

# Model-Checking Plus Testing: From Software Architecture Analysis to Code Testing

A. Bucchiarone<sup>1</sup>, H. Muccini<sup>2</sup>, P. Pelliccione<sup>2</sup>, and P. Pierini<sup>1</sup>

<sup>1</sup> Siemens C.N.X. S.p.A., R. & D.  
Strada Statale 17, L'Aquila, Italy  
antonio.bucchiarone@siemens.it  
pierluigi.pierini@siemens.com

<sup>2</sup> Dipartimento di Informatica, Università dell'Aquila  
Via Vetoio 1, 67100 L'Aquila, Italy  
{muccini,pellicci}@di.univaq.it

**Abstract.** Software Model-Checking and Testing are some of the most used techniques to analyze software systems and identify hidden faults. While software model-checking allows for an *exhaustive* and *automatic* analysis of the system expressed through a model, software testing is based on a clever selection of “relevant” test cases, which may be manually or automatically run over the system.

In this paper we analyze how those two analysis techniques may be integrated in a specific context, where a Software Architecture (SA) specification of the system is available, model-checking techniques are used to validate the SA model conformance with respect to selected properties, while testing techniques are used to validate the implementation conformance to the SA model.

The results of this research are applied to an SDH Telecommunication system architecture designed by Siemens CNX.

## 1 Introduction

Testing and model-checking are between the most important techniques applied in practice to detect and fix software faults.

*Software Model-Checking* [7] analyzes concurrent systems behavior with respect to selected properties by specifying the system through abstract modelling languages. Model-checking algorithms offer an *exhaustive* and *automatic* approach to *completely* analyze the system. When errors are found, counterexamples are provided. The main limitations model-checking suffers are that it may only verify systems expressed through state-based machines, it is more limited than theorem proving, and it suffers of the *state explosion* problem. Moreover, model-checking may be difficult to be applied, since it usually requires skills on formal specification languages.

*Software Testing*, instead, refers to the dynamic verification of a system's behavior based on the observation of a selected set of controlled executions, or

test cases [3]. Testing involves several demanding tasks. However, the problem that has received the highest attention in the literature is, by far, test-case selection: in brief, how to identify a suite of test cases (i.e., a finite set of test cases) that is effective in demonstrating that the software behaves as intended, or, otherwise, in evidencing the existing malfunctions. Clearly, a good test suite is the crucial starting point to a successful testing session.

Comparing model-checking with testing analysis techniques, we may identify the following differences: i) while model-checking offers an *exhaustive* check of the system, testing is based on a clever selection of “relevant” test cases; ii) while model-checking is a completely automated process, testing is usually left to the tester experience; iii) while model-checking requires skills on formal methods, testing may not; finally, iv) while model-checking (usually) helps to find bugs in high-level system designs, testing (in its traditional form) identifies bugs in implementation level code.

Considering the strong complementarity between those two worlds, we believe an integration between model-checking and testing may provide an useful tool to test modern complex software systems.

In this paper we analyze how those two analysis techniques may be integrated in a specific context, where a Software Architecture (SA) [24, 11] specification of the system is available, some properties the SA has to respect are identified and the SA specification has been implemented. Model-checking techniques are used to validate the SA model conformance with respect to selected properties, while testing techniques are used to validate the implementation conformance to the SA model. In the context of this research, the model checker validates the SA conformance to functional properties, while the testing approach provides confidence on the implementation fulfillment to its (architectural) specification (i.e., the so called “conformance testing”).

The *advantages* we expect 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 architectural (functional) properties. Moreover, the test case selection phase is driven by the architectural model. In Siemens CNX systems this approach allows to unify some design steps: in fact, differently from current practices, test cases may be derived from an architectural model, which has been previously checked to be compliant to a set of functional requirements thus ensuring a continuous process from the design step to the integration testing phase.

In proving the validity of this idea, we use the model-checker SPIN in conjunction with the CHARMY framework [6], and apply this approach to a Siemens CNX application.

The following Section 2 motivates our research work by describing some existent related work. Section 3 describes our proposal, by outlining how model-checking is currently used to validate properties on an architectural model and how current testing methodologies allow to test the source code conformance with respect to architectural decisions. The approach is applied to a Siemens

CNX telecommunication application in Section 4, by providing preliminary results. Section 5 draws some considerations while Section 6 concludes the paper and outlines future work directions. A list of abbreviations is provided at the end of the paper.

## 2 Motivations and Approach Outline

Many work have been developed in using model-checking to generate test cases [1, 12, 5, 22, 14]. The key idea that connects these two validation techniques is to generate, by using model checker features, counter-examples successively used to derive test cases.

The generation of counter-examples is obtained identifying (from the requirements) the properties that the system must satisfy. Actually, to obtain test cases the focus must be put on negative requirements, thus the model checker can generate counter-examples automatically turned into executable tests. Typically, the properties negation is obtained by using the *mutation* technique [10] where a mutation is a small syntactic change in the state machine or in the property. A mutant model is obtained by a single mutation operator on the original model. The rationale is to generate test cases for the critical parts of the system.

Unfortunately, the automatic test generation process (by applying model-checking) is not mature in the software industry, for several reasons:

- P1* : due to models complexity, the model checker techniques become inapplicable, thus not allowing to identify test cases;
- P2* : even on little examples, the number of generated test cases causes the intractability;
- P3* : as underlined also by [12] one of the drawback is on assuming the existence of the properties as part of the problem specification.

In this paper we propose a solution which covers problems *P1* and *P2* by *combining model-checking and testing at different abstractions level*. Problem *P3* is also taken into account when using the proposed model-checking approach.

As graphically shown in Figure 1, we use model-checking (MC) techniques in order to validate the architectural model conformance with respect to identified functional properties, and a testing approach which selects test cases driven by the architectural models and the model checking results. The combination of SA-level model-checking and SA-based code testing allows to *guarantee that the architecture correctly reflects important requirements and the code conforms to the software architecture specification*.

This approach produces two benefits: by using an *incremental process* which allows to specify the architecture, model-check it and refine critical sub-parts, we are able to punctually identify details of subsystems identified as critical, working always with smaller models. Together with the fact that model checking is applied to an high-level model (the SA), we reduce the complexity of the model to be analyzed, thus limiting the state explosion problem (i.e., *P1*). We

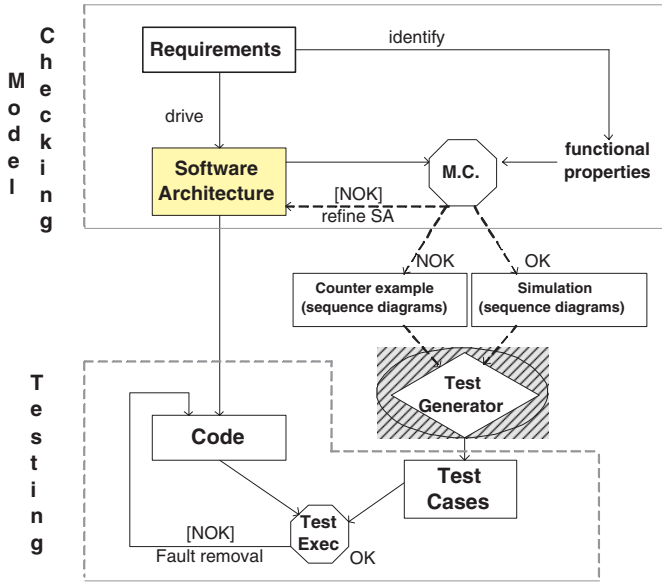


Fig. 1. Approach outline

gain then precision in selecting test-cases maintaining tractable models in terms of dimension and verification time.

### 3 Our Proposal

The approach we are proposing is composed by two correlated activities: validating SA specifications with respect to functional properties through model-checking, and using such models and results to drive an SA-based code testing approach. In the following sections we describe both activities (in Section 3.1 and 3.2) and how to move from the one to the other.

#### 3.1 Validating Architectural Specifications: The CHARMY Approach

In recent years, the study of Software Architecture (SA) [13, 11] has emerged as an autonomous discipline enabling the design and analysis of complex distributed systems through suitable abstractions. SAs support the formal modeling of a system, through components and connectors, allowing for both a *topological* (static) description and a *behavioral* (dynamic) one.

There is not a unique way to specify an SA, and one of the most challenging tasks is on analyzing functional and non functional properties on the selected architecture [19, 11]. Naturally, a good architecture is the one which satisfies at best the system requirements, acting as a bridge between the requirements (the

planned architecture must satisfy) and the implementation code (which has to reflect architectural properties), as advocated in [13].

CHARMY is a framework that, since from the earlier stage of the software development process, aims at *assisting the software architect in designing software architecture and in validating them against expected properties*.

We start specifying an initial, prototypal and even incomplete SA which may identify few (main) components and a set of high-level properties. From this specification we obtain an initial model that can be validated with respect to significant high-level behaviors. This first validation step allows us to gain confidence on the global architectural design and choices, and to identify portions of the SA that might need further and deeper investigation. In fact, when an inconsistency is identified (by producing a counter example), the SA specification may be refined by focussing on relevant system portions. The process is iterated until any critical sub-part is deeply analyzed.

To some extent, our approach is similar to extracting test cases. In eliciting test cases a tester focusses on the critical parts of the system in order to verify his intuitions. In the same way we elicit the properties that represent potentially critical flows and check them on the system model by using model checking techniques.

State machines and scenarios are the source notations for specifying software architectures and behavioral properties they should satisfy, respectively.

CHARMY analyzes these models in order to automatically generate two different outputs, successively used for model-checking purposes. Figure 2 graphically summarizes how the framework works:

- In Step 1, component state machines are automatically translated into a Promela formal prototype. The translation algorithm makes also use of extra information (embedded in the state machine specification) in order to identify which kind of communication is required. SPIN standard checks over Promela may be performed [16];
- In Step 2, scenario specifications (in the form of extended Sequence Diagrams) are automatically translated into Büchi Automata [4]. Such automata describe properties should/should not be verified;
- Finally, in Step 3 the model checker SPIN evaluates the properties validity with respect to the Promela code. If unwanted behaviors are identified, an error is reported.

CHARMY is tool supported and offers a graphical user interface to draw state diagrams and scenarios, a plugin which allows to input existing diagram in the XMI format, and a translation engine to automatically derive Promela code and Büchi Automaton. For more information on the CHARMY project, please refer to [6].

### 3.2 Software Architecture-Based Code Testing

Having a software architecture validated with respect to functional requirements is not enough. In fact, when the architectural information is used to drive the

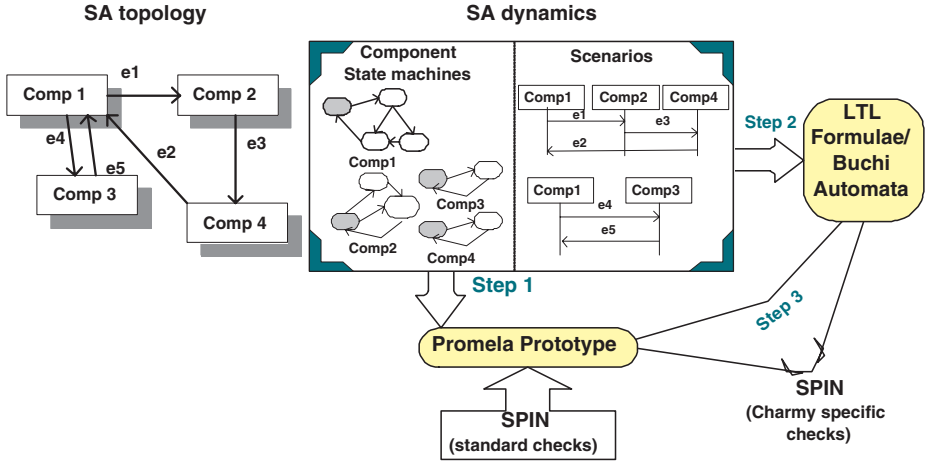


Fig. 2. The CHARMY Framework

source code realization, we need to validate the implementation conformance to the architecture.

SA-based testing is a particular instantiation of specification-based testing, devoted to check that the Implementation Under Test (IUT) fulfills the (architectural) specifications [23, 2]. The abstraction provided by a SA allows testing to take place early and at a higher-level. The SA-based testing activity allows the detection of structural and behavioral problems before they are coded into the implementation. Various approaches have been proposed on testing driven by the architectural specification, as analyzed in [20].

The approach we propose spans the whole spectrum from test derivation down to test execution. Our approach is based on the specification of SA dynamics, which is used to identify useful schemes of interactions between system components, and to select test classes corresponding to relevant architectural behaviors. The goal is to provide a test manager with a systematic method to extract suitable test classes for the higher levels of testing and to refine them into concrete tests at the code level.

In this paper we instantiate/adapt the general SA-based testing framework proposed in [20] (graphically sketched in Figure 3, steps 0-4) to be integrated with model checking techniques. In particular, we here provide guidelines on how the integration happens. Both Figures 1 and 3 will help understanding the idea:

- Step A: the SA is specified through the topology and state diagrams produced in Section 3.1 following the CHARMY approach;
- Step B: when in CHARMY subsystems, considered critical, are identified for further analysis, we intentionally restrict our visibility of the system behaviors by removing “not interesting” behaviors from the original system. To some extent, this approach is similar to extracting test cases. In eliciting test cases a tester focusses on the critical parts of the system in order to verify

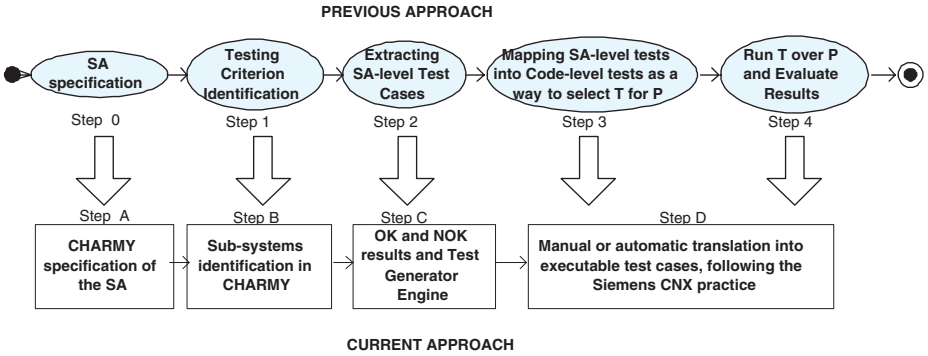
his intuitions. In the same way, we elicit the properties that represent potential flows and check them on the system model by using model checking techniques.

- Step C: when model-checking is run, we may have two different results: the system property of interest is not verified (NOK label in Figure 1), thus counter-examples are found, or it is verified (OK label in Figure 1). In traditional approaches where model-checking is used to identify test cases, counter-examples are generated (by producing properties negation) and automatically turned into tests. In our context, even when a property (i.e., a scenario) is verified on the architectural model (i.e., the Promela model), it may be still of interest as a test case. In fact, since we span from SA checking to code testing, both OK and NOK results are of interest. When a NOK happens, it may depend on two different reasons: the architectural model does not conform to the property (thus we refine the architectural model in order to make it compliant to the functional property (Figure 1)) or we checked the negation of a formula (i.e., NOK means that the property is verified, similarly to what happens in the mutation technique). In some cases, we may still use the counter-example to test if the code conforms to the SA. When an OK result is found, we may simulate the system in order to produce more detailed scenarios which satisfy the property. In fact, we need to remember that the property we wish to verify may be very high-level, by missing many details. Through the simulation process, we may identify more detailed scenarios which verify the high-level property on the architectural (Promela) model. A Test Generator Engine (TGE)(Figure 1) may be used to select only a (relevant) portion of the generated scenarios. The most simple algorithm for a TGE may select just a scenario for each OK property and the NOK scenarios. More precise algorithms may be identified;
- Step D: following the current Siemens CNX practices, such detailed scenarios (called test specifications) may be used to manually or automatically derive executable tests.

## 4 Proposal Application to a Siemens Telecommunication System

In this section we apply the approach just described to a real system developed in Siemens CNX. We initially outline the *Siemens' design process*, thus we describe the *standard telecommunication functional model* in Siemens, as a way to formally describe the SA of telecommunication systems. We thus select, among the set of relevant architecture components we apply the proposed approach, the *Engineering Order Wire (EOW) application*, as an example to show some initial results (in Section 4.1).

In currently applied practices in Siemens CNX, SA and tests design are two parallel activities starting from the system requirements specification. Then two different teams analyse system requirements and divergencies are possible causing errors caught late during the test execution phase. A strong review process



**Fig. 3.** Steps 0-4: General Testing Framework. Steps A-D: Model-Checking plus Testing

(time consuming) is adopted to align all the design phases, eliminating the most part of such type of errors. The application of the MC techniques to validate SA provides a set of simulation results that can be used as a base to define tests. Defining tests in this way unify some design steps. In particular, deriving test traces from MC formal verification of system architecture compliance to the requirements, guarantees consistency of the whole project and, virtually, eliminates the need of the review process.

The standard telecommunication functional model in Siemens is the Synchronous Digital Hierarchy (SDH), a well defined standard by ETSI and ITU-T. Following the standards, a “Functional Model” (for an SDH system) describes the way an equipment accepts, processes and forwards information contained in the transmission signal, defining the internal processes, the internal and external interfaces, the supervision and the performance criteria and relevant recovery actions

The functional model is built around two specific concepts: “network layering”, with a client/server relationship between adjacent layers, and the “atomic functions”, to specify the behaviour of each layer.

A *network layer* is a set of processing and supervision functions described by atomic functions, related to a specific set of transmitted information. Each layer faces to a couple of adjacent layers: a server layer, which provides transport services to the current layer, and a client layer which uses the transport services the current layer provides. Figure 4.a shows some different layers.

Each layer is a group of *atomic functions* (as shown in Figure 4.b). Three basic atomic functions are defined: connection, termination and adaptation functions. They are combined on the basis of combination rules. In addition, application functions (performing specific tasks) should reside at the top of a specific layer as shown for the EOW function in Figure 4.

A *network element* (NE), i.e. a node in the SDH network, is a combination of network layers, thus composed by many atomic functions distributed in different

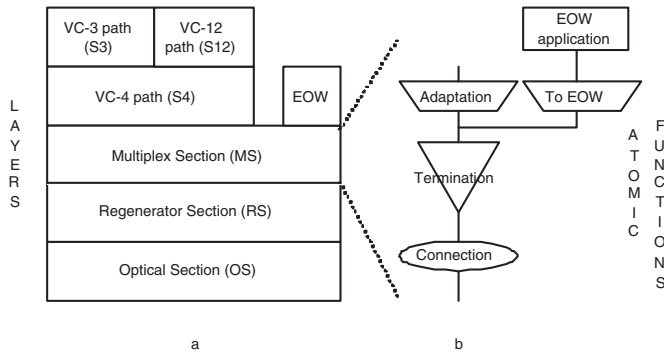


Fig. 4. SDH Layers and Atomic Functions

layers and relevant application functions. Many NE may be interconnected and each atomic function (in a specific layer in a NE) is directly related in a peer-to-peer connection to an identical function, in the same layer but in a different NE. In other words, a direct “virtual network connection” between the two relevant mate functions is created by the transport services.

EOW is an application on top of a specific SDH layer. The EOW application supports a telephone link between multiple NEs by using dedicated voice link channels defined on the SDH frame. An *EOW node* is represented by the EOW application implemented by a SDH NE. An *EOW network* is an overlay network over the SDH network (using the previously mentioned voice link channel). The EOW network is a switch circuit interconnecting several EOW nodes.

An EOW network defines a *Conference*. An EOW node may participate on an EOW conference by being connected to an EOW network by means of EOW ports. An operator (sometimes called subscriber) interfaces the EOW functionalities by means of a handset device. The EOW procedures allow an operator to: i) make a call dialling a selective number, ii) receive a call, iii) enter a conference (with the special number-sign key) when a call is already in progress, and iv) exit the conference (cleanly terminate a call).

#### 4.1 Properties Verification with CHARMY and Test Specification Selection: Initial Results

Given the EOW application and the SDH telecommunication functional model, we initially identified some of the functional properties the EOW application should conform to, and in parallel, we defined an architectural model.

Based on the EOW specification, we identified four functional requirements the system has to satisfy:

- Requirement 1 and 2: when an operator *makes a call* dialling a selective number, the target operator must *receive the call*.

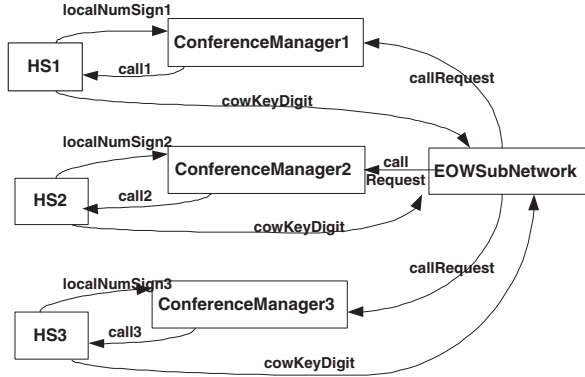


Fig. 5. EOW Software Architecture

- Requirement 3: it must be possible to *enter a busy conference* (with the special number-sign key) when a call is already in progress.
- Requirement 4: It must be always possible to *exit to the conference* (cleanly terminate a call).

We also abstracted away many details of an SDH network and EOW application, and produced an architectural model where each network element is abstracted into a couple of architectural components (the `ConferenceManager` (CM) and the `Handset` (HS) components) while a connector is used to simulate the network connections (the `EOWSubNetwork`). A network model, composed by several network elements, is then abstracted as a set of component instances which may communicate through a connector.

The CM interfaces the EOW network, through ports, by sensing embedded signalling to capture incoming calls and taking into account the conference status (free: when no subscribers are in conference; busy: when calls are in progress). The HS component interfaces and manages the external physical device used by the subscriber. It senses for hook status and key digit pressed and evolves through different internal states to cleanly manage the outgoing and incoming call sequences and signalling taking in care the speech channel connection to the conference. In the following, we take into consideration an EOW network, virtualized by means of a connector component (the `EOWSubNetwork`), with three node instances. Figure 5 shows the EOW architectural configuration of interest. We opted for a configuration with three node instances since this is the minimum configuration required to validate any of the system properties.

Given the architectural specification in Figure 5, the four requirements/properties to be proven have been formalized in the form of CHARMY scenarios. The notation used in Figure 6 is the CHARMY notation for modeling scenarios where messages are typed with prefix `e:` or `r:`. Notation details may be found in [17].

- Requirement 1 and 2: if the HS1 component dials the number to call the HS2 component (`eowKeyDigit`), then component `EOWSubNetwork` must send

a *callRequest* message to the *ConferenceManager2* component that must send *call2* to the *HS2* component.

- Requirement 3: if the *HS3* component makes an *offHook* then it must send the *localNumSign3* to the *ConferenceManager3* to enter the conference.
- Requirement 4: if *HS1*, *HS2* and *HS3* components terminate the call in progress, they must send the *localNumSign* and perform the internal operation *onHook*.

By using the *CHARMY* tool, the three scenarios have been automatically translated into Büchi Automata while the architectural state machine model has been translated into Promela code. *SPIN* has been run and no errors have been detected. This initial analysis (realizing the top part of Figure 1) gives us the assurance that the identified architectural model correctly reflects the three expected behaviors.

In the following of this section, we describe how test specifications may be extracted (thus implementing the mid portion of Figure 1) through simulation.

We thus simulate the Promela prototype obtained from the *CHARMY* tool, through the use of the *SPIN* interactive simulation feature. This feature allows to simulate the Promela model, starting from the model initial state, and selecting a branch at each nondeterministic step. We thus started simulating the Promela prototype by reproducing the properties scenarios previously presented. Through interactive simulation, we are able to get a more detailed description of each scenario, with possible sub-scenarios.

Applying the interactive simulation on the *EOW* model, driven by the first property (Requirements 1 and 2 in Figure 6), we obtained the following test specifications:

```
Config1 ->

[onHook1,
offHook1 -> onHook1] ->

offHook1 ->

[ _ ,
(cbusy==true) -> onHook1 -> offHook1,
(cbusy==false) -> timeout1 -> onHook1 -> offHook1] ->

(cbusy==false) -> eowKeyDigit(...,.) -> callRequest2 ->

*[Config2 -> offHook2 -> onHook2 -> offHook2 -> (cbusy==false),
Config2 -> offHook2 -> onHook2 -> offHook2 -> (cbusy==true),
Config2 -> offHook2 -> onHook2, Config2 -> onHook2 -> offHook2 ->
(cbusy==false), Config2 -> onHook2 -> offHook2 -> (cbusy==true),
Config2 -> offHook2] ->

call2.
```

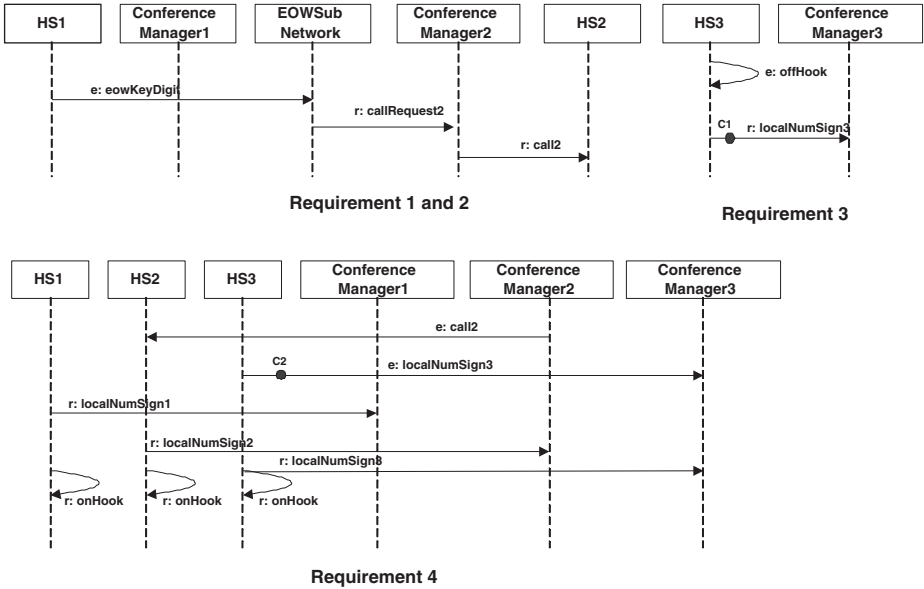


Fig. 6. Properties

This textual description specifies many detailed scenarios (i.e., test specifications) where “ $a \rightarrow b$ ” means  $a$  followed by  $b$ , “[ $a, b$ ]” means  $a$  XOR  $b$ , “[ $-, b$ ]” means  $null$  xor  $b$ , and “ $*a$ ” means that  $a$  may happen everytime between the first and the last operation. Naturally, such test specifications are oriented to test no more than the property of interest, hiding any other (irrelevant) interleaving, and making the test specification more precise and concise.

Following Figure 1, a Test Generation engine (to be implemented) will identify a relevant subset of those test specs which will be converted in test cases by the Test Identification and Execution group in Siemens CNX and run on the EOW application. This activity will be analyzed in future work.

In order to produce other relevant test cases, some refinements have been developed to improve the model, taking into account additional functional and non functional requirements like, DTMF embedded signalling, different calling type, EOW port configurations and so on. Such refinements are currently used to analyze the quality and advantages of the CHARMY iterative specification and analysis process, and more mature results will be reported in future work.

## 5 Some Considerations

Differently from other approaches, we shown how model-checking and testing may be combined to span from requirements to SAs and finally to code. In our approach, since model-checking is used only to validate the architectural specification while test cases are executed over the final implementation, the

idea on how to identify test cases is different. While previous approach use model-checking counter-examples to derive test cases, we may use both verified and not verified scenarios.

The *main advantages* our approach exhibits are listed in the following:

- by applying model-checking at the architectural level, we reduce the modeling complexity and the state explosion problem;
- by using CHARMY we may provide an easy to use, practical, approach to model-checking, hiding the modeling complexity;
- the iterative model-checking approach proposed in Section 3.1 allows to identify the most important properties to be validated, to refine the architectural model and finally to identify critical sub-systems and focus on them;
- through interactive simulation we may identify traces of interest for testing the whole system or just a relevant subsystem. This guarantees a bigger synergy among model-checking and testing application. Since models are small enough, interactive simulation is possible;
- the most important advantage, with respect to how testing has been previously done in Siemens CNX, is that test specifications are identified from the architectural model (instead of directly from requirements) thus creating a bigger synergy between software architects and testers, and eliminating the problem of keeping SA specification and Test specification aligned when requirements change.

The proposed approach also suffers *some limitations*: currently, the Test Generator engine shown in Figure 1 is not yet implemented. This means that the simulated traces need to be manually selected, while it could be automated. The generation of executable tests starting from the test specifications in Siemens CNX is a non automated activity. We started cooperating with the Test Identification group in Siemens CNX in order to make rigorous the translation among test specifications and executable tests. Naturally, even if the iterative model-checking approach at the architectural level reduces the state explosion problem, we still need to carefully handle models dimension and complexity.

Some techniques have been proposed in order to create a model of the implemented system (e.g., [9]) and apply model-checking on it. Our perspective is different from those work. We use model-checking to analyze functional properties at an high-level of abstraction, thus reducing the model complexity.

## 6 Conclusions and Future Work

An architectural specification represents the first, high-level, step in the design of a software system. In this paper we have shown how such specification may be incrementally built, how it may be checked for consistency with respect to some selected functional requirements and how it may be used to select some high-level functional tests which may be successively refined in order to produce test cases.

The approach here presented is partially tool supported even if the generation of test cases from test specifications needs to be better investigated and supported in future work.

In future work we will improve the test specification identification, by improving and automating as much as possible the Test Generator Engines as well as the simulation process. We will also investigate an automated way to produce test cases from test specifications. We are also interested in investigating the use of Rational Quality Architect - Real Time (RQA-RT) to validate an architectural specification and to generate test specifications.

## List of Abbreviations

SA [Software Architecture], MC [Model-Checking], IUT [Implementation Under Test], TGE [Test Generator Engine], EOW [Engineering Order Wire], SDH [Synchronous Digital Hierarchy], NE [Network Element], RQA-RT [Rational Quality Architect-Real Time], HS [Handset], CM [Conference Manager]

## References

- [1] Ammann, P., Black, P.: Abstracting formal specifications to generate software tests via model checking. In *Proceedings of the 18th Digital Avionics Systems Conference (DASC99)*, volume 2, page 10.A.6. IEEE, October 1999. **353**
- [2] Bertolino, A., Inverardi, P.: Architecture-based software testing. In *Proc. ISAW96*, October 1996. **356**
- [3] Bertolino, A.: Software Testing, In SWEBOK: Guide to the Software Engineering Body of Knowledge, IEEE. **352**
- [4] Buchi, R.: On a decision method in restricted second order arithmetic. In *Proc. of the International Congress of Logic, Methodology and Philosophy of Science*, pages pp 1–11. Stanford University Press, 1960. **355**
- [5] Callahan, J., Schneider, F., Easterbrook, S.: Automated software testing using modelchecking. In *Proceedings 1996 SPIN Workshop*, Aug. 1996. **353**
- [6] Charmy Project. Charmy web site. <http://www.di.univaq.it/charmym>, March 2004. **352, 355**
- [7] Clarke, E. M., Grumberg, O., Peled, D. A.: Model Checking. The MIT Press, Cambridge, second edition, year 2000. **351**
- [8] Compare, D., Inverardi, P., Pelliccione, P., Sebastiani, A.: Integrating model-checking architectural analysis and validation in a real software life-cycle. In *the 12th International Formal Methods Europe Symposium (FME 2003)*, number 2805 in LNCS, pages 114–132, Pisa, 2003.
- [9] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000. **363**
- [10] DeMillo, R. A., Lipton, R. J., Sayward, F. G.: Hints on test data selection: Help for the practicing programmer. In *IEEE Comp*, 11(4):34-41, 1978. **353**
- [11] *Formal Methods for Software Architectures*. Tutorial book on Software Architectures and formal methods. In SFM-03:SA Lectures, Eds. M. Bernardo and P. Inverardi, LNCS 2804., 2003. **352, 354**

- [12] Gargantini, A., Heitmeyer, C.L.: Using model checking to generate tests from requirements specifications. In *ESEC / SIGSOFT FSE*, pages 146–162, 1999. 353
- [13] Garlan., D.: Software Architecture. *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc. 2001. 354, 355
- [14] Heimdahl, M.P., Rayadurgam, S., Visser, W., Devaraj, G., Gao, J.: Auto-generating test sequences using model checkers: A case study. In *FATES 2003*. 353
- [15] Holzmann, J.G.: The logic of bugs. In *Proc. Foundations of Software Engineering (SIGSOFT 2002/FSE-10)*, 2002.
- [16] Holzmann, G.J.: *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003. 355
- [17] Inverardi, P., Muccini, H., Pelliccione, P.: Charmy: A framework for model based consistency checking. TR., Dept. of Comp. Science, Univ. of L'Aquila, May 2004. 360
- [18] Peterson, I.: *Fatal Defect: Chasing Killer Computer Bugs*. Random House Publisher, 1995.
- [19] Muccini, H.: Software Architecture for Testing, Coordination Models and Views Model Checking. PhD thesis, University of L'Aquila, year 2002. On-line at: <<http://www.HenryMuccini.com/publications.htm>>. 354
- [20] Muccini, H., Bertolino, A., Inverardi, P.: Using Software Architecture for Code Testing. In *IEEE Transactions on Software Engineering*. Vol. 30, Issue N. 3, March 2004, pp. 160-171. 356
- [21] Pnueli, A.: The temporal logic of programs. In *In Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages pp. 46–57, 1977.
- [22] Rayadurgam, S., Heimdahl, M.P.E.: Coverage based test-case generation using model checkers. In *In 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Apr. 2001. 353
- [23] Richardson, D.J., Wolf, A.L.: Software testing at the architectural level. *ISAW-2* in *Joint Proc. of the ACM SIGSOFT '96 Workshops*, pp. 68-71, 1996. 356
- [24] Shaw, M., Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996. 352