

Rapid System Development via Product Line Architecture Implementation

Mauro Caporuscio, Henry Muccini, Patrizio Pelliccione, Ezio Di Nisio

Dipartimento di Informatica
Università dell'Aquila
I-67010 L'Aquila, Italy
{caporusc,muccini,pellicci,dinisio}@di.univaq.it

Abstract. Software Product Line (SPL) engineering allows designers to reason about an entire family of software applications, instead of a single product, with a strategic importance for the rapid development of new applications. While much effort has been spent so far in understanding and modeling SPLs and their architectures, very little attention has been given on how to systematically enforce SPL architectural decisions into the implementation step.

In this paper we propose a methodological approach and an implementation framework, based on a plugin component-based development, which allows us to move from an architectural specification of the SPL to its implementation in a systematic way. We show the suitability of this framework through its application to the TOOLone case study SPL.

1 Introduction

The Software Product Line (SPL) technology has been introduced for the rapid development of new applications, by reusing as much as possible assets (i.e., components, known requirements or design elements, models, artifacts) identified in relation with specific application domains. The characteristic that distinguishes SPLs from previous efforts is *predictive* versus *opportunistic* software reuse [16]: instead of creating libraries of general purpose components, hoping they will be used in future development, SPLs make a predictive analysis of such artifacts which may be reused and integrated for the development of domain-specific products. Many approaches have been undertaken to deal with SPL during the entire life-cycle. At the requirement level, known notations have been extended to deal with SPL specific concepts (e.g., [13, 7]). At the software architecture level, formalisms have been introduced to model software architecture of product lines (e.g., [9, 11, 25]). At the design level, various approaches have been proposed so far (e.g., [13]). At the implementation level, some approaches and tools have been presented for SPL coding (e.g., [5, 3]).

While systematic tools and approaches have been proposed to specify SPL requirements, architectures and design artifacts, little attention has been paid on how to systematically integrate those steps. In particular, very few attention has been paid on *how to systematically implement the architecture of an SPL*.

Thus, constraints and decisions taken during architectural analysis cannot be easily propagated down to the product implementation.

Goal of this paper is to propose a *methodological support* and an *implementation framework* for establishing an effective and rapid integration and transition from SPL architectural specifications (usually referred as Product Line Architecture – PLA) to the implementation of the selected products in the SPL. The methodological support takes into consideration six different steps (shown in Figure 1), where (a) the PLA is specified in term of mandatory, optional and variant components, (b) a decision process is applied (c) to derive the specification of a certain product architecture (PA), (d) a constraint model is derived from the PLA and PA specifications, (e) the framework is configured taking into account architectural decisions and constraints, and (f) the product implementation is generated. The implementation framework (called N-PLA) fulfills the role of a plugin engine (managing plugins, lookup, communication) and provides capabilities for enforcing PLA level decisions and constraints. The product architecture is implemented using plug-in technology.

The novelty of our approach resides in the following aspects:

- the system development can automatically move from the specification stage to the implementation one;
- PLA-level decisions and choices are automatically enforced into the selected implementation. Structural, communication and dependency constraints identified at the PLA and PA level are identified and handled by the framework. Components are automatically assembled in order to conform to the architectural specification;
- SPL *evolution* is handled by the implementation framework itself. Whenever a component is added, removed or modified in the architectural specification, the framework catches the modification and handles it;
- *reusability*, well addressed at the architectural level by the PLA is reflected and stressed also at the implementation level.

The following of this paper is organized such as Section 2 introduces the concept of PLA, presents our methodology and the N-PLA implementation framework, Section 3 describes the TOOL•one SPL case study and how it is being implemented within the framework. Section 4 introduces related work while Section 5 concludes the paper.

2 From Product Line Architecture Modeling to Implementation

A Software Architecture provides high-level abstractions for representing the structure, behavior, and key properties of complex software systems [12]. Software Architecture models the system in terms of components and connectors, where components represent abstract computational subsystems, and connectors formalize the interactions among components.

When dealing with the software architecture of a product family, the concept of Product Line Architecture (PLA) [8, 15, 11] has been introduced as an emerging evolution to SA modeling for product family. Whereas an SA defines the structure and behavior of a single product, a PLA precisely captures, in a single specification, the overall architecture of a suite of closely-related products [8]. A PLA specification focuses on modeling *mandatory elements* (which are present in the architecture of each product), *optional elements* (which may or may not be present in a product), *variant elements* (which can be mandatory or optional, and can be chosen to be one of a number of different alternatives) [14] (see Figure 1, top-left portion), and explicitly models connection and communication constraints.

Once a number of variation points have been introduced in a PLA, it becomes necessary to be able to resolve variability in order to select one product architecture out of the overall PLA. This process is usually implemented using a *Decision Process* (e.g., [24, 5]) which allows the selection of optional and variant components (Figure 1, bottom-left portion) in order to choose a desired product architecture (PA). While notations and tools have been introduced in order to provide mechanisms to capture an entire family of architectures [24, 25], very few attention has been spent on investigating product line architecture implementation, that is, how architectural decisions and constraints may drive the implementation of a certain product.

In this section we describe our methodological approach and the associated implementation framework for PLA implementation.

2.1 Product Line Implementation: our methodology

Establishing an effective connection between SPL architectural models and their implementation is a relevant problem, as also remarked in [15], ch. 1, “one of the primary tasks of software architects is to establish and communicate to the rest of the team the important concepts necessary for effective software design and implementation”.

In particular, what we believe is really challenging is:

- **enforcing the implementation adherence to PLA-level constraints:** when modeling an SPL and its PLA, many constraints are explicitly or implicitly identified: *structural constraints* are those related to the PLA model, in the form of mandatory, optional and variant elements; *communication constraints* dictate which components may communicate and how; *dependency constraints* establish how the selection of a component may require or exclude the selection of other components.
- **allowing software evolution, both at the PLA and product architecture level:** a relevant problem associated with an architecture is *managing evolution* [6, 11]: the *SPL may evolve*, requiring new versions of existing components or new components; the single *products* (members of the product line) may evolve. New features may be added, letting the product spinning off from the initial SPL; SPL evolution may *affect already deployed products*.

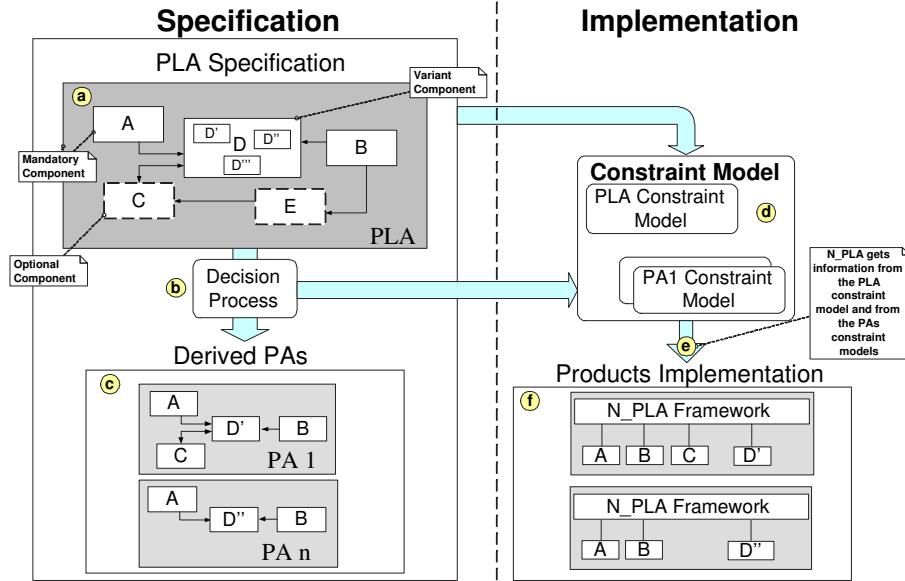


Fig. 1. Metodology Idea

The methodology we propose helps in creating a more deeply connection between the architectural specification and its implementation through six main steps (see Figure 1):

- Step a):** the PLA is specified in terms of mandatory, optional and variant components by using current modeling techniques and tools (PLA modeling tools are outlined in Section 4).
- Steps b and c):** a decision process is applied to the PLA specification for deriving a product architecture (PA) and the related constraint models. Clearly, it is impossible to fully automatize these steps: the product selection needs to be made by the software developer, however, PLA modeling tools allow for automatic PA derivation. The framework automatically generates the PA Constraint Model that is an XML file containing all the choices made by the user. For each selected PA a PA Constraint Model is generated.
- Step d):** structural, communication and dependency constraints expressed in the architectural model are elicited from the PLA specification and automatically embedded in a PLA Constraint Model. This step is fully automated. The PLA Constraint Model is an XML file containing a representation of the PLA in terms of components (mandatory, optional and variant) and relations between them. In other words the PLA Constraint Model is a textual representation (XML format) of the PLA understandable by the N-PLA framework. While the PA Constraint Model is used to startup and instantiate the PA, the PLA Constraint Model is used to guide PA evolution. In fact, even if a PA can evolve, it must always comply to the PLA constraints.

Step e): for each PA, the N-PLA framework selects and loads the desired components from a component repository (i.e., the “plugin” directory). Components are Java bytecode packaged in .jar files and loaded through the **Plugin Manager** component (described in Section 2.2). N-PLA provides instructions on how components have to interact to match with the related Constraint Model. This step is fully automated.

Step f): N-PLA produces the product implementation. In particular, the selected product is implemented through a plugin technology, where mandatory and optional components are implemented as plugin components and, constraints are managed directly by the N-PLA framework.

The use of N-PLA as development environment, allows runtime selection and deselection of components, thus handling runtime modifications. Moreover, whenever a PA or PLA evolves, the framework adapts the implementation to conform to the PA or the PLA specification. In case of PLA evolution, the PLA Constraint Model is updated and each PA generated and instantiated is checked to be conform with the new PLA. In case of PA evolution, if the new PA conforms to the PLA, then the related PA Constraint Model is updated and the implementation modified. If a component is added or removed, a plugin needs to be added or removed. If the way components communicate changes, the communication information in N-PLA needs to be modified. This step is automatized.

In the next Section we show how the N-PLA framework works.

2.2 Product Line Implementation: the N-PLA Framework

The N-PLA framework [2] here proposed supports the designers with automatic transformations from the design to the implementation, automatically enforcing PLA-level decisions and choices into the selected implementation. Structural, communication and dependency constraints identified at the PLA level are mechanically caught and handled by the framework. Components are automatically assembled in order to conform to the architectural specification. N-PLA handles also SPL evolution: whenever a component is added or removed or changed in the architectural specification, the framework catches the modification and handles it.

Many different implementation frameworks have been developed so far for component based programming [23] such as JavaBeans [22] and COM [18]. Although these systems are widely accepted as standard reference models, they are not well suited for our purposes, since they do not allow for components addition and removal at runtime. More precisely, bindings between components are pre-defined and fixed, making architectural mutations impossible. On the contrary, what we need is a framework able to decouple components by achieving both modification of bindings, and components addition and removal.

The N-PLA framework overcomes this problem through a plugin-based infrastructure where components are N-PLA plugins, written in Java and loaded when needed by N-PLA. The framework itself is implemented in Java. Its plugin

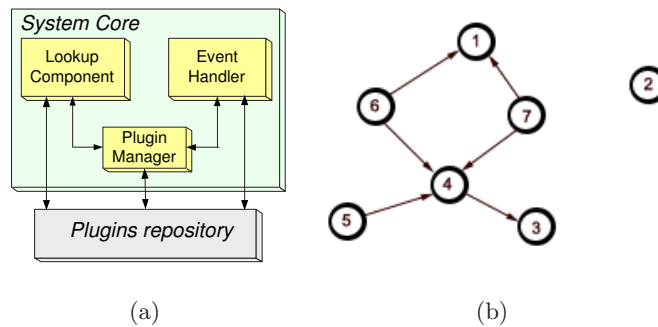


Fig. 2. a) N-PLA Software Architecture b) Plugin loading dependencies

framework architecture is shown in Figure 2.a and is composed by two components:

- a *System Core*, which provides the core elements of the N-PLA plugin framework. While the *Plugin Manager* handles the loading and unloading of plugins, the *Event-based communication Handler* and the *Lookup-Component* manage the communication between components. From one side in the event-based system the components binding policy completely resides into the event handler which is responsible for switching events from a component to the others. Furthermore, architectural link modification will simply reflect in modifying the dispatching policy. Thus, while the event-dispatcher allows for a decoupled communication between components, it has the drawback in the added overhead and in the noise for results retrieving. In the case of synchronous call invocations, it is advised to avoid this communication handler. On the other side the *Lookup-Component*, that is able to retrieve the reference of a component through its ID, requires that the plugin that is interested to communicate with another plugin must necessarily know details on the interface of the target plugin. Moreover, the *Lookup-Component*, extracting information by the PLA XML¹, assures that the communication policies stated into the PLA design are respected.
- a *Plugin Repository*, which contains mandatory, optional and variant components;

While the plugin repository is just a site where the plugin are collected, the system core requires an extensive analysis in terms of its components *Plugin Manager*, *Event-Handler*, and *Lookup-Component*.

Plugin Manager: the plugin manager component is responsible for loading and unloading components. This component receives the PLA XML specifica-

¹ We assume PLA and PA are both modeled in XML (as, for example, in [24]).

tion and extracts all the structural information that are used to define the policy regulating the plugins loading that each implemented PA must respect. In particular:

- each mandatory component must be contained into all implementations;
- optional components can be loaded or not;
- for a variant component the plugin manager assures that at least one of its alternatives is loaded.

For example, referring to Figure 1, the plugin manager, that contains the XML blueprint of the PLA, assures that an incorrect PA composed of only the plugins A and B is not loadable. The Plugin Manager assures also that dependencies between loaded components are respected. Assuming the existence of an acyclic dependency graph, where nodes are components in the PLA, and the direct arcs are the dependencies among components, the algorithm, at each step, loads the plugins (associated to the nodes) without dependencies (exiting arcs). After the deletion of each node representing plugins loaded, and its entering arcs, the process is iterated. In this manner all the plugins are loaded with respect to the stated dependencies. An example is reported in Figure 2.b, where the numbers inside the nodes represent one possible loading order. Whenever the PA or PLA specification have to change, the plugin manager dynamically revises the implementation in order to make it conform to the evolved architecture. If the component that must be removed, substituted or that is involved in the communication that is changing, is in a not stable state (i.e. it is computing or it is waiting for a result), the plugin manager temporary suspends the distribution of calls coming from stable components, waiting for a system stable state.

Event-Handler: the event handler component handles the event-based communications among plugins. Each plugin has an XML file specification that contains:

- plugin name,
- the plugin main class and others required information,
- binding information between received events and their internal methods.

The event is an object that contains the event ID and the list of parameters that should be passed to the bound method. The event handler is implemented as a queue able to maintain the publication order. For example, Figure 3.a shows the binding between an event $\langle E \rangle$ and plugin methods for the component Sub_1 .

The event handler analyzes the XML file for each plugin and builds a table (see the Figure 3.b) containing the event name, the event publisher and the list of components subscribed for the event. Therefore, referring to Figure 3.a and Figure 3.b, when Pub_i submits the event E , the event handler forwards the event to Sub_1, \dots, Sub_n in terms of methods invocation; in the case of Sub_1 the method **meth** of the class *Class* is invoked with parameters “par₁”, \dots , “par_m”. Whenever the PA or PLA specification change, the event handler updates the dispatching policy table, in order to make the implementation conform to the

architectural specification.

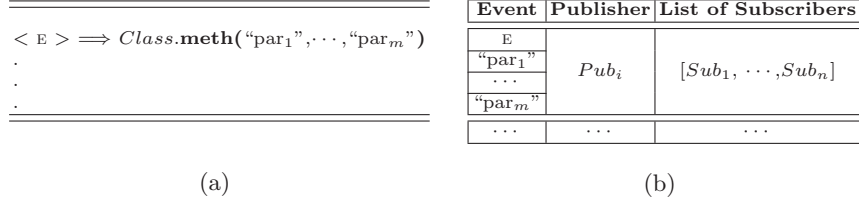


Fig. 3. a) Mapping between events and plugin services b) Event-Handler dispatching policy

Client Component	Server Component
$Comp_1$	$Comp_2$
	$Comp_3$
	...
$Comp_3$	$Comp_1$
$Comp_i$...

Fig. 4. Event Dispatching Policy

Lookup-Component: this component allows for a direct communication between a pair of components. In fact, it is able to discover a component from its ID and to return its interface containing all information needed to profit of the offered services. To do that each plugin specializes the abstract class *PlugData*, contained into the core component, adding the methods representing the exported services. The abstract class *PlugData* contains the method *lookup* that allows for plugins discovery. Thus, each plugin, using its class *PlugData*, that inherits the method *lookup*, can obtain the interface of each other plugin from the plugin ID. In this case the communication constraints can be assured. In fact the Lookup-Component can determine (see Figure 4) with which components a specified component can communicate and in case the communication is not allowed at the PLA level, the Lookup-Component returns *null*.

Currently, the N-PLA framework has been partially implemented and in the prototypal version available so far the *Plugin Manager* and the *Lookup-Component* do not allow for runtime changes (i.e., they work at compile time), and the *Event-Handler* does not handle binding between publisher and subscriber.

3 Product Line Architecture for Software Architecture Analysis: A Case Study

This section describes the application of the proposed methodology for implementing a PLA for Software Architecture (SA) analysis.

The analysis of functional and non-functional properties is a non-trivial task to perform along with the whole software life-cycle. Usually such activity requires the software engineer to have deep knowledge of formal languages and methods. However, since this is in contrast to the necessity to have short time-to-market, such activities are often skipped out. Hence, the use of support tools will be easy and speed up the overall process. In [10], Cortellessa et al. faced such a problem by proposing TOOLone, a framework for integration of functional and non-functional analysis.

Step a): Exploiting the Software Architecture of TOOLone we derived a generic PLA for architectural analysis support tools (shown in Figure 5) that presents the following set of mandatory and optional components:

Mandatory components:

- the *IDE* that implements the main window of the tool and is responsible to manage and coordinate the User Interface and the underlying analysis tools.
- *User Interface* which is a *variant* component for editing formal specification by means of diagrammatic notation. Referring to Figure 5 it is worth noticing that the UML component is actually *variant*. In fact, since UML defines different kind of diagrams, it would be convenient to have the possibility of selecting one or more of them.
- the *XML Model* which manages the shared data structures. The use of XML representation of data allows for easy integration and management of a wide-range of different analysis.
- *Analysis* is a *variant* component responsible for performing the analysis of interest. It contains a set of *variant* components, representing the different kind of analysis, which in turn are composed of subset of *variant* components. In fact, for each kind of analysis there exist a number of various approaches implemented by different components.

Optional Components:

- the *Input Filter* is a *variant* component eventually needed to translate the visual editor output into the XML representation used by the *XML Model* component. In fact there may exist visual editors that already export their output as XML.
- *Analysis Filter*: is a *variant* component amenable to eventually translate or manipulate the XML model depending to the analysis to be carried on.
- *Tool* represents the set of additional components eventually needed to perform the desired analysis.

Step b) and c): From the PLA presented in *Step a*, we have instantiated the PA of CHARMY [1], a tool for supporting the design of Software Architectures and its validation against functional requirements (shown in Figure 6)².

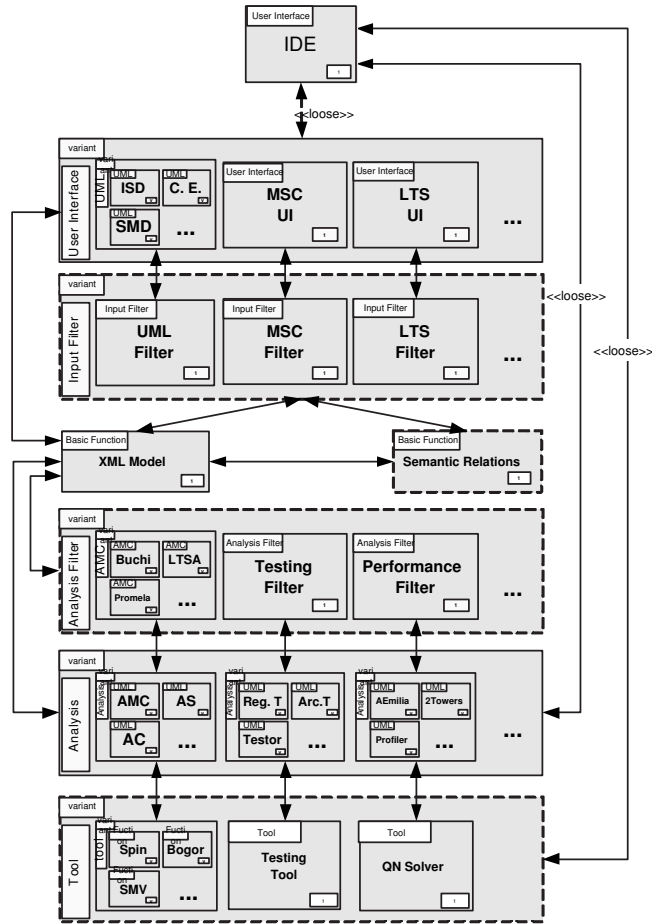


Fig. 5. The PLA for SA Analysis

² Actually, while CHARMY v1.x does not comply to this SA, CHARMY v2.0 is still under construction and it is fully compliant with this work.

In particular CHARMY is implemented by the following components:

Component Repository³:

- the *IDE* which implements the main window of the tool and is responsible to manage and coordinate the editors and the underlining analysis;
- the *CE UI* component which allows editing Software Architecture by means of components and connectors;

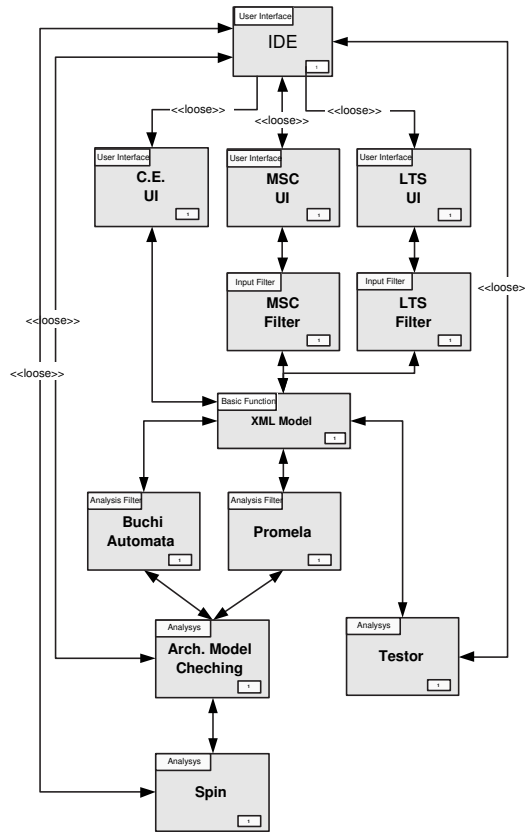


Fig. 6. The PA of CHARMY

- the *LTS UI* editor which permits to specify the component behavior by means of state machines;

³ Even though the Repository should contain all the components described by the PLA, the current prototype implements only those components needed to build the CHARMY PA.

- the *MSC UI* editor which allows expressing interactions among components by means of message exchange;
- while *CE UI* does not need the related Input-Filter (it already exports its output in XML), the *LTS UI* and *MSC UI* come together with their input-filters components which are responsible to translate such diagrams into the XML representation that will be passed to the *XML Model* component;
- on the analysis side, the repository contains the *Architectural Model Checking* component for consistency analysis and the *TeStor* component for model-based testing;
- since the *Architectural Model Checking* requires Büchi automata and Promela code as input, the respective analysis-filters are needed in order to extract and translate data of interest from the XML Model representation.

Since *Architectural Model Checking* does not implement a model checker it self, the *Spin* [20] component is required for performing the overall analysis.

Finally, the XML file that represents the Constraint Model of CHARMY is constructed.

Step d): The PLA Constraint Model of TOOLone is generated. A sketch of the XML file is here reported:

XML of the TOOLone PLA

```

<PLA_Constraint_Model name="Toolone">
  <component type="mandatory" name="IDE" ID= "1">
    <connectionsList>
      <con type="loose">2</con>
      <con type="loose">7</con>
    </connectionsList>
  </component>
  <varcomponent type="mandatory" name="UserInterface" ID= "2">
    <connectionsList>
      <con type="async">1</con>
      <con type="async">4</con>
    </connectionsList>
    <componentAlternatives>
      <varcomponent type="mandatory" name="UML" ID= "2.1">
        <connectionsList>
          <con type="async">3.1</con>
        </connectionsList>
        <componentListAlternatives>
          <component type="mandatory" name="ISD" ID= "2.1.1"/>
          <component type="mandatory" name="C.E." ID= "2.1.2"/>
          <component type="mandatory" name="SMD" ID= "2.1.3"/>
        </componentListAlternatives>
        ...
      </varcomponent>
    </componentAlternatives>
  </varcomponent>
  ...
</PLA_Constraint_Model>

```

Step e): Given such a component repository and the *Constraint Model* derived from the PLA and the PA XML description (Steps c) and d) of Figure 1) we proceed on running N-PLA.

Supposing that the *Decision Process* is correct, N-PLA parses the given PA deriving the set of plugins that composes the initial application configuration

and extracting the constraints and relations between the components. In particular, when the *Plugin Manager* (refer to Figure 2) is launched, it processes the information presented above in order to load right plugins and properly initialize both the *Lookup-Component* and the *Event-Handler*.

Client Comp.	Server Comp.
XML Model	CE UI
	...
	Büchi Automata
	...
Testor	XML Model

Event	Publisher	List of Subscribers
AMC_START	IDE	[Arc. MC]
"depth"		
RESULT	Arc. MC	[IDE]
DRAW	IDE	[CE, MSC, LTS]
"object"		

(a) Lookup Access Policy
(b) Event Dispatching Policy

Fig. 7. Dispatching Policy

In particular:

- The *Plugin Manager* retrieves from the PA the information about tight relations (links without stereotypes in Figure 5 and Figure 6) and build the an access policy table of the *Lookup-Component* (see Figure 7.a). When a plugin p asks for the reference of a plugin q to the *Lookup-Component*, the latter checks if p appears into the table. If there exists a row in which p and q are correlated, then the reference of q will be returned to p .
- The *Plugin Manager* also retrieves information about loosely relations (stereotyped links in Figure 5 and Figure 6) and build the table (shown in Figure 7.b) in which the *Event-Handler* keeps track of the architectural dispatching policy defined by the PA. When a publisher p sends an event e to the *Event-Handler*, the latter checks if such event appears into the table. If there exists a row in which e and p are correlated, then e will be dispatched to all plugins in subscriber list. In particular, Event-dispatching is implemented as method callback respecting the bindings defined into the XML plugin description (an example is given in Figure 8).

Step f): Finally, once all connections among components have been established, the CHARMY product is finished and can be used for performing analysis of interest.

$\langle \text{AMC_START} \rangle \implies \text{AMCClass.start}(\text{depth})$
$\langle \text{AMC_MAXMEM} \rangle \implies \text{AMCClass.setMaxMem}(\text{value})$
...

Fig. 8. Mapping between events and plugin services

4 Related Work

In this section we briefly report on Product Line Architecture Modeling tools and Product Line Implementation processes and tools.

4.1 Product Line Architecture Modeling

Ménage [24] and Koala [25] represent the state of the art in PLA specification. (ACME Studio and UML have been used for product line modeling too).

Ménage [24] is an environment for managing evolving architectures in SPL. It provides a definition of how to represent PLAs (addressing variabilities and also evolution) and a graphical environment which facilitates the specification, in the defined representation, of PLAs. Ménage uses xADL 2.0 [4] XML schemas as the representation format.

Koala, instead, is an ADL specially designed for modeling embedded software for consumer electronics. It inherits from the Darwin architecture description language the main concepts and ideals, even though it is more oriented to notations and concepts commonly used in consumer electronics products. Koala allows specifying hierarchical architectures, it makes a distinction between component types and instances, it permits to construct configurations by instantiating components and connectors and explicitly models optional interfaces.

4.2 Product Line Implementation

KobrA (Component-based Application Development) [5] is a software process- and product- centric engineering approach which uses a component-based approach in each phase of the development process. The KobrA implementation process provides guidelines on how UML artifacts used for specifying the SPL, may be used during the implementation phase. However, and differently from N-PLA, it does not provide implementation support.

Another interesting work has been proposed in [3] where the Pure::Variant approach and tool support for SPL is provided. The tool seems to cover all steps of SPL development, from requirements to product generation. The specification-to-implementation features will be further investigated when the product will be released.

Other tools proposed by the SPL community include Holmes [21], which supports SPL development and whose architecture is based on JavaSpaces, PuLSE-BEAT [19] which supports all steps of the scoping process, and [17] which supports top-down as well as bottom-up traceability in product families. None of such approaches and tools seem to support implementation of SPLs. For sake of completeness, somehow related to our topic are also Koala [25], xADL [4] (with its C2 framework) and DiscoTect [26] (for architecture recovery).

5 Conclusions and Future Work

We presented a methodological approach and the N-PLA implementation framework for implementing product architectures, by enforcing constraints and allowing for reconfiguration. The TOOLone PLA has been presented, together with one PA and some information on how it has been implemented in N-PLA.

With respect to a non-framework implementation, N-PLA imposes to use certain interfaces and packaging rules (as in any framework) but this cost allows for an automated loading of components, communication and constraints automatic handling. It is however necessary to impose a rigid structure in order to provide the N-PLA features.

With respect to other plugin-based frameworks (e.g., Eclipse and MagicBeans) we are currently evaluating N-PLA main advantages and limitations. The added value of N-PLA is on the ability to load the XML representations for the PLA and the PA and to manage consistently the loading of the plugins and the communication between them. Thus, if we think to use another plugin based tool (for example Eclipse), an extension is required in order to provide the potentiality of N-PLA.

Many enhancement are in mind for future work. A validation engine needs to be created for accomplishing two different tasks:

- validating the PLA specification conformance to feature and requirements. In fact, being assured the PA implementation conforms to the PA specification is not enough, if the PLA model itself has not been formally validated a-priori.
- Assuming a PA may have its own properties and constraints, a validation engine needs to be proposed to monitor the implementation execution conformance to the PA specification.

References

1. CHARMY Project. Charmy Web Site. <http://www.di.univaq.it/charmly>, 2004.
2. N-PLA Project. N-PLA Web Page. <http://www.di.univaq.it/di/project.php?id=16>, 2005.
3. Pure::variants. <http://www.pure-systems.com>, 2005.
4. xADL 2.0 Architecture Description Language. <http://www.isr.uci.edu/project-s/xarchuci/>, 2005.
5. C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-Based Product-Line Engineering with UML*. Addison-Wesley, 2001.
6. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, second edition*. SEI Series in Software Engineering. Addison-Wesley Professional, 2003.
7. A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Use Case Description of Requirements for Product Lines. In *International Workshop on Requirements Engineering for Product Lines - REPL02*, Essen, Germany, September 2002.
8. J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ACM Press/Addison-Wesley Publishing Co., 2000.

9. P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Pub Co, 1st edition, August 2001.
10. V. Cortellessa, A. Di Marco, P. Inverardi, F. Mancinelli, and P. Pelliccione. A framework for the integration of functional and non-functional analysis of software architectures. *Electr. Notes Theor. Comput. Sci., Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS), Barcelona*, (116: 31-44 (2005)), march 2005.
11. A. Garg, M. Critchlow, P. Chen, C. van der Westhuizen, and A. van der Hoek. An Environment for Managing Evolving Product Line Architectures. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 358. IEEE Computer Society, 2003.
12. D. Garlan. Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events. In *Formal Methods for Software Architectures*, pages 1–24. Lecture Note in Computer Science, 2804, 2003.
13. H. Gomaa. *Designing Software Product Line with UML - from use cases to pattern-based software architectures*. Object Technology Series. July 2004.
14. M. Jaring and J. Bosch. Representing Variability in Software Product Lines: A Case Study. In *SPLC*, pages 15–36, 2002.
15. M. Jazayeri, A. Ran, and F. van der Linden. *Software architecture for product families: principles and practice*. Addison-Wesley Publishing Co., 2000.
16. C. W. Krueger. Introduction to Software Product Lines. <http://www.software-productlines.com/introduction/introduction.html>, 2004.
17. P. Lago, E. Niemelä, and H. V. Vliet. Tool support for traceable product evolution. In *CSMR*, pages 261–269, 2004.
18. D. S. Platt. *Understanding COM+*. Microsoft Press, 1999.
19. K. Schmid and M. Schank. PuLSE-BEAT – A Decision Support Tool for Scoping Product Lines. In *IW-SAPF-3: Proceedings of the International Workshop on Software Architectures for Product Families*, pages 65–75. Springer-Verlag, 2000.
20. SPIN Home page: <http://cm.bell-labs.com/cm/cs/what/spin/index.html>.
21. G. Succi, J. Yip, and W. Pedrycz. Holmes: an intelligent system to support software product line development. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 829–830. IEEE Computer Society, 2001.
22. Sun Microsystems, Inc. JavaBeans. <http://java.sun.com/products/javabeans>.
23. C. Szyperski. *Component Software. Beyond Object Oriented Programming*. Addison Wesley, 1998.
24. The Ménage Project. *University of California, Irvine*. <http://www.isr.uci.edu/projects/menage>.
25. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.
26. H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. Discotect: A system for discovering architectures from running systems. In *IEEE Proc. of the 26th Int. Conf. on Software Engineering*, pages 470–479, 2004.