone of SPL. Variability refers to the characteristics, performance, strategy, and realization strategy chosen. Their study includes data collection, validation, case studies, models and realization. Different activities involved in the overall process include analysis, data collection, and validation. With respect to capacity variability, both software and hardware cases can be considered. Software realization for capacity variability needs to be given importance as well.

Yu and Ramaswamy [77] proposed an evaluation-based model for configuration management for handling changes in software product line. For handling corrective changes and enhancement changes different approaches were used. The research includes evolution propagation, release path, request path, update path, a feedback path, and report path. It is based on the concept of evolution propagation with core assets, custom assets and products. When changes are made to custom assets, they need not reflect in products already existing. However, the changes are reflected in the future new products only. When changes are made to the custom portion of any product instance, the feedback is not given to custom assets. The rationale behind this is that the configuration management associated with the custom part is not linked with configuration management of custom assets. Various actions considered by them include variant management, baseline management, branch management, evolution management, concurrency management, and version management.

Kalipsiz [78] proposed a framework for building software configuration management system. This researcher opined that configuration management is the
main thing for quality management. Mili et al. [79] explored different issues in reusing software and expressed many benefits of reuse. They include ready to use, can have building blocks ready for improving productivity. There are many benefits of reusing software. They are as follows.

- It is useful to build products faster
- The building blocks and their reuse can help to locate, use and evaluate reusable software products.
- Helps in reducing effort and time.

There are different types of reusing artifacts in SPL. They are as follows.

- Reuse of data
- Reuse of architectures
- Reuse of detailed design and business rules
- Reusing programs

The reusable software development knowledge includes core fragments, logical structures, functional architectures, external knowledge, and environmental knowledge. There are different levels of reuse in the software. They include source domain level, target domain level and object level. Reusable assets can be part of the development and developing reusable assets can be part of development life cycle. In fact, there should be knowledge of the purpose and the interfacing with other components. Mostly the components in the software industry are best known for reusable software. Reusable software components are generally made up of technology known as Component Object Model (COM). This model is adapted from hardware industry where component technology has been around. It is adapted to the software industry as it is possible to have reusable components that have common interfaces. On top of COM, there is Distributed Component Object Model (DCOM) which can be used in a distributed environment for more sophisticated applications and product families.

Godfrey et al. [80] explored the evolution of software in different perspectives. The difference between software evolution and software maintenance is clarified in their work. The changing nature of software is the main reason to have
configuration management. They opined that maintenance and evolution are not same. As a matter of fact, there is IEEE standard for software maintenance as explored by software engineering standards committee [81].

Salehie and Tahvildari [82] discussed about self-adaptive software. The self-adaptiveness has different levels namely general level with self-adaptiveness, major level with self-configuring, self-helping, self-optimizing, and self-protecting. The primitive layer encompasses self-awareness and context-awareness. They explored four adaptation processes with respect to self-adaptive software. The four adaptations are known as monitoring, detecting, deciding, and acting. These four processes are iterative in nature and they run using closed-loop mechanism. With the advent of distributed technologies and related applications, the self-adaptiveness became a very useful phenomenon. As the applications grow in complexity, it is not possible to have manual monitoring of resource allocation and performance. When the monitoring and performance evaluation are automated it is possible to have self-adaption. The self-adaptive architecture is the architecture that can adapt to runtime situations. For instance, when a distributed application is responsible for providing news to end users, it will be able to render services as per the resource availability. When server-side resources are exhausted, it cannot continue further in the same quality model. Then it can adapt to less quality model which can process more number of requests.

When self-adaptation is achieved, it is possible to have flexible software systems that can work in tandem with hardware availability and resource availability. For instance, when the self-adaptive software architecture is in place, it monitors the deployed application in order to see whether it is working fine. Once it finds changes in the external or internal needs, it will try to adapt to the new situation by configuring itself to a different mode. Thus self-configuration plays a vital role in the success of complex software systems where manual configurations are not feasible. As distributed technologies became available in the real world there are two possibilities in the software product line. Integration of existing software applications or product lines and the possibility of new reusable artifacts t can be used in the product development. In the case of artifacts, it is possible to have component technology adapted from hardware industry in order to have reusable artifacts that can be used in
product line development. In the same fashion, when self-adaptation is made possible, the software system can run without worrying about the capacity of servers and other issues that rise in the runtime.

There are different levels in self-adaptation. They include primitive level, major level, and general level. The primitive level includes self-awareness and context-awareness. The major level includes self-optimization, self-protection, self-configuring, and self-helping. The general level contains self-adaption. The adaptation requirements include the questions starting with what, when, where, why, how and who. When these questions are answered, there will be a good architecture for self-adaption in the complex software execution in the real world.

Hamza [83] focused on dynamic and self-adaptive systems for SPL through feature based generation. Pervasive systems are the systems that are distributed in nature. Realization of such systems is possible with feature based models and specific architecture that utilizes software product lines. Due to invent of technologies there has been increasing need for pervasive systems. Especially with the use of web services technology, it is possible to integrate heterogeneous business and to form supply chains. In this context, it is possible to exploit features in order to produce pervasive software systems. The features are associated with different systems such as context-awareness, fault-tolerance, bridging, privacy, domain-specific and pervasive systems.

Cardozo et al. [84] focused on context-oriented programming paradigm using Petri Nets. Petri Nets is one of the modeling approaches that can help in software modeling. Tourwe[85] explored the evolution of frameworks to support self-adaptation in software architectures. The focus was on automated software evolution and also made use of design patterns as part of the framework. A scheme framework was explored in which various design patterns are used. The design patterns include abstract factory, a chain of responsibility, factory method, composite, visitor, strategy, template, and singleton. They focused on identification of change propagation, the concept of avoiding design drift, formal design constraints, support for change propagation, and meta-patterns. The meta-patterns include creation, hierarchy, recursion, connection and unification. They also included software merging
capabilities such as refactoring, conflict identification, identifying emerging conflicts, identifying obsolete methods, and other conflicts.

Estublier and Clemm [86] studied the impact of the general software engineering research on the usage of software configuration management in the real world. According to them, SCM changed significantly from the 1970s to 2000s. From simple applications to very complex and distributed systems evolved over a period of time that forced the SCM to grow to adapt changes. Software configuration management has its roots in the software engineering research. The SCM systems are evaluated from home-grown, critical software, versioning, with mainframe to ad-hoc system, large software, workspace, the local team in Unix to off-the-shelf, any software, process modeling, anywhere and everybody concept. The evolution refers to the modern application development and SCM.

Gallardo et al. [87] focused on the replication of software components. When software is constructed on time, it will be useful. In this context, reusing components can help to build applications in short span of time. Haupt et al. [88] explored Virtual Machine(VM) product line known as CSOM/PL. This SPL was built by using multi-dimensional separation of concerns. The concern is an aspect that can have a cross cutting presence in SPL. It was adapted to build VM implementations. Their research revealed that VM product line could improve configurability, maintainability, and understandability. Gunther and Fischer [4] explored “rbFeatures” which is nothing but an extension of feature-oriented programming which is used in SPL. By processing Ruby and non-Ruby files, they could generate variants that support feature-oriented programming.

Kroon [89] proposed an SCM model known as layered configuration management. This configuration management is applied to SPL. The layered configuration model contains four levels of layers. The first layer handles version management, branch management, baseline management, and branched baseline management. The second layer contains customization management and feature management. The third layer contains composition management, while the last layer handles product customization management.
In the case of configuration management, the focus points include identification artifacts in level one, identification of variation points in level two, identification of feature compositions in level three, and identification of customizations in level four. With respect to tool support, the functional requirements include basic configuration management, feature management, customization management, composition management, and product customization management. The use cases in SCM include create a product line, edit product line, create a feature, create a composition, create a product, edit feature, edit composition, and edit product. These are service side SCM activities. The client side includes create a working location, check out product line, check out feature, checkout composition, check out product, check-in feature, check in configuration, check the product, and resolve conflicts if any. Other general use cases include propagating feature to composition, propagate feature to a product, promote composition to feature, promote a product to feature, register to feature, register to composition, and register to a product. Thus layered configuration management includes various artifacts and aspects to be considered.

2.4 CHANGE MANAGEMENT

Crnkovic [90] explored change management in SCM. The change process explored is based on the Capability Maturity Model (CMM) and SCM tool. Every change is verified and when it leads to a stable version, then the state is considered to be changed and configuration of artifacts is changed. Change process model is a very important concept in SPL. The reason behind this is that the product lines are built by reusing existing artifacts and building new ones. The concept of product line evolved from reusability of code components. With the concepts like core and custom assets, it is possible to have commonality and variability addressed while making new products.

When a file is subjected to changes, it needs to be configured with a new version number. This is required as there should be two copies of the file that is the file before changes are made and the file after changes are made. When a file is not modified, its state is same and it need not be reconfigured again. Change management
process needs to be controlled. The process includes creating change request, approving change requests, and modifying products accordingly and releasing them. In the process, the modified files are subjected to the software configuration management process. The modified files are kept in the released product. There are four phases in controlling change requests. They include planning phase, implementation phase, verification and integration phase. The change requests in the development process include initiating the project, modifying files, completing changes, testing, approving and releasing. There are certain measurements for change requests. They are CR current status, accumulated CRs, latest changes, CR life length, CR type and priority.

Rajabi and Lee [10] made a comparative study Business Process Modelling (BPM) as part of change management. German and Hindle [91] proposed change metrics that are used to measure fine-grained changes in software. Those metrics are known as modification-aware change metrics. The change metrics they incorporated are entity change metrics, MR-scoped change metrics, time-based change metrics, event-triggered change metrics, modification-unaware metrics, and change metrics that need not measure code. A change metric is something that is applied before and after making a change. Therefore, every metric is associated with two states known as a state before making changes and the state after making changes.

There are different entities and relationship among them in a version control system. For instance, the entities include author, Modification Request (MR), revision and file. The metrics for measuring change needs are to be organized. The metrics include entity metrics, MR scored metrics and time-based metrics. The metrics are modification-aware metrics. They are aware of issues before and after modification. Modification-awareness and modification-unawareness are two categories of metrics. Event triggered change metrics are used in case of certain events. Those events include releases, entity, metric-based event, and author-based events. Modification-unaware metrics that are related to object oriented metrics include delta in the case of a count of entity type, a ratio of change pertaining to an entity type, and change in measurements. Essentially the change metric comes under modification-aware metric.
Ahsan et al. [92] explored on change requests in software engineering. They proposed a method for automatic classification of change requests. Their method works on effort estimation, bug triage that is allocating bug to the suitable developer, and impact analysis. Towards this end, they employed multi-label machine learning methods. The multiple labels include time spent, developer name, and a list of source files associated with the change request. This is called multi-labelled CR data on which multiple methods such as algorithm adaptation and problem transformation are applied. They evaluated their approach with precision and recall measures. Ambros and Lanza [82] proposed a framework named Churrasco that takes care of modeling of software evolution and visualization of the modeling and analysis. It was also known as collaborative software evolution analysis.

Collaborative software evolution is explored in terms of Churrasco framework. The framework has different parts. They are extensible evolutionary meta-model, meta-base, SVN module, web portal and annotation module. The evolutionary meta model is used for internal representation, meta-base is meant for dynamic object relational mapping for persistence. The SVM module is used to keep track of testing aspects and managing bug history. The web portal is used to have web based interface while the visualization module is meant for helping intuitiveness in the analysis. The annotation module is meant for collaborative analysis. The meta model with respect to software configuration management includes different components such as author, commit, comment, action, version, artifact, module, bug report, bug attachment, and bug activity. Accessibility and flexibility are the two important benefits of the framework.

Thao et al. [93] introduced an SCM system for that supports product derivation in SPL. A tool known as MoSPL that can handle versions of different artifacts in SPL was built. Among components, the MoSPL takes care of managing derivation relationships and logical constraints. They explored different versioning models such as composition versioning model, total versioning model, product versioning model, the structure of versioning of components and relationship management. Relationships exploited by the tool include mandatory, optional, alternative, implies, and excludes. Product versioning model as part of software
configuration management with respect to product lines was explored. Composite versioning model was employed where multiple parts are involved in the versioning. Drawbacks of rule-based approach were discussed. The drawbacks of rule-based approach are as follows.

- Indirect representation of configuration can cause issues.
- There is no guarantee for version immutability.
- There is a problem of inconsistency.
- There is a problem of recursion.

They introduced total versioning model. As time goes on and according to changes made versions are created. There is version proliferation problem that needs to be addressed. The model includes version numbers and each version is somewhat different from another version of the same product or artifact. They proposed and implemented a configuration management tool for SPL. Their tool was named as MoSPL which makes use of XML in the process of configuration management. Since XML is the neutral data representation format, they used it to have intermediate data. The XML format is machine-readable, platform independent and device independent. They also used Java as a programming language and UML for modeling. The UML model is used to have a visualization of the artifacts. The tool was a prototype implementation of Molhado framework that supports SPL configuration management and also product derivation.

Svahnberg and Bosch [94] opined that software evolution is a complex process as there is inter-dependency among artifacts used in SPL. Finally, they made certain guidelines for the evolution of SPL. They include avoiding changes to interfaces, making general interfaces to components, separating application and domain behavior of SPL, keeping the SPL intact, detecting and using common functionality, rewrite components and implementations when required, and avoid hidden assumptions in the source code. BSSC [95] focused on providing a guide to configuration management for better results.
Clements et al. [96] explored different project management aspects that apply to SPL which include structuring the organization, risk management, planning, funding, data collection, tracking, metrics, customer interface management and configuration management. Differences between traditional product development and SPL in terms of core asset development, product development, and organizational management were identified. Core assets are reusable artifacts. Product development is done based on customer needs and every product is made up of many reusable assets. Project management is the process of ensuring management activities pertaining to SPL. They include planning, executing, monitoring and controlling.

In the case of traditional project management, the teams were focusing on producing a single product and having time and effort for the same. In modern software development, product lines are given importance. The traditional concept of the project is changed to the new paradigm. The new paradigm shift considers a set of products known as product family or software product line. The key product line activities are the development of core assets, development of products based on the core assets, and managing them using software configuration management activities adapted to product lines. There are certain differences between traditional products and product lines. They are discussed in terms of scoping, planning, and developing core assets. It also involves producing individual products. Organizational management compared include developing a business case, technology forecasting, establishing policies, producing a product line, training, and product line adaptation. There are certain management practices that are applicable to all projects. They include structure of the organization, risk management, planning, funding, data collection, tracking and metrics, customer interface management and configuration management.

Building a business case includes market analysis, scoping, market analysis, technology forecasting and a business case. The scoping includes building a business case, market analysis, product-line scope, and understanding relevant domains.

Amhed et al. [97] explored key business processes that are to be considered in SPL. They include market orientation, the order of entry, brand, strategy, business vision, strategic planning, assets management, innovation and financial management.
They opined that these are important activities that can help organizations who want to be in the business of software product family.

2.5 PRODUCT DERIVATION APPROACHES IN SPL

Rabiser et al. [1] focused on the key activities involved in product derivation. Pro-PD and DOPLER were the two tools proposed and compared. The key activities involved include translating customer requirements, coverage analysis, creating product-specific requirements, implementing scope requirements, creating test-cases that are product-specific, allocating requirements and create guidelines for decision makers. Tawhid and Petriu [98] explored the concept of model transformation in SPL through product model derivation. The product model derivation is the process of mapping between the features present in the feature model to actual implementations in the SPL. As product derivation is very important part of SPL, they focussed on model transformation which can be automated. The model transformation can convert the model from one form to another form. The models include platform independent model (PIM) and platform specific model (PSM). These models, as the names imply, are considered with respect to the platform. For instance, when a model is related to code in the specific language, it is called platform specific model. The examples for platform independent models include UML models, DFD and ERD. UML MARTE profiles are also used to work with model transformations. UML MARTE can help in the understanding model as it is machine readable. The UML software models extended by MARTE can provide a highly programmable interface for model transformations. The UML models become more and more intuitive when MARTE profiles are associated. This can help in software product line while developing and managing.

They explored a case study using e-Commerce to demonstrate feature dependency. The features in a product line can have dependencies. When feature models are built, it is possible to have details of SPL and thus use them in the product derivation. In this thesis, the feature models are used to construct ontology for a better programmable interface. Thus dynamic configuration management and quality product derivation are achieved. The e-Commerce SPL with MARTE profiles is the
best candidate to understand the power of feature models and the need for organizing them and using them in machine readable format. The researchers also focused on application engineering process using e-commerce case study. Even the class diagrams can be produced with MARTE profiles. The use cases and classes in the case study can have an implicit and explicit mapping. There are model transformation rules considered. The model transformation rules are applied when the model is transformed from one format to another format. An important observation made by the researchers is that SPL approach improves productivity besides reducing the time taken to realize products.

Product derivation is an important activity in SPL and its configuration management. SPL configuration management can help in deriving new products easily. However, there are different approaches that can be used to do so. Rabiser et al. [99] proposed a framework that contains key activities for product derivation. The activities are categorized into three parts. Preparing for derivation is the first category which will result in derivation guidelines. The second category leads to actual product derivation while the third category activities take care of additional development and testing the product for improving its quality. Pedrycz et al. [100] opined that neural networks and fuzzy logic can be applied to improve software quality. Siegmund et al. [101] explored product derivation in software product line by measuring nonfunctional properties. They effort was to optimize products in SPL towards nonfunctional properties. User-defined and non-functional properties are used to derive high-quality product. Nonfunctional properties are classified into three categories direct assigned properties, inferred properties and runtime properties. The first category properties are considered fully or partly as features. The second category properties are either measured in isolation or products are used to infer the properties. Runtime properties are the properties that surface in the running product. Used memory, power consumption, and performance are the examples for runtime properties. By mapping directly assigned properties to product line model, and by measuring inferred properties and runtime properties lead to optimization of product derivation.
Guana and Correal [102] explored a quality-driven approach for SCM. They built a model-driven strategy in order to derive a new project based on the needs of customer and quality requirements. They used the strategy to improve the selection process when a new product is derived. However, they followed domain centric approach and improved the possibility of selecting most suitable artifacts while defining a new product in SPL. Deelstra et al. [15] focused on a framework for product derivation in SPL. They opined that there was less research done on the product derivation. They studied product derivation issues in SPL and worked out product derivation strategies for two companies namely Thales Netherland B.V, and Robert Bosch GubH.

2.6 VARIABILITY-AWARE DESIGN PATTERNS AND ONTOLOGY

Recently multiple universities collaborated to expedite research on variability-aware software. Wanner et al. [103] focused on the mission of National Science Foundation (NSF) for specific research on variability–aware software. The research resulted in key findings such as variability-aware storage, variability-aware software and variability-aware task scheduling and so on. Guthaus and Wilke [104] focused on variability-aware clock design. They could find different variations in terms of process, voltage, temperature and cross talk. This kind of approach is also applicable to SPL. Seidl et al. [105] proposed variability-aware approach known as generative SPL development based on design patterns. They opined that variability-aware design pattern is one of the best practices that can be used in SPL and its configuration management. This model can govern configuration knowledge and improve the quality of configuration management of SPL. In [106] feature models are used to know how best the variability can help in improving SPL development, configuration, and product derivation. The design pattern specification can help in building SPL that can reap advantages of design patterns that are key building blocks of reusability and systematic software development.

Leung and Lui [73] focused on the variable information for testing software systems. They employed the historical test cases available for finding variability aware information for performing tests on software variability. In the integrated
software product family, this kind of testing could lead to more robust products to be derived. While design patterns are very useful for reusability and systematic approach in software design, finding relationships among design patterns can help in promoting variability in SPL. Hachemi et al. [74] focused on software patterns and their extraction automatically for improving the variability-aware design in software product lines. The software patterns obtained can be used further to find hidden relationships among them. The extraction of relationships can help in making well-informed decisions. Ayora et al. [75] proposed a framework named “VIVACE” for evaluating variability support in information systems that are process-aware. Especially, workflow management systems with different variabilities can be leveraged with the framework.

Christopher et al. [16] a design pattern approach is followed for investigating designs and the reuse of them in software development. They performed pattern mining which could help in building various software products that are having commonality with certain customized or variable functionalities. Myllarniemi et al. [76] proposed a case study for presenting performance variability in SPL. Based on customer needs, constraints, and tradeoffs they built a model for performance variability. This model provides insights of performance variability in SPL which helps in better configuration and product derivation. Mikael et al. [94] evaluated variability concepts for Software Product Line Engineering (SPLE). The concepts they explored include 150% models and transformation models.

Becker [107] proposed a model for variability in SPL to manage variability systematically and efficiently. Parra et al. [108] explored Aspect Oriented Programming (AOP) and design patterns to enhance SPL and runtime adaptation. Tizzei et al. [109] also studied AOP for assessing design stability of SPL by comparing Object Oriented(OO) and Aspect Oriented (AO) approaches. Fortier et al. [110] proposed design patterns for dealing with variability in SPL pertaining to mobile software. The impact of variability in different domains was studied. Hammouda et al. [111] studied feature-driven variability and design patterns for separation of concerns for better change management.
Alves et al. [30] analyzed variability and commonality in variation management between RE (Requirements Engineering) and SPL. Schuster [112] employed role modeling for pattern-based SPL design. Roles present in a variability-aware design pattern are mapped to elements of artifacts in SPL. The variability-aware design patterns were explored for enhancing SPL.

According to Asikainen et al. [18] variability refers to the ability of a software system to get extended or customized or reconfigured. They threw light on domain ontology named as “Kumbang” for modeling variability features in SPL. They made use of UML profiles and natural language to describe the semantics of their domain ontology. The representation of knowledge is done using ontology. In other words, the modeling of SPL its variability features i.e., knowledge is represented graphically to simplify and comprehend with ease. On the contrary to this, Luiz et al. [98] proposed a state-based modeling for representing SPL and its safety analysis. Therefore, it can conceive that ontology can also be used to have such safety analysis and aligned to activities of configuration management and product derivation. Sudarsan et al. [113] proposed a framework for modeling product life cycle management. This could apply to families of products as well. Moreover, they used an ontology to achieve modeling. That resulted in fine-grained information representation that could help in SPL activities. D.J.Wu [114] promoted the concept of software agents for knowledge representation and management thus leading to the configuration management of complex software systems.

Ontology is one of the classic domains for knowledge representation. Shahri et al. [115] proposed an approach for configuration management using ontologies. They used standard OWL-DL language for encoding version restrictions and component constraints. As OWL language is machine readable it is a suitable candidate for knowledge representation. With ontology inconsistencies in components is identified and test configurations are done automatically. According to them, there are many advantages of using ontology in software configuration management. The advantages are as follows.

- Version restrictions and configuration constraints can be formalized. It will help in better knowledge representation.
• Helps in automatic detection of issues in configurations by improving consistency checking, concept satisfiability, classification and realization.

• Configuration management planning, configuration item identification, configuration ontology, version management and version identification, release creation, system building are all the aspects that are benefited using ontology.

Especially software configuration dependencies and related constraints can be identified and adapted effectively using ontology usage in configuration management. In the same fashion, ontology can help in better analysis while products are derived as it renders programmable interface.

2.7 MORE ON SPL CONFIGURATION MANAGEMENT

This section provides the review of the literature on various approaches to configuration management of SPL. There are many conventional SCM tools such as CVS, SVN, GIT, Mercurial [116] and so on. Study on these tools revealed that they support many common features besides having unique features as shown in Table 2.1.

Clements and Northrop [13] proposed a generic model for SCM as shown in Figure 2.1. The configuration management considers various artifacts such as core assets, custom assets, and products derived. Every product that is under the purview of configuration management should have a product-in-use which has been released and being used by customers. The main focus of this model is the development of core assets and custom assets. There exists one to one relationship between product instances and product-in-use. The model supports the development of core assets and custom assets independently. When all instances are under SCM, they can be accessed with ease. The evolution of artifacts takes place based on customer feedback while the updated product instances are released again.

Due to the nature of SPL, there are some issues to be resolved according to Staples [11]. The issues include multi-dimensional nature of SPL configuration, changes to SPL architecture, different release constraints of products, chance of
duplicating efforts in disguise of reusability. In order to handle these issues, traditional SCM is not sufficient.

Table 2.1: Commonalities and differences among SCM tools

<table>
<thead>
<tr>
<th>#</th>
<th>Requirements</th>
<th>CVS</th>
<th>SVN</th>
<th>GIT</th>
<th>Mercurial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The product model uses an object ID</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>The product model is based on a file system</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>The version model uses files only</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>The version model supports files and directories</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>The version model is state-based</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>The version model supports predecessor/successor relationships</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>7</td>
<td>The version model supports alternative relationships</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>8</td>
<td>The version model supports merge relationships</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>9</td>
<td>Both models are integrated in a product-centered way</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>10</td>
<td>Both models are integrated in a version-centered way</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>11</td>
<td>The storage uses backward deltas to save disk space</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>12</td>
<td>The storage uses references to common contents to save disk space</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>13</td>
<td>The SCM system synchronizes a central repository and workspaces by &quot;commit/checkout&quot;</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>14</td>
<td>The SCM system supports the replication of the repository</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>15</td>
<td>The SCM system supports the synchronization of distributed repositories per &quot;push/pull&quot;</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>16</td>
<td>The SCM system supports pessimistic locks</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>17</td>
<td>The SCM system supports optimistic locks</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Staples suggested guidelines to overcome the issues. They are nonusage of versions for more recent core assets and using branch versions specific to products in some cases. Van Gurp [117] proposed the combination of SVN and variability tools towards better configuration management. He opined that SVN is suitable as it supports file system, annotations, and copy by reference for versioning. In the generic
SCM model described by Clements and Northrop [13] core assets, custom assets and product instances are kept under configuration management. For each product instances under SCM, there is a corresponding product in use.

Van Ommering [8] also tried in a similar fashion to combine the traditional SCM tools and component based products as part of SPL for configuration. The component used has three kinds of interfaces such as diversity interfaces, requires interfaces, and provides interfaces. The environment is provided for these interfaces. For fine-tuning components, diversity interfaces are used. The component needs are reflected by required interfaces. Thus many components are combined to form composite components which are used to produce products.

Variation management is an important activity in SPL configuration. As shown in the Figure 2.2 the pre-processors such as #ifdef are used to handle variation at the component level. Temporary variation concept is used to handle branch variation. Temporary variation refers to moving a package to a product just before the product is released for use thus avoiding introduction of bugs when new features are incorporated [20]. Divide and conquer is the phenomenon used in SPL configuration management. In this approach, variation management is given much focus. There are
variation management clusters under sequential time, a parallel time and domain space with respect to files, components, and products. Files are associated with version management, branch management, and variation point management. The components are associated with baseline management, branched baseline management, and customization management. The products are associated with composition management, branched composition management, and customization composition management. The traceability can help reduce or prevent product line decay. There are certain tasks associated with traceability in case of configuration management of SPL.

![Variation management clusters](image)

Figure2.2: Variation management clusters

The granularity of SCM is shown where different variations are in columns while the files, components, and products are in rows. Evolution of software artifacts is represented by variation in parallel time. The same in domain space is different for different products. Files are used to create variations in components and the components and their variations exist in products. Files and components under variations in sequential time and parallel time can be configured using traditional SCM tools. However, component composition tools are required to configure products. Component composition and software mass customization are important aspects in a product line as described by Krueger [7]. According to Krueger products are nothing but transient outputs. Common and variant artifacts are subjected to changes.

The composition of components is the process in which different components are used to form a product. Customization refers to the process of adding specialities
to a product. These two activities need configuration management in order to provide versions and track them over time. Context approach was suggested by Krueger. A possible composition for various components can be represented by a context. This is controlled by SCM. Customized components can be used to derive customized products. In this process variation points play an important role.

![Image](image.png)

**Figure 2.3:** Variation management in SPL

For SPL configuration management Krueger’s model can be used. It includes files, components, and products. Krueger [7] describes an approach that uses conventional SCM tools. The core assets and product line instantiation infrastructure are kept under SCM as shown in Figure 2.3.

The feature based product line was focused by Van Deursen et al. [118]. In the product line named “DocGen”, focus was on source-level packages for feature based approach. A package can share functionality provided by parent packages or it can implement a feature. By merging multiple packages a product is built. Feature descriptions are used in SCM for variation management. In [119] feature based variation management is described using Feature Oriented Domain Analysis method (FODA). Product instances are generated using an FDL definition. For instance, deriving a product for a customer DocGen needs selection of variable features, packages, and configuration switches. Packages are independent of any traditional SCM tool available.
Buchmann et al. [60] proposed a model-driven product line while Guana and Correal [61] proposed a quality driven approach for the same. Their work is focused on the configuration management of SPL. Buchmann et al. presented a method and developed a tool for model-driven SPL engineering, used FORM method integrated with SCM domain to have a feature model for SCM for SPL. They built a prototype in OO domain model that supports feature model. On the other hand, Guana and Correal followed a different approach for SCM of SPL. The model-driven approach was used for finding critical points in SPL architecture. Their strategy is able to choose a product automatically when quality and functional requirements are provided and used mobile healthcare SPL for experiments. Selection of reusable assets using quality attribute driven approach is the important contribution towards SCM for SPL.

According to Buchmann et al, model drive approach is useful for product derivation. The engineering process is categorized into domain engineering and application engineering. The domain engineering includes domain, domain analysis, feature model and developing a configurable domain model. The application engineering model contains application specific requirements; configure features, feature configuration, and configuring domain. The challenges here include feature model, domain model, mapping and configuration. The domain analysis they made include the SCM domain, feature model, a sample configuration, the modularization approach they focused include different architectural principles such as separation of concerns, abstraction, information hiding, loose coupling, and high cohesion. They focused variability at different levels of granularity. Action level, method level, class level, and package level was included.

In the generic SCM model described by Choi and Scacchi [24] core assets, custom assets and product instances are kept under configuration management. For each product instances under SCM, there is a corresponding product in use. Van Gurp [117] proposes coupling variation modeling tools with subversion to support product derivation. Kruger [7] describes an approach that uses conventional SCM tools. Core assets and product line instantiation infrastructure are kept under SCM. According to Krueger products are generated and are not kept under SCM. All changes are made in
core assets and custom assets. Since changes occur in the core assets, a product that uses the latest version gets the new changes. Dependency among components and products is manually maintained.

Molhodo SPL [93] is a prototype to solve the evolution problem at the configuration management level instead of at the source code at the programming language level. Changes can propagate from the core assets and custom assets to products or from products to core assets and custom assets. The proposed method allows product specific changes to shared components without interfering with the changes made to the referred component in the core and custom asset. To support product specific changes to shared core assets and custom assets in order to avoid interference between the product’s changes and changes to the core assets and custom assets, the core asset and the custom asset creates a product specific branch to support the changes. When a product developer checks in their product with changes to a shared core asset or custom asset, the core assets and custom assets create an automatic branch to support it. The subsequent check-in of changes to this shared asset for that particular product creates more versions of the product specific support branch created earlier.

In [153] Farahani et al proposed change management and version management models to cover two main activities of Configuration Management for Software Product Lines. In the change Management and version management models, different scenarios for change propagation in Domain Engineering and Application Engineering phase are proposed. Nesrine et al[154] proposed a Model Driven Approach in order to reduce the time taken for product derivation. In [155] Cuong et al proposed an architecture-centric approach that includes PLA (product line Architecture)implementation mechanism and a strategy of automatically deriving product specific architecture and code from PLA.