Software of the Future Is the Future of Software?

Paola Inverardi

Dipartimento di Informatica,
Università dell'Aquila
inverard@univaq.it

Abstract. Software in the near ubiquitous future (Softure) will need to cope with variability, as software systems get deployed on an increasingly large diversity of computing platforms and operates in different execution environments. Heterogeneity of the underlying communication and computing infrastructure, mobility inducing changes to the execution environments and therefore changes to the availability of resources and continuously evolving requirements require software systems to be adaptable according to the context changes. Softure should also be reliable and meet the user's performance requirements and needs. Moreover, due to its pervasiveness, Softure must be dependable, which is made more complex given the highly dynamic nature of service provision. Supporting the development and execution of Softure systems raises numerous challenges that involve languages, methods and tools for the systems thorough design and validation in order to ensure dependability of the self-adaptive systems that are targeted. However these challenges, taken in isolation are not new in the software domain. In this paper I will discuss some of these challenges, what is new and possible solutions making reference to the approach undertaken in the IST PLASTIC project for a specific instance of Softure focused on software for Beyond 3G (B3G) networks.

1 Introduction

Software in the near ubiquitous future (Softure) will need to cope with variability, as software systems get deployed on an increasingly large diversity of computing platforms and operates in different execution environments. Heterogeneity of the underlying communication and computing infrastructure, mobility inducing changes to the execution environments and therefore changes to the availability of resources and continuously evolving requirements require software systems to be adaptable according to the context changes. At the same time, Softure should be reliable and meet the user's performance requirements and needs. Moreover, due to its pervasiveness, Softure must be dependable, which is made more complex given the highly dynamic nature of service provision.

Supporting the development and execution of Softure systems raises numerous challenges that involve languages, methods and tools for the systems thorough design and validation in order to ensure dependability of the self-adaptive systems that are targeted.

U. Montanari, D. Sannella, and R. Bruni (Eds.): TGC 2006, LNCS 4661, pp. 69–85, 2007.
© Springer-Verlag Berlin Heidelberg 2007

However these challenges, taken in isolation are not new in the software domain. Adaptable and re-configurable systems do exist in many software application domains from tele-communication to the software domain itself, e.g. operating systems. Dependable systems have been intensively investigated and methods and tools exist to develop them. Hence what are the new challenges for Softure? In this paper I will discuss some of these challenges and possible solutions making reference to the approach undertaken in the IST PLASTIC [1] project for the specific instance of Softure as software for Beyond 3G (B3G) networks. I will try to highlight what I consider innovative and futurist for software and what I simply consider software for the future. The ultimate thesis of this paper is that Softure requires to rethink the whole software engineering process and in particular it needs to reconcile the static view with the dynamic view.

The paper is structured as follows. In the following section I discuss the Softure characteristics in order to identify the two key challenges: *adaptability* and *dependability*. Section 3 discusses and compares different notions of adaptability with different degrees of dependability. This discussion will bring me to consider the Softure issues in a software process perspective. Section 4 proposes a new software process and discusses it in the scope of the PLASTIC project [1].

2 Softure Challenges: Setting the Context

Softure is supposed to execute in an ubiquitous, heterogeneous infrastructure under mobility constraints. This means that the software must be able to carry on operations while changing different execution environments or *contexts*. Execution contexts offer a variability of resources that can affect the software operation. *Context awareness* refers to the ability of an application to *sense* the context in which it is executing and therefore it is the base to consider (Self-) adaptive applications, i.e. software systems that have the ability to change their *behavior* in response of external changes.

It is worthwhile stressing that although a change of context is measured in terms of availability of resources, that is in quantitative terms, an application can only be adapted by changing its behavior, i.e. its functional/qualitative specification. In particular, (Physical) Mobility allows a user to move out of his proper context, traveling across different contexts. To our purposes the difference among contexts is determined in terms of available resources like connectivity, energy, software, etc. However other dimensions of contexts can exist relevant to the user, system and physical domains, which are the main context domains identified in the literature [2]. In the software development practice when building a system the context is determined and it is part of the (non-functional) requirements (operational, social, organizational constraints). If context changes, requirements change therefore the system needs to change. Context changes occur due to physical mobility, thus while the system is in operation. This means that if the system needs to change this should happen dynamically. This notion leads to consider different ways to modify a system at run time that can happen

in different forms namely *(Self-)adaptiveness/dynamicity/evolution* and at different levels of granularity, from software architecture to line of code.

Softure needs also to be dependable. *Dependability* is an orthogonal issue that depends on Quality of Service (QoS) attributes, like performance and all other—bilities. Dependability impacts all the software life cycle.

In general dependability is an attribute for software systems that operate in specific application domains. For Softure I consider dependability in its original meaning as defined in [3], that is *the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers ... Dependability includes such attributes as reliability, availability, safety, security.* Softure encompasses any kind of software system that can operate in the future ubiquitous infrastructure. The dependability requirement is therefore extended also to applications that traditionally have not this requirement. Dependability in this case represents the user requirement that states that the application must operate in the unknown world (i.e. out of a confined execution environment) with the same level of reliance it has when operating at home. At home means in the controlled execution environment where there is complete knowledge of the system behavior and the context is fixed. In the unknown world, the knowledge of the system is undermined by the absence of knowledge on contexts, thus the dependability requirement arises also for conventional applications. Traditionally dependability is achieved with a comprehensive approach all along the software life cycle from requirements to operation to maintenance by analyzing models, testing code, monitor and repair execution.

Therefore the overall challenge is to provide dependable assurance for highly adaptable applications. Since dependability is achieved throughout the life cycle many software artifacts are involved, from requirements specification to code. In the rest of this paper I will consider as such artifacts only *models* that is idealized view of the system suitable for reasoning, developing, validating a real system. Models can be functional and non-functional and can represent different level of abstractions of the real system, from requirements to code. My research bias is on Software Architecture, therefore I will often consider software architectural systems' models. An architectural model allows the description of the static and dynamic components of the system and explains how they interact. Software architectures support early analysis, verification and validation of software systems. Software architectures are the earliest comprehensive system model along the software lifecycle built from requirements specification. They are increasingly part of standardized software development processes because they represent a system abstraction in which design choices relevant to the correctness of the final system are taken. This is particularly evident for dependability requirements like security and reliability and quantitative ones like performance.

3 Adaptability: 3 Examples from My Own Bag

In this section I discuss the notion of adaptability. According to what presented so far, adaptability is the ability to change a system according to context

variations, e.g. driven by QoS requirements. However, the change should maintain the *essence* of the system that from now on I will call *invariant*.

In Sections 3.2, 3.3, and 3.4, I will focus on evolving systems that change through adaptation. In order to classify them I propose to use a 4 dimension metric: the four *Ws*.

3.1 The Four Ws

The systems I consider can change through adaptability either their *structure* and/or their *behavior*. The four Ws characterize the nature of the change along the following four dimensions:

- *Why* there is the need to change?
- *What* does (not) change?
- *When* does the change happen?
- *What/Who* manages the change?

Why: This dimension makes explicit the need for the change. In a Software Engineering perspective this change is always done to meet requirements. It can be because the requirements evolved or it can be that the system does not behave properly according to the stated requirements. It is also worthwhile mentioning that requirements can be functional and non functional requirements. The former class captures the qualitative behavior of a software system, its functional specification. The latter defines the systems's quantitative attributes like, performance, reliability, security, etc. In the following I will provide 3 examples of functional and non functional adaptation that have been developed in the Software Engineering research group at University of L'Aquila.

What: Here we discuss the part of the system that is affected by the change. Referring to architectural models, changes can affect the structure and/or the behavior. For the structure, components can get in and out, new connectors can be added and removed. For the behavior components can change their functionality and connectors can change their interaction protocols.

When: This dimension captures the moment during the systems lifetime in which the change occurs. It does not mean that the change happens necessarily at run time. This dimension is related with the Static versus Dynamic issue.

What/Who: This is the description of the mechanisms to achieve the change. It can be a configuration manager or it can be the system itself. Involves monitoring the system to collect relevant data, evaluating this data, make a decision about the change alternatives and then perform the actual change.

3.2 Synthesis

Synthesis is a technique equipped with a tool that permits to assemble a component based application in a deadlock free way [35,21,34]. Starting from a set of Commercial Off The Shelf (COTS) components, Synthesis assembles them together according to a so called connector-based architecture by synthesizing

a connector that guarantees deadlock-free interactions among components. The code that implements the new component representing the connector is derived, in an automatic way, directly from the COTS (black-box) components interfaces. Synthesis assumes a partial knowledge of the components' interaction behavior described as finite state automata plus the knowledge of a specification of the system to assemble given in terms of Message Sequence Charts [18,19,20].

Furthermore it is possible to go beyond deadlock if we have a specification of the behavioral integration failure to be avoided. This specification is an implicit failure specification. Actually we assume to specify all the assembled system behaviors which are failure-free rather than to explicitly specify the failure. Under these hypotheses Synthesis automatically derives the assembling code of the connector for a set of components. The connector is derived in such a way to obtain a failure-free system. It is shown that the connector-based system is *equivalent* according to a suitable equivalence relation to the initial one once depurated of all the failure behaviors.

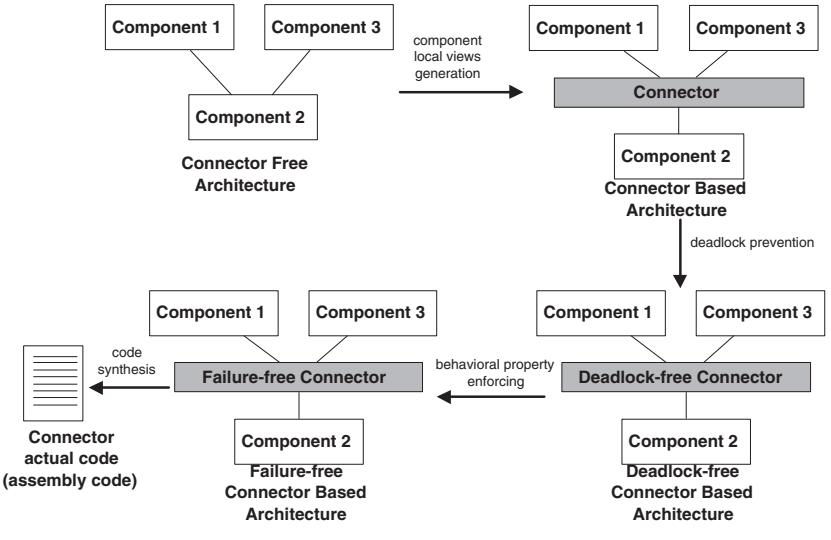


Fig. 1. The Synthesis Application Adaptation

As illustrated in Figure 1, the Synthesis framework realizes a form of system adaptation. The initial software system is *changed* by inserting a new component, the connector, in order to prevent interactions failures.

The framework makes use of the following models and formalisms. An architectural model, the connector-based architecture that constrains the way components can interact, by forcing interaction to go through the connector. A set of behavioral models for the components that describe each single component's interaction behavior with the external context in the form of label transition systems (LTS). A behavioral equivalence on LTS to establish the equivalence among

the original system and the adapted one. Temporal logic to specify the behavioral integration failure to be avoided, and then Buchi Automata and model checking to synthesize the failure-free connector specification. From the connector specification the actual code can then be automatically derived.

Let us now analyze the Synthesis approach to adaptation by means of the four *Ws* metric:

- *Why* there is the need to change? Here the purpose of the change is to correct functional behavior. That is to avoid interaction deadlocks and/or enforce a certain interaction property P. This adaptation is not due to change of context, since it is not driven by quantitative parameters. The change here aims at correcting a functional misbehavior.
- *What* does (not) change? It changes the topological structure and the interaction behavior. A new component is inserted in the system and the overall interaction behavior is changed. The invariant part of the system is represented by all the correct behaviors. The proof that the adaptation preserves the invariant is by construction.
- *When* does the change happen? It happens at assembly time, thus prior to deployment and execution. Thus it is actually part of the development process.
- *What/Who* manages the change? An external entity: The developer through the Synthesis framework.

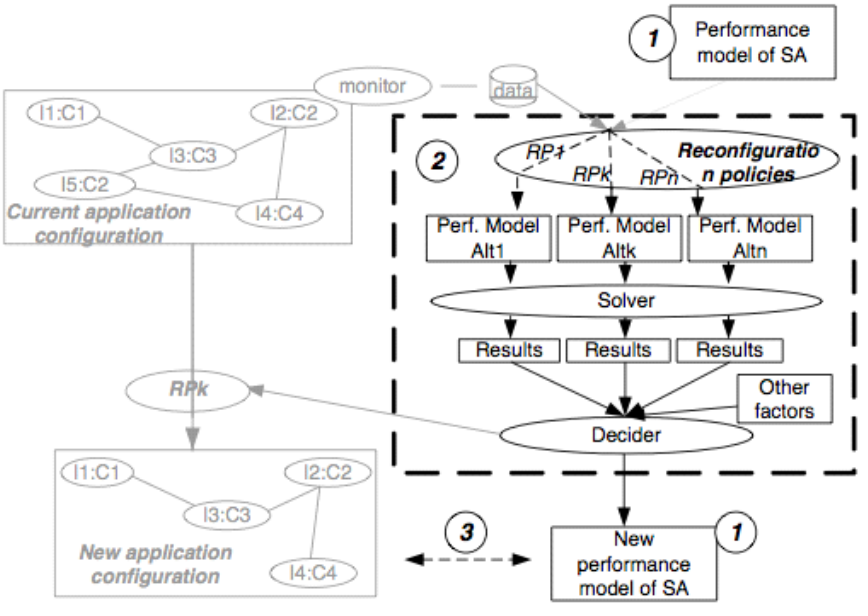
3.3 Performance

The work presented in this section, discusses PFM a framework to manage performance of software system at runtime based on monitoring and model-based performance evaluation [17]. The approach makes use of Software Architectures as abstractions of the managed application also at run time when the system is operating.

The framework monitors the performance of the application and, when a performance problem occurs, it decides the new application configuration on the basis of feedback provided by the on-line evaluation of performance models of several reconfiguration alternatives. The main characteristic of this approach is the way such alternatives are generated. In fact, differently from other approaches we do not rely on a fixed repository of predefined configurations but, starting from the data retrieved by the on-line monitoring (that represents a snapshot of the system current state), we generate a number of new configurations by applying the rules defined within the reconfiguration policy. Once such alternatives have been generated we proceed to the on-line evaluation by predicting which one of them is most suitable for resolving the problem occurred. In particular, the choice of the new system configuration might consider several factors, such as, for example, security and reliability of the application, and resources needed to implement the new configuration.

In this approach performance evaluation models are used to predict the system performance of the next system reconfiguration alternative. To this aim,

each eligible system configuration is described by means of a predictive model instantiated with the actual values observed over the system until the moment of the performance alarm. The models are then evaluated and on the basis of the obtained results the framework decides the reconfiguration to perform over the software system. Therefore, the predictive models representing the software system alternatives are evaluated at run time and this poses strong requirements on the models themselves. PMF has been experimented to manage the performance of the *Siena* publish/subscribe middleware [7,6]. The experiment shows that the usage of predictive models improves the decision step. The system reconfigured with the chosen alternative has better performance than the other alternatives generated during the reconfiguration process. The configuration alternatives we experimented all deal with structural changes of the *Siena* network topology in order to improve messages routing.



(b) Flow of the activities in an adaptation step

Fig. 2. Adaptation for performance

In Figure 2 the PMF components are represented. It is worthwhile stressing that all the described 4 steps are carried on at run time, while the system is operating. Note that predictive models are commonly used to carry on quantitative analysis at development time, while the system is under construction [9]. Their use at execution time raises a number of challenging research issues like: what data are relevant to collect? The collected data is more fine-grained than the performance model parameters, how can data be used? Models have to be

modified and evaluated online this means that they must require fast solution techniques. However fast solution techniques usually apply for simple predictive models, then which performance model should be used? How is the decision on the next configuration taken? The answers to all these questions should consider different aspects like security, resources availability, and so on.

Let us now consider PMF with the four Ws metric:

- *Why* there is the need to change? The change aims to correct non-functional behavior, i.e. adjust Performance. This change is context dependent.
- What does (not) change? In the *S*IENA experiment the topological structure is going to be modified, while the overall behavior is kept equivalent. That is the change does not affects the routing capabilities of the *S*IENA network.
- When does the change happen? The change happens at run time, while the system is operating and it is enacted trough run time monitoring of performance parameters.
- What/Who manages the change? An external to the system entity, that is the PMF framework provides support to the whole re-configuration process as shown in Figure 2.

3.4 Resource Aware Applications

This framework aims at developing and deploying (Java) adaptable application. It supports the development of applications that are *generic* and can be correctly adapted with respect to a dynamically provided context, which is characterized in terms of available (hardware or software) resources, each one with its own characteristics. To attack this problem we use a declarative and deductive approach that enables the construction of a generic adaptable application code and its correct adaptation with respect to a given execution context [8,10,11]. Inspired by Proof Carrying Code (PCC) [12,15] techniques, we have defined a formal setting which enables us to reason about Java program adaptability with respect to resource usage. We use first-order logic formulas to model the code behavior with respect to the resources which characterize the execution context. The adaptation process is carried out by using theorem proving techniques that try to derive a formal proof that the code behavior can be correctly adapted to the given context. Provided that the proof exists, by construction it gives information on how the adaptation has to be done. The adapted code is thus by construction certified to correctly work with respect to the execution context resources availability. In Figure 3 we show the components of the framework's architecture.

The *Development Environment* is a standard Java development environment where the developer can write applications. We only assume that the applications are written according to some framework programming guidelines that easy their (generic) management. The output of this step is an extended Java program representing a generic program.

The *Abstract Resource Analyzer* produces from the application written in the *Development Environment* a declarative description of its characteristics in terms

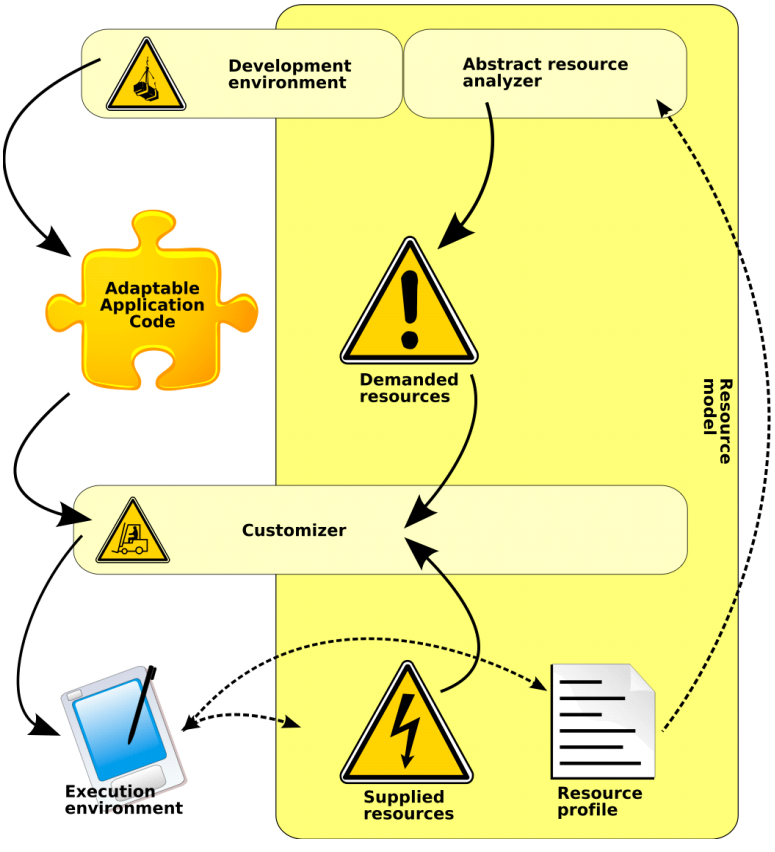


Fig. 3. Adaptation for resource consumption

of resource demands. It is an abstract semantics that interprets the applications with respect to a well defined *Resource Model*, and extracts the information according to that model.

The *Customizer* carries out the actual adaptation of the application before deploying it in the target environment for execution. This step produces a standard Java application.

The *Execution Environment* can be any device that will host the execution of the application. Typically the *Execution Environment* will be provided by Personal Digital Assistants (PDA), mobile phones, smart phones, etc. From this point of view, the *Execution Environment* is not strictly part of the framework we are presenting here. However it must be characterized by a declarative description of the resources it provides that we assume to be provided by the component itself.

The *Resource Model* characterizes resources and provides metrics to allow reasoning on the adaptation in order to be able to choose the *"best"* one according to the adaptation policy.

Let us analyze the resource aware framework with the four Ws metric:

- Why there is the need to change? The change allows to correctly utilize the host device resources. Therefore it is driven by non functional requirements and it is context dependent.
- What does (not) change? The service/application core behavior does not change. It changes the quantitative semantics (resource consumption) of the service implementation. The logic of the adaptation has been programmed by the developer.
- When does the change happen? The framework manages the adaptation at deployment time. That is as soon as the information on the execution context becomes available.
- What/Who how is the change managed? The deployment framework carries out the whole adaptation process. The application when in execution is completely customized and works like a standard Java application.

Summarizing in this section I have presented three examples of adaptation that differ with respect to several dimensions. One issue that is raised by the *when* dimension in the four Ws metric is whether *adaptability* is static or dynamic. The system adapts at run time, how and when the adaptation is computed or carried out does not change the problem, it is just a matter of *cost*. The cost I am referring to here is the cost of carrying out the adaptation maintaining the original integrity of the part of the application that does not change, i.e. the *invariant*. Thus if the application A that exhibits property P is changed into an application A' and the change is supposed to preserve the property P, then this means that also A' must satisfy P. For example the property P could be type integrity, thus we require that the change does not undermines type integrity in the changed application. Obviously, in this case, carrying out the change statically, i.e. before the system is running permits to prove type integrity of A' in a less expensive way than if done at run time.

4 Softure: The Process View

In this section I cast the above discussed challenges in a process view. The process view focusses on the set of activities that characterize the production and the operation of a software system. These activities are traditionally divided into activities related to the actual production of the software system and activities that are performed when the system can be executed and goes into operation. Specification, Design, Validation, and Evolution activities vary depending on the organization and the type of system being developed. Each Activity requires its Language, Methods and Tools and works on suitable artifacts of the system. For validation purposes each artifact can be coupled with a *model*. Models are an idealized view of the system suitable for reasoning, developing, validating a real system. To achieve dependability a large variety of models are used from behavioral to stochastic. These models represent the systems at very different levels of

abstraction from requirements specification to code. The ever growing complexity of software has exacerbated the dichotomy development/static/compile time versus execution/dynamic/interpreter time concentrating as many analysis and validation activities as possible at development time.

Software puts new requirements on this standard process. The evolutionary nature of Software makes unfeasible a standard approach to validation since it would require before the system is in execution to predict the system behavior with respect to virtually any possible change. Therefore in the literature most approaches, that try to deal with the validation of dynamic software systems, concentrate the changes to the structure by using graph and graph grammars formalisms or topological constraints [25,23,22,24,26,27]. As far as changes to behavior are concerned, only few approaches exist that make use either of behavioral equivalence checks or of the type system [4,28,29] or through code certification [12,30]. If dependability has to be preserved through adaptation whatever the change mechanism is, at the time the change occurs, a validation check must be performed. This means that all the models necessary to carry on the validation step must be available at run time and that the actual validation time becomes now part of the execution time.

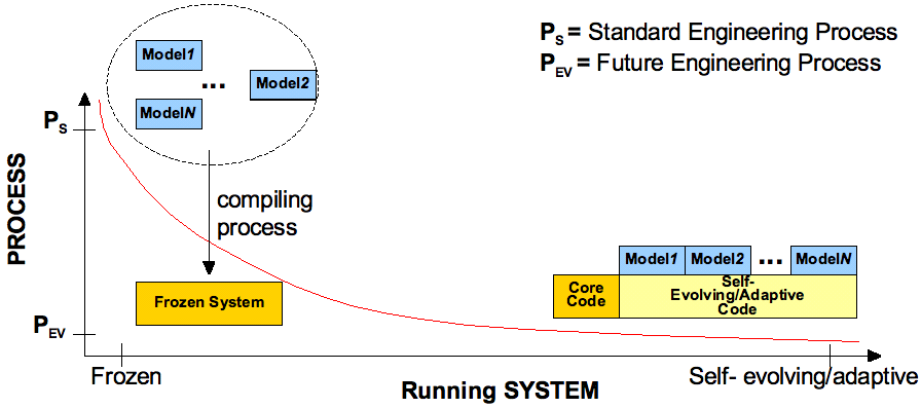


Fig. 4. The Future Engineering Process

The Future development process therefore has to explicitly account for complex validation steps at run time when all the necessary information are available. Figure 4 represents the process development plane delimited on one side by the standard process and on the other side by the future development one. The vertical dimension represents the static versus dynamic time with respect to the analysis and validation activities involved in the development process. The horizontal axis represents the amount of adaptability of the system, that is its ability to cope with evolution still maintaining dependability. The standard development process carries out most of the development and validation activities before the system is running that is during development. The result is a running

system that, at run time, is frozen with respect to evolution. Considering development processes that allow increasingly degrees of adaptability permits to move along the horizontal axis thus ideally tending to a development process that is entirely managed at run time. In the middle we can place development processes that allow larger and larger portions of the system to change at run time and that make use for validation purposes of artifacts that can be produced statically. In the following section I introduce an instance of the Future Engineering Process that has been proposed in the scope of the PLASTIC project.

4.1 PLASTIC

The PLASTIC project aims to offer a comprehensive provisioning platform for software services deployed over B3G networks (see Figure 5). A characteristic of this kind of infrastructure is its heterogeneity, that is it is not possible to assume that the variety of its components' QoS is homogenized through a uniform layer. PLASTIC aims at offering B3G users a variety of application services exploiting the network's diversity and richness, without requiring systematic availability of an integrated network infrastructure. Therefore the PLASTIC platform needs to enable dynamic adaptation of services to the environment with respect to resource availability and delivered QoS, via a development paradigm based on Service Level Agreements and resource-aware programming.

The provided services should meet the user demand and perception of the delivered QoS, which varies along several dimensions, including: type of service, type of user, type of access device, and type of execution network environment.

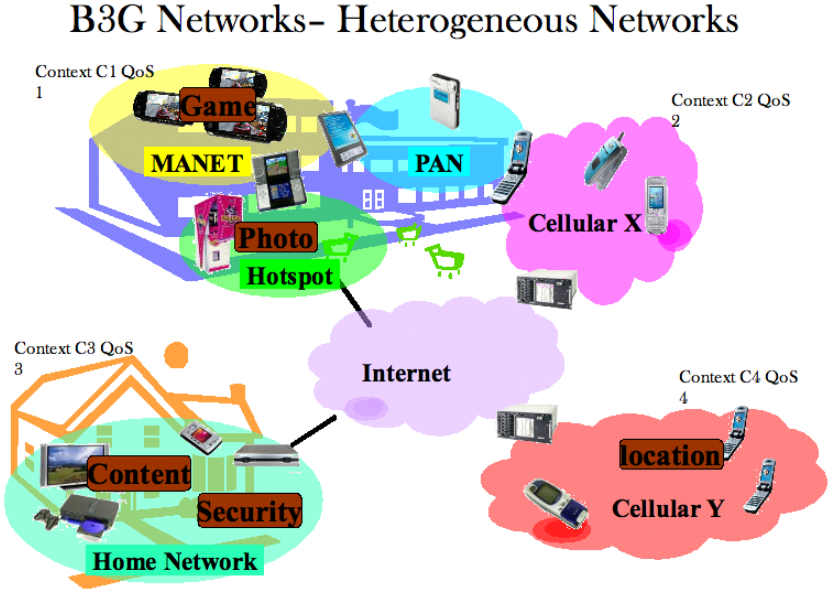


Fig. 5. B3G Networks

Referring to the challenges discussed in Section 2, this means that services must be *dependable* according to the users expected QoS.

This demands for a software engineering approach to the provisioning of services, which encompasses the full service life cycle, from development to validation, and from deployment to execution.

The PLASTIC answer to the above needs is to offer a comprehensive platform for the creation and provisioning of lightweight, adaptable services for the open wireless environment. Supporting the development of resource-aware and self-adapting components composing adaptable services requires focusing on the QoS properties offered by services besides the functional ones. The whole development environment is based on the PLASTIC Conceptual Model [1]. Recently, several approaches to conceptualize the world of services have been proposed. The PLASTIC model takes the move from the SeCSE conceptual model [31,32] that it has been suitably extended to reflect all the concepts related to B3G networks and service provision in B3G networks. In particular it focusses on the following key concepts:

– *Service level agreement* that clearly set commitment assumed by consumers and providers and builds on services descriptions that are characterized functionally, via a service interface and non-functionally via a Service Level Specification *SLS*.

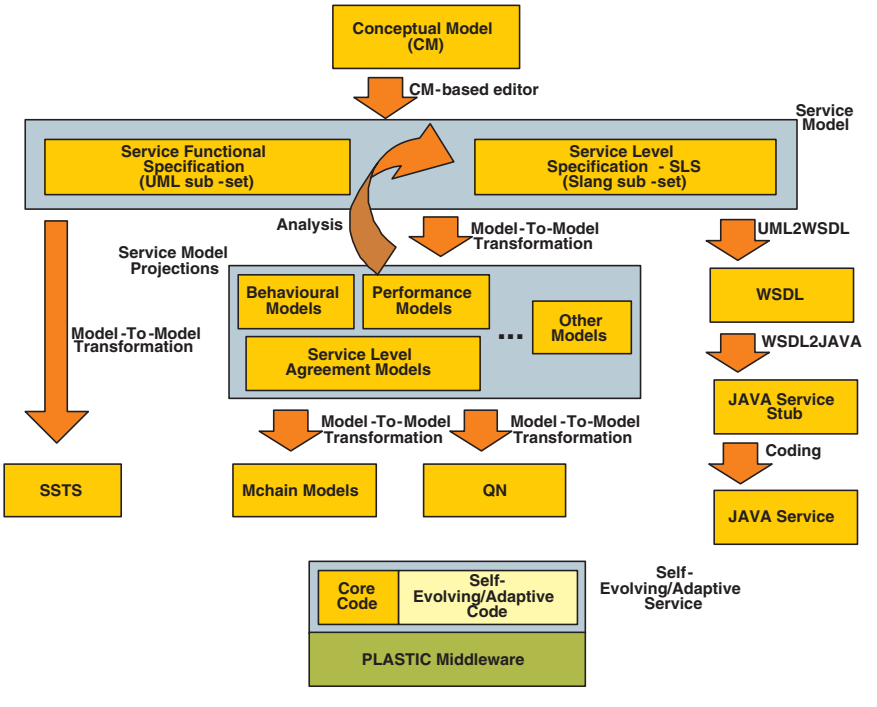


Fig. 6. The PLASTIC Development Process

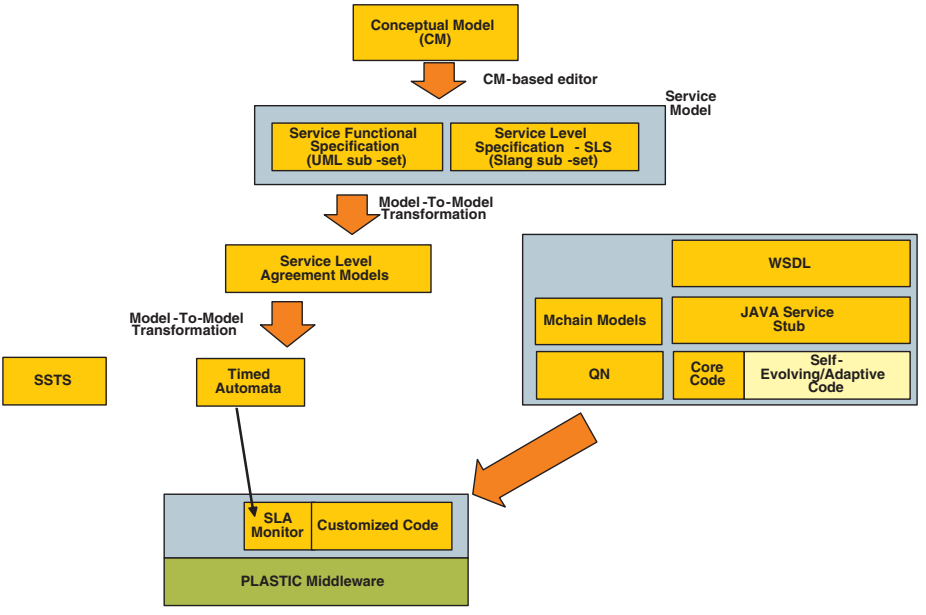


Fig. 7. The PLASTIC Deployment Process

– *Context awareness and adaptation* as the context is a key feature distinguishing services in the vast B3G networking environment. B3G networking leads to have diverse user populations, changing availability in system resources, and multiple physical environments of service consumption and provisioning. It is then crucial that services adapt as much as possible to the context for the sake of robustness and to make themselves usable for given contexts.

As illustrated in Figure 6 adaptability is achieved by transferring some of the validation activities at run time by making available models for different kind of analysis. In particular stochastic models and behavioral ones will be made available at run time to allow the adaptation of the service to the execution context and service on line validation, respectively.

In PLASTIC all the development tools will be based on the conceptual model exploiting as much as possible model-to-model transformations. The definition of a service will consists of a functional description and of a Service Level Specifications that defines the Quality of Service characteristics of the service. The overall service description is obtained by means of an iterative analysis specification phase that makes use of behavioral and stochastic models. These models suitably refined with pieces of information coming from the implementation chain, will then be made available as artifacts associated to the service specification.

With respect to the spectrum presented in Figure 4 the PLASTIC development process will present a limited form of adaptability as shown in Figure 7. The components implementing PLASTIC services will be programmed using the resource

aware programming approach presented in Section 3 by using Java. In PLASTIC adaptation happens at the time the service request is matched with a service provision. This match has to take into account the user's QoS request and the service SLS and the result of the match will produce the Service Level Agreement (SLA) that defines the QoS constraints of the service provision. During this matching process in order to reach an SLA the service code might need to be adapted, according to the resource aware approach, thus resulting in a customized service code that satisfies the user's QoS request and results in a SLA.

5 Conclusions

In this paper I have discussed my point of view on software in the future. Adaptability and Dependability will play a key role in influencing models, languages and methodologies to develop and execute future software applications. In a broader software engineering perspective it is therefore mandatory to reconcile the static/compile time development approach to the dynamic/interpreter oriented one thus making models and validation technique manageable lightweight tools for run time use. There are several challenges in this domain. Programming Language must account in a rigorous way of *quantitative* concerns, allowing programmers to deal with these concerns declaratively. Models must become simpler and lighter by exploiting compositionality and partial evaluation techniques. Innovative development processes should be defined to properly reflect these new concerns arising from software for ubiquitous computing. I presented the PLASTIC approach to service development and provision in B3G networks as a concrete instance of the problem raised by Softure. The solutions we are experimenting in PLASTIC are not entirely innovative *per se* rather they are used in a completely new and non trivial fashion. Summarizing my message is that in the Softure domain it is important to think and research *point to point* theories and techniques but it is mandatory to re-think the whole development process in order to cope with the complexity of Softure and its requirements.

Acknowledgments

The author would like to acknowledge the IST project PLASTIC that partially supported this work and all the members of the PLASTIC Consortium and of the SEALab at University of L'Aquila for joint efforts on all the research efforts reported in this paper.

References

1. PLASTIC IST STREP Project: Home page on line at: `http://www-c.inria.fr:9098/plastic`
2. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In: IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, US (1994)

3. IFIP WG 10.4 on Dependable Computing And Fault Tolerancs http://www.dependability.org/wg10.4/
4. Allen, R., Garlan, D.: A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology 6(3), 213–249 (1997)
5. Magee, J., Kramer, J.: Concurrency: State models & java programs. Wiley publisher, Chichester (1999)
6. Caporuscio, M., Carzaniga, A., Wolf, A.L.: Design and evaluation of a support service for mobile, wireless publish/subscribe applications. IEEE Transactions on Software Engineering (December 2003)
7. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems 19(3), 332–383 (2001)
8. Inverardi, P., Mancinelli, F., Nesi, M.: A Declarative Framework for adaptable applications in heterogeneous environments. In: Proceedings of the 19th ACM Symposium on Applied Computing (2004)
9. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based Performance Prediction in Software Development: A Survey IEEE Transaction on Software Engineering (May 2004)
10. Inverardi, P., Mancinelli, F., Marinelli, G.: Correct Deployment and Adaptation of Software Applications on Heterogenous (Mobile) Devices. In: ACM Proceedings Workshop on Self-Healing Software (2002)
11. Mancinelli, F., Inverardi, P.: Quantitative resource-oriented analysis of Java (adaptable) application. In: ACM Proceedings Workshop on Software Performance (2007)
12. Necula, G.C.: Proof-Carrying Code. In: Jones, N.D. (ed.) Proceedings of the Symposium on Principles of Programming Languages, pp. 106–119. ACM Press, Paris, France (1997)
13. Necula, G.C., Lee, P.: Proof-Carrying Code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University (September 1996)
14. Necula, G.C., Lee, P.: Safe kernel extensions without run-time checking. In: Proceedings of the Symposium on Operating System Design and Implementation, Seattle, Washington, pp. 229–243 (October 1996)
15. Necula, G.C., Lee, P.: Efficient Representation and Validation of Logical Proofs. In: Pratt, V. (ed.) Proceedings of the Symposium on Logic in Computer Science, pp. 93–104. IEEE Computer Society Press, Indianapolis, Indiana (1998)
16. Necula, G.C., Lee, P.: Safe, untrusted agents using proof-carrying code. In: Vigna, G. (ed.) Mobile Agents and Security. LNCS, vol. 1419, pp. 61–91. Springer, Heidelberg (1998)
17. Caporuscio, M., Di Marco, A., Inverardi, P.: Model-based system reconfiguration for dynamic performance management. Journal of Systems and Software (to appear, 2006)
18. Itu telecommunication standardisation sector, itu-t reccomendation z.120. Message Sequence Charts (msc'96), Geneva
19. Uchitel, S., Kramer, J., Magee, J.: Detecting implied scenarios in message sequence chart specifications. In: ACM Proceedings of the joint 8th ESEC and 9th FSE, ACM Press, Vienna (2001)
20. Uchitel, S., Kramer, J.: A workbench for synthesising behaviour models from scenarios. In: proceeding of the 23rd IEEE International Conference on Software Engineering (ICSE'01), Toronto, Canada (May 2001)

21. Inverardi, P., Tivoli, M.: A compositional synthesis of failure-free connectors for correct components assembly. In: proceedings of the 6th ICSE Workshop on Component-Based Software Engineering (CBSE6): Automated Reasoning and Prediction at 25th ICSE 2003, Portland, Oregon, USA (May 3-10, 2003)
22. Georgiadis, I., Magee, J., Kramer, J.: Self-organising software architectures for distributed systems. In: Proc. of the 1st Work. on Self-Healing Systems (WOSS02), pp. 33–38. ACM Press, New York (2002)
23. Hirsch, D., Inverardi, P., Montanari, U.: Graph grammars and constraint solving for software architecture styles. In: Proc. of the 3rd Int. Software Architecture Workshop (ISAW-3), pp. 69–72. ACM Press, New York (1998)
24. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: Proc. of the 4th ACM SIGSOFT Symp. On Foundations of Software Engineering (FSE-4), pp. 3–14. ACM Press, New York (1996)
25. Metayer, D.L.: Describing software architecture styles using graph grammars. IEEE Trans. Software Engineering 24(7), 521–533 (1998)
26. Taentzer, G., Goedicke, M., Meyer, T.: Dynamic change management by distributed graph transformation: Towards onfigurable distributed systems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, Springer, Heidelberg (2000)
27. Baresi, L., Heckel, R., Thne, S., Varr, D.: Style-Based Refinement of Dynamic Software Architectures. WICSA, 155–166 (2004)
28. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting Software Architecture to Implementation. In: proceedings of ICSE 2002 (May 2002)
29. Aldrich, J.: Using Types to Enforce Architectural Structure. University of Washington Ph.D. Dissertation (August 2003)
30. Barthe, G.: Mobius, securing the next generation of java-based global computers. ERCIM News (2005)
31. SeCSE Project, `http://secse.eng.it`
32. Colombo, M., Di Nitto, E., Di Penta, M., Distante, D., Zuccalá, M.: Speaking a Common Language: A Conceptual Model for Describing Service-Oriented Systems. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, Springer, Heidelberg (2005)
33. Autili, M., Cortellessa, V., Marco, A.D., Inverardi, P.: A conceptual model for adaptable context-aware services. In: Proc. of International Workshop on Web Services Modeling and Testing (WS-MaTe2006) (2006)
34. Autili, M., Inverardi, P., Navarra, A., Tivoli, M.: SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems Proc. of International Conference on Software Engineering (ICSE 2007) - Tool Demos Session (to appear)
35. Inverardi, P., Tivoli, M.: Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, Springer, Heidelberg (2003)