CHARMY: A Plugin-based Tool for Architectural Analysis

H. Muccini, P. Pelliccione, and M. Stoduto

Dipartimento di Informatica, Universitá dell'Aquila,
Via Vetoio 1, 67100 L'Aquila, Italy
[muccini,pellicci]@di.univaq.it

Abstract. While at the code level, many tools have been proposed so far for software analysis, very limited support has been provided for analyzing Software Architectures. In this paper, we present CHARMY, an architectural tool for SA-based analysis which allows to validate the conformance of an SA specification with respect to expected properties. We describe the tool's features, how it may be used and its benefits.

1 Introduction

With the advent and use of software specifications, source code no longer has to be the single source for analysis: formal, informal and model-based specifications can be used for this purpose [21, 9]. Even if much work has been done in introducing new formalisms and languages which may allow for analysis at the specification level, unfortunately very few has been done in the direction of analysis automation.

In particular, by focussing on Software Architecture (SA) [10] level specifications, a lot of work has been done in introducing formal [17] and informal [18] specification languages. SA specifications have been integrated by both researchers in industry and academia in their software development processes (e.g. [11, 1]) and thus they have recognized how expensive (even if useful) may be to deal with SAs. The effort is justified only if the SA artifacts are extensively used for multiple purposes. In particular, industries are currently increasing their interests in *analyzing and validating architectural choices*, both behavioral and quantitative. SA-based analysis methods have been introduced to provide several value-added benefits, such as system deadlock detection, performance analysis, component validation and much more (as reported in [9]). However, very few tools have been proposed to support SA-level analysis, many of them are not anymore supported or difficult to be introduced in an industrial context. Thus, *how to automate the SA-based analysis process in a way useful for current industrial needs* is one of the topics which require major attention.

This paper introduces CHARMY, a tool which allows to specify the SA of a given software system through diagrammatic, UML-based notations, and to *validate the architectural specification conformance to certain functional requirements*. CHARMY offers a graphical user interface to draw state diagrams and

scenarios, a translation engine to automatically derive Promela code [12] and Büchi Automaton [3] which may be model-checked through the SPIN model checker [12], and a feature which allows to output information in the XMI format. Moreover, the tool has been organized as a plugin framework to simply the introduction of new features and to help the integration with other existing analysis tools.

The tool main benefits are that it is UML-based (thus easily integrable in industrial contexts), it automatically produces a formal prototype of the SA and model-check it with SPIN without requiring formal languages skills, it is extensible, due to its plugin architecture.

The remaining of this paper is organized as follows: Section 2 introduces the CHARMY tool with its features, requirements and architecture. Section 3 analyzes some related work on tools for SA-based functional analysis. Section 4 provides some considerations while Section 5 describes future work.

2 CHARMY: A Tool for Model-Based Validation and Verification

CHARMY [15, 6] is an open source tool 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.* In the following we describe the CHARMY features (Section 2.1), the requirements (Section 2.2) and finally the tool architecture (Section 2.3).

2.1 CHARMY Features

CHARMY is composed of a *graphical editor* which allows to specify the software architecture, a *translator utility* which converts diagrams in the languages comprehensible by the model checking engine (at present the model-checker SPIN [12]), a *repository* which stores models for subsequent reuse.

The *graphical editor* supports the specification of a SA, allowing for both a topological (static) description and a behavioral (dynamic) one [10]. The *topology editor* allows to specify the SA topology in terms of components, connectors and relationships among them, where components represent abstract computational subsystems and connectors formalize the interactions among components (Figure 1.a). The *thread editor* allows to specify the internal behavior of each component in terms of state machines (one for each component's thread)(Figure 1.b). Even if many formal specification languages have been proposed to specify SAs, we use an UML-based notation (stereotyped class diagrams for the topology and state diagrams for the behavior) which is the most common used tool in current industrial practice.

Whenever the SA specification is available, a *translator utility* is used to obtain from the model-based SA specification, a formal executable prototype in Promela (the specification language of SPIN)(Figure 1.step1). When the Promela

code is generated, we can use the SPIN standard features to find, for example, deadlocks or parts of states machines that are unreachable (Figure 1.step2).

Note that the effort to have this formal system prototype in Promela is minimal since the translation process is fully automated.

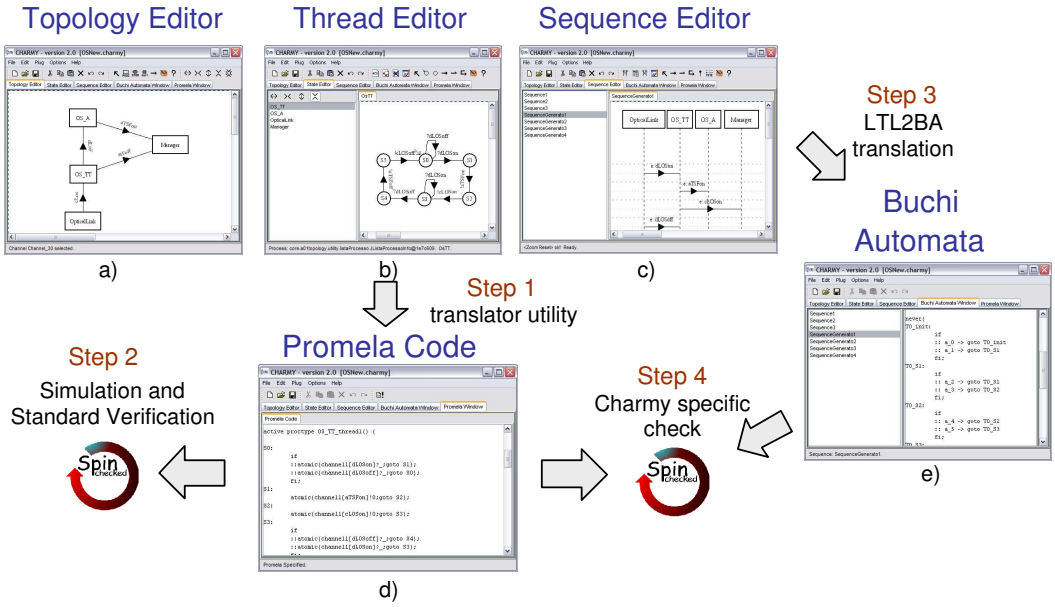

Fig. 1. The Tool

More sophisticated analysis require to specify behaviors the system should satisfy. The typical instrument used is the temporal logic; SPIN makes use of the Linear-time Temporal Logic (LTL) [20] used to produce Büchi automata. However, in an industrial contexts it is unfeasible to write by hand complex LTL formulae, as pointed out by Holzmann in [13]. To this extent, he proposes a tool to write temporal properties in a graphical notation. Differently, we specify behavioral properties using the familiar formalism of sequence diagrams and automatically translate them 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. The graphical editor involved in this step is the *sequence editor* (Figure 1.c), while the translation algorithm is called *LTL2BA* and is detailed in [15] (Figure 1.step3).

The CHARMY tool performs several checks, at the graphical editor level, in order to find specification errors that can be statically discovered: a) in each state diagram it is impossible to introduce two states with the same name; b) each state diagram must contain one and only one initial state; c) for each send (receive) message in a component, the user must insert a receive (send) message

in another component; d) sequence diagrams must contain only messages already inserted into the state diagrams; e) the sender and the receiver for a message must be the same components in the sequence diagrams and in the state diagrams, i.e., a message "m" can be inserted in a sequence diagram between a pair of components only if the sender state machine sends "m" and the receiver state machine receives "m". ; f) messages with same name must have the same number of parameters.

2.2 CHARMY Requirements

The fundamental requirement the tool have to respect is the extensiveness in many directions: i) we want to be able to extend the *kind of analysis* we may perform on a software architecture. For example, we recently extended the theory behind the tool in order to integrate compositional verification techniques for architectural analysis [5] and for CHARMY integration in the standard life-cycle process [8]. ii) We want CHARMY to be *integrated with other existing tools*. We are currently implementing a new feature which allows to describe the software architecture through standard UML tools, by means of an XMI representation, which may be imported from CHARMY. We developed a feature that, exploiting JSpin [16] (a java interface for SPIN), aims at embedding SPIN in CHARMY. iii) Moreover, we want CHARMY to be *open with respect to the engines* used to perform analysis. Since we are not tied down to use the model checker SPIN, we could think to use others model checkers.

2.3 CHARMY Software Architecture

The sentences in the previous section pilot to a plugin architecture, easily extensible by adding new components to the initial core.

The CHARMY tool architecture is shown in Figure 2. Taking a look to the CHARMY `Core` macro-component, 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 software architecture topology, the state machines and the scenarios.

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.

When a new plug requires to be created and integrated into the core system, some aspects need to be carefully analyzed: *i*) how a new plug may be implemented, *ii*) how the core system may recognize the plug and use it, *iii*) how core and plug components may interact.

Figure 3 graphically summarizes some of such aspects.

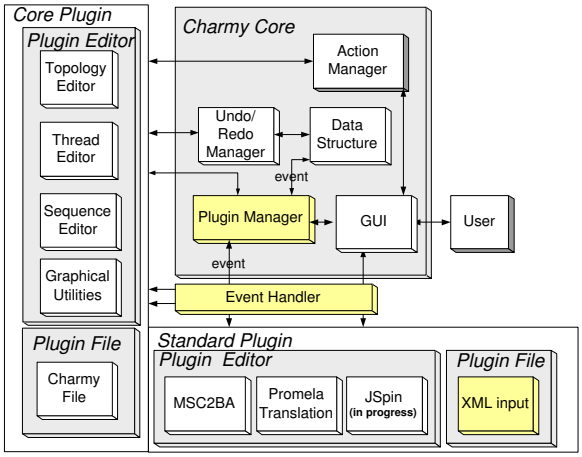

Fig. 2. The CHARMY Plugin Architecture

i) Implementing a new plug: when a new component needs to become a plugin, it has to implement two interfaces: the "IMainTabPane" and the "IFile-Plug". The IMainTabPane interface handles the data information related to the windows. Here we have methods which allow to receive information from the editor components (Topology, Sequence and Thread components). The IFile-Plug interface, instead, needs to be implemented when the plug requires to save or open a file. When a plug is created, it needs to reside in a specific folder, subfolder of the "plugin" dir.

ii) Recognizing the new plug: when a new plug is created and wants to be inserted, the core system needs to be informed about this. The solution we adopted has been to create an .xml file (called plugin.xml, as in Eclipse [19]) which contains all information needed.

iii) Interaction: when a data if modified inside the core system, an event is sent by the `Event Handler` component to the plug. This event informs the plug of which kind of modification has been made over the data (e.g., insert, modify, delete) and sends a clone of the data itself to the plug. A plug, in order to receive the event, has to be registered as a listener of the event itself. The event is sent to all such plugs which are listeners.

3 Related Work

When dealing with tools for functional analysis of software architectures, we may distinguish between proposed tools, still supported tools and tools suitably usable in industrial contexts.

In the first type of tools (i.e., the proposed ones) we may list all of those (mainly academic) introduced in the '90s to model and analyze specific Architec-

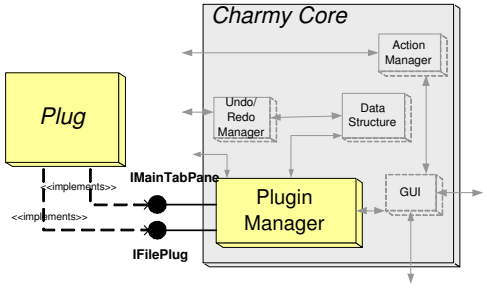

Fig. 3. Plug and Core

ture Description Languages (ADLs) (e.g., Aesop, ArTek, C2, Darwin, LILEANNA, MetaH, Rapide, SADL, UniCon, Weaves, Wright) [17].

Currently, only ACME's tool *AcmeStudio* and C2's tools *ArchStudio* and *Dradel* seems to be still supported and in use. Moreover, Darwin has been recently revisited in order to introduce behavioral aspects into its standard structural specification.

All such still-in-use tools are somehow easy to use, even if none of them make use of UML-like notations. Their main limitation is that each of them focusses on a particular analysis technique, leaving other techniques unexplored. Moreover, any of them use a different notation for SA specification, thus making even more difficult any integration.

With the introduction of CHARMY, instead, we hope that its UML-based architectural specification and its extensibility will help the software architecture community to identify a standard way to specify SAs and to create a unique ground for building and integrating different SA-based analysis tools.

4 Some Considerations

CHARMY has been thought in order to be *easily integrated in industrial projects*: the model checker engine complexity is hidden, providing the software engineer an automated, easy to use tool which takes in input the architectural models in an UML-based notation, creates the prototype and automatically analyzes the prototype reducing as much as possible the human intervention.

By using CHARMY the software architect *saves also time and improves space efficiency*. In fact, CHARMY provides guidelines on how to model the system and automatically generates an optimized Promela code, thus allowing an exhaustive analysis through model-checking. The usual problems of state explosion and model memory size are mitigated, without requiring particular knowledge to users. In fact, experience shows that there is generally a big difference in efficiency and memory size between models developed by a "casual" user and models developed by an "expert" user.

This version of the tool is freely inspired by the *Eclipse* plugin architecture, event if it is not Eclipse compliant. Instead of building CHARMY as a set of

plugs for Eclipse, we preferred to create our in-house tool in order to put in place lightweight framework. We are currently evaluating how much our plugin architecture differs from Eclipse, and how our plugin architecture may be improved by ameliorating the way the core system and the plugs interact.

CHARMY has been used in *several case studies* both industrial and academic: *NICE* a joint work with Marconi Mobile Lab. NMS C2 (L'Aquila-Italy) that operates in a naval communication environment [8]. *Siena* and COMETA a publish/subscribe middleware and its extension to handle the mobility [4]. *Engineering Order Wire (EOW)* a joint work with Siemens C.N.X. S.p.A., R. & D. (L'Aquila-Italy). EOW is an application that supports a telephone link between multiple equipments by using dedicated voice link channels [2].

5 Future Work

Interesting extensions are planned to take place.

As described in Section 2, the model checker currently used as verification engine is Spin. Since we are not tied down to use the model checker SPIN, we are currently investigating the use of SMV or Bogor as model-checking engines. In the case of Bogor, it is very interesting to take advantage of its plugin structure in order to define a customized algorithm search.

Recently much effort focusses in techniques that operate on the input of the model checker (models) in order to continue improving time and space efficiency: abstraction, symmetry and compositional reasoning [7] are the currently evaluated solutions. The plugin SA of CHARMY will allow the introduction of new features to handle those new techniques. For example, we are currently developing a plugin to use CHARMY in a compositional approach, as theoretically treated in [5].

The SA topology editor will be extended by following the representation provided by common architecture description languages. By using existing architectural languages, we may also reuse existing dependence analysis [22] and architectural slicing [23] techniques, already automated by other tools.

References

1. L. Bass, P. Clements and R. Kazman. Software Architecture in Practice, 2nd edition. SEI Series in Software Engineering, *Addison-Wesley Professional*, 2003.
2. A. Bucchiarone, H. Muccini, P. Pelliccione, and P. Pierini. Model-Checking plus Testing: from Software Architecture Analysis to Code Testing. In Proc. Int. Workshop on Integration of Testing Methodologies, ITM '04. October 2004.
3. R. Buchi, J. 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. Standford University Press, 1960.
4. M. Caporuscio, P. Inverardi, and P. Pelliccione. Formal analysis of architectural patterns. In *First European Workshop on Software Architecture - EWSA 2004*, 21-22 May 2004, St Andrews, Scotland, UK.

5. M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proceedings of the International Conference on Software Engineering (ICSE 2004)*, Edimburgh, 2004.
6. Charmy Project. Charmy web site. http://www.di.univaq.it/charmy, February 2004.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2001.
8. D. Compare, P. Inverardi, P. Pelliccione, and A. Sebastiani. 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. 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.
10. D. Garlan. Software Architecture: a Roadmap. In A. Finkelstein (Ed.), *ACM ICSE 2000, The Future of Software Engineering*, pp. 91-101, 2000.
11. C. Hofmeister, R. L. Nord, and D. Soni. *Applied Software Architecture*. Addison Wesley, 1999.
12. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003.
13. J. G. Holzmann. The logic of bugs. *In Proc. Foundations of Software Engineering (SIGSOFT 2002/FSE-10)*, 2002.
14. P. Inverardi, H. Muccini, and P. Pelliccione. Automated Check of Architectural Models Consistency using SPIN. In *the Automated Software Engineering Conference Proceedings (ASE 2001)*. San Diego, California, Nov 2001.
15. P. Inverardi, H. Muccini, and P. Pelliccione. Charmy: A framework for model based consistency checking. Technical report, Department of Computer Science, University of L'Aquila, May 2004.
16. jSpin - A Java GUI for Spin. http://stwww.weizmann.ac.il/g-cs/benari/jspin/
17. N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering, 2000, 26(1), pp. 70-93.*
18. N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, and J.E. Robbins. Modeling Software Architectures in the Unified Modeling Language. ACM Trans. on Software Engineering and Methodology, vol. 11, no. 1, pages 2-57 (January 2002).
19. Object Technology International, Inc. Eclipse Platform Technical Overview. Technical report, IBM. www.eclipse.org/whitepapers/eclipse-overview.pdf, July 2001.
20. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages pp. 46–57, 1977.
21. D. J. Richardson and P. Inverardi. ROSATEA: International Workshop on the Role of Software Architecture in Analysis E(and) Testing. ACM SIGSOFT Software Engineering Notes, vol. 24, no. 4, July 1999.
22. J. A. Stafford, A. L. Wolf, and M. Caporuscio. The Application of Dependence Analysis to Software Architecture Descriptions. In [9], pp. 52-62.
23. J. Zhao. Applying Slicing Technique to Software Architectures. In Proc. Fourth IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS98), pp.87-98, August 1998.