

development (domain engineering) involves the creation of common assets and the evolution of the assets in response to product feedback, new market needs, etc. Product development (application engineering) creates individual products by reusing the common assets, gives feedback to core asset development, and evolves the products. Management includes technical and organizational management, where technical management is responsible for requirement control and the coordination between core asset and product development activities.

There are two main software product line adoption approaches: *big bang* and *incremental* [10]. With the big bang approach, core assets are developed for a whole range of products prior to the creation of any individual product. With the incremental approach, core assets are incrementally developed to support the next few upcoming products. In general, the big bang approach has a higher return on investment but involves more risks, while the incremental approach has lower entry costs but higher total costs. The four common software product line adoption situations are: *independent*, *project-integration*, *reengineering-driven*, and *leveraged* [10]. Under the independent situation, a product line is created without any pre-existing products. Under the project-integration situation, a product line is created to support both existing and future products. Under a reengineering-driven scenario, a product line is created by reengineering existing legacy systems. And the leveraged situation is where a new product line is created based on some existing product lines.

Some common product line evolution strategies are: *infrastructure-based*, *branch-and-unite*, and *bulk-integration* [10]. The infrastructure-based strategy does not allow deviation between the core assets and the individual products, and requires that new common features be first implemented into the core assets and then built into products. Both the branch-and-unite and the bulk-integration strategies allow temporal deviation between the core assets and the individual products. The branch-and-unite strategy requires that the new common features be reintegrated into the core assets immediately after the release of the new product, while the bulk-integration strategy allows the new common features to be reintegrated after the release of a group of products.

2.2. Simulation

A software process is a set of activities, methods, practices, and transformations that people use to develop and maintain software and associated products, such as project plans, design documentations, code, test cases, and user manuals [4]. Adopting a new software process is

expensive and risky, so software process simulation modeling is often used to reduce the uncertainty and predict the impact. Software process simulation modeling can be used for various purposes and scopes, and have been supported by many technologies [3]. The software product line process simulator described in this paper is for long-term organization strategic management, and is implemented in DEVJSJAVA [1], a Java implementation of the Discrete Event System Specification (DEVS) modeling formalism [1].

The external view of a DEVJSJAVA model is a black box with input and output ports. A model receives messages through its input ports and sends out messages via its output ports. Ports and messages are the means and the only means by which a model can communicate with the external world. A DEVJSJAVA model is either "atomic" or "coupled". An atomic model is undividable and generally used to build coupled models. A coupled model consists of input and output ports, a finite number of (atomic or coupled) models, and couplings. The couplings link model ports together and are essentially message channels. They also provide a simple way to construct hierarchical models.

To execute atomic and coupled models, DEVJSJAVA uses distinct atomic and coupled simulators that support incremental simulation model development. These simulators can execute in alternative settings (i.e., sequential, parallel, or distributed). An important feature of the DEVS framework is the ability for models to seamlessly execute either in logical or (near) real-time. Furthermore, due to its availability of its source code and object-oriented design, DEVJSJAVA can be extended to incorporate domain-specific (e.g., Software Product Line) logic and semantics.

2.3. COPLIMO

In the simulator, COMPLIMO [2] is used as the cost model to provide cost estimates. COPLIMO is a COCOMO II [9] based model for software product line cost estimation, and has a basic life cycle model and an extended life cycle model. The basic life cycle model has two sub-models: a development model for product line creation and a post-development model for product line evolution. Although the basic life cycle model has many simplification assumptions, it is thought to be good enough for early stage product line trade-off considerations [2]. The basic model also can be easily extended to the extended life cycle model, which allows products have different parameter values instead of the same values. In the implementation, the cost model is designed as a plug-in model, thus other cost models can be plugged in to meet other needs.

3. Approach

A simulation framework and a software cost model were used to develop the simulator. Although DEVJSJAVA [1] and COMPLIMO [2] are currently used, they can be replaced by other suitable simulation frameworks and cost models. This section presents the abstract software product line engineering model, the specifics of the simulation models, the assumptions, and the simulation tool.

3.1. Abstract product line model

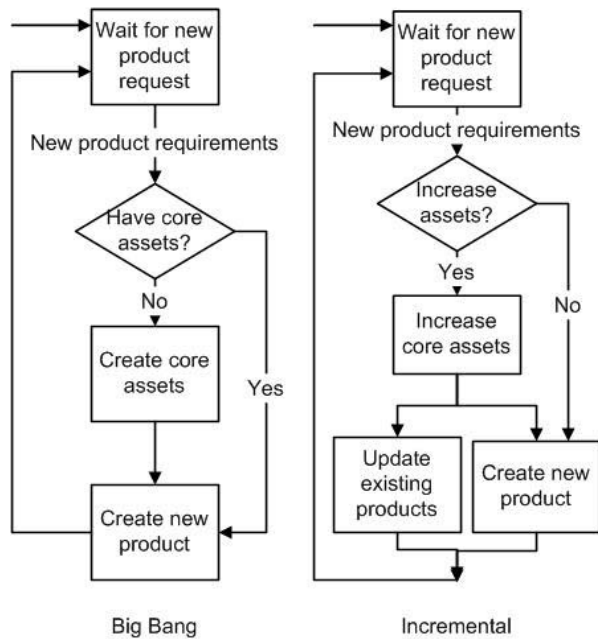


Figure 3.1. SPL adoption approaches

Software product line engineering typically involves a creation phase and an evolution phase [10]. Currently the simulator provides two options (big bang and incremental) for the creation stage and two options (infrastructure-based and branch-and-unite) for the evolution stage. In the following, we will discuss the costs associated with those cases in detail.

With the big bang adoption approach, core assets are first developed to meet the requirements for a whole range of products. Products are then developed by reusing the core assets [10]. Figure 3.1 illustrates the process flow. Costs associated with this approach include core asset development costs and new product development costs.

With the incremental adoption approach, the core assets are incrementally developed to meet the requirements of the next few upcoming products, new

products are developed by reusing the existing core assets, and existing products are updated after the change of the core assets. Figure 3.1 depicts the process flow. Costs associated with this approach include core asset development costs, new product development costs, and existing product maintenance costs. Compared to the big bang approach, the incremental approach has higher product development costs because of the incompleteness of the core assets, and extra product maintenance costs as the result of a short-term planning penalty.

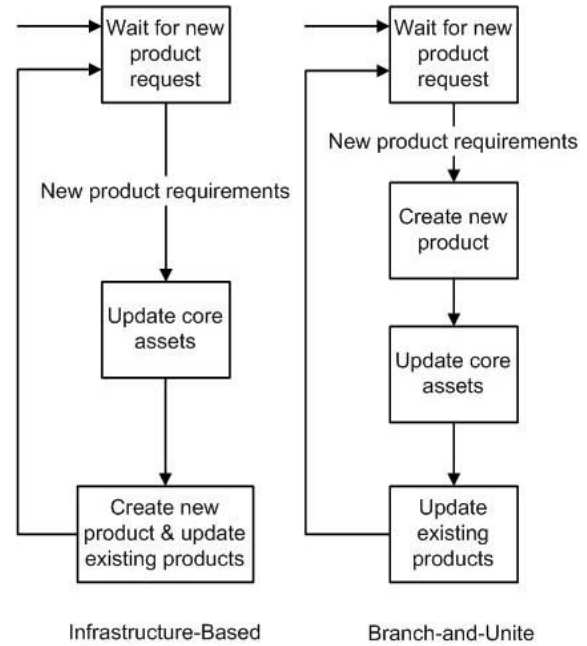


Figure 3.2. SPL evolution approaches

With the infrastructure-based product line evolution strategy, the process for building a new product is the following: core assets are updated by incorporating new common requirements, and the new product is developed and existing products are updated. Figure 3.2 shows the process flow. The COPLIMO [2] basic life cycle model assumes that a change to a product causes the same percentage of change on reused code, adapted code, and product unique code. So if the change rate caused by new product requirements is α , then the costs for one product development iteration include the costs of maintaining the core assets with a change rate of α , the costs of developing the new product, and the costs of maintaining existing products with a change rate of α .

With the branch-and-unite product line evolution strategy, the process for building a new product is the following: the new product is developed, core assets are updated to incorporate new common features, and existing products are updated (including the newly

created product). Figure 3.2 illustrates the process flow. If α is the change rate caused by new product requirements, then the costs for one product development iteration include the costs of developing a new product with $(1-\alpha)$ percentage of the reuse rate, the costs of maintaining the core assets with a change rate of α , and the costs of maintaining existing products (including the newly created one) with a change rate of α . Product maintenance costs are higher in this case because it has one more product to update, the newly created one. The new product is first created with new features that are not supported by the core assets, then after the core assets are updated to incorporate the new features the new product needs to be updated to keep consistent with the core assets. The new product development costs are also higher with this approach, because of the lower reuse rate.

3.2. Model development

Twelve DEVSJAVA [1] models were developed to model software product line engineering activities. Figure 3.3 shows a UML diagram depicting the hierarchical structure of the model. Some time constraints are imposed in the simulator: for each atomic model, jobs are processed one by one in FIFO order (or in combination with some priority).

The PLPEF (Product Line Process Experimental Frame) is the top level coupled model and contains a Product Line Process instance and an Experimental Frame instance.

The Product Line Process models software product line engineering activities. It contains an instance of Technical Management, Core Asset Development, and Employee Pool, and a finite number of Product Development instances. The number of Product Development models to be included depends on the number of projected products in the product line. The Product Line Process receives market demands and dispatches them to Technical Management. It sends out requirements (generated by Technical Management and Maintenance Requirement Generator) and development reports (generated by Core Asset Development and Product Development), which can be used for process monitoring.

The Employee Pool models human resource management. It receives resource request and resource return messages, and sends out reply messages to grant resources. Currently, Employee Pool manages the resource requests in either a pure FIFO manner or a FIFO manner where new development jobs are given higher priority. Before starting any development activity, a resource request must be sent to Employee Pool. A

project cannot be started until the requested resources are granted from the Employee Pool. If the number of employees in the employee pool is not less than the requested number of employees, the requested amount of employees will be granted. Otherwise, if the number of available employees meets the minimum employee level (a model parameter, between $5/8$ and 1), then the number of available employees will be granted. In that case, a job can be started with fewer resources but longer development time. In other cases, the employee pool will not grant any resources until enough resources are returned.

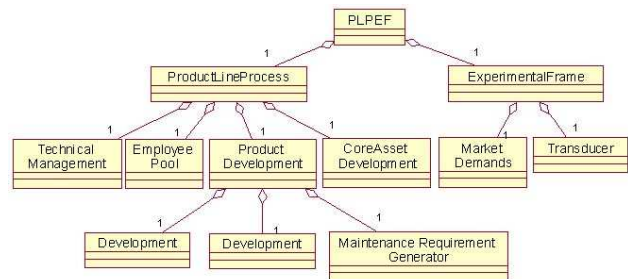


Figure 3.3. Model hierarchical structure

The Product Development models the application engineering activity. It has a Development instance for product creation and inter-product synchronization (development instance), a Development instance for inner-product maintenance (maintenance instance), and a Maintenance Requirement Generator instance. When the Product Development gets the first requirement, the development instance starts product creation, once that is done, the Maintenance Requirement Generator sends out a requirement to maintain the product for N years (the number of years in the product life cycle), which starts the maintenance instance. After N years, the Maintenance Requirement Generator sends out a stop message, which stops the maintenance activity and the acceptance of new development requirements.

The Development models a general software engineering activity. When a new requirement is received, Development sends a resource request to the Employee Pool, waits for the reply from the Employee Pool, starts developing activity when the resources are granted, then returns resources to the Employee Pool and sends a report to Technical Management upon completion. The Development model will stop accepting new requirements when it receives a stop message on its stop port, which means the product reaches the end of its life cycle and needs to be phased out.

The Maintenance Requirement Generator models product maintenance requirement generation. Once a new product is released, it sends out a requirement to maintain the product for N years (the number of years in

the product life cycle), and sends a stop message when the product reaches the end of the product life cycle.

The Core Asset Development models domain-engineering processes. Currently, it is modeled in the same way as the Development model. The domain engineering is not modeled as the same as the application engineering, because in practice technical management often collects the core asset feedback from product development and issues the core asset requirements in the context of product development.

Table 3.1. Behavior of technical management

Stage	Approach	Activities
Creation	Big bang	<ol style="list-style-type: none"> 1. Create core assets if they do not exist 2. Create new product by fully reusing core assets
	Incremental	<ol style="list-style-type: none"> 1. Increase core assets if necessary 2. Create new product by fully reusing core assets 3. Update existing products
Evolution	Infrastructure-Based	<ol style="list-style-type: none"> 1. Update core assets 2. Create new product by fully reusing core assets and updating existing products (excluding the newly created product)
	Branch-and-Unite	<ol style="list-style-type: none"> 1. Create new product by partially reusing core assets 2. Update core assets 3. Update existing products (including newly created product)

The Technical Management models requirement generation and control as well as the coordination between core asset and product development. It receives market demands (which are processed in FIFO order), generates requirements for core asset or product development, and accepts development reports. Which requirements will be generated, when will they be generated, and where they will be sent depend on the selected product line adoption and evolution approaches. Technical Management coordinates core asset development and product development activities through keeping the requirement generation in a certain order. Table 3.1 summaries the behavior of Technical Management according to the given strategies.

The Experimental Frame consists of a Market Demand instance and a Transducer instance. It feeds Product Line Process with inputs and receives Product Line Process' outputs.

The Market Demand models the demands for new products from the market. It sends out a new product request after a certain interval, which can be set through the model parameter, "interval".

The Transducer observes product line engineering activities for a certain amount of time. During the observation period, it receives development requirements and reports, and tracks the generating and finishing time of each requirement. At the end of a simulation run it will output some statistical information to a file.

3.3. Assumptions

In the simulator, we made a number of assumptions as follows.

1. All the employees have the same capability and can work on any project.
2. If task B needs to make use of the results from task A, task B cannot start until task A is finished.
3. Product maintenance starts right after the release of the product and the maintenance activity holds the staff until the product is phased out.

The assumptions made by the COPLIMO [2] basic life cycle model are:

4. All products in the product line have the same size, the same fractions of reused (black-box reuse) code, adapted (white-box reuse) code, and product unique code, and the same values for cost drivers and effort modifiers.
5. For each product in the product line, the size, the values of cost drivers, and the values of effort modifiers remain constant over time. For each product, the fractions of reused code, the fractions of adapted code, the fractions of product unique code remain constant during the time when core assets stay the same.
6. A change to a product will cause the same percentage of change to reused code, adapted code, and product unique code.

Assumption 2 states that concurrency between interdependent tasks is not supported in the current version, which will be supported to some extent in the future. Assumption 4 can be relaxed by using COPLIMO [2] extended life cycle model, which allows products to have different parameter values. Assumption 5 can be relaxed by allowing users to specify the change trend. Assumption 6 can also be relaxed by allowing users to specify the change rate on different portions of the products. Because COPLIMO is currently used as the underlying cost model, its assumptions are adopted in the simulator. If another cost model is used, these assumptions would be replaced by those made by the new cost model.

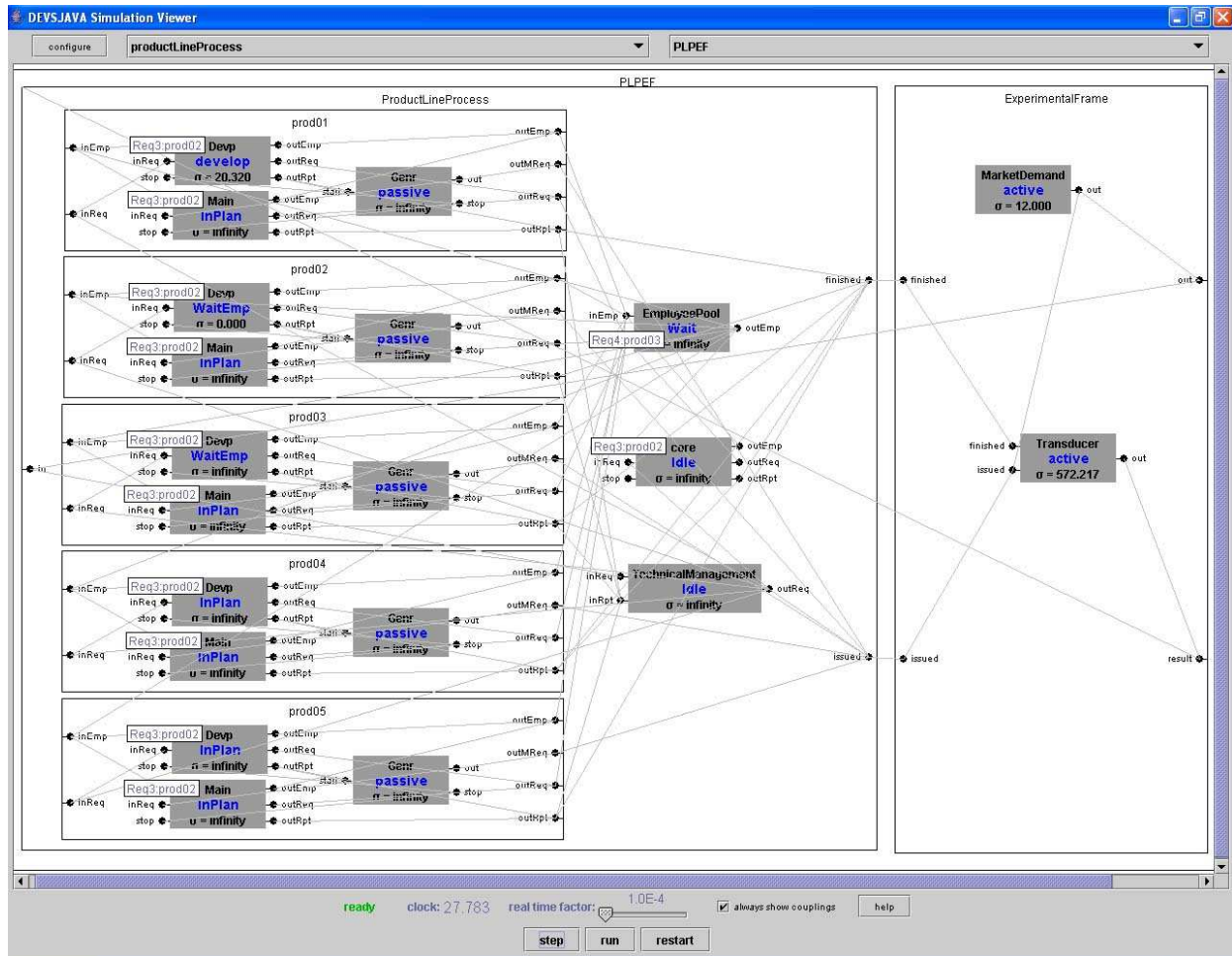


Figure 3.4 Simulation tool in execution

3.4. Simulation tool

The simulation tool was developed in Java and runs in the DEVSJAVA [1] environment. Figure 3.4 shows the user interface. The upper part of the interface shows the current running model and its package, which are “PLPEF” and “productLineProcess”, respectively. The middle part shows the models and their hierarchical relationships. The bottom part contains execution control components. The “step” button allows running the simulator step by step, the “run” button allows executing the simulator to the end, and the “restart” button allows starting a new simulation run without quitting the system. The “clock” label displays the current simulation time in the unit of months, and selecting the “always show couplings” checkbox will allow couplings between models to be displayed. The simulation speed can be manipulated at run time to allow execution in near real-time or logical time (slower/faster than real-time).

Figure 3.4 shows that at time 27.783, Core Asset Development is idle, Products 1 is under initial development, Product 2 and 3 are waiting for resources, and Products 4 and 5 are in planning. The messages tell that Product 2 just received requested resources and Product 3 just sent out a resource request. Because of the lack of resources, the Employee Pool cannot grant the requested resources and is waiting for more resources to be returned, which in turn puts Product 3 in wait. Technical Management is idle, Market Demand generates new product requirement in every 12 months. Finally, and Transducer is observing the simulation. This case shows a situation where limited resources cause development activity delay.

At the end of each simulation run, a result table is generated similar to Table 3.2. The table has two sections that are divided by a blank line. The top section lists the created products, their first release time (FR), time-to-market (TTM), initial development effort (DPM), initial development time (DT), accumulated development and maintenance effort (APM), accumulated

development and maintenance time (AT), and the number of finished requirements (FR). The bottom section summarizes the total product line evolution effort (TPM), the time when all the requirements are finished (FT), the average annual development effort (APM), the number of requirements generated (TR), and the average time-to-market (ATM). The unit of effort is person-months and the unit of time is months.

Table 3.2. Simulation Result

	FR	TTM	DPM	DT	APM	AT	FR
core	27.8	27.8	582.2	27.8	651.6	50.5	3
p01	48.1	48.1	217.7	20.3	443.0	159.0	4
p02	48.1	36.1	217.7	20.3	443.0	159.0	4
p03	68.4	44.4	217.7	20.3	443.0	159.0	4
p04	79.8	43.8	217.7	20.3	424.2	149.6	3
p05	100.1	52.1	217.7	20.3	405.4	140.3	2
TPM	2810.1						
FT	220.1						
APM	153.2						
TR	20						
ATTM	44.9						

4. Results

In this section some simulation cases are presented to illustrate the use of the simulator and to demonstrate the analytical capability of the simulator.

4.1. Overview

The inputs to the simulator include *general* parameters and *product* (core asset) parameters. The general parameters are used to describe software product line process attributes and organization characteristics. These parameters include the maximum number of products that will be supported by the product line, the number of products that will be created during the creation stage, the product line adoption and evolution approaches, the number of employees in an organization, and the market demand intervals. The product (core asset) parameters are primarily determined by the employed cost model (COPLIMO, in this case). These parameters include the size of the product (core assets) in source lines of code, fraction of product unique code, fraction of reused code, fraction of adapted code, percentage of design modified, percentage of code modified, and percent of integration required for modified software. In addition, a number of parameters related to reuse and maintenance are included, such as software understanding of product unique code, software understanding of adapted code, unfamiliarity with

product unique code, unfamiliarity with adapted code, and average change rate caused by new market demands.

To study the effect of resources, adoption approaches, and evolution approaches on software product line engineering, we ran the simulator seven times using the same basic parameter values. Accordingly, we varied the number of resources, the type of adoption approach, and type of evolution approach.

Table 4.1. Scenarios

Scenario	Market Demand Interval	Resources	Single product only	Big bang	Incremental	Infrastructure-based	Branch-and-merge
1	12	50		✓		✓	
2	12	40		✓		✓	
3	12	30		✓		✓	
4	12	50			✓	✓	
5	12	50		✓			✓
6	12	50			✓		✓
7	12	50	✓				

4.2. Effect of resources

The inputs to Scenarios 1, 2 and 3 only differ in the values of number of employees (50, 40, and 30, respectively). The results differ in time-to-market, as shown in Figure 4.1. At the beginning, there is little difference, as time progresses, the gap of time-to-market between resource-constrained and non-resource-constrained scenarios increases. The reason is that as more products are developed, more resources for product maintenance are required, thus less resources are left for new product development, which may increase the resource waiting time and in turn result in a longer time-to-market. The effort associated with Scenario 3 is smaller than the other two cases. That is because in Scenario 3, when Product 10 is released at 202.69, Product 1 and 2 are already phased out (at the time 168.1). Accordingly, no effort is needed to update those two products due to the change of the core.

4.3. Effect of adoption approach

The inputs to Scenarios 1 and 4 only differ in product line adoption approach (big bang and incremental, respectively). The results differ in time-to-market, as shown in Figure 4.2. As specified by the inputs, the core

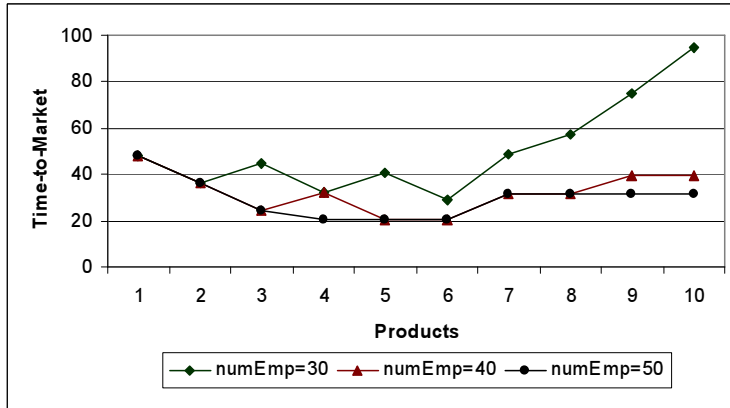


Figure 4.1. Effect of resources

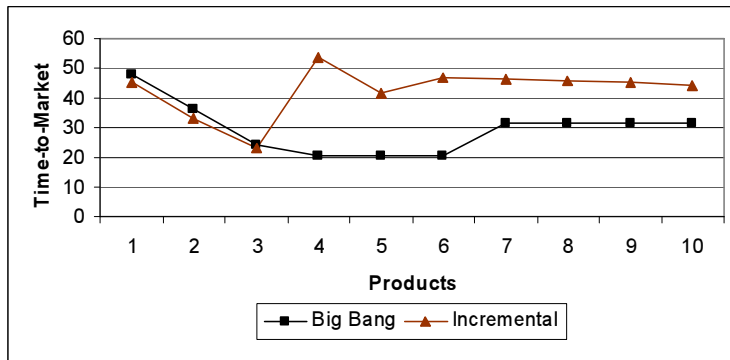


Figure 4.2. Effect of adoption approach

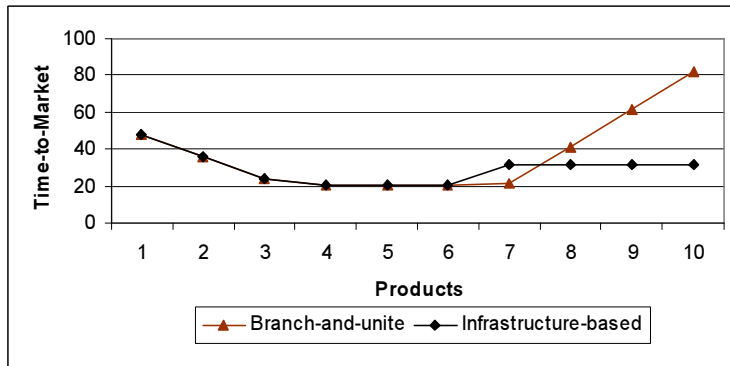


Figure 4.3. Effect of evolution approach

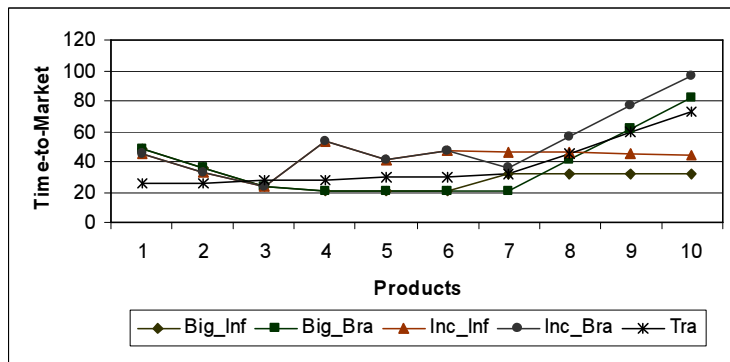


Figure 4.4. Effect of combined adoption and evolution approach

assets are developed in two steps with the incremental approach. The first increment happens right before the development of Product 1, and implements half of the core assets. The second increment happens right before the development of Product 4, and implements the rest of the core assets. For the first three products, the incremental approach appears to have shorter time-to-market, mainly because fewer core assets means less time is required for asset development. As request for Product 4 comes, with the incremental approach, the development of the new product can not be started until the rest of the core assets have been implemented. So, we see a big jump in time-to-market from Product 3 to 4. The incremental approach results in higher total effort (6173.36) than the big band approach (5338.47). That is the nature of its process flow.

4.4. Effect of evolution approach

The inputs to Scenarios 1 and 5 differ in product line evolution approach (infrastructure-based and branch-and-unite, respectively). Figure 4.3 shows the comparison of the results in time-to-market. As specified by the inputs, the evolution stage starts from Product 7. For Product 7, the branch-and-unite approach has smaller time-to-market because the product gets developed earlier and does not have to wait for core asset updates. For the later products, the branch-and-unite approach results in longer time-to-market because it requires extra effort to rework new products and imposes more task dependencies, thus reducing concurrency. The total effort of the branch-and-unite approach (5340.37) is only a slightly higher than the infrastructure-based approach (5338.47). That is because when Product 9 and 10 are released, some early products have already been phased out, so the costs for updating existing products are reduced.

4.5. Effect of adoption and evolution approaches

A situation an organization might face is the need to determine which software development and evolution approaches best fit its goals. Scenarios 1 and 4 – 7 show the alternatives the organization might have. Scenario 7 is the case where a traditional software development approach (single product only) is taken, where products are created and evolved independently.

Figure 4.4 shows the comparison of the results in time-to-market. As we can see, the big bang with infrastructure-based approach has the shortest average time-to-market, and the incremental with branch-and-unite approach has the longest average time-to-market. The traditional approach has the shortest time-to-market for the first two products, the longest time-to-market on

the third product, then its time-to-market stays between the incremental and the big bang approaches, afterwards its time-to-market starts climbing dramatically but still stays in between the branch-and-unite and infrastructure-based approaches. In our experiment, the reuse rates are not very high (30% for both black-box and white-box reuse) and the product is relatively small (100KSLOC), so the traditional product development time is only slightly longer (about 5 months) than product line engineering approaches. In the case of branch-and-unite evolution approach, the dependencies imposed by that approach overweighs the benefits of reusing the core assets. The total effort of Scenario 1 and 4-7 are 5338.47, 5340.37, 6173.36, 6194.03, and 8760.55, respectively. As we have expected, the traditional approach requires considerably more effort. By large-scale reuse, product line approaches generally result in smaller code size to development and maintain. Thus, the total effort on creating and evolving the products in a product line is smaller.

4.6. Validation of model and results

Several steps have been taken to verify and validate the model. First, the results of the simulator have been compared with the results of COCOMO II [9] to make sure the mathematic calculations are correct, and the results are the same (ignoring rounding errors). Second, the results of the simulator have been compared with the common knowledge about the product line, and we feel the results confirm to the common knowledge. Third, a initial small set of experts have reviewed the simulation results, and they feel that the results are consistent with what have been observed in the real world and the abstract model reflects the real process flow at a high level. In future investigations, we plan to continue soliciting expert feedback and compare simulation results with real product line data.

5. Related work

Cohen [7] presents an approach for making a software product line investment determination. The approach uses three factors to justify software product line investment: applications, benefits, and costs. Applications include the number of projected products in the product line, the time they will be developed, and their annual change traffic; benefits consists of the tangible and intangible goals the organization wishes to achieve through a product line approach; costs are the life cycle costs associated with core assets and individual products. Costs are affected by some factors, such as

costs of reuse, degree of reuse, and core assets change rate. Our cost estimation method is consistent with the Cohen approach but provides more capabilities.

Regnell et al. use a simulator to study a specific market-driven requirement management process [5]. The goal of simulator is to help in exploring bottleneck and overload situations in the requirement engineering process, investigating which resources are needed to handle a certain frequency of new requirements, and analyzing process improvement proposals. The specific process is modeled using queuing network and discrete event simulation [6]. Our simulator also uses discrete event simulation, but its purpose is to study life cycle issues for a product family instead of a portion of a software engineering process for a single product.

Riva and Delrosso recently discussed issues related to software product family evolution [11]. They state that a product family typically evolves from a copy-and-paste approach to a mature software platform. They point out some issues that harm the family evolution, such as organization bureaucracy, dependencies among tasks, slower process of change, and the new requirements that can break the architectural integrity. Their notion of product family used in that paper is different from the definition of a product line [8]. Creating a product family by copy-and-paste is not a product line approach, because the product line approach emphasizes a disciplined strategic reuse, not opportunistic reuse. A product line is actually a product family that has already evolved to a mature software platform. Our simulation results also show that in some cases dependencies imposed by product line approaches result in slower market response than the traditional software engineering approach.

6. Conclusions and future investigations

Software product line engineering promises of reduced cost while still supporting differentiation makes adoption and continued use of the associated approaches attractive. However, in order to make appropriate planning, decision tools are necessary. In this paper, we described a simulator that is intended to support early stage decision-making. The simulator provides both static and dynamic information for the selected software product line engineering process. The statistical result generated at the end of the simulation can be used for trade-off analysis. Stepping through the simulator helps analyzing product line processes, uncovering problems, and improving the understanding of software product line evolution.

Currently the simulation tool supports the study of independent product line initiation using big bang or

incremental product line adoption approaches and infrastructure-based or branch-and-unite product line evolution strategies. Our future investigations include providing estimates for other software product line initiation situations and approaches, allowing concurrency between inter-dependent tasks to some extent, providing probabilistic demand intervals, incorporating other cost models, and removing a number of the simplification assumptions. Furthermore, we plan to validate the model by comparing the results with real product line data and getting more expert feedback. Also, we want to combine the simulator with an optimization model, so users can specify their end-goal criteria and then allow the simulator to search for the best results.

7. References

- [1] B.P. Zeigler and H.S. Sarjoughian, "Introduction to DEVS Modeling & Simulation with JAVA(TM): Developing Component-based Simulation Models", 2003, <http://www.acims.arizona.edu/SOFTWARE/software.shtml>.
- [2] B. Boehm, A.W. Brown, R. Madachy, and Y. Yang, "A Software Product Line Life Cycle Cost Estimation Model", USC, June 2003
- [3] M. I. Kellner, R. J. Madachy, and D. M. Raffo, "Software Process Modeling and Simulation: Why, What, How", *The Journal of Systems and Software*, April 1999, pp. 91-105.
- [4] M. Paulk, et al., "Key Practices of the Capability Maturity Model", Version 1.1, Tech. Rept. CMU/SEI-93-TR-25, Software Engineering Institute, Feb 1993.
- [5] M. Höst, B. Regnell, et al, "Exploring Bottlenecks in Market-Driven Requirements Management Processes with Discrete Event Simulation", *The Journal of Systems and Software*, Dec 2001, pp. 323-332.
- [6] J. Banks, J.S. Carson, and B.L. Nelson, *Discrete-Event System Simulation*, 2nd Ed., Prentice Hall, Aug 2000.
- [7] S. Cohen, "Predicting When Product Line Investment Pays", Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications, Toronto Canada, 2001, pp. 15--18.
- [8] P. Clements and L.M. Northrop, *Software Product Lines -- Practices and Patterns*, Addison-Wesley, Aug 2001
- [9] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0," *Annals of Software Engineering Special Volume on Software Process and Product Measurement*, Science Publishers, Amsterdam, The Netherlands, 1995, pp. 45 - 60.
- [10] K. Schmidt and M. Verlage, "The Economic Impact of Product Line Adoption and Evolution", *IEEE Software*, Jul/Aug 2002, pp. 50-57.
- [11] C. Riva and C.D. Rosso, "Experiences with Software Product Family Evolution", Proceedings of International Workshop on Principles of Software Evolution, Helsinki Finland, Sep 2003, pp. 161-169.