Architectural Verification of Black-box Component-Based Systems

Antonia Bertolino[1], Henry Muccini[2], and Andrea Polini[1]

[1] Istituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche
via Moruzzi, 1 – 56124 Pisa, Italy
{antonia.bertolino, andrea.polini}@isti.cnr.it
[2] Dipartimento di Informatica,
University of L'Aquila
Via Vetoio, 1 - L'Aquila, Italy
muccini@di.univaq.it

Abstract. We introduce an original approach, which combines monitoring and model checking techniques into a comprehensive methodology for the architectural verification of Component-based systems. The approach works by first capturing the traces of execution via the instrumented middleware; then, the observed traces are reverse engineered into Message Sequence Charts, which are then checked for compliance to the Component-based Software Architecture, using a model checker. The methodology has been conceived for being applied indifferently for validating the system in house before deployment and for continuous validation in the field following evolution. A case study for the first case is here illustrated.

1 Introduction

Two antithetical approaches which emerge today for the verification of large complex distributed systems are *model-based* and *monitoring*. These two approaches are generally used in different stages of the software life cycle, and serve different purposes. The former enforces the rigorous derivation of a set of test cases from the system model, and aims at validating before deployment that the implemented system behaviour actually conforms to the modeled one. The latter collects and analyse runtime data during system execution to identify failures and to evaluate critical quality and performance attributes in the field.

In this paper we describe an original approach we are working on, which draws from both model-based verification and monitoring concepts, and combines their respective strengths into a comprehensive methodology for verification in a continuum between in-house and in the field. In particular we describe here how the behaviour of a component assembly is validated against the corresponding software architecture.

A software architecture (SA) provides high-level abstractions for representing the structure, behavior, and key properties of complex software systems [12]. SA-driven development assigns to the SA specification a central role in the software life cycle, both to the phase of design and integration, and to analysis and testing activities. Most methodologies for SA-based analysis and testing generally assume a model-driven approach, in which the SA specification constitutes the reference model and the system is subject to a thorough and accurate validation of the required architectural properties before being deployed. In such paper we describe the validation phase and techniques for the off-line testing.

Advances in SA have greatly contributed to the advent of the *Component-Based paradigm* of development. In fact, the SA specification provides the blueprint for developing systems by properly composing "pieces" of software against it. A Component-Based Software System (CBS) can be roughly considered as an assembly of reusable components, designed to meet the quality attributes identified during the architecting phase [9]. A component can be defined [20] as a unit of composition, with contractually specified interfaces. In a CB approach, a big challenge is posed by the scarce information that is generally available about the components. Various approaches for testing CBSs have been recently proposed spanning over a varying spectrum of assumptions made on the metadata accompanying the component, which can be merely in form of pre- and post-conditions, or even as detailed as state machines. However, such a paradigm needs also to assume there is a time for validation before deployment; with the fast and ceaseless increase of systems complexity and pervasiveness, and the consequent emergence of dynamic global SAs, such a model needs to be revised.

In antithesis to a proactive approach to testing such as model driven, several proposal for *monitoring*, or passive testing, are today spreading. Referred to with differing terminology, such as "monitoring", "tracing", and similar, what this approach to verification foresees is to observe the system during execution and to profile the obtained traces with different purposes. Hence, while in model-based testing the system must be stimulated so to reproduce some predefined behaviour, in monitoring the actual behaviour is observed and a posteriori analysed to see whether this conforms to desired properties. This prevents the burden of reproducing preselected test sequences as in model driven testing; we adopt the model-based approach in that we derive some architectural properties from the specification that we want to verify, and we verify the collected traces against these properties. In particular, for the latter purpose, we apply model-checking techniques, by which we check that the CBS derived traces conforms with the expected event sequences in the CBSA model.

Our medium-term objective is to define an approach that can be equally applied to off-line and on-line validation of CBS systems. With off-line testing we refer to the test of the assembly before its deplyment in the final environment and real usage. This means that in the case of off-line validation the system must be stimulated with test cases appositely derived. Moreover in such case it will make sense to derive the test cases from the same models used for the following verification steps. On-line validation is meant to verify the behaviour of the system during its normal execution. In such case a particularly interesting goal of the approach is to ensure that the "core" of the implemented system fulfills the SA expectations even when new plugin component are attached to the system, as figuratively illustrated in Figure 1.

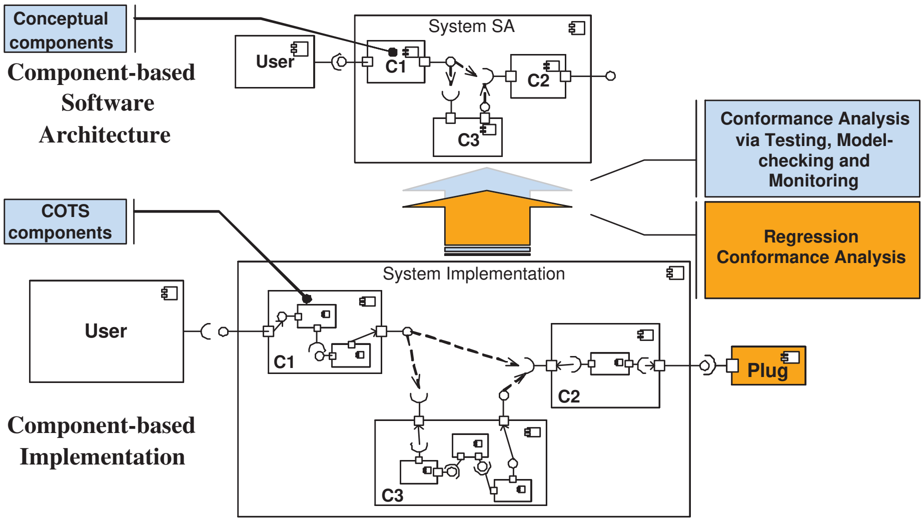

Fig. 1. Architectural Verification of Component-based Systems

In the next two sections we provide an overview of the proposed approach, and of related work. Then, in Sections 4 and 5 we describe the monitoring and model-checking steps, and in Section 6 we discuss the application of the approach to a case study. Finally some conclusions of results reached so far are drawn in Section 7.

2 Related Work

As discussed in the introduction of this paper, our approach wants to integrate, in a novel approach, (model-based) verification applied at the architecture level, monitoring applied during system execution and model-checking techniques covering implementation and software architectures. Much research has been conducted in these areas, and main results are briefly surveyed in this section.

Monitoring and Black-Box Monitoring: Due to the complexity of understanding and configuring modern complex systems, several different approaches to monitor their functioning on-line have been recently proposed. For a recent comprehensive assessment of strategies and testing opportunities for profiling deployed software we refer to [10], while for a quite interesting approach that shares many of the problems and goals with our approach we mention [1]. In the latter, the authors however adopt a different solution, since they instrument the architectural description, and not the middleware. Moreover, they require the developers to define a set of rules used to analyse the traces. Another interesting approach to derive execution traces using Aspect-Oriented Programming is presented in [15]. However even in this case no analysis technique is proposed.

SA-based Model-Checking: Software Model-Checking [8] analyzes concurrent systems behavior with respect to selected properties by specifying the system through abstract modeling languages. Model-checking algorithms offer an *exhaustive* and *automatic* approach to *completely* analyze the system. When errors are found, counterexamples are provided. Initial approaches for model-checking at the architecture level have been provided by the Wright architectural language [5] and the Tracta approach [16]. More recently, Fujaba [4], Æmilia [6], and CHARMY [19] have been proposed. Fujaba is an approach tool supported for real-time model-checking of component-based systems: the system structure is modeled through UML component diagrams, the real-time behavior is modeled by means of real-time statecharts (an extension to UML state diagrams), properties are specified in TCTL and the UPPAAL model-checker is used as the real-time model checker engine. Æmilia is an architectural description language based on the stochastic process algebra EMPAgr: initially introduced for performance analysis, it permits to apply symbolic model-checking. TwoTowers 5.1 is a software tool for the verification of Æmilia specifications. CHARMY [19] is our proposal to model-check software architecture compliance to certain functional temporal properties. The software architecture is specified according to the CHARMY UML-based specification of software architecture. More details will be provided in Section 5.

Integration of analysis techniques: Integration of analysis techniques is a topic which is recently receiving some attention in the software engineering community (e.g., [3]). In [17] the authors integrate testing and monitoring activities, both applied over component-based systems. While testing is used to collect information on components interaction, monitoring is successively employed to identify anomalous interactions when components are added or modified in the original system. More related to some of the authors experience [7], we recently integrated model-checking and testing activities during the life-cycle, where model-checking techniques have been used to validate the SA model conformance with respect to selected properties, while testing techniques have been utilized to validate the implementation conformance to the SA model.

3 Approach Overview

A big synergy relates CB development and SA (the latter being the model that should lead the assembly of a set of components to form the required system): when developing components, our focus is on identifying reusable entities, with well defined interfaces and proved quality. When building component-based systems (CBS), we move our focus to assemble the components so to build a highquality system. When modeling the software architecture of a component-based system (CBSA), our goal is to provide a high-level blueprint on how real components are supposed to be assembled (according to styles and patterns, constraints, and rules).

Therefore a CBSA specification plays a major role in validating the quality of the assembly (even before the CBS components are developed or bought). The main objective of the approach we propose is to verify the coordination properties of components which are part of a CBS, against the specified CBSA. The verification process we propose is composed by different steps, where:

CBSA Specification for Analysis: The CBSA of the system under analysis is specified in terms of a structural model (which describes components, connectors, interfaces, and ports) and a behavioral model (which specifies the internal expected behavior and coordination of the CBS components).

Testing Input Selection: In case of off-line testing how do we select the test cases to be executed? As the basis assumption of this approach is that a detailed model of the real component is not available (assuming the component is off the shelf), we use the only information that is anyhow available (it may be in various forms), i.e. the expected Input/Output functions of the components that can be retrieved from the architecture specification.

Monitoring Black-box Component-Based Systems: We execute the implementation on the selected test cases, by observing the traces of execution via monitoring techniques. Traditionally, monitoring techniques are realized by instrumenting the component code in order to capture desired information from execution. However, since components can be black box with no available code, we cannot instrument the component in traditional ways. For this purposes, we adopted a middleware instrumentation.

Model-Checking CBS conformance to CBSA: The execution traces are used to check the CBS conformance to the CBSA specification. We remind here that while the CBSA specification described the intended/expected system usages, the CBS execution traces (obtained via monitoring) represents how the implemented CBS works. Model-checking techniques are then utilized to compare expected and real behavior.

In this paper we will focus our attention on the Monitoring and Model-Checking activities, while future work will investigate how such technologies can be used for verifying dynamically evolving CBSs. For this purpose, we will distinguish, as showed in Figure 1, among *architectural components* (the abstract ideal components of the SA, specified by their interfaces and their expected model of interactions), *concrete components* (the components of the implemented CBS, obtained by refining SA components or by adapting existing components), and *real components* (the building blocks of the concrete components, and can be, for instance, Commercial-off the Shelf – COTS – components). Concrete and real components may coincide, or a concrete component could be obtained by the assembly or wrapping of real components.

4 Monitoring Black-box CB Systems

Monitoring is the activity intended to collect and check information about specific properties and behavior of a system during its real execution. Different elements and issues can be recognised in a monitoring setting:

1. the **monitored system** is the software whose behaviour and properties are of interest. The functionalities provided by such system are those that are in general relevant for an external user.
2. the **monitoring system** is the software that collects specific information on the monitored system. In general the monitoring system accepts as input the information to be observed and collected, and the constraints on the behaviour/properties of the system that must be respected.
3. If a violation is detected, the monitoring system communicates the anomaly to a **controlling component**. The latter, that in some cases can also be a human agent, should put in place a set of actions to manage the anomalous condition and restore the system to a correct state (note this step is out of the scope for the proposed approach).

Observability of the monitored system is certainly the basic element constraining what can be monitored: it refers to what an external observer can notice about the system/component behaviour and properties evolution. In the fortunate case that the system has been developed having already in mind what is necessary to check, the set of information that can be observed generally encloses the set of what is necessary to monitor, but unfortunately this is not the general case.

Monitoring becomes particularly tough when the source code of a software component cannot be directly accessed and modified, as is the rule when black-box reuse of software components externally acquired is considered. In such a situation the only information that can be accessed by an external agent are those expressly made available by the component developer. In CB programming this will generally only include the information passed trough the public interface.

Our approach requires to collect specific information for checking that the interactions among the concrete components are actually allowed by the architectural description. Two basic elements are needed to put in place this kind of verification activity:

1. a technique for representing the concrete interactions in a way that will be suitable for the architectural checking step
2. a mechanism for observing component interactions as they happen at run-time

In our approach the execution traces are abstracted to Message Sequence Charts (MSCs) reporting the whole signature for each invoked method. It is worth noting that these diagrams will not report any internal interactions within a *concrete component* implementation, given that its implementation is transparent to the system integrator.

The most obvious mechanism to observe component interactions might seem wrapping each concrete component with code that in some way records all the incoming and outgoing invocations. However, a concurrent behaviour of the wrapped component (e.g., the simultaneous invocation of different methods exposed by the component by at least two different threads of execution, or the internal activation, within the invoked component, of another thread) hinders the identification of the correct relations among incoming and outgoing call to and from the component. An example can aid to understand the problem. In Figure 2 component B invokes (b) component C while this component is already processing an invocation (a) from component A. In this situation a standard Wrapper of the C component could not recognize the thread that gave rise to the invocation "c" towards component D. So when situations such as that represented in the figure happen, it becomes impossible, using only a "wrapping" approach, to correctly associate outgoing invocations with the corresponding incoming ones.

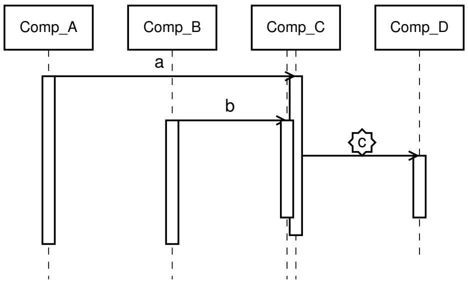

Fig. 2. Wrapping cannot easily manage concurrent invocations

An alternative way to monitoring, the one we chose, is to base such activity on the run-time environment of the application under verification, finding suitable mechanisms to observe the inteactions among the components "following" threads of executions. In a local approach based on Java, for instance, this would mean that we have to base the monitoring on direct interactions with the Java Virtual Machine (JVM) or we have to define an appropriate version of the JVM directly inserting monitoring features on it. In a distributed setting, instead, this would mean to insert the monitoring activity as part of the middleware level and not at the application level as in the case of the wrapping approach. At this point the problem to solve is how can we retrieve the needed information to derive meaningful execution traces. As stated before we can assume the knowledge only of the interfaces to be implemented by the concrete component. However inserting the monitoring "below" the application will permit to increase the observability of the executed application to all the invocations it makes to the middleware. At this point the idea is that the only additional information we need is to observe all the actions and interactions on and among the processes (thread of execution) activated by the execution of the application. So starting with the "main" thread of the application we should observe, behaving as a sort of debugger, when new processes are activated, when they interact or when they stop. At the same time we should be able to record when a process during its execution "hits" one of the methods of the interface implemented by one concrete component. For each process the invocation sequence are stored in different files successively used to recreate the traces that have been actually executed. Considering the scenario in Figure 2, the result of the monitoring will be two different files, the first reporting the sequence of invocations "C.a","D.c" and the second the sequence "C.b", providing the precise information on the call sequences.

The above approach is implemented into a proof-of-concept monitoring tool called Theseus, from the mythological character that used a thread to trace the path to the way out from the minotaur's maze. In the current version Theseus can only monitor the execution of non distributed CBS (implemented using the Java language); we are currently working to add support also for Java Remote Method Invocations.

Theseus implementation is based on the Java Platform Debugger Architecture (JPDA) API (Application Programming Interface). The tool takes as input the `.class` files containing the interfaces defined for the concrete components, then using the API defined in the JDA, it asks to the JVM to notify when methods on this interfaces are invoked. When this happens the JVM is stopped and all the information concerning the invocation are retrievied and stored in the corresponding file; then the machine can then be restarted. The JVM bloking behaviour is consequence of the fact that the API and the JVM have not been explicitly developed for monitoring purpose, and a non stopping behaviour will raise the risk of loosing relevant information given the occurence of successive method calls.

The tool Theseus also records all the invocations made on instances of the `java.lang.Thread` class, that in Java ands in particular in the JVM are used as the abstraction of processes. In such manner the tool can recognize when a Thread interacts with another via a notify, or also when it activates another thread creating and starting it, permitting to manage the issues raised by the presence of concurrency. In Figure 3 the main window of Theseus is shown.

5 Model-Checking CBS conformance to CBSA in TANDEM

As soon as execution traces have been collected after monitoring the component-based system implementation, they have to be checked for consistency to expected architectural behaviors. This validation phase has to identify if and how much the (behavior of) realized system complies to what has been previously specified at the architecture level. In fact, while execution traces denote the real system behavior when submitted to certain inputs, architecture-level behavioral models identifies the expected behavior.

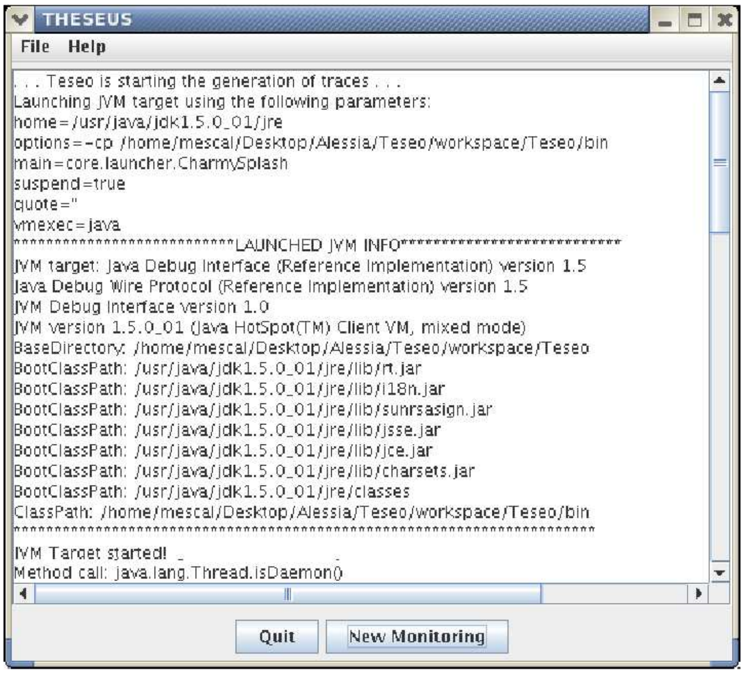

Fig. 3. Theseus start interface

The architectural model-checking approach we take in place here is CHARMY, a model-based approach for architectural checking.

CHARMY[2] enables the **specification** of a software architecture through diagrammatic (UML-based) notations, and the *verification* of the architectural specification conformance with respect to certain temporal properties, representing how architectural elements are supposed to be coordinated. By focussing on CHARMY main features, we have that:

Specification: CHARMY allows the specification of a software architecture by means of both a topological (static) description and a behavioral (dynamic) one [11]. The specification of the SA topology is realized in terms of stereotyped UML 2.0 component diagrams, where components represent abstract computational subsystems and connectors formalize the interactions among components. The internal behavior of each component and the coordination of the interacting components is specified in terms of stereotyped UML 2.0 state machines.

Verification: once the SA specification is available, a translation engine automatically derives from the model-based SA specification, a formal executable prototype in Promela (the specification language of SPIN) [13]. 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.

Figure 4 graphically summarizes how the tool supporting the CHARMY approach works: the CHARMY tool editor allows the graphical specification of the SA topology and behavior and the properties in terms of UML diagrams. In step 1, component state machines are automatically translated into a Promela formal prototype. Once the Promela model is produced SPIN standard checks may be performed. In Step 2, scenario specifications (in the form of extended Sequence Diagrams) are automatically translated into Büchi automata (the automata representation for LTL formulae). Such automata describe properties to be verified. Finally, in Step 3 SPIN evaluates the properties validity with respect to the Promela code. If unwanted behaviors are identified, an error is reported.

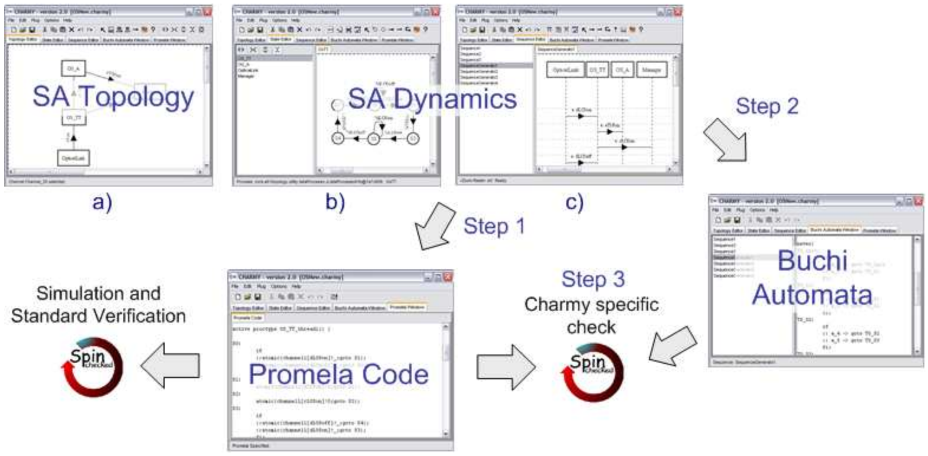

Fig. 4. Tool Support for CHARMY Main Features

6 Applying the Approach to the CHARMY Plugin System

As a case study to experiment the approach we have taken the CHARMY plugin system. We provide an outline of the CHARMY software architecture and its specification (Section 6.1). Then, we identify some monitored traces and show how CHARMY can check some properties over its architectural model (Section 6.2). We conclude this section with some considerations and evaluation of results (Section 6.3).

6.1 The CHARMY Plugin Architecture and its Specification

The CHARMY architecture is composed by two main parts: the CHARMY `Core` and the `Plugin Package`.

The CHARMY `Core` macro-component is composed of the `Data Structure` component, the `Plugin Manager` and `File Manager` which allows the handling of plugins of type editor and file, respectively, the `GUI` which receives stimuli from the system users, and the `Event Handler` which handles all those events generated by plugins. The CHARMY `core` handles the plugin management by specifying: *i*) how a new plug should be implemented, *ii*) how the core system has to recognize the plug and use it, and *iii*) how the core and plug components should interact. Figure 5 graphically summarizes the interfaces to be implemented in order to plug a new component in the system. Details on how to implement and recognize a new plug, and plugs interaction are provided in [14].

The `Plugin Package` contains a set of plugins to specify and analyze software architectures. The *Topology, State*, and *Sequence* editors permit to edit the software architecture topology, the architectural state machines and the scenarios, respectively. The *PSC2BA* and *Promela Translation* plugins allow an automatic translation from sequence diagrams to Büchi automata and from state machines to Promela code. Such translations permit the application of model-checking techniques at the software architecture level. The *TesTor* component allows the generation of architectural test specifications. The *Composit* component allows for compositional analysis of middleware-based SA. For more details, please refer to [19].

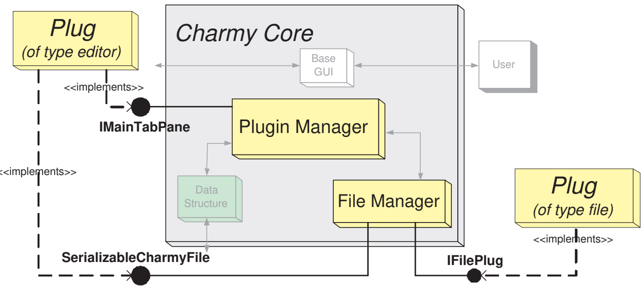

Fig. 5. Plug and Core

By means of the Promela Translation plugin, the CHARMY architectural specification has been automatically translated into a formal executable prototype in Promela, and checked through SPIN standard checks (step 1 in Figure 4). After few modeling and checking iterations, we produced a stable correct (with respect to SPIN checks) architectural specification.

6.2 Validating the CHARMY Implementation with respect to its Architectural Specification

The CHARMY core and plugs implementation has been realized in the last three years at the Computer Science Department, University of L'Aquila. When moving from version 0 to version 1, the tool implementation has been re-structured to make it plugin-based. More recently, when moving from version 1 to the current version 2.0 beta, some minor modifications have been made, while re-thinking some interfaces and adding some utilities. Many plugins have been created and unit tested. Since many of them have been realized thanks to students support, our confidence on the plugin-core integration correctness has been mainly based on beta testing.

We then decided to use the approach as a means to validate the plugin-core integration (i.e., to check if the integration of a new plug in Charmy may violate the Charmy core standard behavior). We submitted the Charmy system implementation (together with its plugins) to three different analysis: *i*) we plugged the Topology Editor component to monitor and check its correct integration into the Charmy system, *ii*) we monitored an initially faulty version of the ExplodePlugin component, in order to obtain an error trail enabling the localization of the fault, and *iii*) we injected a fault on the Topology Editor final version, in order to evaluate the approach ability to precisely localize an expected fault.

Analysis of type i): When running Charmy with the Topology Editor plugged into the system, we can observe via Theseus only a subset of the entire flows of events (since Theseus in its current version can monitor only interfaces). In particular, it can collect information on how the Charmy Core loads the plugin, how the copy/cut/undo operations are performed, how the resulting topology diagram is stored and closed.

Figure 6 shows one of the typical traces produced when monitoring the Topology Editor/Core integration. As we can see from the figure, the scenario is quite complex, since it records a quite long list of events. In any case, all the internal (to the plug) operations are not observed, since they are implemented without any specific interface.

Indeed, analyzing the scenario by hand, will be extremely expensive. Instead, the scenario has been drawn into Charmy, automatically translated into a Büchi automaton representation, and the SPIN verification has been run. No errors have been detected.

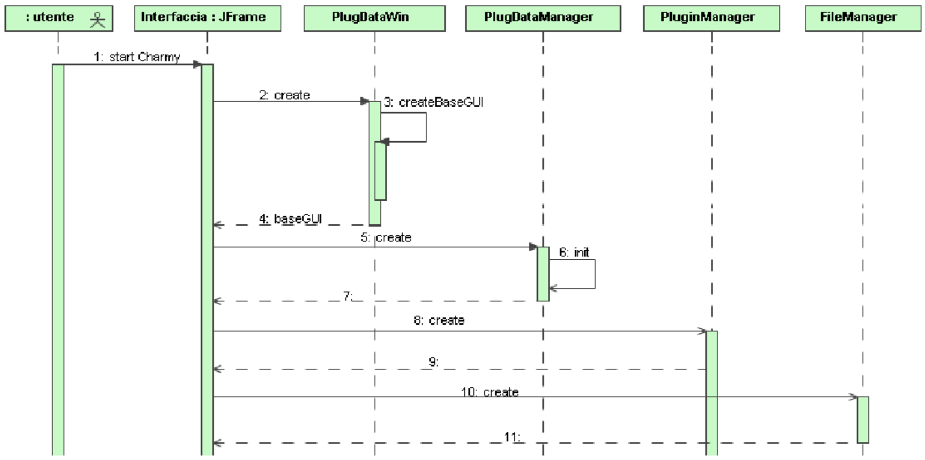

Fig. 6. Monitoring the Topology Editor interaction with the Charmy Core

Analysis of type ii): In this second scenario, we selected the ExplodePlugin plugin we knew had a bug. This plugin has been realized for testing purposes. It creates incrementally a multitude of plugins to be loaded in Charmy. Our goal was to evaluate how much Charmy scales and performs when a multitude of plugins are plugged into the Charmy Core. We knew in advance the plugin was buggy, but we did not know where the fault was localized.

When running the Theseus monitoring activity, many scenarios have been recorded. For sake of space, we do not report them here. When applying the Charmy model-checking approach to this scenario, the expected error trail has been detected, highlighting the location of the first undesired (with respect to the architectural model) interaction.

Analysis of type iii): Ultimately, we tested the monitoring + model-checking ability to precisely localize faults. While analysis *ii*) permits to localize the fault, analysis *iii*) allows approach users to check how much the localized fault corresponds to the expected (injected) one. Then, this activity represents a validation of the approach precision in localizing faults.

A fault has been injected in the Topology Editor component. When loading this faulty version into Charmy, and after Theseus monitoring, many traces have been collected (not reported for space limits). When running the SPIN verification feature, an error trail has been raised, indicating the unwanted behavior.

Indeed, we cannot expect the approach to identify all possible injected faults. Only architectural coordination faults can be detected. Moreover, the fault localization ability of the approach strongly depends on the Theseus ability to monitor events. Since in its current version Theseus monitors only such services implemented via interfaces, the approach localizes the first architectural interaction affected by the injected fault.

An interesting discover we made thanks to the application of the approach has been that the students introduced many architectural mismatches using direct reference to classes instead of using the defined interfaces. By executing the Charmy tool in a monitored environment we did not notice any strange behaviour; but subsequently the traces collected by Theseus did not expose all the interactions we expected. Indeed we could see all the initial invocations made by the core components to correctly initialise all the installed plug-ins, but after these invocations we could observe less invocations with respect to the expected ones. Initially we thought there was an error in the Theseus implementation previously tested only on small case studies. After having analysed the code of Theseus without finding a solution to the problem we tried to investigate whithin the Charmy code. During this investigation we discovered that often students preferred to use direct casting on objects or however direct reference to external classes. So we were faced with an implementation containing many architectural mismatches affecting the possibility of substituting components with new versions implemented by different classes. Clearly this architectural mismatches strongly affected our capability of analysis since Theseus is only able to trace invocations occurring through interfaces. Indeed this kind of mismatches could be discovered through static analysis, assuming any tool is available for code-level checking of architectural properties.

6.3 Considerations and Evaluation of Results

With the use of the approach, we can test the system coordination, by integrating monitoring and model-checking techniques. If the system implementation behaves accordingly to the system specification, an ok message is raised. Otherwise, an error trace identifies where the execution trace differs from the expected one.

When an anomalous behavior is known, the approach allows the detection and localization of known but unwanted faults. It acts as a sort of debugger, identifying which components are behaving incorrectly when integrated. In both cases, the ability of the approach to detect and precisely localize a fault can be evaluated via fault injection. Moreover, with respect to traditional specification-based testing techniques, the approach permits to analyze the implementation conformance to external input/output (as in traditional IOLTS-based testing) and also the implementation conformance to the entire architectural trace (by model-checking the execution scenarios compliance to the architectural model).

According to the experiment conducted over the Charmy system, we here propose some initial considerations and evaluation of results (being conscious more deeper investigation is required for providing more informative insights):

Automation: so far, a quite relevant part of the approach is supported by tools. The architectural specification is made through the Charmy editors. Test cases generation can be realized according to [18]. The monitoring activity has been supported by the Theseus tool. The model-checking activity is realized through the integration of Charmy features and the model-checking engine SPIN;

Costs: monitoring black box components is generally an invasive and expensive activity. Since Theseus requires to stop the JVM at each time an information needs to be collected, the monitoring activity for a complex execution may required up to ten minutes. However, we are working on modifying the JVM in order to reduce the monitoring time. Regarding the model-checking activity, time effort is quite limited, since properties of interest "p" are submitted to an existential check (i.e., check if "p" exists in the system model "m").

Scalability: even if more experimental results are needed for a finer evaluation, the approach seems to be able to scale to larger systems (assuming an improvement on the monitoring activity performance).

7 Conclusions and Future Work

We have presented a novel approach to the SA-driven verification of CBSs, in that it combines the strengths of either approaches, trying to overcome the inherent difficulty of reproducing a predefined sequence of events of model-based testing, but enriching the power of monitoring with a rigorous model-check stage of the obtained traces. The goal of the approach is to verify that some important SA properties are indeed satisfied by the implemented CBS, and continue to be so even after evolution. The approach is in fact conceived as a comprehensive methodology to be used without interruption both for in-house validation (off-line testing), and for continuous verification in use (on-line testing). The goal we pursue is quite ambitious, and of course we are not yet done, but several pieces of the approach are already implemented. In this paper we have already presented some promising results by applying the approach for off-line testing of the CHARMY architecture. Even though the approach proposed can show appealing opportunity, its usage should be carefully evaluated, particularly for the case of on-line testing. Monitoring being executed concurrently with the application strongly affects performance. The monitoring technique we propose seems to be particularly powerfull but if not supported by adequate tools particularly expensive. In the current implementation performance can be reduced up to ten times when many threads are started. We are currently investigating some tool improvements in order to make it applicable for the run-time monitoring of multi-threades applications.

8 Acknowledgements

The authors wish to thank Alessia Bardi and Ezio Di Nisio for their important contribution to the implementation of the tools enabling the proposed approach.

This work is partially supported by the PLASTIC Project (EU FP6 STREP n.26955): Providing Lightweight and Adaptable Service Technology for pervasive Information and Communication. http://www.ist-plastic.org.

References

1. An Approach for Tracing and Understanding Asynchronous Systems. ISR Tech. Report UCI-ISR-02-7, 2002.
2. CHARMY Project. Charmy Web Site. http://www.di.univaq.it/charmy, 2004.
3. *2nd International Workshop on "Rapid Integration of Software Engineering techniques" (RISE 2005)*, Heraklion Crete, GREECE, September 2005. LNCS.
4. Fujaba Project. http://wwwcs.uni-paderborn.de/cs/fujaba/publications/index.html, 2005. U.Paderborn, Sw Eng. Group.
5. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–249, July 1997.
6. M. Bernardo, L. Donatiello, and P. Ciancarini. *Performance Evaluation of Complex Systems: Techniques and Tools*, chapter Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language. LNCS, 2459:236-260, September 2002.
7. 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*, LNCS, vol. 3236, pp. 351 - 365 (2004), October 2004.

8. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The *MIT Press*, Cambridge, second edition, 2000.
9. I. Crnkovic and M. Larsson, editors. *Building Reliable Component-based Software Systems*. Artech House, July 2002.
10. S. Elbaum and M. Diep. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. *IEEE Trans. on Software Engineering*, 31(8):1–16, August 2005.
11. D. Garlan. Software Architecture: a Roadmap. In *ACM ICSE 2000, The Future of Software Engineering*, pages 91–101. A. Finkelstein, 2000.
12. D. Garlan. Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events. In *Formal Methods for Software Architectures*, pages 1–24. LNCS, 2804, 2003.
13. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003.
14. P. Inverardi, H. Muccini, and P. Pelliccione. CHARMY: An Extensible Tool for Architectural Analysis. In *ACM Proc. European Software Engineering Conference/the Foundations of Software Engineering (ESEC/FSE)*, September 2005.
15. Kimmo Kiviluoma, Johannes Koskinen, and Tommi Mikkonen. Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'06)*, pages 181–190, July 17-20 2006. Portland, Maine, USA.
16. J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *I Working IFIP Conf. Sw Architecture, WICSA1*, 1999.
17. Leonardo Mariani and Mauro Pezze'. Behavior Capture and Test: Automated Analysis of Component Integration. In IEEE Computer Society, editor, *In 10th IEEE International Conference on Engineering of Complex Computer Systems*, Shangai (China), 16-20 June 2005.
18. H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. *IEEE Trans. on Software Engineering*, 30(3):160–171, March 2003.
19. P. Pelliccione. *CHARMY: A framework for Software Architecture Specification and Analysis*. PhD thesis, Computer Science Dept., U. L'Aquila, May 2005.
20. C. Szyperski. *Component Software. Beyond Object Oriented Programming*. Addison Wesley, 1998.