An Architecture-Centric Approach for Producing Quality Systems

Antonia Bertolino[1], Antonio Bucchiarone[1,2], Stefania Gnesi[1], and Henry Muccini[3]

[1] Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" (ISTI-CNR), Area della Ricerca CNR di Pisa, 56100 Pisa, Italy
{antonia.bertolino, stefania.gnesi}@isti.cnr.it
[2] IMT Graduate School, Lucca
antonio.bucchiarone@imtlucca.it
[3] Dipartimento di Informatica, Universitá dell'Aquila, Via Vetoio 1, 67100 L'Aquila, Italy
muccini@di.univaq.it

Abstract. Software architecture has been advocated as an effective means to produce quality systems. In this paper, we argue that integration between analysis conducted at different stages of development is still lacking. Hence we propose an architecture-centric approach which, by combining different technologies and tools for analysis and testing, supports a seamless tool-supported approach to validate required properties. The idea is to start from the requirements, produce a validated model of the SA, and then use the SA to derive a set of conformance test cases. In this paper, we outline the process, and discuss how some existing tools, namely QuARS, ModTest, CowTest and UIT, could be exploited to support the approach. The integrated framework is under development.

1 Introduction

Based now on more than a decade of research in this new discipline, the Software Architecture (SA) [18] has today become an important part of software development processes (e.g. [22,9]). From SA application in practice, software developers have in fact learned how SA specifications permit to obtain a better quality software system, while reducing realization time and cost. In current trends SA description is used for multiple purposes: while some companies use the SA description just as a documentation artifact, others make also use of SA specifications for enhanced analysis purposes, and finally many others use the architectural artifacts to (formally or informally) guide the design and coding process [28,12].

We focus here on the first two purposes, i.e., using SA for documentation and analysis. Indeed, notwithstanding the enormous progresses in software technologies promoted by the affirmation of UML, open standards specifications and new

R. Reussner et al. (Eds.): QoSA-SOQUA 2005, LNCS 3712, pp. 21–37, 2005.
© Springer-Verlag Berlin Heidelberg 2005

development paradigms, the problem of assuring as early as possible the adequacy and dependability of a software system still takes a preponderant portion of the development cycle cost and time budgets. Considering the SA, analysis techniques have been introduced to understand if the SA satisfies certain expected properties, and tools and architectural languages have been proposed in order to make specification and analysis techniques rigorous and to help software architects in their work [33,11]. Even though much work has been done on this direction, many limitations still need to be handled: *i*) analysis results from one phase of the software process are not capitalized for use in the following phases, *ii*) tool support remains limited, fragmented, and in demand of high investment for introduction, *iii*) use in industrial projects generally requires specific training needs and extra requirements. In analyzing the difficulties behind wider industrial take-up of SA methodology, one practical obstacle is that the existence of a *formal* modeling of the software system cannot be assumed in general. What we may reasonably presuppose for routine industrial development, rather, is a semi-formal, easy to learn, specification language. We thus employ and extend the standard UML as our notation for modeling SAs. Moreover, the applied methods should be *timely* (even incomplete models should allow to start analysis), and *tool supported* (automated tool support is fundamental for strongly reducing analysis costs).

We here interpret the term “Quality of Software Architecture” as implying that the SA specification becomes the main artifact used to analyze the entire system. We hence propose an *SA-centric approach for producing quality systems*: such an approach starts from the Natural Language (NL) definition of requirements, analyzes them with an automated tool-supported method, continues up to the specification and validation of an architectural model, and finally selects architectural-level test cases that are then used to test the implementation conformance with respect to the SA specification. Such process is meant to address the requirements imposed by items *i*), *ii*) and *iii*) above, and is conceived to be tool supported, model-based, and applicable even on incomplete specifications.

We do not claim that our proposal is a consistent and complete process for SA-centric software development. On the contrary, we emphasize that our proposed approach addresses specifically the analysis aspects of development, in order to eventually produce a quality system with validated properties. Our aim is to develop a seamless analysis process centered on the SA, which proceeds side by side along with development. We do not deal here with the activities entitles to the production of the artifacts, such as requirements or SA design, which we analyze.

Note that to realize the proposed analysis process we have taken a sort of “component-based” approach, in that instead of producing the supporting tools from scratch, we exploit tools and technologies already available, by adapting and suitably assembling them. This in the long term may not be the most effective solution, in the sense that perhaps other more suitable tools could instead be adopted. However, our interest is rather in analyzing the feasibility and advantages of a seamless analysis centered on SA, rather than in implementing a

highly efficient approach. Therefore, we leverage on results produced in previous research projects aimed at addressing specific stages of development. The emphasis of this paper is in the attempt of combining specific analysis techniques into an integrated SA-centric analysis process.

The paper is organized as follows: after introducing in Section 2.1 the most significant SA-centric development processes proposed so far, we outline our proposed analysis approach in Section 2.2. We introduce the exploited approaches and their tool support in Section 3, then we show some initial results and issues in integrating these approaches and tools (Section 4). Conclusions and future work are briefly drawn in Section 5.

2 SA-Based Analysis Process: State of the Art and Our Proposal Outline

The dependability of a software system strongly depends on the software development process followed to produce it, thus, on the selected activities and on how the system evolves from one activity to another. Several papers advocate the use of architectural specifications to identify deficiencies soon in the process, but only few systematic processes have been proposed to implement such requirement. Currently, automating the analysis stages that accompany the process of transiting from requirements to architectures and from architectures to code still remains subject of research.

2.1 Related Work

SA-based development processes, SA integration with the other development phases, and SA-based analysis are the main topics related to our work. We here survey some relevant works on each topic.

Software Architecture-Based Development Process: The Unified Software Development Process (UP) [24], is an iterative and incremental, use case driven, and architecture-centric software process. In the UP, an architecture specification is not provided in a single phase in the development process, but it embodies a collection of views created in different phases. In the Hofmeister, Nord and Soni' book on applied software architecture [22], an architectural specification is composed by four different views: *conceptual*, *module*, *execution*, and *code*. In the "architecture-based development" proposed in [10], the authors present a description of an architecture-centric system development where a set of architecture requirements is developed in addition to functional requirements. The process comprises six steps: *i*) elicit the architectural requirements (expressing quality-based architectural requirements through quality-specific scenarios), *ii*) design the architecture (using views), *iii*) document the architecture (considered as a crucial factor in the SA success), *iv*) analyze the architecture (with respect to quality requirements), *v*) realize and finally, *vi*) maintain the architecture. The requirements are informally validated, the architecture is specified

through different views, the architecture is analyzed through the Architecture Tradeoff Analysis Method (ATAM). This process differs from traditional development in which SA is used for design perspectives since it focuses on capturing system qualities for analysis. KobrA (Component-based Application Development) [5] is a software process- and product-centric engineering approach for software architecture and product line development. KobrA uses a component-based approach in each phase of the development process, it uses UML as the modelling language, it exploits a product line approach during components development and deployment, and it makes a clear distinction between the Framework Engineering stage and the Application Engineering stage (typical of product line developments).

For completeness, we also cite TOGAF [2], a detailed method and a set of guidelines to design, evaluate and build enterprise architectures and Catalysis [34], a methodology that makes special emphasis on component based development.

Software Architecture Integration with other Phases: To increase the overall system quality, two main topics need to be carefully addressed: *i*) how different phases in the software process should be mutually related, *ii*) and how results from one phase should be propagated down to the other phases.

Even if such processes take into consideration the different stages in the software process, they do not specify how requirements, architectures and implementation have to be mutually related. This is still an open problem advocated and investigated by many researchers [3,29,21]. In [3,29], ways to bridge the gap between requirements and software architectures have been proposed. Egyed proposes ways to trace requirements to software architecture models [21] and software architecture to the implementation [26].

Software Architecture-Based Analysis: In the research area of Software Architecture much work is oriented on architectural specification and analysis [11,33]. Many analysis methods and tools, based upon formal Architecture Description Languages (ADLs), have been proposed in order to perform analysis and model checking [25], behavioral analysis of concurrent systems [14], deadlock detection [4,15], architectural slicing [35] and testing [27]. Today, only few ADLs with their analysis capabilities are still supported. Many UML-based approaches have been proposed in order to make the adoption of SA specification and analysis feasible in industrial contexts [16].

2.2 Our Proposal Outline

Previous SA-based processes have provided guidelines on how to move from one stage to another and on how to analyze a software architecture. However, techniques and tools which allow to formalize and implement such techniques are not illustrated.

What we envision, instead, is a tool supported, SA-centric analysis process which allows to specify and analyze requirement specifications, specify the architecture and validate its conformance to requirements. Once the SA has been iden-

tified, both system tests and the system implementation are derived from it. Test specifications are identified from the architectural model (rather than directly from requirements) thus unifying some design steps, guaranteeing consistency of the whole project and, virtually, eliminating the need of review processes. During test execution, if faults are identified, the only artifact to be revised is the implementation (since the SA has been proved to be correct). Figure 1 summarizes this process (the labels A,B,C and D have been introduced in order to track correspondence with Figure 4). Both requirements, architectures and implementation are revised whenever specification or conformance errors are detected. The entire analysis process is driven by model-based specifications. The validation steps are tool supported.

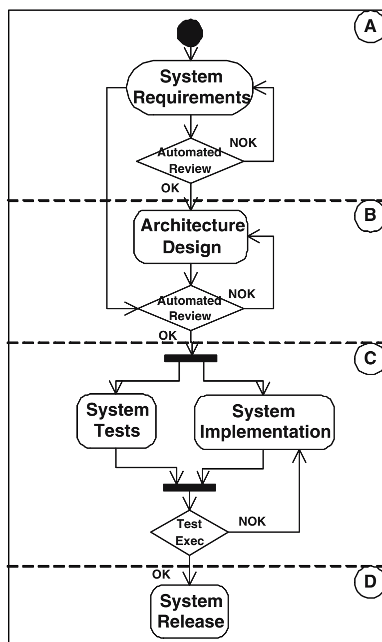

Fig. 1. Software Architecture-Based Analysis Process

In the following of this paper we present three different approaches which are the results of previous research projects carried on in independent way. By an appropriate integration approach, these three tools can support the entire analysis process we propose. The three tools are: QuARS (Quality Analyzer for Requirements Specifications), which supports the analysis of natural language requirements in a systematic and automatic way; ModTest, a framework for SA-based model-checking driven testing, which allows to validate the SA model with respect to requirements and to select test procedures; and CowSuite which supports the generation of executable test procedures from UML diagrams, and

their management via weights coefficients. The execution of such test procedures allows us to analyze the system conformance with respect to the SA.

Indeed, other approaches and analysis tools can be selected for integration. However, in this initial attempt to create an architecture-centric approach for producing quality systems we selected those we are more familiar with.

3 The Three Approaches in Isolation

3.1 QuARS

The tool QuARS (Quality Analyzer for Requirements Specifications) provides a systematic and disciplined analysis process for Natural Language (NL) requirements. The application of linguistic techniques to NL requirements allows their analysis from a lexical, syntactical or semantic point of view. For this reason it is proper to talk about, e.g., lexical non-ambiguity or semantic non-ambiguity rather than non-ambiguity in general. For instance, a NL sentence may be syntactically non-ambiguous (in the sense that only one derivation tree exists according to the applicable syntactic rules) but it may be lexically ambiguous because it contains wordings, which do not have a unique meaning.A Quality Model is the formalization of the definition of the term "quality" to be associated to a type of work product. The QuARS methodology performs the NL analysis by means of a lexical and syntactic analysis of the input file in order to identify those sentences containing defects according to the quality model in Figure 2 [19]. When the Expressiveness analysis is performed, the list of defective sentences is displayed by QuARS and a log file is created. The defective sentences can be tracked in the input requirements document and corrected, if necessary. Metrics measuring the defect rate and the readability of the requirements document under analysis are calculated and stored. The tool QuARS(Quality Analyzer for Requirement Specifications) provides a systematic and disciplined analysis process for NL requirements. The development of QuARS has been driven by the objective to realize a tool modular, extensible and easily usable. QuARS is based on the quality model, shown in Figure 2 [19]. It is composed of a set of high-level quality properties for NL requirements to be evaluated by means of syntactic and structural indicators directly detectable and measurable looking at the sentences in requirement document. These properties can be grouped in three categories:

- **Expressiveness:** it includes those characteristics dealing with incorrect understanding of the meaning of the requirements. In particular, the presence of ambiguities in and the inadequate readability of the requirements documents are frequent causes of expressiveness problems.
- **Consistency:** it includes those characteristics dealing with the presence of semantic contradictions in the NL requirements documents.
- **Understandability:** it includes those characteristics dealing with the lack of necessary information within the requirements document.

	Indicator	Description
Expressiveness	*Vagueness*	It is pointed out when parts of the sentence hold inherent vagueness, i.e. words having a non uniquely quantifiable meaning
	Subjectivity	It is pointed out if sentence refers to personal opinions or feeling
	Optionality	It reveals a requirement sentence containing an optional part (i.e. a part that can or cannot considered)
	Weakness	It is pointed out in a sentence when it contains a weak main verb
	Under-specification	It is pointed out in a sentence when the subject of the sentence contains a word identifying a class of objects without a modifier specifying an instance of this class
Consistency	*Under-reference*	It is pointed out in a Requirement Specifications Document (RSD) when a sentence contains explicit references to: not numbered sentences, documents not referenced into and entities not defined nor described into the RSD itself
Understandability	*Multiplicity*	It is pointed out in a sentence if the sentence has more than one main verb or more than one direct or indirect complement that specifies its subject
	Implicity	It is pointed out in a sentence when the subject is generic rather than specific.
	Comment Frequency	It is the value of the CFI (Comment Frequency Index). [CFI= NC / NR where NC is the total number of Requirements having one or more comments, NR is the number of Requirements of the RSD]
	Unexplanation	It is pointed out in a RSD when a sentence contain acronyms not explicitly and completely explained within the RSD itself

Fig. 2. Quality Model

The Functionalities provided by QuARS are:

1. Defect identification: QuARS performs a linguistic analysis of a requirement document in plain text format and points out the sentences that are defective according to the expressiveness quality model. The defective sentences can be automatically tracked on the requirement document to allow their correction.
2. Requirements clustering: The capability to handle collections of requirements, i.e. the capability to highlight clusters of requirements holding specific properties, can facilitate the work of the requirements engineers.
3. Metrics derivation: QuARS calculates metrics (The Coleman-Liau Formula and the defect rate) during the analysis of a requirements document.

QuARS satisfies the requirements identified in Section 1. The Quality Analysis by QuARS allows the automatic generation of validated requirements documents that will be used for the definition of Functional Properties *(goal i)*. The quality analysis is supported by QuARS*(goal ii)*. QuARS can be applied even on partial specifications, it does not require formal modeling and it is tool supported *(goal iii)*.

3.2 ModTest

ModTest is an architecture-centric approach for model-checking based testing [13]. It is based on the idea that SA-based testing and exhaustive analysis through model-checking can be successfully integrated in order to produce highly-dependable systems. In particular, ModTest works in a specific context, where the SA specification of the system is available, some properties the SA has to respect are identified, and the system implementation is available.

Model-checking techniques are used to validate the SA model conformance with respect to selected properties, while testing techniques are used to provide confidence on the implementation fulfillment to its architectural specification. (The architecture-centric approach is graphically summarized in Figure 3).

The *advantages* we obtain are manifold: we apply the model-checking technique to higher-lever (architectural) specifications, thus governing the state-explosion problem, while applying testing to the final implementation. We check and test the architecture and the system implementation with respect to properties elicited from requirements. Moreover, the test case selection phase is driven by both the requirements and the architectural model.

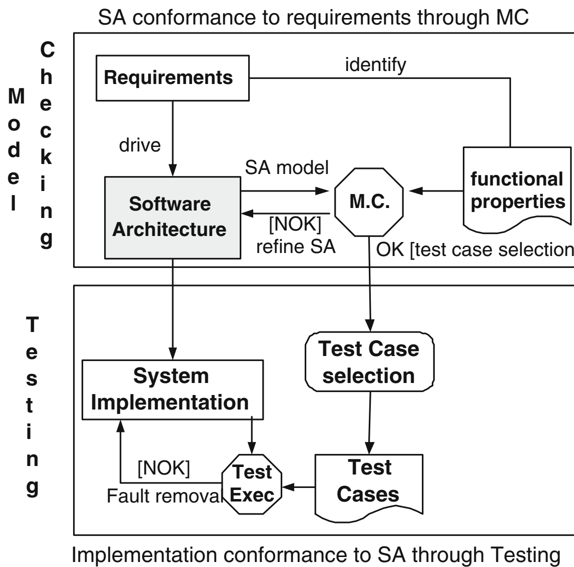

Fig. 3. ModTest

ModTest has been implemented by using two different technologies: Charmy [1], a framework which allows to model check architectural specifications with respect to selected properties, and TeStor [32] an algorithm which permits to extract test specifications from Charmy outputs.

Following the Charmy approach, the software architect specifies the system architecture, in an UML-based diagrammatic way. Both a structural and a behavioral viewpoints are taken into account. The SA topology is specified in terms of components, connectors and relationships among them, where components represent abstract computational subsystems and connectors formalize the interactions among components. The internal behavior of each component is specified in terms of state machines. Once the SA specification is available, a translation feature is used to obtain from the model-based SA specification, a formal executable prototype in Promela (the specification language of SPIN)[23].

On the generated Promela code, we can use the SPIN standard features to find, for example, deadlocks or parts of states machines that are unreachable. Behavioral properties modeled using the familiar formalism of sequence diagrams, are automatically translated into Büchi automata (the automata representation for LTL formulae). Each sequence diagram represents a desired behavioral property we want to check in the Promela (architectural) prototype. (More details may be found on [31]).

Whenever the SA specification has been validated with respect to certain requirements, the SA itself is used to identify test procedures to be used to provide confidence on the implementation fulfillment to the architectural specification. TeStor (our test sequence generator algorithm) gets in input the behavioral model of the software under test (i.e., the SA behavioral model) and a set of test generation directives (inSD), and outputs a set of sequence diagrams (outSD) representing the test specification. An inSD represents the property previously checked with Charmy (used as a test directive) while an outSD contains the sequence of messages expressed by the inSD, enhanced/completed with information gathered by the components' state machines. (More details may be found on [32]).

The Charmy approach is tool supported: via a *topology editor*, the SA topology is specified in terms of components, connectors and links. The *thread editor* allows to specify the internal behavior of each component. The *sequence editor* allows to draw sequence diagram representing desired behavioral properties we want to check. TeStor has been implemented as a plugin component. The user selects the inSD from a list of scenarios, then she runs the TeStor algorithm by activating the plugin module and the corresponding outSDs are automatically generated. The tool is available on [1].

ModTest satisfies the requirements identified in Section 1. Model-checking and testing are integrated in a unique process, thus allowing to reuse model-checking results for generating test specifications (goal *i*). The approach is tool supported, from SA specification to test procedures generation (goal *ii*). ModTest can be applied even on partial specifications, it does not require formal modeling, and it is tool supported (goal *iii*).

3.3 Cow_Suite Test Strategy

Cow_Suite [8] is a methodology originally conceived for test planning and generation, since the early stages of system analysis and modeling, based on a UML design. The name Cow_Suite stands for *COW*test plu*S UIT* Environment, and as the name implies it combines two original components:

1. CowTest (Cost Weighted Test Strategy) is a strategy for test prioritization and selection;
2. UIT (*U*se *I*nteraction *T*est) is a method to derive the test cases from the UML diagrams.

These two components work in combination, as CowTest helps decide which and how many test cases should be planned from within the universe of testcases

that UIT could derive for the system under consideration. In the remainder of this section we very shortly present the main characteristics of these two components. Further information can be retrieved from [17].

CowTest: Essentially, the CowTest strategy considers the diagrams composing the design and, by using their mutual relationships, organizes them into a hierarchical structure. More precisely, considering in particular the Actors, the Use Cases (UCs) and the Sequence Diagrams (SDs), they are first organized in an oriented graph called the Main Graph, which is then explored by using a modified version of the Depth-First Search algorithm for producing a forest of Trees, which constitute the basic hierarchical structures of the CowSuite approach. Each level in each tree evidences a different degree of detail of the system functionalities and represents for testing purposes a specific integration stage.

The central feature of CowTest then is that the nodes of the derived trees are annotated by the test manager with a value, called the weight, belonging to the [0,1] interval and representing its relative "importance" with respect to the other nodes at the same level: the more critical a node the higher its weight (the tool by default provides initially uniform weights which can be easily modified). Different criteria can be adopted to define what importance means for test purposes, e.g., the component complexity, or the usage frequencies, such as in reliability testing. Oftentimes, these criteria are not documented or even explicitly recognized, but their use is implicitly left to the sensibility and expertise of the managers.

The basic idea behind CowTest is requiring the test manager to make explicit these criteria and supporting them with a systematic tool strategy to use such information for test planning.

In the context of this proposed architecture-centric approach, it is our intention to enrich the phase of requirement analysis (normally focused on modeling concerns), also with the managerial concern of introducing an evaluation of the importance of the analysed requirements. Clearly, in this case we will consider weights which reflect architectural concerns. The CowTest approach fits quite well into the proposed process, as an help to manage the apportioning of testing effort onto the architectural specification.

For each node (which will correspond to a SD, i.e. to the description of a system architectural behaviour) the final weight is then computed as the product of all the nodes weights on the complete path from the root to this node. These final weights can be used for choosing amongst the tests to execute, in two different manners:

1. by fixing the number of test cases: then CowTest selects the most suitable distribution of the test cases among the functionalities on the basis of the leaves weights.
2. by fixing a functional test coverage as an exit criterion for testing. In this case CowTest can drive test case selection, by highlighting the most critical system functionalities and properly distributing the test cases.

UIT: Largely inspired to the Category Partition Method [30], UIT systematically constructs and defines a set of test cases for the Integration Testing phase by using the UML diagrams as its exclusive reference model and without requiring the introduction of any additional formalisms.

UIT was originally conceived [7] for UML-based integration testing of the interactions among the objects, or objects groups, involved in a SD. For each given SD, UIT automatically constructs the Test Procedures. A Test Procedure instantiates a test case, and consists of a sequence of messages, and of the associated parameters. UIT is an incremental test methodology; it can be used at diverse levels of design refinement, with a direct correspondence between the level of detail of the scenarios descriptions and the expressiveness of the Test Procedures derived. For each selected SD, the algorithm for Test Procedures generation is the following:

Define Messages_Sequences: Observing the temporal order of the messages along the vertical dimension of the SDs, a Messages_Sequence is defined considering each message with no predecessor association, plus, if any, all the messages belonging to its nested activation bounded from the *focus of control* [20] region.

Analyze possible subcases: the messages involved in a derived Messages_Sequence may contain some feasibility conditions (e.g., if/else conditions). In this case a Messages_Sequence is divided in subcases, corresponding to the possible choices.

Identify Settings Categories: the Settings Categories are the values or data structures that can influence the execution of a Messages_Sequence.

Determine Choices: for each Message choices represent the list of specific situations or relevant cases in which the messages can occur; for the Settings Categories, they are the set or range of input data that parameters or data structures can assume.

Determine Constraints among choices: to avoid meaningless or even contradictory values of choices inside a Messages_Sequence, constraints among choices are introduced.

Derive Test Cases: for every possible combination of choices, for each category and message involved in a Messages_Sequence a test case is automatically generated.

Combined Approach: The combination of CowTest and UIT seems to well satisfy requirements posed in the Introduction. The approach is *(goal ii)* tool-supported with the CowSuite tool (some adaptation is of course needed); its two features of trading-off between extensive testing and limited effort, by means of CowTest weights, and of using standard UML diagrams, without requiring any extra annotation for derivation of test procedures, make it *(goal iii)* purposely conceived for industrial usage. Finally, with regard to req *(goal i)*, we intend (see Figure 4) to combine the usage of CowTest with the requirement analysis stage to find appropriate Architectural relevant weights, and to use as an input the SD produced by ModTest.

4 Integration

4.1 Methodologies Integration

In this section we instantiate the analysis process in Figure 1 via the integration of the three methodologies previously introduced into a smooth architecture-centric process. The result is the process sketched in Figure 4:

a1) Requirements are specified with commercial tools, e.g., RequisitePro, Doors, AnalystPro. Natural Language (NL) requirement documents are automatically produced (by commercial tools such as SoDA, DOORSRequireIT) in the form of .txt or .doc files;

a2) QuARS takes in input a NL requirement document, makes a quality analysis and gives in output a log file which lists the indications of requirements containing defects or not. If defects are detected by QuARS (NOK arrow), it points to some defects lowering the quality of the requirements document, a refinement activity (*Requirements Revision*) is initiated, followed by another quality analysis step. At this stage, use-cases can be identified and a first distribution of CowTest weights is performed. If defects are not detected (OK arrows), the architectural specification can start on the basis of the approved requirement document.

a3) NL Requirements are used to define some high-level functional properties that the system must satisfy.

a4) Requirements are used to drive the Software Architecture (SA) definition, that will be designed into Charmy;

a5) In parallel the NL Functional Properties are formalized in the appropriate form for Charmy (for example PSC [6]).

a6) By using Charmy we have an easy to use, pratical approach to model, and check architectural specifications;

a7) Whenever the SA specification does not properly implement selected requirements/properties (NOK arrows), the SA itself needs to be revised (*SA Revision*). Thanks to the model-checker outputs, and following the Charmy iterative process, we identify specific portions of the SA which need to be better specified. Thus, starting from a first system SA high level abstraction, which identifies few main subsystems and a set of high level properties, we obtain a model that can be validated with respect to significant high level behaviors. This first validation step allows us to gain confidence on the global design architectural choices and to identify portions of the SA that might need further and deeper investigation. The subsequent steps are to zoom into the potentially problematic subsystems by identifying implied sub-properties;

a8) The sequence diagrams representing the properties validated through Charmy is the input for TeStor. Through the TeStor algorithm we identify traces of interest for testing the whole system or just a relevant subsystem. The TeStor output represents test sequence specifications (outSD);

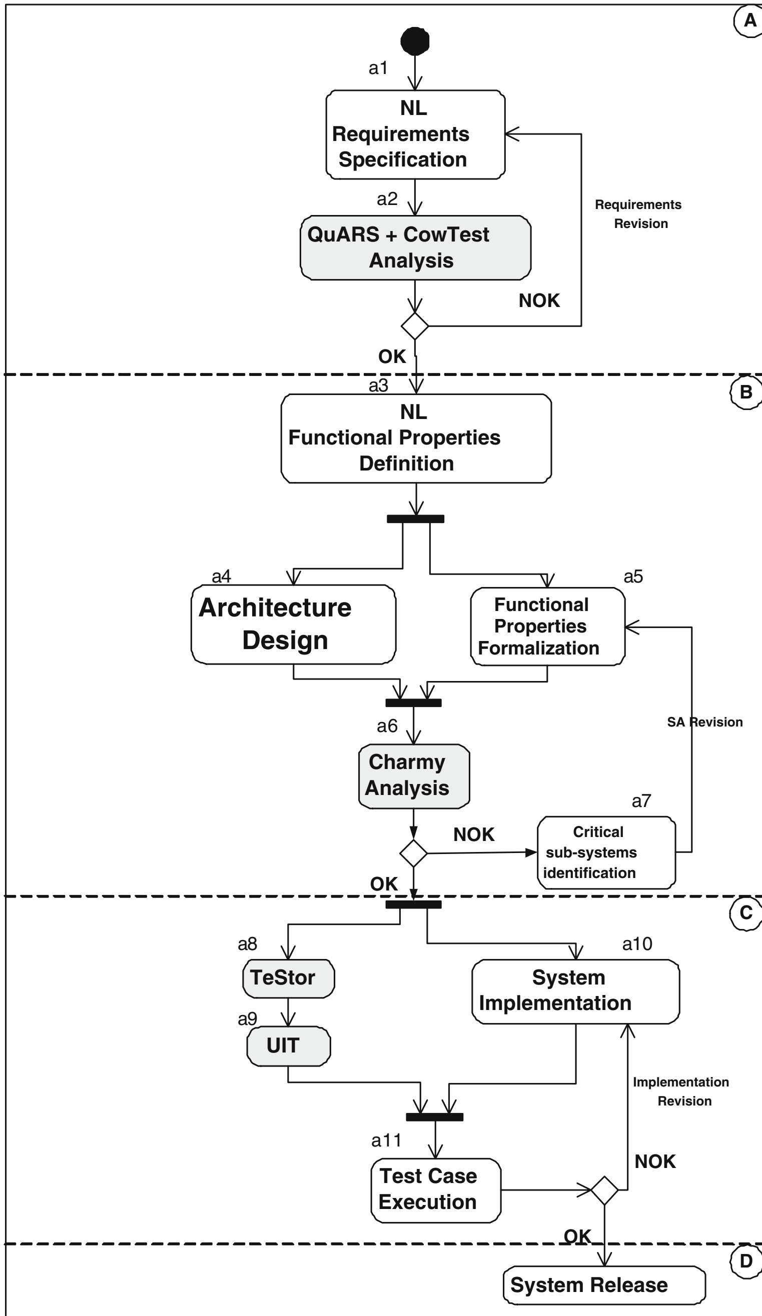

Fig. 4. Instantiating the SA-based Analysis Process

a9) The TeStor output (outSD) represents the input for the UIT methodology. UIT constructs the Test Procedures (TP) using solely the information retrieved from the outSD. A Test Procedure is a set of detailed instructions for setting-up, executing, and evaluating the results of a given test case. The number of Test Procedures associated with each outSD is assigned on the basis of the test effort distribution made according to the CowTest procedures.

a10) In parallel with the testing process, the SA is implemented;

a11) In the last step, the Test Procedures are executed on the system code. If the system testing doesn't introduce errors the system is released (OK arrows) otherwise, the system implementation has to be revised (NOK arrows).

4.2 Tools Integration

In order to perform a complete analysis of a software architecture we need an easy to use and automatic framework that starting from requirements specification, is able to validate the SA from qualitative points of view (Requirements, SA and Testing validation). For this aim the Charmy tool architecture, that is shown in Figure 5, can be of help. Taking a look to the Charmy `Core` macro-component,

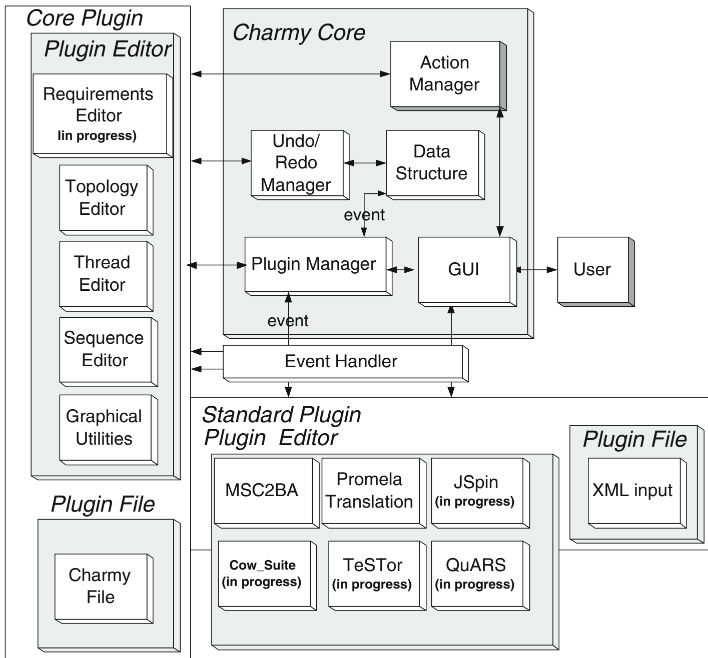

Fig. 5. The Charmy Plugin Architecture

it is composed by the `Data Structure` component, the `Plugin Manager` which allows to handle the plug of a new component in the core system, the `GUI` which receives stimuli by the users, and activates the `Action Manager` and the `Event Handler`.

The `Core Plugin` meta-component contains a set of core plugs, which allow to edit the NL system requirements, software architecture topology, the state machines and the scenarios (functional properties).

The `Standard Plugin` contains a set of standard plugs, which allow to implement the translation from sequence diagrams to Büchi automata and from state machines to Promela code. Moreover, this component contains the new plug `XML Input` and will contain the others.

The plugin SA of CHARMY will allow the introduction of new features. We are currently adapting the existing tools, COWSUITE, TESTOR and QUARS, as plugins to insert them in CHARMY, so as to have a complete framework able to specify and analyze a software architecture along its life-cycle process. Indeed, the tools integration requires a certain effort, since existent tools (QUARS written in Tcl/Tk and COWSUITE in Visual Basic) need to be ported to Java.

5 Conclusions and Future Work

As rightly claimed in the QoSA 2005 CFP, progress in software quality research is to be expected by joining research efforts of several groups and from disciplines which have proceeded separately so far. Our modest proposal is meant exactly as a little step towards this ambitious vision, by proposing the combination of a tool for requirements analysis, a tool for SA model-checking and model-based testing, and a tool for test planning and derivation based on a UML design.

In future work we wish to apply the entire analysis process to a selected case study, in order to improve the integration analysis and empirically evaluate main advantages and limitations. Regarding tool support, we plan to port existent tools inside the CHARMY plugin architecture.

References

1. CHARMY Project. Charmy Web Site. http://www.di.univaq.it/charmy, 2004.
2. TOGAF 8: The Open Group Architecture Framework. `http://www.opengroup.org/architecture/togaf/`, 2005.
3. STRAW '03: Second Int. Workshop From Software Requirements to Architectures, May 09, 2003, Portland, Oregon, USA.
4. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–249, July 1997.
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. M. Autili, P. Inverardi, and P. Pelliccione. A graphical scenario-based notation for automatically specifying temporal properties. Technical report, Department of Computer Science, University of L'Aquila, 2005.

7. F. Basanieri and A. Bertolino. A Practical Approach to UML-based Derivation of Integration Tests. In *Proceeding of QWE2000, Bruxelles*, November 20-24.
8. F. Basanieri, A. Bertolino, and E. Marchetti. The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects. In *Proceeding of UML 02, LNCS 2460, Dresden, Germany*, pages p. 383–397.
9. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, second edition.* SEI Series in Software Engineering. Addison-Wesley Professional, 2003.
10. L. Bass and R. Kazman. Architecture-Based Development. Technical report, Carnegie Mellon, Software Engineering Institute, CMU/SEI-99-TR-007, 1999.
11. M. Bernardo and P. Inverardi. *Formal Methods for Software Architectures, Tutorial book on Software Architectures and Formal Methods.* SFM-03:SA Lectures, LNCS 2804, 2003.
12. R. J. Bril, R. L. Krikhaar, and A. Postma. Architectural Support in Industry: a reflection using C-POSH. *Journal of Software Maintenance and Evolution*, 2005.
13. A. Bucchiarone, H. Muccini, P. Pelliccione, and P. Pierini. Model-Checking plus Testing: from Software Architecture Analysis to Code Testing. In *Proc. International Testing Methodology workshop*, Lecture Notes in Computer Science, LNCS, vol. 3236, pp. 351 - 365 (2004), October 2004.
14. R. Chatley, S. Eisenbach, J. Kramer, J. Magee, and S. Uchitel. Predictable dynamic plugin systems. In *FASE*, pages 129–143, 2004.
15. D. Compare, P. Inverardi, and A. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. In *Science of Computer Programming*, pages 33(2):101–131, February 1999.
16. V. Cortellessa, A. D. Marco, P. Inverardi, H. Muccini, and P. Pelliccione. Using UML for SA-based Modeling and Analysis. In *Int. Workshop on Software Architecture Description & UML. Hosted at the Seventh International Conference on UML Modeling Languages and Applications*, Lisbon, Portugal, October, 11-15 2004.
17. CowSuite. http://www.isti.cnr.it/researchunits/labs/se-lab/software-tools.html.
18. D. Garlan. Software Architecture. In *Encyclopedia of Software Engineering, John Wiley & Sons*, 2001.
19. S. Gnesi, G. Lami, G. Trentanni, F. Fabbrini, and M. Fusani. An Automatic Tool for the Analysis of Natural Language Requirements. In *Computer Systems Science & Engineering Special issues on Automated Tools for Requirements Engineering*, volume Vol 20 No 1, 2005.
20. O. M. Group. OMG/Unified Modelling Language(UML) V2.0, 2004.
21. P. Grünbacher, A. Egyed, and N. Medvidovic. Reconciling Software Requirements and Architectures with Intermediate Models. *Journal for Software and Systems Modeling (SoSyM)*, accepted for publication, 2004.
22. C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture.* Addison-Wesley, 1998.
23. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, September 2003.
24. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison Wesley, Object Technology Series, 1999.
25. J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *First Working IFIP Conference on Software Architecture, WICSA1*, 1999.
26. N. Medvidovic, P. Grünbacher, A. Egyed, and B. Boehm. Bridging Models across the Software Life-Cycle. *Journal for Software Systems (JSS)*, 68(3):199–215, December 2003.

27. H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. *IEEE Trans. on Software Engineering*, 30(3):160–171, March 2003.
28. G. Mustapic, A. Wall, C. Norstrom, I. Crnkovic, K. Sandstrom, and J. Andersson. Real world influences on software architecture - interviews with industrial system experts. In *Fourth Working IEEE/IFIP Conference on Software Architecture, WICSA 2004*, pages 101–111, June 2004.
29. B. Nuseibeh. Weaving Together Requirements and Architectures. *IEEE Computer*, 34(3):115–117, March 2001.
30. T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. In *Communications of the ACM*, pages pp. 676–686.
31. P. Pelliccione, P. Inverardi, and H. Muccini. Charmy: A framework for Designing and Validating Architectural Specifications. Technical report, Department of Computer Science, University of L'Aquila, May 2005.
32. P. Pelliccione, H. Muccini, A. Bucchiarone, and F. Facchini. Deriving Test Sequences from Model-based Specifications. In *Proc. Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005)*, Lecture Notes in Computer Science, LNCS 3489, pages 267–282, St. Louis, Missouri (USA), 15-21 May, 2005 2005.
33. D. J. Richardson and P. Inverardi. ROSATEA: International Workshop on the Role of Software Architecture in Analysis E(and) Testing. In *ACM SIGSOFT Software Engineering Notes, vol. 24, no. 4,*, July 1999.
34. D. D. Souza and A. C. Wills. Objects, components, and frameworks with UML. The Catalysis approach. Addison-Wesley, 1998.
35. J. Zhao. Software Architecture Slicing. In *Proceedings of the 14th Annual Conference of Japan Society for Software Science and Technology*, 1997.