The Journal of Systems and Software 83 (2010) 723–744

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A classification and comparison of model checking software architecture techniques

Pengcheng Zhang [a,c], Henry Muccini [b,*], Bixin Li [a]

[a] School of Computer Science and Engineering, Southeast University, China
[b] Dipartimento di Informatica, University of L'Aquila, L'Aquila, Italy
[c] College of Computer and Information Engineering, Hohai University, China

ARTICLE INFO

Article history:
Received 28 August 2008
Received in revised form 10 September 2009
Accepted 11 November 2009
Available online 16 December 2009

Keywords:
Software architecture
Model checking

ABSTRACT

Software architecture specifications are used for many different purposes, such as documenting architectural decisions, predicting architectural qualities before the system is implemented, and guiding the design and coding process. In these contexts, assessing the architectural model as early as possible becomes a relevant challenge. Various analysis techniques have been proposed for testing, model checking, and evaluating performance based on architectural models. Among them, model checking is an exhaustive and automatic verification technique, used to verify whether an architectural specification conforms to expected properties. While model checking is being extensively applied to software architectures, little work has been done to comprehensively enumerate and classify these different techniques.

The goal of this paper is to investigate the state-of-the-art in model checking software architectures. For this purpose, we first define the main activities in a model checking software architecture process. Then, we define a classification and comparison framework and compare model checking software architecture techniques according to it.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

Software Architecture (SA) has emerged as a principle method of understanding large-scale structures of software systems (Shaw and Garlan, 1996; Shaw and Clements, 2006). The typical use of SA is as a high-level design blueprint of the system to be used during system development and later for maintenance and reuse. At the same time, SA can be used by itself to analyze and validate architectural choices, both behavioral and quantitative, complementing traditional code-level analysis techniques. More recently, new model-driven techniques and architectural programming languages have been introduced to guide the design and coding process from an architectural artifact (ArchJava, 2007; Fujaba RT Project, 2009; Hacklinger, 2004). In summary, SA specifications are used for many purposes (Mustapic et al., 2004; Bril et al., 2005; Bernardo and Inverardi, 2003): as a *documentation artifact*, for *analysis*, and to *guide the design and coding process*.

Most of these activities rely on the *selection of the right architecture* with respect to certain quality requirements that need to be satisfied. Thus, the problem of assuring as early as possible the correctness of an SA occupies increasing importance in the development life-cycle, especially since SA artifacts are becoming the foundation for building families of software products.

Analysis techniques and tools have been introduced to understand whether the SA satisfies certain expected qualities (Bernardo and Inverardi, 2003; de Lemos et al., 2008; QOSA, 2005-2007). Model checking, deadlock detection, testing, performance analysis, and security are, among the others, the most investigated analysis techniques to assess the architectural quality and to predict final system characteristics. Among the techniques that allow designers to perform exhaustive verification of the systems (such as theorem provers, term rewriting systems and proof checkers), model checking (Clarke et al., 2000; Queille and Sifakis, 1982) has the advantage that it is completely automatic. The user provides a model of the system and a specification of the property to be checked and the model checker returns either true if the property is verified, or a counter-example if the property is violated. The counter-example is particularly important since it shows a trace that leads the system to an error. For these and other reasons, the application of model checking has achieved great success in finding subtle errors in complex industrial designs such as sequential circuits, communication protocols and digital controllers (Clarke et al., 2000).

Recently, model checking has also been used to assess whether an SA specification satisfies desired architectural properties (Inverardi et al., 2001; Bose, 1999; Barber et al., 2001; Ciancarini and Mascolo, 1999; Corradini and Inverardi, 1998; He et al., 2002; Jerad and Barkaoui, 2005; Magee et al., 1999; Tsai et al., 1997; Colvin et al., 2008; Björnander et al., 2009). Although many different approaches have been proposed, little work has been done to provide a comprehensive classification and comparison.

* Corresponding author.
E-mail addresses: pchzhang@seu.edu.cn, pchzhang@hhu.edu.cn (P. Zhang), henry.muccini@di.univaq.it (H. Muccini), bx.li@seu.edu.cn (B. Li).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.11.709

The goal of this paper is to classify and compare model checking SA techniques. For this purpose, we first define the model checking SA problem. Then, we identify a set of entities and attributes based on domain knowledge and on existing work. We then select papers in the scope of this work and classify them according to the identified entities. Differences and similarities are highlighted. The main contributions of this paper are: the formalization of the model checking SA problem, the identification of a set of classification entities and attributes, a survey of state-of-the-art model checking SA techniques, and a detailed comparison of selected model checking SA techniques.

The remainder of this paper is organized as follows: Section 2 outlines related work. Section 3 defines the model checking SA problem, describes the comparison framework and the scope of this survey. In Section 4, state-of-the-art model checking SA techniques are classified and compared according to the framework. Section 5 draws conclusions.

2. Related work

Although many different techniques have been proposed to model check software architectures, little work has been done on their comprehensive classification and comparison.

The closest paper related to this study is the work done by Tsai and Xu (2000), which compares formal verification techniques applied over software architecture specifications. In order to provide a systematic comparison of formal verification techniques applied to software architecture specifications, the authors convert architectural specifications into a labeled transition system model (considered as a bridge between the architecture description languages and the input languages of different verification tools). Such a model is then translated into the input format of the existing formal verification tools in order to analyze architectural properties. This work, while significant, has two shortcomings: first, it focuses on applying formal verification tools to architectural specifications, and not specifically on the topic of model checking software architectures. Second, it is quite old and many new model checking techniques have been proposed since its publication in 2000.

Dobrica and Niemelä (2002) propose a survey of architecture analysis methods. They also propose a framework for comparing eight surveyed approaches, to discover similarities and differences. The evaluation framework they propose covers the following aspects: the specific goals of the method, the evaluation technique included in the method, the quality attributes assessed by the method, the stakeholders' involvement in the evaluation process, the SA description and views used in the analysis method, the method's activities, the reusability of an existent knowledge base, and the method validation. In contrast to our proposal, they cover scenario-based, evaluation and maintenance methods, while not focussing on model checking SA techniques.

Babar and Gorton (2004) extend Dobrica's and Niemela's work (Dobrica and Niemelä, 2002) and the authors' previous survey by proposing a wider perspective and a more detailed framework for comparing SA evaluation methods. Moreover, new techniques are surveyed, not available at the time of the previous study. The main entities analyzed in that paper are: *context*, *stakeholders*, *contents*, and *reliability*.

Garlan and Schmerl (2006) note the importance of formal approaches to specify architectures and outline several techniques for formally specifying and analyzing software architectures. Different from our proposal, their paper focusses on research carried out at Carnegie Mellon University in the ABLE Project, as opposed to a comprehensive survey.

Even if not directly related to the model checking SA study presented in this paper, there are also several surveys about architecture description languages (Vestal, 1993; Clements, 1996; Medvidovic and Taylor, 2000). Among them, in Medvidovic and Taylor (2000) the authors propose a classification and comparison framework for architectural languages based on the following main elements: components (including interfaces, types, semantics and etc.), connectors (including interfaces, types, semantics, etc.), architecture configuration, and tool support. Our paper will reuse and extend some of the findings illustrated in their work.

3. The classification and comparison framework

This section proposes a framework for comparing existing model checking SA techniques. As a driver for the selection of the main parameters for classifying and comparing state-of-the-art model checking SA approaches, Section 3.1 defines the model checking SA problem. Based on such a definition, Section 3.2 provides an outline of the various entities and attributes forming the framework. Section 3.3 specifies the scope of this work, while Section 3.4 lists and provides a brief description of surveyed model checking SA techniques.

3.1. The model checking SA problem

Model checking is a formal verification technique whose aim is to analyze, through an exhaustive and automatic approach, systems' behavior with respect to selected properties. It takes as input a system model and a property specification. The output is "true", or false with a counter-example when an error is detected.

SA is the earliest model of the whole software system created along the software life-cycle. SA describes (through formal languages or models) how a system is structured into components and connectors and how these elements interact with each other (i.e., configurations).

The *model checking SA* problem consists in *deciding whether an SA specification satisfies certain SA properties*. The input to this process is an SA specification and a set of SA properties. The model checker processes the SA inputs in order to provide the output. There could be cases in which the SA inputs cannot be directly elaborated by the model checker, thus an intermediate translation activity may be required. Fig. 1 graphically summarizes the main artifacts (drawn as boxes) and activities (drawn as rounded boxes) in model checking SA.

3.2. The framework entities and attributes

The framework shown in Fig. 2 is used to classify and compare model checking SA approaches. It has two main goals:

- Understanding the main artifacts and activities that take place when model checking software architectures.
- Understanding the techniques' associated qualities (such as, usability, reliability, scalability, and expressiveness).

The selection and formation of the framework's entities and attributes is drawn from a number of sources, such as previous papers, domain knowledge, and conformance to the survey goals.

As far as the former goal is concerned, and by referring to Fig. 1, we identify the following main entities: the *input*, the *computation*, and the *translation*. Those three entities have been selected to reflect the model checking SA process described in Section 3.1 and in-line with some entities and attributes presented in similar work (e.g., Tsai and Xu, 2000 and Babar and Gorton, 2004) or in other comparative studies (e.g., Naslavsky et al., 2006). Note that the *output* artifact, while not discussed in this goal, will be analyzed as part of the quality entities and attributes in the second goal.

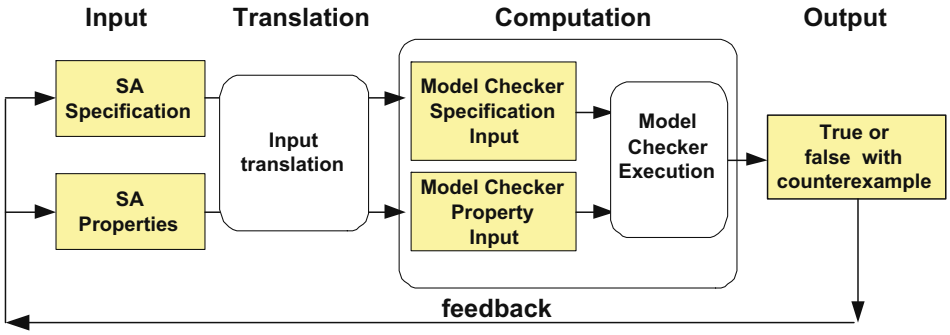

Fig. 1. Main activities in model checking SA.

According to the model checking SA problem, the *input* entity can be decomposed into two main entities: the *specification of the SA* and the *specification of SA properties*. For the first input, various specification languages have been proposed, spanning from informal (box-and-line) to formal (ad hoc notations) (Medvidovic and Taylor, 2000; Medvidovic et al., 2007). Each *architectural description* language and notation has certain *modeling features* (e.g., components, connectors, ports, interfaces) (Medvidovic and Taylor, 2000). The latter input to the model checking SA process consists of a set of architectural *properties*. Properties can be specified via different *property description* languages and notations (e.g., LTL, CTL, CTL*), can expose different *features* (i.e., modeled elements), and can be used to specify different *types* of properties (e.g., safety and liveness).

The existence of *tool support* for both SA specification and properties specifications is also noted.

The second macro-entity focuses on the *computation*, that takes as inputs the specifications previously analyzed and produces an output. The study classifies model checking SA approaches according to the *model checking engine* used by the surveyed approaches.

The third macro-entity considers the *translation* activity, that consists of the process of making the SA inputs in a format readable by the model checking engine. We focus on classifying existing approaches according to the types of *translation* required (i.e., no translation, partial translation, or total translation) and its level of automation.

Sections 4.1, 4.2, and 4.3 will analyze such macro-entities in detail.

As far as the second goal is concerned (i.e, understanding the techniques' qualities), we selected the following entities: *usability*, *reliability*, *scalability*, and *expressiveness*. They have been then elaborated based on related studies.

Based on ISO/IEC 9126-1:2001,[1] ISO 9241-11,[2] and the IEEE 610.12-1990[3] standards, three are the sub-characteristics considered for *usability*: *understandability*, *learnability*, and *operability*.

The *reliability* entity will classify model checking SA approaches according to a subset of the 4-level maturity scale as proposed in (Babar et al., 2004): inception, development, refinement, and dormant.

As far as *scalability* is concerned, two main entities are considered: the model checking SA *ability to specify complex architectures* and the *ability to model check complex inputs*.

The *expressiveness* entity will analyze which type of properties can be checked by the surveyed model checking SA techniques.

Sections 4.5–4.7 and 4.8 analyzes in detail these macro-entities. The table in Fig. 2 summarizes the selected entities and attributes.

3.3. The scope of this survey

Once we have defined the entities and attributes of interest for our framework, the paper selection process has been driven by the following criteria: first of all, we selected papers that present work on architecture description languages (ADLs) suitable for model checking. Starting from an informal survey on ADLs conducted in Dipartimento di Informatica, University of L'Aquila, we found 52 ADLs and model-based architectural notations. We analyzed all of those to check whether they have been integrated with model checking features. From this first study, we identified the Wright, CHAM, Darwin/FSP, Archware, Æmilia, CBabel, AutoFocus, and Fujaba as appropriate model checking SA approaches.[4] Our second criterion was to select the model checkers that have been directly applied at the architectural level. Starting from the model checkers listed at the YAHODA Verification Tool Database,[5] we analyzed whether they have been applied to architectural artifacts. Accordingly, we added to the list of model checking SA techniques SAM, Garlan et al., Bose, Arcade, Zanolin et al., and CHARMY. The last criteria was to include the papers explicitly focusing on model checking architectural specifications. After a search conduced on Google Scholar, IEEE, ACM DL, DBLP, and other sources, we discovered PoliS and Lfp.

While it is difficult to guarantee that every relevant approach is included in this survey, we have made an effort to find and examine all the techniques proposed in the last 15 years related to model checking software architectures.

3.4. The surveyed model checking SA techniques

Sixteen model checking SA techniques are considered in this study. A brief chronological outline of the surveyed techniques is presented below, and summarized in Fig. 3.

Allen and Garlan (1997, 1998) is an architectural specification language proposed in 1997 to support more direct formal specification and analysis of software architectures. Wright can be considered the first seminal work on model checking SA. Wright uses CSP and FDR for formally specifying, analyzing and checking architectural connections. Its extension in Allen et al. (1998) addresses the problem of specifying and analyzing dynamic architectures. *Tsai et al.* proposed an incremental method to verify SA of real-time systems (Tsai et al., 1997); a specification language to model and verify desired properties of complex and real-time sys-

[1] ISO/IEC 9126. Software Engineering – Product quality. Part 1: quality model, June 2001.

[2] ISO 9241. Ergonomics Requirements for Office with Visual Display Terminals (VDTs) Part 11: Guidance on usability, 1998.

[3] IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology, December 1990.

[4] Citations and information about the selected approaches are reported in the next section.

[5] http://anna.fi.muni.cz/yahoda/, maintained by the ParaDiSe laboratory at the Faculty of Informatics, Masaryk University Brno.

Goal #	Macro - Entities	Entities	Sub - Entities	Attributes	Brief Explanation
Goal #1	Input	SA specification	Architectural description languages and notations	formal ADL, formal process algebra, formal specification, model-based	What form of SA descrip tion is used (e.g., formal, informal, model -based, etc.)? Which specific architectural language?
			Architectural modeling features	components, components interface, connectors, connectors interface, components and connectors semantics, types, configurat ions	Which are the features provided by the SA language for modelling software system's conceptual architecture?
		SA property	Property description language and notation	existing, new ad hoc notations, graphical	What form of SA property description is used (e.g., existing, ad hoc notations, graphical, textual, others etc.)? Which property specification language?
			Property modeling features	components, components interface, connectors, connectors interface, types, configuration	Which are the features pro vided by the SA property language for model checking conceptual architecture (e.g., safety, liveness, fairness)?
			Type of property	safety, liveness, compatibility, equivalence	Which are the type of properties that can be specified?
		Tool support		type of tool (available, internal, not available), last release, version	Is the SA or SA property specification supported by a tool?
	Computation	Model checking engine		generic engines, ad hoc engines, no engines	Which are the model checking engines utilized by model checking SA techniques?
	Translation	Translation type		no translation required, SA specification translation, property translation	Is the input directly usable by the model checker?
		Tool support		automated SA specification translation, automa ted SA property translation, no tool support	Is the translation process automated?
Goal #2	Usability	Understandability	Explicit configuration	explicit, implicit	Is the SA configuration a first class element ?
			Graphical notations	graphical, grap hical & sound, textual	Is the model checking SA description and SA property modelling semantically sound?
		Learnability	Output Interpretation	easy to comprehend, difficult to comprehend	Is the output comprehensible to a software architect?
		Operability	Automation	none, low, medium, high	Which is the degree of automation in model checking SA techniques?
	Reliability	Maturity of the method		development, refinement, dormant	What is the level of maturity (inception, development, refinement or dormant)?
		Methods validation		single and small, multiple and industrial	Has the method been validated? How has it been validated?
	Scalability	Ability to specify complex architectures	hierarchical/ compositional modeling	yes, not	Does the model checking SA techniqu e support hierarchical modelling?
			textual and/or graphical specification	textual, graphical, textual and graphical	Is the SA specification textual?
		Ability to model check complex inputs	Abstraction and reduction	abstraction, reduction, known scalabi lity	Does the model checker support abstraction and minimization?
	Expressiven ess	SA specification expressiveness vs. SA property expressiveness		(component modeling /component Property), (connector modeling / connector property), (comp. Interface modelin g / comp. Interface Property), (conn. Interface modeling / conn. Interface Property), (component & connector Types / type property)	Can the desired properties be checked by the model-checking SA approach?

Fig. 2. The framework parameters.

tems has been introduced as part of the approach. The Chemical Abstract Machine (CHAM) (Compare et al., 1999; Corradini et al., 2006) and its model checking features have been proposed in '98 to specify dynamic and reconfigurable architectures in terms of molecules and reaction rules and to model check them. Bose (1999) presented a method which automatically translates UML models of SA for verification and simulation through SPIN (Holzmann, 1997); roles and coordination policies are specified through

Name	Year	Principal Investigator(s)	References
Wright	1997	Allen and Garlan	Allen and Garlan (1997, 1998)
Tsai	1997	Tsai	Tsai et al., 1997
CHAM	1998	Inverardi and Wolf	Compare et al., 1999; Corradini et al., 2006
Bose	1999	Bose	Bose 1999
Darwin/FSP	1999	Magee, Kramer, Giannakopoulou	Giannakopoulou, 1999; Magee et al., 1999
PoliS	1999	Ciancarini and Mascolo	Ciancarini and Mascolo, 1999; Ciancarini et al., 2000
Arcade	2001	Barber, Graser, and Holt	Barber et al., 2001
Archware	2001	Warboys and Oquendo	Oquendo et al., 2004; Oquendo, 2004; Mateescu and Oquendo, 2006
CHARMY	2001	Pelliccione, Inverardi, Muccini	Inverardi et al., 2001; Pelliccione et al., 2009
AEmilia	2002	Bernardo	Balsamo et al., 2003; Aldini et al., 2009
SAM	2002	He, Ding, Deng, Yu	Shi and He, 2003; Yu et al., 2004; He et al., 2004; He, 2005
Garlan et al.	2003	Garlan, Khersonsky, Kim	Garlan et al. (2003)
CBabel	2003	Braga and Sztajnberg	Braga and Sztajnberg, 2003; Rademaker et al., 2005; Braga et al., 2009
Zanolin et al.	2003	Zanolin, Ghezzi, Baresi	Zanolin et al. 2003
AutoFocus/Auto Focus 2	2005	Philipps, Slotosch	MunichUT, 2006
Fujaba	2005	Giese	Burmester et al., 2005; Becker et al., 2006; Burmester et al., 2007
Lfp	2005	Jerad and Barkaoui	Jerad and Barkaoui, 2005; Jerad et al., 2007; Jerad et al., 2008

Fig. 3. The 17 surveyed model checking SA approaches.

a mediator component and model checked. *Darwin/FSP* (Giannakopoulou, 1999; Magee et al., 1999) is a model checking SA approach introduced to analyze behaviors of architectures for concurrent and distributed systems: the SA is specified in Magee et al. (1995) ADL via the FSP (Magee and Kramer, 2006) process algebra. In '99, Ciancarini and Mascolo proposed a formal SA coordination model called *PoliS* (Ciancarini and Mascolo, 1999; Ciancarini et al., 2000) based on nested tuple space. The PoliS specification is hierarchically structured, and denotes a tree of nested spaces whose structure evolves *dynamically at run-time*. It can reason on changes and evolution with respect to structural and behavioral constraints. *Arcade* (Barber et al., 2001) applies model checking to DRA (domain reference architecture) to provide software architects early feedbacks on safety and liveness properties. The main goal of the *ArchWARE* (Oquendo et al., 2004; Oquendo, 2004; Mateescu and Oquendo, 2006) project is to specify and analyze evolvable SA at run-time; one of the project outcomes has been a model checking technique for evolvable architectures. CHARMY (Inverardi et al., 2001; Pelliccione et al., 2009) is a framework for model checking the SA compliance to desired functional temporal properties. It intends to provide an automated, easy to use tool for the model-based design and validation of concurrent and distributed SAs. *Æmilia* (Balsamo et al., 2003; Aldini et al., 2009) is an ADL based on the stochastic process algebra $EMPA_{gr}$: it allows for model checking, performance and security analysis. *SAM* (Shi and He, 2003; Yu et al., 2004; He et al., 2004; He, 2005) uses Petri Nets to define the behavior models of components and connectors while temporal logic is used to specify properties of components and connectors. SAM is dedicated to distributed systems, and can be used to model embedded as well as real-time systems. Garlan et al. (2003) present an approach for model checking a publish-subscribe style architecture where there is not a fixed finite state model, and components can be added and removed at *run-time*. *CBabel* (Braga and Sztajnberg, 2003; Rademaker et al., 2005; Braga et al., 2009) is a declarative language, which allows the description (and checking) of SAs as a composition of selected modules and connectors. The CBabel ADL is suitable to model evolvable and reconfigurable architectures. In Zanolin et al. (2003), an alternative method for modeling and validating publish-subscribe architectures is proposed by *Zanolin et al. AutoFOCUS* (MunichUT, 2006) is a model-based tool for the development and verification of reliable, distributed, and embedded systems. In AutoFOCUS, static and dynamic aspects of the system are modeled and an integrated tool for modeling, simulation, and validation is provided. *Fujaba* (Burmester et al., 2005; Becker et al., 2006; Burmester et al., 2007) is an approach tool supported for real-time model checking of component-based systems modeled through UML component diagrams and real-time statecharts. Part of Fujaba is the Fujaba Real-Time Tool Suite which is specifically tailored for the architectural modeling of *distributed*, *embedded real-time and mechatronic* systems with the Mechatronic UML approach.[6] *Lfp* (Jerad and Barkaoui, 2005; Jerad et al., 2007; Jerad et al., 2008) is a formal language dedicated to the description of distributed embedded systems' control structure and it is used to verify properties of *distributed embedded* SA. It inherits characteristics from both ADLs and coordination languages and also supports the model checking attribute.

4. Model checking software architectures: the study

This section presents a detailed overview and comparison of existing model checking SA techniques along the dimensions discussed in Section 3.2. We highlight representative techniques and support our arguments with a running example. The example is a well known Producer_Buffer_Consumer (PBC) architecture. The architecture is composed of three types of components: Producer, Consumer and Buffer. The Producer produces the input data/service which is consumed by the Consumer. The Buffer is used to coordinate the interactions between Consumer and Producer. The chosen example is deliberately kept simple and wants to provide to the reader a flavor of which kind of solution an approach may provide for a particular model checking SA problem. While applying the different model checking SA techniques to the PBC architecture, structural and behavioral aspects will be specified according to the language adopted by each single approach. Fig. 4 shows a box-and-line representation of the PBC architecture.

The following of this section is structured in this way: Section 4.1 will analyze the model checking SA input parameters. Section 4.2 will analyze the model checking engines employed in the surveyed techniques. Section 4.3 will cover those parameters related to the transition from SA inputs to the model checking inputs. Section 4.5 will analyze the usability of the surveyed techniques. Section 4.6 will focus on the reliability of the existing model checking SA techniques. Section 4.7 will cope with the scalability while Section 4.8 will analyze the expressiveness of the selected techniques. A summary of goal one findings is presented in Section 4.4. A summary of goal two findings is presented in Section 4.9.

4.1. Goal 1 – model checking software architecture: the input

There are two main *inputs* to be considered in the model checking SA process: the *specification of the SA* and the *specification of SA properties* (as summarized in Fig. 2). For the former input, this study will focus on the *architectural description*, *modeling features*, and *tool support* entities. For the latter input, this work will focus on the *property description*, *property modeling features*, *type of property*, and *tool support* entities.

4.1.1. SA specification – architectural description languages and notations

The *architectural description languages and notations* entity here analyzed is utilized to investigate what form of SA description[7] is utilized by the model checking SA (surveyed) techniques.

[6] We omit from this work other formal analysis and checking capabilities of the tool which do not fall into model checking such as, for example, invariant checking of dynamic structures (Becker et al., 2006), hybrid behavior (Burmester et al., 2007), and automatic refinement synthesis of component behavior (Henkler et al., 2009).

[7] This terminology has been taken from Medvidovic and Taylor (2000).

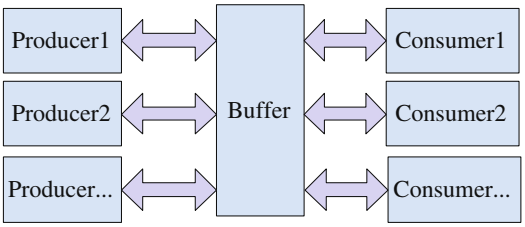

Fig. 4. The SA of the PBC example.

According to Medvidovic and Taylor (2000); Medvidovic et al. (2002), SAs can be specified in various ways: through formal specification languages (architecture description languages – ADLs or other general formal languages), model-based diagrammatic notations, or box-and-line notations. Three kind of architectural descriptions are considered in the surveyed approaches: *formal languages*, *model-based* notations, and *both formal and model-based* notations. Importantly to remark, we hereafter refer to "model-based" as equivalent to "diagrammatic", and not as opposite to "formal" (most of the model-based notations we discuss have, in fact, a clearly formalized syntax and semantics).

Fig. 5 graphically summarizes how the surveyed techniques fit into one of the three categories while Fig. 6 provides information on the architectural languages utilized by the surveyed techniques.

4.1.1.1. Formal languages. Many of the surveyed approaches rely on a formal (non diagrammatic) specification of the architecture to be submitted to model checking, as shown in Fig. 5. The first sub-classification to be done is in between formal ADL and general formal languages (i.e., non architectural specific).

Formal languages – formal ADL specifications: An ADL, as described in Medvidovic and Taylor (2000) is a language that explicitly models architectural concepts such as components (with interfaces), connectors (with interfaces), component/connector types, and their configurations. In our surveyed techniques, Wright, Darwin/FSP, ArchWARE, Æmilia, CBabel, and CHAM all use a formal ADL specification for SA modeling.

Among the six identified above, four ADLs make use of *process algebraic* languages. Wright ADL is based on process algebra CSP. In the Darwin/FSP ADL, the components' behavior are specified via the finite state process (FSP) (Magee and Kramer, 2006) algebra in terms of labeled transition systems. The PADL component-oriented process algebra is used by Æmilia. The ArchWare ADL (also called π-ADL Oquendo, 2004) is based on an extension of the modal mu-calculus to specify SAs in *ArchWare*.

The *CBabel* ADL is a declarative language which allows the description of an SA as a composition of selected modules and connectors. The *CHAM* description of an SA consists of a syntactic description of the static components of the architecture (the molecules), and a specification of reaction steps used to describe how the system dynamically evolves.

Formal languages – general formal specification (non architectural specific): Although general formal specification languages do not have all the attributes to be considered as an ADL, some of them have been used/extended for SA modeling. The model checking SA approaches Tsai, PoliS, Arcade, SAM, and Garlan et al. all rely on existing general formal specifications.

The Tsai method uses an object-oriented specification language based on Horn logic to model the SA. PoliS is a *coordination language* used to model SA, based on nested tuple space. Arcade (architecture analysis dynamic environment) uses DRA (domain reference architecture) to represent an SA and provides analysts and developers with early feedback from safety and liveness evaluations. SAM is a general formal framework for specifying and analyzing hierarchical SAs: it uses Petri Nets to define the behavior of components and connectors. Garlan et al. in their model checking SA proposal use finite state machine models to deal with a special and (not fixed) finite state model of the SA.

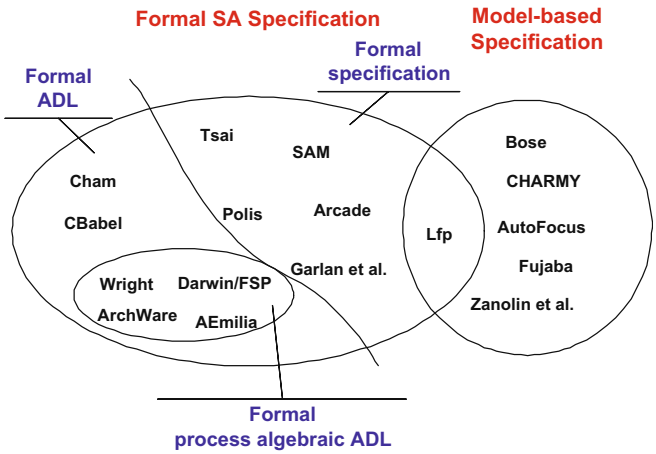

Fig. 5. Formal or model-based SA specification.

4.1.1.2. Model-based notations. Recently, UML-based notations have been introduced for specifying SAs (Medvidovic et al., 2002). In the context of this study, Bose, CHARMY, Zanolin et al., Fujaba, and AutoFocus make use of model-based notations, in terms of UML diagrams, for SA modeling. They all start from the standard UML diagrams (UML 1.x for Bose, UML 2.0 for AutoFocus, both UML 1.x and UML 2.0 for CHARMY, and UML statecharts for Zanolin et al.). Bose uses a stereotyped class diagram to specify the architecture composed of components and mediators. CHARMY uses stereotyped class and state diagrams to model the SA topology and behavior. AutoFocus uses different views (based on stereotyped class, sequence, and state diagrams) to model the structure and the behavior of an SA. Fujaba uses UML 2.0 component diagrams, while components' behavior is expressed through a real-time extension to statechart.

4.1.1.3. Formal and model-based notations. Recently, some researchers started combining the advantages of both formal and model-based specification for model checking SA. Lfp belongs to this category. Lfp is a graphical coordination language providing facilities of an ADL. LfP can also be linked to an UML-based methodology. It aims to provide a UML description with structured information enabling formal verification and automatic generation of distributed programs.

4.1.2. SA specification – modeling features

This entity investigates which modeling features (i.e., architectural concepts) are provided by the architectural notations. (This attribute, while already analyzed in Medvidovic and Taylor (2000), is now applied to a more up-to-date list of architecture description languages, and it is also applied to model-based notations).

While there is no a standard agreement on what goes in an architectural description, some concepts such as *component (with interface)*, *connector (with interface)*, *component/connector's semantics*, *component/connector types*, and *configurations* are considered standard elements to be presented in any architectural language (xArch, 2007; CMU, 1998; Medvidovic and Taylor, 2000). This section is organized to show how software architecture, in the model checking SA techniques, are specified in terms of the aforementioned elements.

Fig. 7 summarizes SA specification modeling features.

	Wright	Tsai	CHAM	Bose	Darwin/FSP	PoliS	Arcade	Archware	CHARMY
SA Specification Language	Wright	Extend Horn logic clause	CHAM	UML Model	Darwin and FSP	PoliS	DRA	Archware ADL	State and Sequence diagram
Specification Language Type	Formal ADL	Formal	Formal ADL	Model-based	Formal ADL	Formal	Formal	Formal ADL	Model-based

	AEmilia	SAM	Garlan et al.	CBabel	Zanolin et al.	AutoFOCUS	Fujaba	Lfp
SA Specification Language	PADL	Petri net and Temporal Logic	State machine model	CBabel	UML statechart diagram	Four different models	UML 2.0 component diagrams, RT statecharts	LfP
Specification Language Type	Formal ADL	Formal	Formal	Formal ADL	Model-based	Model-based	Model-based	Formal & MB

Fig. 6. Architectural language and type.

4.1.2.1. Components and components' interface modeling. Components are explicitly modeled by *formal specifications* in Wright, Darwin/FSP, ArchWare, Ämilia, Garlan et al., and by all the model-based SA specifications (Bose, CHARMY, Zanolin et al., Fujaba, and AutoFOCUS).

Other formal specification languages refer to components in an *implicit* way, i.e., they use different concepts or names to explicitly model components. CHAM uses molecules to model components in the CHAM ADL. In PoliS, components are represented by tuple or tuple spaces in the PoliS coordination language. In SAM, a component is *hierarchical* and modeled via *Petri Nets*. In CBabel, a component is modeled by a *module*. Lfp uses LfP-classes to model components.

Tsai and Arcade do not explicitly model components.

Component interfaces are supported by most of the *formal approaches*, and by some of the *model-based approaches*, even if with some variation on the terminology and type of information. For example, it is referred as a *port* in Wright, a *service* in Darwin/FSP, a *port* in Bose, ArchWare, CBabel, SAM, AutoFOCUS and Lfp, an interface point in Fujaba, and an *interaction* in Ämilia.

Formal approaches, such as CHAM, PoliS, and Garlan et al., and model-based approaches, such as CHARMY and Zanolin et al. , do not model component *interface*.

4.1.2.2. Connectors and connector's Interface modeling. Connectors are explicitly modeled by Wright, SAM, Zanolin et al., Archware, Bose, Ämilia and Lfp, even if with some specific terminology, e.g., "connectors" in Wright, SAM and Archware, "mediator" components in Bose, "middleware" in Zanolin's p/s architecture, and "medias" in Lfp, and architectural element in Ämilia (that comprises both components and connectors).

Other approaches provide only an *implicit* representation of connectors (i.e., connectors are not first-class entities, thus, they cannot be sub-typed or reused, and a behavioral specification is missing). A connector in Darwin is specified through a *binding*, and in CBabel is a *link*. But they do not have interfaces and a semantic description. In PoliS connectors are described as a set of *tuples*. A connector is modeled as a channel between components

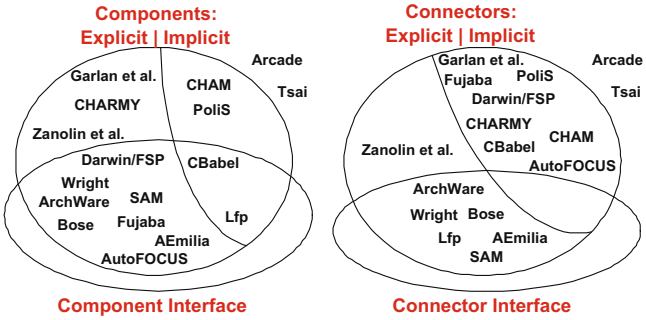

Fig. 7. SA specification modeling features.

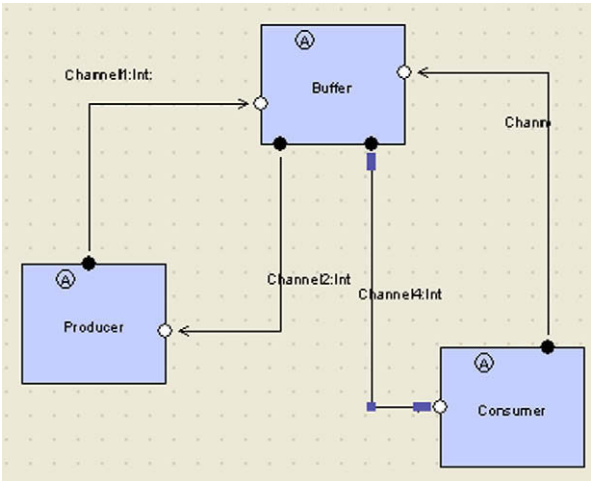

Fig. 8. Connectors of PBC architecture using AutoFOCUS, where each connector is modeled as a channel from a source port (filled circles) to a destination port (empty circles).

in CHARMY, and is modeled as a channel from a source to a destination port in AutoFOCUS (Examples of channels of PBC architecture using AutoFOCUS shown in Fig. 8). In Fujaba, a connector is modeled from provided interface to required interface. CHAM and Garlan et al. support an implicit modeling of connectors.

Among all the approaches that model connectors explicitly, Wright, Archware, Lfp, and Bose have the ability to model *connector interfaces* explicitly.

Whenever connectors are only implicitly modeled, connector interfaces are not specified.

Tsai and Arcade do not model connectors.

4.1.2.3. Components and connectors' semantics. A component, with interface or not, has an underlying formal specification to describe its semantics.[8] Fig. 9 summarizes the semantic models for components and connectors of each approach.

The semantics for a *component* in CHAM and PoliS is rule-based. The semantics for components in Bose, CHARMY, Garlan et al., Zanolin et al., Fujaba, Darwin/FSP, and AutoFOCUS are based on state machines. Bose, CHARMY and Garlan et al. use finite state machines. Zanolin et al. uses statecharts. Fujaba makes use of real-time statecharts. The semantic models for components in Wright, Darwin/FSP, Archware, Ämilia are based on process algebra, i.e., CSP, labelled transition systems (LTS), higher-order typed pi-calculus, stochastic process algebra, respectively. LfP-components semantics are based on state-transition automata. They are specified through LfP-BD diagrams that describe, in hierarchical way, the desired behavior. The semantic models for components in

[8] The term "semantics" is used as in Medvidovic and Taylor (2000) for identifying the high-level model of a component/connector behavior.

	Wright	Tsai	CHAM	Bose	Darwin/FSP	PoliS	Arcade	Archware	CHARMY
Component semantics	CSP	-	Reaction rules	Finite state machine	LTS	Rules	-	Higher Order Typed pi-Calculus	Finite state machine
Connector semantics	CSP	-	None	Finite state machine	None	Rules	-	Higher Order Typed pi-Calculus	None

	AEmilia	SAM	Garlan et al.	CBabel	Zanolin et al.	AutoFOCUS	Fujaba	Lfp
Component semantics	Stochastic process algebra	Petri-net	Finite state machine	Methods in object theory	Statechart	State machine with pre and post condition	Real-time statechart	Lfp-BD
Connector semantics	Stochastic process algebra	Petri-net	Finite state machine	None	Statechart	None	Real-time statechart	Lfp-BD

Fig. 9. Semantics models for components and connectors of each approach.

module **BUFFER** { in port int (int item) put ; in port int (void) get ;}

module **PRODUCER** { out port int (int item) put }

module **CONSUMER** { out port int (void) get ; }

Fig. 10. The CBabel SA description of components BUFFER, PRODUCER and CONSUMER, where BUFFER has two "in" ports while both PRODUCER and CONSUMER have an "out" port.

CBabel are based on methods in object theory. Tsai and Acade do not model components or connectors explicitly, so a semantic model is not provided.

An example of the PBC semantics in CBabel is shown in Fig. 10. Since a component in CBabel has ports, the CBabel behavioral specification provides port semantics. A PBC architecture and the behavioral specifications using CHARMY is depicted in Fig. 11. Please note that since a CHARMY component does not have ports, its formal specification specifies the behavior of the whole component, without focussing on ports behavior.

A *connector*, when modeled explicitly, has an underlying formal specification to model its behavior. Typically, connector's behavior is specified using the same mechanism utilized for specifying component's behavior. For example, Bose uses a state machine diagram to model connector's semantics.

4.1.2.4. Component/connector types. Component/connector types are abstractions that encapsulate functionality into reusable blocks, and allow the instantiation of components/connectors multiple times in a single architecture or allow reuse across architectures (Medvidovic and Taylor, 2000).

Component and connector types are explicitly modeled by Wright, Bose, Darwin/FSP, Archware, Æmilia, CBabel, and Lfp. More specifically, Bose permits a typed specification of components, mediators, and spaces. Darwin/FSP's components can also be typed. In Æmilia, any architectural element can be typed. In CBabel, any module or connector can be typed first and instantiated at runtime. Lfp's module can also be typed and instantiated.

The other model checking SA specification languages make use of instances only.

4.1.2.5. Configuration's modeling. Among the *formal ADLs*, Wright, Æmilia and Archware explicitly model SA *configurations*. The other ADLs and formal specifications Darwin/FSP, CBabel, SAM, PoliS, Garlan et al., Tsai and Arcade do not explicitly model SA *configurations*. They all use the SA basic elements to explicitly form an SA configuration. In the *model-based* notations, CHARMY and AutoFOCUS provide an explicit configuration, while configurations are not explicitly modeled by Bose, Zanolin et al. and FUJABA. (An example of implicit modeling of a configuration using Fujaba is shown in Fig. 12). A configuration is modeled by LfP through LfP-AD. LfP-AD contains the system's architecture description.

4.1.3. SA property – property description languages and notations

This section, as well as the next one, classifies existing techniques according to how SA properties are described and modeled.

SA properties can be of various kinds, and they have been defined in slightly different ways and used for different purposes. In Perry and Wolf (1992) properties are used to constrain the choice of architectural elements, that is, to define constraints on the elements to the degree desired by the architect. In Bass et al. (2003), an SA is composed by components, their visible properties, and the relationships among them. In Shaw and Garlan (1996), a property is considered to be an integral and first-class entity on an SA. In Kazman et al. (1996), the authors propose a scenario-based approach for evaluating architectures to determine their fitness with respect to certain properties.

While SA properties can be of different types (e.g., security, reliability, performance, usability Kazman et al., 1996), this section will focus on the properties to be checked by existing model checking SA approaches – specifically, *temporal* properties, such as safety and liveness.

Through the *property description languages and notations* attribute in our study, we want to investigate what form of SA property description is utilized by the model checking SA surveyed approaches. In particular, an SA property can be specified through *existing* property specification languages, *new* property specification languages, and *existing logics through graphical notations*.

Fig. 13 graphically summarizes how the surveyed approaches fit into the three categories.

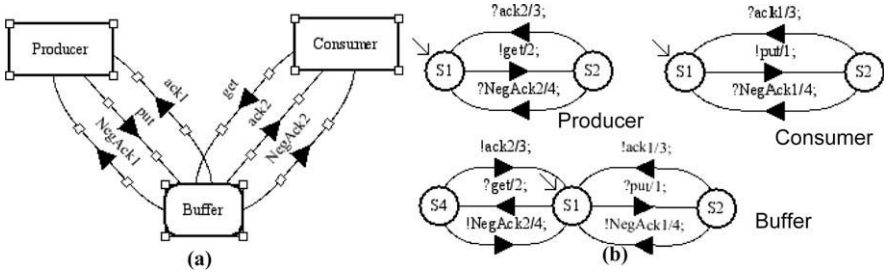

Fig. 11. The PBC architecture using CHARMY, where (a) shows the three components, (b) shows the state machine diagram of each component.

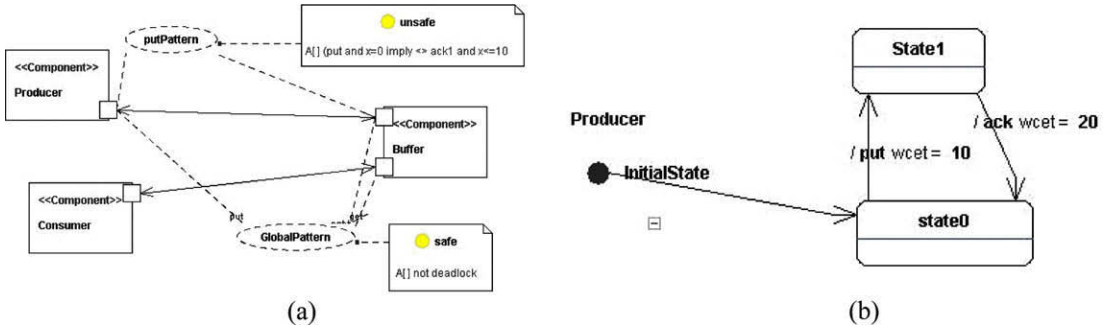

Fig. 12. A simple PBC architecture in Fujaba, (a) shows the implicit configuration and coordination pattern where deadlock-free and timed liveness properties are specified. An example of a real-time statechart of component Producer is shown in (b).

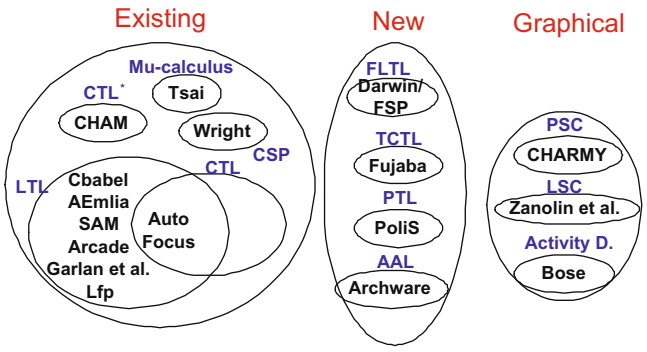

Fig. 13. Existing, new or graphical property specification languages.

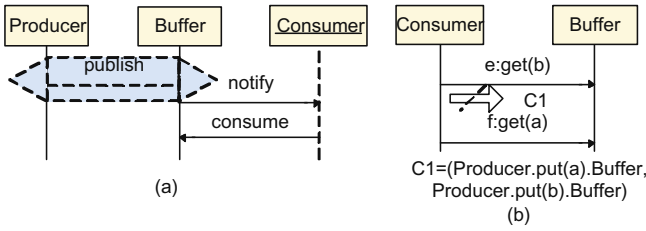

Fig. 14. (a) shows a property specified in LSC. If a Producer publishes his service on Buffer, then the Buffer must notify the Consumer that has to consume the service. Note that the message shown by a dashed line is an existing message while other messages are required. (b) shows a property represented in PSC, that is, if the Buffer is a stack, it must satisfy the first-in and last-out operator, i.e., if get(b) and get(a) happen, put(a) and put(b) must happen in order. Note that "e:" means an existing message type while "f:" means a fail message type.

4.1.3.1. Existing property specification languages. LTL, CTL, CTL*, mu-calculus and CSP are the four existing formalisms used in the surveyed techniques for specifying temporal properties. LTL is the most common used logic which is used by CBabel, Æmilia, Bose, Arcade, LfP, SAM and Garlan et al. Some other approaches can support both CTL and LTL property checking (i.e., AutoFOCUS). CTL* and mu-calculus (used by the CHAM and Tsai, respectively) are purely conceptual property specification languages, not supported by tools. CSP is used by Wright to model properties.

4.1.3.2. New property specification languages. Some new property specification languages have been proposed, typically extending or modifying existing logics.

Darwin/FSP uses FLTL (Giannakopoulou and Magee, 2003) which augments LTL with fluents to express fluent-based properties. To verify properties of real-time system, Fujaba's uses TCTL, which is an extension of CTL with time. PoliS uses PTL, a CTL dialect: the main difference between PTL and CTL is that PTL is based on multi-spaces, so that formulae can be evaluated in a space. ArchWare uses pi-AAL (architecture analysis language) (Mateescu and Oquendo, 2006) defined as an extension of the modal mu-calculus, which is designed to describe run-time and evolvable architectural properties.

4.1.3.3. Existing logics through graphical notations. Writing temporal properties correctly is a difficult task, as recognized by Holzmann, Dwyer, and colleagues in Holzmann (2002) and Dwyer et al. (1999), respectively. For this reason, higher level notations have been introduce to simplify this task. LSC (Damm and Harel, 2001), PSC (Autili et al., 2007), and activity diagrams are three types of graphical (yet formal) notations used by the surveyed approaches to manage this problem. One merit of them is that they provide a graphical notation easily understandable and usable.

Zanolin et al. use LSC as the property specification language. CHARMY uses PSC as the property specification language. Bose uses activity diagrams.

Two properties represented by LSC and PSC are shown in Fig. 14. The second property can be specified using the following LTL formula:

$$[\,]\,(((\mathtt{put(a)}\ \&\ !\mathtt{get(a)})\ \mathtt{U}\ (\mathtt{put(b)}\&\ !\mathtt{get(a)}\ \&\ !\mathtt{get(b)}\,)\,)\ -\!>\ <\!>\ (\ \mathtt{get(a)}\ \&\ !\mathtt{get(b)}\ -\!>\ <\!>\ \mathtt{get(b)}).$$

While the LTL formula is not easily understandable, the same property expressed in the PSC formalism (Fig. 14.b) appears more intuitive and closer to natural language description (even if still formal). Recently, PSC has been extended with time constructs (Zhang et al., 2010; Zhang et al., 2008) to model timing properties for real-time systems.

4.1.4. SA property – modeling features

By going deeper into the architectural elements (i.e., components, connectors, etc.) considered when expressing the SA property, the table in Fig. 15 graphically summarizes the architectural elements a property can refer to. Based on the table, we can notice that ArchWare and Wright have the richest SA property language, since all the considered architectural elements can be specified within it. Æmilia can specify all the elements except type. Darwin/FSP, AutoFocus, and Fujaba[9] can express properties about components and their interfaces. Bose, Darwin, ArchWare, CBabel, and Lfp permit one to specify properties about component types, while

[9] It has to be noted that properties for types, interfaces, and all other modeling artefacts could be specified by structural specifications similar to the ones explained in Becker et al. (2006). However, in most cases, other verification techniques are required (like Becker et al., 2006).

		Wright	Tsai	**CHAM**	Bose	Darwin/FSP	PoliS	Arcade	Archware	CHARMY
Property modeling features	Component	Y	N	Y	Y	Y	Y	N	Y	Y
	Interface	Y	N	N	N	Y	N	N	Y	N
	Connector	Y	N	N	Y	N	N	N	Y	N
	Interface	Y	N	N	N	N	N	N	Y	N
	Types	Y	N	N	Y	Y	N	N	Y	N
	Configuration	Y	Y	Y	Y	Y	Y	Y	Y	Y

		AEmilia	SAM	**Garlan et al.**	**CBabel**	**Zanolin et al.**	**AutoFOCUS**	**Fujaba**	Lfp
Property modeling features	Component	Y	Y	Y	Y	Y	Y	Y	Y
	Interface	Y	N	N	N	N	Y	Y	N
	Connector	Y	Y	N	N	Y	N	Y	Y
	Interface	Y	N	N	N	N	N	N*	N
	Types	N	N	N	Y	N	N	N*	Y
	Configuration	Y	Y	Y	Y	Y	Y	Y	Y

Fig. 15. SA property modeling features. A "N" means that the SA property language of the selected approach (see columns) cannot express properties about the selected architectural element (see rows). (* in Fujaba, properties for types, interfaces, and all other modeling artefacts can be specified by structural specifications as in Becker et al. (2006)).

all the others refer only to component instances. All the surveyed approaches permit one to specify configuration properties.

4.1.5. SA property – type of property

This section focuses on the properties to be checked by existing model checking SA approaches, such as safety, liveness, compatibility, and equivalence.

Most of the surveyed approaches support both safety and liveness. A *safety* property expresses that "something (bad) never happen". A *liveness* property expresses that "something (good) will eventually happen".

CBabel focusses only on safety, while PoliS focus only on liveness. Tsai, Garlan et al., and Fujaba verify timed temporal properties for real-time systems. All the others focus on both safety and liveness.

Other SA properties aim at checking *compatibility* or *equivalence*. Compatibility properties are aimed to verify that components/connectors collaborate to achieve the desired interactions. Equivalence properties, instead, aims at verifying component behaviors equivalence.

Wright, Archware and Æmilia can deal with both temporal properties and equivalence checking. Archware can verify not only general temporal properties, such as safety and liveness, but also compatibility or equivalence between different components or connectors. Æmilia can verify safety properties, compatibility, and inter-operability between different components. The properties that can be verified by Wright are compatibility checking (port-roles and instance-style) and assertion checking for deadlock analysis of finite CSP processes.

4.1.6. Model checking SA inputs – tool support

Through the *tool support* attribute associated with the model checking SA *input* entity, we investigate if a tool exists for modeling the SA and SA properties. (This attribute, while already partially analyzed in Medvidovic and Taylor (2000), is now applied to a different set of architecture description languages and model-based notations, focusses on a different set of attributes, and classifies SA property specifications, not considered in Medvidovic and Taylor (2000)).

Specifically, we indicate whether a tool is *available* (we assume a tool is available if at least a prototypal version is available for download), *internal* (in case the authors refer to an internal prototype not publicly available), or *not existent*, and what feature it supports (i.e., specification of the SA structure, specification of the SA behavior), and the date of the last available release.

Figs. 16 and 17 show the main findings of tool support parameter.

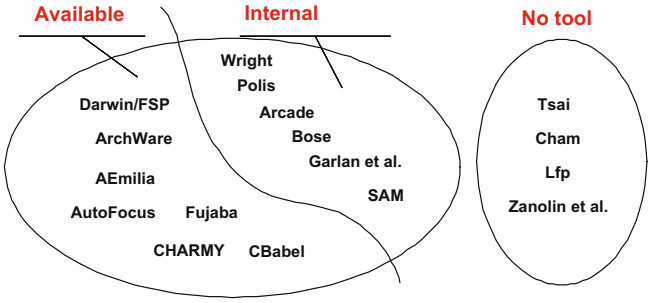

Fig. 16. Tool support for SA and SA property specification.

4.1.6.1. Available Tools for SA and SA property specification. Seven modeling tools are classify as available (see Fig. 16). They are the SA Assistant/LTSA (used by Darwin/FSP),[10] the ArchWare tool[11] (for ArchWare), the CHARMY editors (used by CHARMY),[12] TwoTowers[13] (used by Æmilia), AutoFOCUS editors[14] (used by AutoFOCUS), the Fujaba editor[15] (used by Fujaba), and CBabel.

CBabel provides some prototypal tool[16] for SA and SA property specification.

AutoFOCUS and CHARMY provide tool support for writing temporal properties in a graphical way. Fig. 18 shows the AutoFOCUS tool support for SA property specification. CHARMY uses the PSC notation for supporting temporal properties specifications in LTL.

4.1.6.2. Internal tools. Wright, Bose, PoliS, Arcade, SAM, and Garlan et al., refer in their papers to tools used internally to automate the SA specification or property steps. However, since those prototypes not publicly available, we refer to them as internal tools.

4.1.6.3. No tool. Tsai, CHAM, Lfp, and Zanolin et al. do not provide any tool support for SA and SA property specification. Tsai and CHAM do not refer to any specific model checking engine, and they do not even provide any modeling tool support. Lfp does not provide tool support, that is in the future work list. In Zanolin et. al, even if the authors make use of LSC for property modeling, they do not use the existing play engine tool (Harel and Marelly, 2003) for LSC modeling.

[10] Downloadable at http://www.doc.ic.ac.uk/ltsa/darwin/.
[11] Downloadable at http://www-valoria.univ-ubs.fr/ARCHLOG/ArchWare-IST/.
[12] Downloadable at http://www.di.univaq.it/charmy/.
[13] Downloadable at http://www.sti.uniurb.it/bernardo/twotowers/.
[14] Downloadable at http://autofocus.in.tum.de/autofocus-bin/download.
[15] Downloadable at http://wwwcs.uni-paderborn.de/cs/fujaba/downloads/index.html.
[16] Downloadable at http://www.ic.uff.br/cbraga/vas/cbabel-tool/.

	Wright	Tsai	CHAM	Bose	Darwin/FSP	PoliS	Arcade	Archware	CHARMY
Tool for SA specification	Wright	-	-	Rationale tools	SA Assistant and LTSA	PoliSMC	Prototype	ArchWare/ADL compiling tools	The Charmy Topology and state machine editor
Tool for Property specification	Wright	-	-	Rationale tools	Property editor in LTSA	PoliSMC	Prototype	ArchWare/AAL model-checking tools	The Charmy PSC editor
Type of tool	internal	-	-	internal	available	internal	internal	available	available
Structure/behav	B	B	B	S & B	S & B	S & B	S & B	S & B	S & B
Last release date	-	-	-	-	June 2006	-	-	Dec 2004	February 2006
Version	-	-	-	-	Version 3.0	-	-	Version 1.0	Charmy 2.0

	AEmilia	SAM	Garlan et al.	CBabel	Zanolin et al.	AutoFOCUS	Fujaba	Lfp
Tool for SA specification	AEmilia Compiler in TwoTower	prototype	prototype	Cbabel tool	-	AutoFOCUS	Component diagrams and RT statecharts editors in Fujaba	-
Tool for Property specification	Property editor in TwoTower	prototype	prototype	Cbabel tool	-	Property editor in MCUI	Component diagrams and RT statecharts editors in Fujaba	-
Type of tool	available	internal	internal	available	-	available	available	-
Structure/behav	S & B	S & B	B	S & B	B	S & B	S & B	S & B
Last release date	January 2006	-	-	Nov 2004	-	Oct 2005	August 2007	-
Version	TwoTowers 5.1	-	-	Cbabel prototype	-	Version 2	Fujaba Tool Suite 5.0.4	-

Fig. 17. Tool support for SA and SA property specification – detailed.

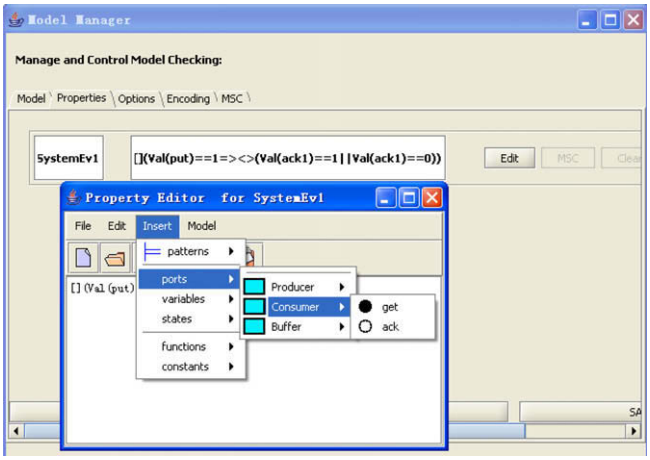

Fig. 18. A screenshot of AutoFOCUS tool supporting for SA property, which shows that the tool can help software architects to specify the SA property concerning ports, variables, states, etc. in SA using the pre-defined patterns.

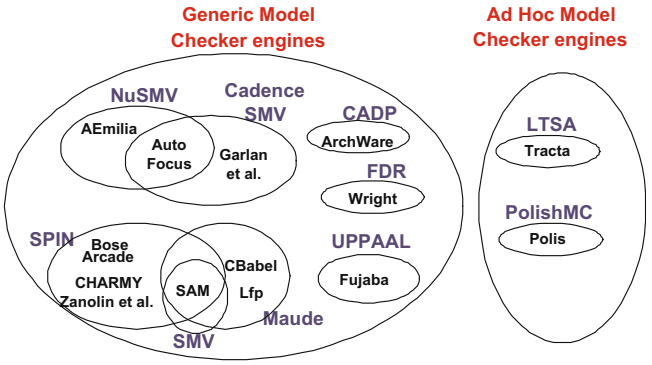

Fig. 19. Generic or ad hoc model checker engines.

4.2. Goal 1 – model checking software architecture: the computation

So far, this study has focussed on the inputs of model checking SA approaches. This section, instead, will focus on the engines used by the surveyed approaches for model checking the architectural inputs. According to the surveyed overview provided in Section 3.2 and summarized in Fig. 2, this section focusses on the model checking engines utilized by the model checking SA techniques.

4.2.1. Model checking engines utilized in the model checking SA approaches

Many model checkers are employed in the existing model checking SA approaches. Some of them are pre-existing, some others are created ad hoc. The framework will analyze the main characteristics of those model checkers utilized in the surveyed model checking SA techniques, by making an explicit distinction between *existing model checking engines*, and *ad hoc engines*. A third category is created for those approaches not referring to any specific engine. Fig. 19 summarizes how the surveyed approaches fit into this category.

4.2.1.1. Reuse of existing engines. Most approaches make use of existing model checking engines, such as FDR (Formal System Ltd. Fdr2, 2005), SMV (McMillan, 1993), SPIN (Holzmann, 1997), Maude (CLavele et al., 2004), UPPAAL (UPPAAL, 2005) and CADP (INRIA, 2007), when implementing the model checking SA techniques.

Wright makes use of the (CSP) model checker FDR version 1.2. Properties are specified in CSP. The properties that can be verified are compatibility checking (port-role and instance-style) and assertion checking for deadlock analysis of finite CSP processes. FDR supplies some compression functions for reducing the number of states and transitions.

Bose, Arcade, CHARMY, *and Zanolin et al., and SAM* make use of *SPIN* as their model checker engine. SPIN (Holzmann, 1997) is a tool designed for the design and verification of distributed systems. Its specification language is PROMELA, whereas its semantics model is based on finite automata. Correctness requirements can be expressed directly in LTL. LTL formulas are automatically translated into PROMELA never-claims, which represent the Büchi automaton corresponding to the negation of these formulas. SPIN performs model checking on-the-fly. It uses an efficient depth-first search algorithm that is compatible with all modes of verification supported by the tool, for example exhaustive search, bit-state hashing and partial-order reduction techniques. These techniques, together with state compression, are used for dealing with large state spaces.

AutoFOCUS, Garlan et al., SAM, and Æmilia model checking SA approaches make use of the *SMV* family of model checking engines for formal verification purposes. *SMV* has three versions: the original SMV (McMillan, 1993), *Nu-SMV* (ITC-irst, 2005) and *Cadence SMV* (CSMV, 2000). The original SMV is a tool for checking finite state systems against specifications in the temporal logic CTL. It has been developed by researchers at Carnegie Mellon University (CMU), it supports a flexible specification language, and uses an OBDD-based symbolic model checking algorithm for efficiently checking whether CTL specifications are satisfied by the system.

MC Engine Name	Used by	Input languages	Supported Logics	Types of Property	Reduction method	Specific Features	Type of Checking
FDR	Wright	CSP	CSP	assertion checking , compatibility checking	compression functions	Explicit connector	Model Checking and Equivalence Checking
SPIN	Bose, Arcade, Zanolin et al., CHARMY	PROMELA (CSP-like)	LTL or Buchi automaton	Safety and liveness	POR, on the fly, state compression	Distributed Systems	Model Checking
PoliMC	PoliS	PoliS	PTL	Safety and liveness	Local search	Based on multiset rewriting system	Model Checking
LTSA	Darwin/FSP	FSP	FLTL	Safety, liveness and fairness	Compositional minimization	Concurrent systems	Model Checking
SMV	SAM	SMV languages	CTL	Safety, liveness and fairness	OBDD	Hardware and embedded system	Model Checking
NuSMV	Aemilia, AutoFocus		CTL or LTL		OBDD, BMC	Customizable and extensible	Model Checking
Cadence SMV	Garlan et al., AutoFocus		CTL, LTL or SMV		OBDD, compositional and abstraction,	Refinement verification	Model Checking and Theorem Proving
CADP	ArchWare	LOTOS (and others)	Temporal logics and modal mu-calculus	Compatibility checking,safety, liveness and fairness	On the fly, BDD, compositional verification	Probabilistic Model Checker	Model Checking and Equivalence Checking
Maude	Cbabel, Lfp	Maude	LTL	Safety and liveness	On the fly, BDD	Based on multiset rewriting system	Model Checking
UPPAAL	Fujaba	Timed-automata	TCTL	Safety and timed liveness	New data structures and memory reductions, BMC	Real-Time	Model Checking

Fig. 20. Comparison of different model checker engines.

Nu-SMV is a re-implementation of SMV, and is aimed at being customizable and extensible. Nu-SMV extends SMV with LTL model checking and bounded model checking. It supports bounded model checking. Cadence SMV is an entirely new model checker based on compositional system and views abstraction as a way to fight state explosion problem. SMV's properties are specified in CTL. *Nu-SMV* and *Cadence* support both LTL and CTL formulae.

CBabel, Lfp, and SAM make use of *Maude* as their common model checking engine. Maude (CLavele et al., 2004) is a tool which uses object theory of rewriting logic to model concurrent and distributed systems and to check some temporal properties using its pre-defined modular model checker. Maude engine's input is based on rewriting logic which is a formalization of static and dynamic behavior of distributed systems. Maude uses LTL as the properties specification language.

ArchWare uses *CADP* (INRIA, 2007). CADP is a toolbox for protocol engineering, which offers a wide range of functionalities, from interactive simulation to the most recent formal verification techniques. It allows for model- and equivalence checking. The input language of CADP is LOTOS, but it also accepts other input languages such as finite state machines and networks of communicating finite state machines. CADP can support several model checkers for various temporal logics and modal mu-calculus, and several verification algorithms including exhaustive verification, on-the-fly verification, symbolic verification using BDD, and compositional verification based on refinement.

Fujaba uses *UPPAAL* (UPPAAL, 2005) for model checking its specifications. UPPAAL is a model checker engine for real-time systems: the tool is designed to verify systems that can be modeled as networks of timed automata extended with integer variables, structured data types, and channel synchronization. UPPAAL's input language is timed I/O automata or timed interface automata. UPPAAL's properties are formalized as a subset of TCTL formulae.

4.2.1.2. Ad hoc engines. Darwin/FSP and Polis, differently from all the other model checking SA approaches, provide their own specific model checkers for SA-level model checking.

Darwin/FSP uses LTSA (the labeled transition system analyzer) (Magee et al., 1999). LTSA allows for model animation and model checking, deadlock detection, liveness properties checking. It uses the finite state process (FSP) algebra as its input language. Properties are specified in FLTL (Fluent LTL), which is designed to provide a uniform framework supporting both action- and fluent-based property specifications, as well as their combination. LTSA supports compositional minimization verification.

PoliS makes use of the *PoliMC* engine. PoliMC is an ad hoc engine designed for its input language (i.e., PoliS). It supports checking a simple logic called *PTL* (Polis Temporal logic) which is a CTL dialect to represent properties. The main difference between PTL and CTL is that PTL is based on multi-sets, thus all formulas are evaluated in special tuple spaces, referred as contexts. The tool evaluates formulae in a special tuple space, thus reducing the search area of verification.

4.2.1.3. No engines. CHAM and Tsai, while focussing on theoretical aspects of model checking SA, do not rely on any specific model checking engine.

A comparison of different model checker engines is shown in Fig. 20. Note that, while this survey refers to and provides information on the model checking engines used in the surveyed approaches, further details are out of scope for this work. Interested readers can refer to, for example (Clarke et al., 2000; Holzmann, 1997; Peled et al., 2009).

4.3. Goal 1 – model checking software architecture: the translation

Not all SA inputs can be directly processed by the model checking engines. In many cases, a translation activity is needed to make the input processable by the model checker.

Since formal ADLs, formal general purpose notations, and model-based notations for model checking SA specifications have their underlying formal semantics (see Fig. 9), the translation between the architectural language and the model checker input is straightforward and consists in a mapping among concepts between two formal languages. Instead, in case of informal languages, the lack of a clear semantics associated to the input elements, might generate ambiguities during the translation process. Indeed, the way a model from the software architecture domain is mapped into the associated in the model checking domain is a subjective matter, and can be realized in different ways by different engineers.

This section discusses the *translation type and automation* (i.e., what kind of translation is required, if any, and tool support for its automation).

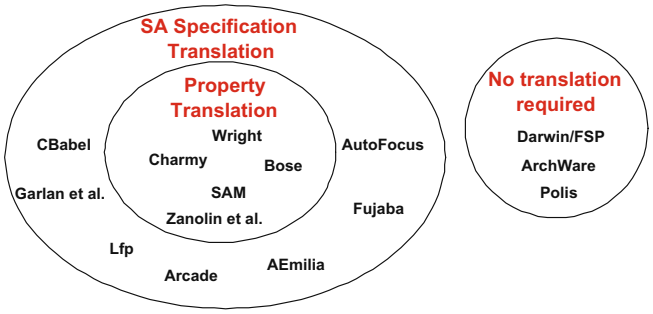

Fig. 21. Input translation.

4.3.1. From SA inputs to MC inputs: translation and automation

This section classifies the model checking SA approaches according to the need for translating the SA specification or the SA property into the model checker inputs. There are five possible alternatives: *no translation is required*, either the *SA specification* or the *SA property* needs translation, *both* need to be translated, and *it does not apply*.

Fig. 21 summarizes the main findings of input translation.

No translation required: Darwin/FSP, PoliS and Archware are the only model checking SA approaches whose architectural specification inputs (both SA specification and SA properties) do not require any translation. Darwin/FSP and PoliS have their internal, ad hoc, model checking engine and for this reason the SA specification/property notation is the input taken from the model checking engine. In ArchWare, the ArchWare ADL compiler produces a set of execution traces on which the temporal properties are evaluated. The AAL properties are checked on the execution traces by using the ArchWare model checker. So the SA and the properties are not translated, but they are handled by specific tools developed within the project.

SA specification translation: many model checking SA techniques require a translation of the SA specification into an input readable by the model checker. A CSP-like Wright specification is translated into CSP by the Wright tool. A DRA model in Arcade is automatically translated into Promela by a prototype tool. Petri Net in SAM is automatically translated into the SMV language by a prototype tool. User-defined state machines in Garlan et al. are also translated into SMV language by a prototype tool (not available). The CBabel ADL is translated into the Maude language by the CBabel tool. The Æmilia tool responsible for the translation process is TwoTowers 5.1. The model checking user interface (MCUI) tool controls the translation process of AutoFOCUS. The Fujaba SA specification is also translated into timed automata through the Vanilla tool. A plug-in in the CHARMY framework controls the translation from state machines to Promela. The SA specification of Bose is translated into Promela by SODA, which is the Rational tools for model capture (Rational Rose) and document generation.

Lfp and Zanolin et al. are the only two model checking SA techniques that, while requiring an SA translation, do not provide any translation tool support.

Property translation: By focussing on properties and their translation, we notice that Wright Bose, ArchWare, CHARMY, SAM, and Zanolin et al. require one to translate the property specification input into the model checking input, because all of those approaches make use of dedicated property specification languages. A CSP-like Wright property specification is translated into CSP by the Wright tool. Pre- and post-conditions in DRA models are translated into

	SA Specification Translation			Property Specification Translation			
	SA spec Input	**MC SA input**	**SA Specification Translation Tool**	**Property spec input**	**MC property input**	**Property Specification Translation Tool**	**Translation type**
Wright	Wright(CSP-like)	CSP	Wright	CSP-like	CSP	Wright	SA specification and Property
Tsai	Extended Horn logic clause	-	-	Mu-calculus	-	-	Does not Apply
CHAM	CHAM	-	-	CTL*	-	-	Does not Apply
Bose	UML Model	Promela	SODA	Activity Diagram	LTL	SODA	SA specification and Property
Darwin/ FSP	Darwin and FSP	FSP	-	FLTL	FLTL	-	No translation required
PoliS	PoliS	PoliS	-	PTL	PTL	-	No translation required
Arcade	DRA	Promela	Prototype (unavailable)	Pre and post conditions	Buchi Automata	Prototype (unavailable)	SA specification and Property
Archware	Archware ADL	LOTOS (and others)	-	AAF/MC	Temporal logics and modal mu-calculus	-	No translation required
CHARMY	State and Sequence diagram	Promela	A plus-in in CHARMY	PSC	LTL/ Buchi Automata	A plus-in in CHARMY called PSC2BA	SA specification and Property
AEmilia	PADL	SMV languages	TwoTowers 5.1	LTL	CTL, LTL	-	SA specification
SAM	Petri net	SMV languages	Prototype (unavailable)	LTL	CTL	Prototype (unavailable)	SA specification and Property
Garlan et al.	State machine model	SMV languages	Prototype (unavailable)	LTL	CTL, LTL, SMV	-	SA specification
CBabel	CBabel	Maude	CBabel tool	LTL	LTL	-	SA specification
Zanolin et al.	UML statechart diagram	Promela	Manual	LSC	Buchi Automata	Manual	SA specification and Property
Auto FOCUS	Four different models	SMV languages	MCUI	CTL, LTL	CTL, LTL, SMV	-	SA specification
Fujaba	UML 2.0 component diagrams, RT statecharts	Timed-automata	Vanilla tool	TCTL	TCTL	-	SA specification
Lfp	LfP	Maude	Manual	LTL	LTL	-	SA specification

Fig. 22. Mapping between MC SA inputs and model checkers inputs: details.

Promela by a prototype tool. Property specification in LSC used in Zanolin et al. is translated into Büchi automata: however, there is no tool to support this translation. Bose's activity diagrams are automatically translated into Büchi automata through the SODA tool. PSC in CHARMY is translated into Büchi automata by a CHARMY plug-in. Properties in Darwin/FSP, PoliS, Arcade, Æmilia, Garlan et. al, CBabel, AutoFocus, Fujaba, and Lfp, instead, *do not require any translation*, being them specified directly through the model checker native property language.

SA specification and Property translation: Among the approaches, we note that Bose, SAM, CHARMY, Arcade and Zanolin et al. require translation of both SA inputs (SA specification and SA property) into the model checking inputs.

Does not Apply: Since Tsai and CHAM do not refer to any specific model checking engines, this classification does not apply.

Fig. 22 provides details about the mapping.

4.4. Goal 1 – summary

Different model checking SA approaches use different input languages. The SA specification can be written in a formal way, through model-based notations, or by using both formal and model-based notations. The choice among those different solutions relies on the degree of formality required versus the practicality of the proposed methods. As shown in the previous sections, the surveyed approaches typically rely on formal textual notations, while more recent approaches make use of model-based notations. SA properties are typically specified using existing languages, even if some new formal languages or graphical abstractions have been provided by some approaches.

The surveyed approaches also provide different support for architectural element modeling. This strongly affects the formal verification a software architect can perform: if, for example, an architect is interested in checking properties about connectors and interfaces, the SA specification needs to explicitly cover such concepts and the property language needs to be adequate for their specification. For this reason, information about components, connectors, interfaces, types and instances, semantics, and configuration has been collected. It can be noted, for example, that components and connectors can be explicitly or implicitly specified.

Further considerations can be provided by focussing on those model checking SA approaches making use of the same engine. Bose, Arcade, CHARMY, and Zanolin et al. use SPIN as their model checking engine, thus they all use Promela and LTL formulae (or Büchi automata) as the model checking input languages. Among them, Bose, CHARMY, and Zanolin et al. use a model-based notation, while Arcade a formal DRA specification. Zanolin et al. has no tool support (while all the other three approaches provide some tool support). *Æmilia, Garlan et al., and AutoFocus* rely on SMV-based model checkers. While Garlan et al. is only partially tool supported, both Æmilia and AutoFocus are supported by tools. Properties are modeled in the model checker property language. *CBabel and Lfp* make use of Maude. They have quite similar characteristics, even if CBabel provides some tool support, not provided by Lfp.

When translating the SA inputs into the model checking inputs, there are two typical scenarios: the model checking SA approach

	SA Specification				SA Property				Translation and Computation			
	What form of SA description is supported? Which architectural language?	Which are the features provided by the SA language?	Which types of systems can be specified and verified?	Is the SA specification supported by a tool?	What form of SA property description is available?	Which are the features provided by the SA property language?	Which are the type of properties that can be specified?	Is the SA Property specification supported by a tool?	SA Input translation required?	Property translation required?	Is the translation automated?	MC Engine
Wright	Wright	Components, connectors, configurations, roles, glues	Architecture with explicit connector	Internal tool	CSP-like	Comp, Comp Interf, Con, Con Inter, Config	Compatibility checking, deadlock analysis	Internal tool	Y	Y	Y	FDR
Tsai	Extend ed Horn logic clause	Non explicity	Complex and real-time systems	No	Mu-calculus	Config	Safety and liveness	No	Does not apply	Does not apply	-	-
CHAM	CHAM	Molecules, reaction rules	Dynamic and reconfigurable SA	No	CTL*	Comp, Config	Temporal property	No	Does not apply	Does not apply	-	-
Bose	UML Model	Components, mediator components	A special SA style called mediated architecture	Internal tool	Activity diagram	Comp, Con, Types, Config	safety and liveness	Internal tool	Y	Y	Y	SPIN
Darwin/ FSP	Darwin and FSP	Composite and primitive Components, interfaces, ports, channels	Concurrent and distributed SA	Available tool	FLTL	Comp, Comp Interf, Type, Config	Safety, liveness	Available tool	N	N	-	LTSA
PoliS	PoliS	Tuple space, rules	Structure evolves dynamically at run-time	Internal tool	PoliS TL	Comp, Config	Liveness	Internal tool	N	N	-	PolisMC
Arcade	DRA	Non explicitly	Domain reference model	Internal tool	Pre-condition and post-condition	Config	Safety and liveness	Internal tool	Y	Y	Y	SPIN
Archware	Archware ADL	Components, connectors, communications	Evolvable SAs at runtime	Available tool	Archware AAL	Comp, Comp Interf, Con, Con Inter, Type, Config	Safety, liveness,	Available tool	N	N	-	CADP
CHARMY	State and Sequence diagram	Components, connectors, channels	Distributed system with concurrent and complex interactions	Available tool	PSC	Comp, Config	Scenarios-based temporal properties	Available tool	Y	Y	Y	SPIN
AEmilia	PADL	Architecture element type, architecture topology	A family of architecture	Available tool	LTL	Comp, Comp Interf, Type, Config	Safety, such as deadlock	Available tool	Y	N	Y	NuSMV
SAM	Petri net	Components, connectors, constraints	Distributed system with hierarchical SA,	Internal tool	LTL	Comp, Con, Config	Safety and liveness	Internal tool	Y	Y	Y	SMV
Garlan et al.	State machine model	Components, event, styles	Publish-subscribe architectures	Internal tool	LTL	Comp, Config	Time properties in LTL	Internal tool	Y	N	Y	Cadence SMV
CBabel	CBabel	Modules, ports, links	Evolvable and reconfigurable architectures	Available tool	LTL	Comp, Type, Config	Safety	Available tool	Y	N	Y	Maude
Zanolin et al.	UML statechart diagram	Event, middlewares, components	Publish-subscribe architectures	No	LSC	Comp, Con, Config	Global temporal properties	No	Y	Y	Y	SPIN
Auto FOCUS	Four different views	Components, ports, links	Reliable embedded SA	Available tool	LTL, CTL	Comp, Comp Interf, Config	Safety, liveness and user-defined temporal properties	Available tool	Y	N	Y	NuSMV and Cadence SMV
Fujaba	UML 2.0 component diagrams, RT statecharts	Components, port, connector and patterns	Distributed, embedded real-time systems and mechatronic systems	Available tool	LTL, CTL	Comp, Comp Interf, Config	Safety, timed liveness	Available tool	Y	N	Y	UPPAAL
Lfp	LfP	Classes, ports, medias	Distributed embedded systems control structure	No	LTL	Comp, Con, Type, Config	Liveness	No	Y	N	Y	Maude

Fig. 23. Summary of the goal 1 attributes.

relies on *external* model checking engines (which implies that the SA specifications and/or SA properties have to be translated into the model checker input languages and successively checked), or the model checking SA approach comes with *its own* model checking engine (which means that the SA specification and SA properties are used directly as the model checker input). While considering formal or model-based notations for SA specification, the translation process is similar and can be typically automated. In most cases, SA properties are already expressed in the model checker input language, so no translation is required.

A summary about the parameters related to the first goal is provided in the table in Fig. 23.

4.5. Goal 2 – model checking software architecture: usability

According to the ISO/IEC 9126-1:2001 standard,[17] usability is characterized by *"a set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users*". The ISO/IEC 9126 further decomposes the usability into more concrete sub-characteristics, such as *understandability*, *learnability*, *operability*, and *attractiveness*. According to the ISO 9241-11[18] standard, instead, usability can be defined as *"the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use*." The IEEE Std. 610.12-1990[19] refers to usability as *"The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component*."

In the following, we focus on some usability sub-characteristics that we have been able to analyze in model checking SA techniques. These are: *understandability*, *learnability*, and *operability*. We now discuss how the model checking SA approaches can be classified according to those attributes.

4.5.1. Understandability

According to the ISO/IEC 9126-1:2001 standard, understandability represents the capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use. In Reference (Medvidovic and Taylor, 2000) it is remarked that "explicit configuration" ADLs (e.g., those modeling both components and connectors separated from configurations) have the best *potential* to facilitate understandability of architectural structure. Moreover, "semantically sound" graphical notations (i.e., those interchangeable with the textual notation) is another means of achieving understandability.

Based on those sources, we identified the following parameters to be considered: *explicit configuration* and *graphical notations*.

4.5.1.1. Explicit configuration. As already analyzed in Section 4.1.2 (under the *Configurations modeling* paragraph), Wright, Ämilia, Archware, CHARMY, and AutoFOCUS are the only architectural description and modeling languages that provide an explicit model of the architecture configuration separately from components and connectors. In those languages, configuration is considered as a first-class architectural concept, independently defined from other modeling features. All the other architectural languages provide an implicit description of the architecture configuration. Therefore, based on Medvidovic and Taylor (2000), Ämilia, Archware, CHARMY, and AutoFOCUS are the techniques with the potential to facilitate understandability.

4.5.1.2. Graphical notations. Based on Reference (Medvidovic and Taylor, 2000), "semantically sound" graphical notations improve usability. In the specific domain of interest, we analyze those model checking SA techniques providing a semantically sound graphical notations for expressing the architecture and architectural properties.

As discussed in Section 4.1.1, some are the architectural descriptions providing a graphical notation for SA modeling: Bose, CHARMY, AutoFocus, Fujaba, Zanolin et al., and Lfp provide such a feature. However, according to Reference (Medvidovic and Taylor, 2000), only "semantically sound" graphical notations improve understandability. While looking at the model-based notations listed above, we can recognize that only AutoFocus and Fujaba provide interchange between textual and graphical specifications (since they both provide either graphical and textual notations, with a well formalized semantics), thus can facilitate understandability.

As discussed in Section 4.1.3, these are property specification languages that provide a semantically sound graphical notation for expressing properties: the property sequence chart graphical notation for properties in CHARMY, the Live Sequence Charts notation used by Zanolin et al., and the Activity diagrams utilized in Bose.

While considering both the architectural and property descriptions, we note that Bose, CHARMY and Zanolin et al. are the only three surveyed techniques providing a graphical representation of both properties and architecture descriptions. However, their graphical notation for SA specification cannot be considered sound.

4.5.2. Learnability

The ISO/IEC 9126-1:2001 standard defines learnability as *"The capability of the software product to enable different users to learn how to use it*". It is then remarked that learnability includes several attributes, being *comprehensible input and output* one of those. The IEEE Std. 610.12-1990 also points out that the ease to prepare the inputs and interpret the outputs is a fundamental part of usability.

Accordingly, we identified the *output interpretation* of the model checking SA process as the main parameter to be considered. (While this parameter could have been discussed under the understandability sub characteristics, the reasoning above convinced us to discuss this attribute under learnability).

4.5.2.1. Output interpretation. Understanding the counter-example obtained by running the model checking SA analysis and then using it for improving the SA model can be an issue, as already discussed in Goldsby et al. (2006). In fact, if the output is expressed in terms of the model checking technical language, it can become difficult to comprehend by a software architect. By analyzing the surveyed model checking SA approaches, we noticed that only three approaches, that are Arcade, CHARMY, and Autofocus, provide a more readable output with respect to the one provided by the model checking engine. Specifically, *Arcade* provides a graphical representation of the output called Architecture Trace Diagram (ATD) that is obtained by mapping a SPIN counter-example back on the architectural DRA notation. An ATD is expressed in terms of service invocations, data and event exchanges, assignments to attributes, satisfaction of pre-conditions and post-conditions, and state transitions from the DRA behavioral model, familiar to the architect and domain expert. Any property violation is reported in the ATD formalism, and annotated with a message indicating the type of error and what caused it. CHARMY provides a plug-in called SIMULATEIT which allows to visualize the simulation outputs in terms of state machines and scenarios. It allows a complete hiding of SPIN simulation technical information. *Autofocus*

[17] ISO/IEC 9126. Software Engineering – Product quality. Part 1: quality model, June 2001.

[18] ISO 9241. Ergonomics Requirements for Office with Visual Display Terminals (VDTs). Part 11: Guidance on usability, 1998.

[19] IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology, December 1990.

uses Message Sequence Charts (MSCs) to visualize the generated counter-examples from the SMV engine. Components, ports, messages with corresponding variables are shown in the generated MSC. Three types of MSCs can be generated: "black-box MSCs", hiding the intra-components communication while focussing on inter-components communication, "component MSCs", showing messages internal to the selected component, and "basic MSCs", showing all atomic components in the verified system.

4.5.3. Operability

Operability is defined in ISO/IEC 9126-1:2001 standard as the capability of the software product to enable the user to operate and control it. According to Reference (Zeiß et al., 2007), the need for an external, non-automatable actions strongly diminish operability. Based on this comment, we selected *automation* as an operability parameter.

4.5.3.1. Automation. Fig. 24 summarizes partial results already discussed in Sections 4.1.6, 4.2.1, and 4.3. From the table we note that Tsai and CHAM are those providing the lower support for automation: they do not have an associated model checking engine, and do not provide means for the input modeling and translation. Then, Zanolin et al. and Lfp while having associated a model checker, do not provide tool support for input modeling and translation. (In the specific case of Lfp, the property specification translation is not required). Darwin/FSP, ArchWare, Æmilia, CBabel, AutoFocus, CHARMY and Fujaba, instead, are techniques offering a higher degree of automation.

4.6. Goal 2 – model checking software architecture: reliability

According to the ISO/IEC 9126-1:2001 standard, reliability refers to "*the capability of the software product to maintain a specified level of performances when used under specified conditions*" and it is characterized by *maturity*, *fault tolerance*, and *recoverability*. In Babar and Gorton (2004), reliability is defined as the maturity of a method and its validation. In Reference (Babar et al., 2004), the four maturity levels are: (i) *inception* (method recently surfaced and has not been validated yet for example), (ii) *development* (method is being validated with results appearing in credible literature), (iii) *refinement* (method has been validated in various domains and refinements are continuously being made to various techniques), and (iv) *dormant* (method has one or few publication or no validation has been performed).

In the context of our model checking SA research, we classify the approaches as follow:

- *Dormant:* since Tsai, Bose, Arcade, PoliS, Garlan et al., CBabel, Lfp, CHAM and Zanolin et al. have not being validated with large and industrial case study applications and/or no further research is currently carried on, we classify those methods as dormant;
- *Refinement:* Wright, SAM, CHARMY, Darwin/FSP, Æmilia, AutoFOCUS, Archware and Fujaba pertain to this category. Wright has been successfully used in HLA case studies. SAM has been validated in many case studies such as security protocols (Dai et al., 2004). Recently, it has been extended to model service-oriented architecture. CHARMY has been validated through complex industrial studies and recently extended with new testing and simulation features. Darwin/FSP has been applied and validated through different case studies, such as alternating-bit protocol, and a Reliable Multicast Transport Protocol (RMTP). New features are currently being introduced. AutoFOCUS has been validate through several case studies and it is currently being updated through AUTOFOCUS 2 in the AutoPLACE project. Fujaba has been applied to model check mechatronic systems and its framework is under continuous evolution. Archware has been applied to the FKMS system for distributed database management and the Agile integrated process system. Æmilia has been applied to many small-size case study and has been recently extended with security analysis and model checking features.

Fig. 25 also reports whether the approach has been applied to single and small (S&S) or multiple and industrial (M&I) case studies.

4.7. Goal 2 – model checking software architecture: Scalability

In Bondi (2000) scalability is defined as a desirable property of a system, a network, or a process, which indicates its ability to either

		Wright	Tsai	CHAM	Bose	Darwin/FSP	PoliS	Arcade	Archware	CHARMY
Input Automation	Tool for SA specification	Wright	not available	not available	Rationale tools	SA Assistant and LTSA	PoliSMC	Prototype	ArchWare/ADL compiling tools	The Charmy Topology and state machine editor
	Tool for Property specification	Wright	not available	not available	Rationale tools	Property editor in LTSA	PoliSMC	Prototype	ArchWare/AAL model-checking tools	The Charmy PSC editor
	Type of tool	internal	not available	not available	internal	available	internal	internal	available	available
Traslation Automation	SA Specification Translation Tool	Wright	translation not required	translation not required	SODA	translation not required	translation not required	Prototype (unavailable)	translation not required	A plug-in in CHARMY
	Property Specification Translation Tool	Wright	translation not required	translation not required	SODA	translation not required	translation not required	Prototype (unavailable)	translation not required	The PSC2BA plug-in in CHARMY
Model Checking Automation	Tool for Model Checking	FDR	not available	not available	SPIN	LTSA	PolisMC	SPIN	CADP	SPIN

		AEmilia	SAM	Garlan et al.	CBabel	Zanolin et al.	AutoFOCUS	Fujaba	Lfp
Input Automation	Tool for SA specification	AEmilia Compiler in TwoTower	prototype	prototype	Cbabel tool	not available	AutoFOCUS	Component diagrams and RT statecharts editors in Fujaba	not available
	Tool for Property specification	Property editor in TwoTower	prototype	prototype	Cbabel tool	not available	Property editor in MCUI	Component diagrams and RT statecharts editors in Fujaba	not available
	Type of tool	available	internal	internal	available	not available	available	available	not available
Traslation Automation	SA Specification Translation Tool	TwoTowers 5.1	Prototype (unavailable)	Prototype (unavailable)	Cbabel	not available	MCUI	Vanilla	not available
	Property Specification Translation Tool	translation not required	Prototype (unavailable)	translation not required	translation not required	not available	translation not required	translation not required	translation not required
Model Checking Automation	Tool for Model checking	NuSMV	SPIN/SMV/ Maude	Cadence SMV	Maude	SPIN	NuSMV/Cadence SMV	UPPAAL	Maude

Fig. 24. Automation in SA modeling, SA properties specification, translation, and model checking.

		Wright	Tsai	CHAM	Bose	Darwin/FSP	PoliS	Arcade	Archware	CHARMY
Reliability	Maturity of method	Refinement	Dormant	Dormant	Dormant	Refinement	Dormant	Dormant	Refinement	Refinement
	Methods validation	M & I	S & S	S & S	S & S	M & I	S & S	S & S	M & I	M & I

		AEmilia	SAM	Garlan et al.	CBabel	Zanolin et al.	AutoFOCUS	Fujaba	Lfp
Reliability	Maturity of method	Refinement	Dormant	Dormant	Dormant	Dormant	Refinement	Refinement	Dormant
	Methods validation	M & I	S & S	S & S	S & S	S & S	M & I	M & I	S & S

Fig. 25. Reliability of model checking SA approaches.

handle growing amounts of work in a graceful manner, or to be readily enlarged.

The scalability of model checking techniques is still prevented by the state explosion problem. As remarked by Gerald Holzmann in Holzmann (2002), no currently available model checking approach cannot suffer from this problem. State explosion occurs either in systems composed of (not too) many interacting components, or in systems where data structures assume many different values. The number of global states easily becomes intractable. To solve this problem, many methods have been developed by exploiting different approaches. They can be classified into two disjoint sets Caporuscio et al. (2004, 2008). The first set, which we call External Methods, includes techniques that operate on the input of the model checker (models), such as symmetry, hierarchical modeling and compositional reasoning. The second set, that we call Internal Methods, considers algorithms and techniques used internally to the model checker to efficiently represent transition relations between concurrent processes, such as Binary Decision Diagrams and partial order reduction techniques.

In the context of MC SA, scalability can be defined as the ability to handle large and complex SA specifications and to model check them. Thus, the external method is related to the *ability to specify complex architectures*: in this respect, we analyze whether the surveyed approaches provide a *hierarchical/compositional modeling* of the SA structure and behavior (according to Reference (Medvidovic and Taylor, 2000), compositionality strongly impacts scalability) and if they allow *textual* SA specifications (since textual specifications more easily scale with respect to graphical notations). About the internal methods, it is related to the approaches *ability to model check complex inputs*. In this respect, we consider the model checker support for *abstraction, reduction* and other state compression techniques.

Indeed, other parameters like the degree of automation can affect scalability. For this information, you may refer to Section 4.5.

Fig. 26 reports how the surveyed approaches fit into the proposed classification schema.

Based on the results shown in Fig. 26, we can conclude that Darwin/FSP, ArchWare, AutoFOCUS and Fujaba provide a more scalable input since they both provide hierarchial modeling and textual specifications. LfP provides the ability of hierarchically modeling LfP-components behavior. Tasi, CHAM, PoliS, Arcade, CHARMY, Æmilia, CBabel while not providing a hierarchial modeling, allow for a textual SA specification. Bose, SAM, Garlan et al., and Zanolin et al. are those techniques with the lower level of scalability.

We can also conclude that Tasi and CHAM do not have abstraction and reduction techniques, since they do not use any kind of model checker. *Bose, Arcade,* CHARMY, *and Zanolin et al., and SAM* make use of *SPIN* as their model checker engine. Consequently, they support the reduction techniques of SPIN model checker such as state compression, on-the fly, and partial order reduction. *AutoFOCUS, Garlan et al., SAM, and Æmilia* model checking SA approaches make use of the *SMV* family of model checking engines for formal verification purposes. The SMV model checker supports OBDD, etc. *CBabel, Lfp, and SAM* make use of *Maude* as their common model checking engine. Maude supports BDD and on-the-fly techniques. *ArchWare* uses *CADP* (INRIA, 2007), which supports compositional on-the fly. *Fujaba* uses *UPPAAL* (UPPAAL, 2005) for model checking its specifications. *UPPAAL* supports memory reduction technique. *Darwin/FSP* uses LTSA (the labeled transition system analyzer) (Magee et al., 1999). LSTA supports compositional minimization technique. *PoliS* makes use of the *PoliMC* engine, which supports local state searches.

4.8. Goal 2 – model checking software architecture: expressiveness

This parameter wants to classify model checking SA approaches based on their expressiveness. Expressiveness is here defined as the model checking SA technique ability to check architectural concepts (e.g., components, connectors, interfaces) specified at the architectural level. For this extent, we do compare the expressiveness of the SA modeling language with the expressiveness of the SA property language. The table in Fig. 27 summarizes the classification. Columns 2-6 interleave information about the SA expressiveness (i.e., ability to model architectural elements) and SA property expressiveness (i.e., ability to specify a property about

	Wright	Tsai	CHAM	Bose	Darwin/FSP	PoliS	Arcade	Archware	CHARMY
hierarchical/ compositional modeling	No	No	No	No	Yes	No	No	Yes	No
textual and graphical SA specifications	Textual	Textual	Textual	Graphical	Textual and Graphical	Textual	Textual	Textual	Textual and Graphical
Abstraction and reduction in model checkers	compression functions	-	-	SPIN supports state compression, on-the fly, and POR	LTSA supports compositional minimization	PolishMc supports local searches	SPIN supports state compression, on-the fly, and POR	CADP supports compositional on-the fly	SPIN supports state compression, on-the fly, and POR

	AEmilia	SAM	Garlan et al.	CBabel	Zanolin et al.	AutoFOCUS	Fujaba	Lfp
hierarchical/ compositional modeling	No	Yes	No	No	No	Yes	Yes	Yes
textual and graphical SA specifications	Textual	Graphical	Graphical	Textual	Graphical	Textual and Graphical	Textual and Graphical	Textual and Graphical
Abstraction and reduction in model checkers	NuSMV supports OBDD and BMC	SPIN, SMV, and Maude model checkers related abstractions and reductions	Cadence SMV supports composition and abstract	Maude supports BDD and on-the-fly	SPIN supports state compression, on-the fly, and POR	NuSMV and Cadence SMV support	UPPAAL supports memory reduction	Maude supports on-the-fly computation and BDD

Fig. 26. Scalability of model checking SA approaches.

Approach	Components/ Connectors		Interface		Type
	Component modeling / Component Property	Connector modeling / Connector property	Comp. Interface modeling / Comp. Interface Property	Conn. Interface modeling / Conn. Interface property	Component & Connector Types / type property
Wright	Yes/Yes	Yes/Yes	Yes/Yes	Yes/Yes	Yes/Yes
Tsai	No/No	No/No	No/No	No/No	No/No
CHAM	Imp/Yes	Imp/No	No/No	No/No	No/No
Bose	Yes/Yes	Yes/Yes	Yes/No	Yes/No	Yes/Yes
Darwin/FSP	Yes/Yes	Imp/No	Yes/Yes	No/No	Yes/Yes
PoliS	Imp/Yes	Imp/No	No/No	No/No	No/No
Arcade	No/No	No/No	No/No	No/No	No/No
Archware	Yes/Yes	Yes/Yes	Yes/Yes	Yes/Yes	Yes/Yes
CHARMY	Yes/Yes	Imp/No	No/No	No/No	No/No
AEmilia	Yes/Yes	Yes/Yes	Yes/Yes	Yes/Yes	Yes/Yes
SAM	Yes/Yes	Yes/Yes	Yes/No	Yes/No	No/No
Garlan et al.	Yes/Yes	Imp/No	No/No	No/No	No/No
Cbabel	Imp/Yes	Imp/No	Yes/No	No/No	Yes/Yes
Zanolin et al.	Yes/Yes	Yes/Yes	No/No	No/No	No/No
AutoFOCUS	Yes/Yes	Imp/No	Yes/Yes	No/No	No/No
Fujaba	Yes/Yes	Imp/No	Yes/Yes	Yes/Yes*	Yes/Yes*
Lfp	Imp/Yes	Yes/Yes	Yes/No	Yes/No	Yes/Yes

Fig. 27. Expressiveness of model checking SA approaches ("Imp" stays for implicit)(* if we consider the complete power of FUJABA RT as explained above).

an architectural element). The surveyed technique provide an adequate degree of expressiveness when a concept expressed by an architectural description (i.e., components, components' interfaces, connectors, connectors' interfaces, and types) can be expressed by an architectural property.

From the table we can notice that:

Explicit Component, Connectors, and types: only Wright, Bose, Archware, and Lfp allow for the explicit modeling and checking of both components, connectors, and types.

Explicit Component and Component's interface: only Wright, Darwin/FSP, Archware, Æmilia, AutoFOCUS and Fujaba support both component and component's interface checking.

Type-level checking: both Wright, Bose, Darwin, Archware, Æmilia, CBabel, and Lfp allow for both an instance-level and type-level checking (i.e., model checking not just the component/connector instances, but also properties about their types).

All explicit: only ArchWare and Wright permits to explicitly model and check both components, connectors, their interfaces, types and instances.

Component interface modeling and SA property: only in a few cases, (i.e., Bose, Lfp) the language supports component's interface modeling while it is not possible to write a property enabling its check. Specifically, Bose allows modeling interface for components and connectors, while its property specification language (i.e., activity diagrams) only allows one to specify component-related properties, while discarding interface-related properties. Similarly, Lfp allows the modeling of components' and connectors' interfaces, while its property specification language does not permit to specify properties about interfaces.

Configuration modeling and property: it is always possible to write and check a property about an architectural configuration (not reported in the table).

4.9. Goal 2 – summary

Different model checking SA approaches provide different degrees of usability, reliability, scalability, and expressiveness. The table in Fig. 28 provides an overall view of the data collected in the previous sections related to the second goal. Considerations follow, based on the study carried on.

Focussing on usability, we can notice that AutoFOCUS results to be the most usable model checking SA technique (providing an optimal support on the four entities under analysis), followed by CHARMY and Fujaba.[20] Tsai, CHAM, and Lfp result to be the less usable techniques.

Archware, CHARMY, Æmilia, AutoFOCUS, and Fujaba are the most reliable model checking SA techniques, since they are in a refinement stage of maturity and have been applied to multiple industrial studies.

Darwin/FSP, Archware, AutoFOCUS, and Fujaba are the most scalable approach, since they all support hierarchical/compositional modeling and a textual specification of the architecture.

Wright and Archware are the most expressive model checking SA techniques among those surveyed in this paper, being the only two supporting the criteria identified in Section 4.8.

5. Conclusions

Based on the model checking SA techniques classification and comparison proposed in this paper, this section identifies how the topic of model checking software architectures is progressing, and directions for future work.

5.1. Progress identification

This section identifies the progress in model checking SA according to the following aspects:

From theoretical study to practical application: While initial work on model checking SAs tackle the problem from a purely theoretical view point (see for example CHAM and Tsai), more recent techniques provide both a theoretical and a practical approach. Some techniques, that have mature tool support, total or partial transparency, and readable output (e.g., AutoFOCUS and CHARMY) have been applied to industrial studies.

From special ADLs to model-based specifications: While ADLs can properly solve domain-specific modeling needs and provide a solid means for verification, model-based notations are becoming more

[20] We here assume to give to the different parameters the same weight.

	Usability				Reliability		Scalability			Expressiv.
	Is the SA configuration a first class element ?	Is the model checking SA description/ SA property modelling semantically sound?	Is the output comprehensible to a software architect?	Which is the degree of automation in model checking SA techniques?	What is the level of maturity	Has the method been validated?	Does the model checking SA technique support hierarchical modelling?	Is the SA specification textual?	Does the model checker support abstraction and minimization?	Can the desired properties be checked by the model-checking SA approach?
Wright	Explicit	Textual/Textual	difficult	medium	dormant	S & S	NO	Textua	Compression functions	Comp, Comp Int
Tsai	Implicit	Textual/Textual	difficult	none	dormant	S & S	No	Textual	None	None
CHAM	Implicit	Textual/Textual	difficult	none	development	S & S	No	Textual	None	Comp
Bose	Implicit	Graphical/ Graphical and sound	difficult	medium	dormant	S & S	No	Graphical	state compression, on-the fly, and POR	Comp, Con, Type
Darwin/ FSP	Implicit	Textual/Textual	difficult	high	development	M & I	Yes	Textual and Graphical	compositional minimization	Comp, Comp Int, Type
PoliS	Implicit	Textual/Textual	difficult	medium	dormant	S & S	No	Textual	supports local searches	Comp
Arcade	Implicit	Textual/Textual	easy	medium	dormant	S & S	No	Textual	state compression, on-the fly, and POR	None
Archware	Explicit	Textual/Textual	difficult	high	refinement	M & I	Yes	Textual	compositional on-the fly	all
CHARMY	Explicit	Graphical/ Graphical and sound	easy	medium	refinement	M & I	No	Textual and Graphical	state compression, on-the fly, and POR	Comp
AEmilia	Explicit	Textual/Textual	difficult	high	refinement	M & I	No	Textual	OBDD and BMC	Comp, Comp Int, Type
SAM	Implicit	Textual/Textual	difficult	medium	dormant	M & I	Yes	Graphical	abstractions and reductions	Comp, Con
Garlan et al.	Implicit	Textual/Textual	difficult	medium	dormant	S & S	No	Graphical	composition and abstract	Comp
CBabel	Implicit	Textual/Textual	difficult	high	dormant	S & S	No	Textual	BDD and on-the-fly	Comp, Type
Zanolin et al.	Implicit	Graphical/ Graphical and sound	difficult	low	dormant	S & S	No	Graphical	state compression, on-the fly, and POR	Comp, Con
Auto FOCUS	Explicit	Graphical and sound/ textual	easy	high	refinement	M & I	Yes	Textual and Graphical	OBDD and POR	Comp, Comp Int
Fujaba	Implicit	Graphical and sound/ textual	difficult	high	refinement	M & I	Yes	Textual and Graphical	memory reduction	Comp, Comp Int
Lfp	Implicit	Graphical/Textual	difficult	low	dormant	S & S	No	Textual and Graphical	on-the-fly computation and BDD	Comp, Con, Type

Fig. 28. Summary of the goal 2 attributes.

popular and of wider application in industrial contexts. As already summarized in Fig. 5, most of the recently introduced techniques make use of model-based notations for model checking purposes.

From simple to complex architectures: While earlier methods were used to verify simple architectures (for example, send-receiver and client-server architecture), recent techniques can verify more complex systems. For example, Fujaba is currently used to specify and check mechatronic (hardware and software) systems, CHARMY has been used for verifying telecom and space systems.

From generic to domain-specific architectures: The trend towards SA specification and verification consists in moving from generic to domain-specific software architecture specification and verification. Most of the recently proposed model checking approaches apply to specific domains (e.g., distributed embedded systems, safety critical systems, p/s architectures).

5.2. Future directions for model checking SA

Based on the results of this study, this section tries to identify how this research area is progressing. To identify what we believe should be future research directions on the topic, we highlight what we believe is still missing in the integration among the model checking and SA communities, and where research from the two communities are moving.

Model-driven architecture: Shifting the focus of software development from coding to modeling is one of the main achievements of model-driven architecture (OMG, 2001) (MDA), which separates the application logic from the underlying platform technology and represents them with precise semantic models. Consequently, models are primary artifacts retained as first-class entities, and can be manipulated by means of automated model transformations, in order to move in an automated way from models to implementation.

SA plays a fundamental role in MDA, being an SA the earliest model of the whole software system created along the software life-cycle. Current research in the SA community is investing effort on understanding how to bridge the gap between requirements and software architecture (Nuseibeh, 2001; STRAW '03, 2003), and software architecture and coding (ArchJava, 2007; Hacklinger, 2004; Fujaba RT Project, 2009).

New model checking SA approaches should allow software architect to check the adequacy of the SA specification according to both the requirements and the real system implementation, in order to guarantee traceability analysis from requirements to code.

Integration of model checking with other V&V techniques: Many are the reasons for integrating model checking with other verification and validation techniques. Firstly, increased interest in designing dependable systems has favored the proliferation of analysis techniques each one based on a slightly different notation. As an immediate consequence, each specification language provides constructs that nicely support some specific analysis and leave other techniques unexplored. The resulting fragmentation induces the need to embrace different notations and tools to perform different analysis at the architecture level: for instance, supposing an organization is interested in model checking and performance analysis, a comprehensive result is obtained only using two different architectural languages, with different analysis methods and tools. Additionally, whenever the performance model needs to be modified, the model checking model must be manually adjusted (based on the performance results) and re-analyzed, causing frequent misalignments among models. Moreover, integration can overcome mutual limitations. For example, model checking and testing can be integrated for making the v&v process automated (via model checking), for reducing the state explosion problem (via testing), for improving perpetual v&v (from requirements to code).

New model checking SA approaches should be developed to allow easy integration of various analysis techniques in the same framework. Such integration can be realized through the definition of an extensible language to be used for various analysis techniques, and the development of extensible tools for

plugging in new analysis tools. Initial research on this direction has been presented in Malavolta et al. (2009), Garlan and Schmerl (2006).

Dynamic software architecture: Modern software architectures dynamically evolve, either because components are modified, replaced or removed, or because new components are plugged in. These run-time modifications may be due to the need to recover from functional or performance malfunctions, or may also be the natural consequence of system evolution. Dynamic SA is today becoming the prevalent paradigm in, e.g., in service-oriented architectures, publish/subscribe architectures, peer-to-peer distributed architectures. For dynamic systems we might still assume that SA provides the reference model, that has the capability to evolve with new components, defining the way in which these can interact with the "core" structure. In particular, each time components are removed, modified or added, the system should be checked again against the required SA properties.

New model checking SA approaches should be devised for checking dynamic SAs. Approaches for checking evolving architectures have been proposed in Allen et al. (1998), Garlan and Schmerl (2006) (the verification engine and approach used extend those proposed in Allen and Garlan (1997) with different forms of consistency, completeness, and equivalence checking), Darwin/FSP (and specifically Kramer and Magee, 1998), CBabel, Zanolin, and Archware methods.

Service-oriented architecture: Service-oriented computing is considered to be a challenging and promising computing paradigm that utilizes services as fundamental elements for developing applications. Service-oriented architecture (SOA) is a component model that inter-relates different functional units of an application, called services, through well-defined interfaces and contracts between them. Most benefits of this kind of SA come from service composition which combines existing services following a certain pattern to form a new value-added service. Service composition requires dynamic reconfiguration of services.

New model checking techniques should be devised in order to verify the correctness in service composition.

Software product line architectures: A software product line architecture (SPLA) (Bosch, 2000) captures, in a single specification, the architecture of a suite of closely-related products (rather than specifying the architecture for a single software system). The techniques for doing so are rooted in the disciplines of SA and configuration management, and focus on a distinction between mandatory elements (which are present in the architecture of each and every product) and variation points (which define the dimensions along which the architectures of the individual products differ from each other). A single product line architecture may have many variation points that are often orthogonal to each other: as a result, hundreds and sometimes thousands of product architectures can be formed by a single product line architecture.

New techniques are necessary for model checking product line architectures. Being an SPLA the high-level specification of an entire family of systems, by checking the SPLA we get feedback on any products derivable from the SPLA.

SA specific model checking engines: As illustrated in Fig. 19, most of the surveyed methods make use of existing model checking engines, translating SA specifications into the model checker input. While this approach permits the reuse of existing model checking engines, standard model checking engines do not provide primitives for architectural concepts, thus the constructed state model is usually not optimized.

The Bogor (Robby et al., 2004) extensible software model checking framework is a possible solution to this problem. It has been demonstrated that its application to software architectures can improve the verification performance up to 1000 times (Baresi et al., 2007).

Model checking probabilistic SA properties: Model checking probabilistic SA properties (Grunske, 2008; Zhang et al., 2009; Grunske and Zhang, 2009) is also an interesting future work, since probabilistic properties are considered the most important requirements for software in medical, avionic, automotive and telecommunication systems. These probabilistic properties are required to express non-functional requirements such as availability, reliability, safety, security, and performance.

New techniques are necessary for model checking probabilistic properties for SA. Some primary work have been done in Grunske et al. (2007), Colvin et al. (2007) and show promising results.

Acknowledgments

The authors wish to thank Marco Bernardo and André van der Hoek for their constructive comments on previous versions of this paper. The authors also thank the anonymous reviewers for their earlier valuable feedback to improve this work. Many thanks to the following colleagues who provided important feedbacks when preparing the final version of the paper: Luciano Baresi, Marco Bernardo, Kamel Barkaoui, David Garlan, Holger Giese, Paola Inverardi, Jeff Kramer, Jeff Magee, Cecilia Mascolo, Radu Mateescu, Flavio Oquendo, and Alexandre Sztajnberg. We are especially indebted with David Garlan who provided detailed comments on the paper.

Henry Muccini's work is supported by the EU IST CONNECT (http://connect-forever.eu/) No 231167 of the FET – FP7 program, the Italian PRIN d-ASAP project, and the Italian FIRB ARTDECO (Adaptive infRasTructure for DECentralized Organizations) project.

Pengcheng Zhang and Bixin Li's work is supported partially by the Natural Science Foundation of Jiangsu Province of China under Grant No. BK2007513, partially by National High Technology Research and Development Program under Grant No. 2008AA01Z113, partially by the National Natural Science Foundation of China under Grant No. 60773105 and partially by the Program for New Century Excellent Talents in University under Grant No. NCET-06-0466.

References

Aldini, A., Bernardo, M., Corradini, F. (Eds.), 2009. A Process Algebraic Approach to Software Architecture Design. Springer.

Allen, R., Garlan, D., 1997. A formal basis for architectural connection. ACM Trans. Software Eng. Methodol. 6 (3), 213–249.

Allen, R., Douence, R., Garlan, D., 1998. Specifying and analyzing dynamic software architectures. In: FASE, pp. 21–37.

ArchJava, 2007. ArchJava Project. <http://archjava.org/>.

Autili, M., Inverardi, P., Pelliccione, P., 2007. Graphical scenarios for specifying temporal properties: an automated approach. Automat. Software Eng. 14 (3), 293–340.

Babar, M.A., Gorton, I., 2004. Comparison of scenario-based software architecture evaluation methods. In: APSEC, pp. 600–607.

Babar, M.A., Zhu, L., Jeffery, D.R., 2004. A framework for classifying and comparing software architecture evaluation methods. In: Australian Software Engineering Conference, pp. 309–319.

Balsamo, S., Bernardo, M., Simeoni, M., 2003. Performance evaluation at the software architecture level. In: Formal Methods for Software Architectures, Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM, pp. 207–258.

Barber, K.S., Graser, T., Holt, J., 2001. Providing early feedback in the development cycle through automated application of model checking to software architectures. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, pp. 341–345.

Baresi, L., Ghezzi, C., Mottola, L., 2007. On accurate automatic verification of publish-subscribe architectures. In: Proceedings of the 29th International Conference on Software Engineering (ICSE07), Minneapolis (MN, USA), May.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, second ed. SEI Series in Software Engineering, Addison-Wesley Professional.

Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D., 2006. Symbolic invariant verification for systems with dynamic structural adaptation. In: Proceedings of the 28th International Conference on Software Engineering (ICSE), Shanghai, China. ACM Press, pp. 72–81.

Bernardo, M., Inverardi, P., 2003. Formal Methods for Software Architectures, Tutorial Book on Software Architectures and Formal Methods. SFM-03:SA Lectures, vol. 2804, LNCS.

Björnander, S., Grunske, L., Lundqvist, K., 2009. Timed simulation of extended aadl-based architecture specifications with timed abstract state machines. In: Architectures for Adaptive Software Systems, Fifth International Conference on the Quality of Software Architectures, QoSA, pp. 101–115.

Bondi, A.B., 2000. Characteristics of scalability and their impact on performance. In: WOSP '00: Proceedings of the Second International Workshop on Software and Performance. ACM, New York, NY, USA, pp. 195–203.

Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. ACM Press/Addison-Wesley Publishing Co..

Bose, P., 1999. Automated translation of uml models of architectures for verification and simulation using spin. In: Proceedings of the 14th IEEE International Conference on Automated Software Engineering, pp. 102–109.

Braga, C., Sztajnberg, A., 2003. Towards a rewriting semantics for a software architecture description language. Electron. Notes Theor. Comput. Sci. 95, 148–168.

Braga, C., Chalub, F., Sztajnberg, A., 2009. A formal semantics for a quality of service contract language. Electron. Notes Theor. Comput. Sci. 203 (7), 103–120.

Bril, R.J., Krikhaar, R.L., Postma, A., 2005. Architectural support in industry: a reflection using C-POSH. J. Software Mainten. Evolut. 33.

Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M., 2005. The Fujaba real-time tool suite: model-driven development of safety-critical, real-time systems. In: ICSE, pp. 670–671.

Burmester, S., Giese, H., Henkler, S., Hirsch, M., Tichy, M., Gambuzza, A., Müch, M., Vöcking, H., 2007. Tool support for developing advanced mechatronic systems: integrating the Fujaba real-time tool suite with camel-view. In: Proceedings of the 29th International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA. IEEE Computer Society Press, May, pp. 801–804.

Caporuscio, M., Inverardi, P., Pelliccione, P., Compositional verification of middleware-based software architecture descriptions. In: Proceedings of the International Conference on Software Engineering, pp. 221–230.

Ciancarini, P., Mascolo, C., 1999. Model checking a software architecture. In: Proceedings of the ROSATEA: International Workshop on the Role of Software Architecture in Analysis and Testing, vol. 24.

Ciancarini, P., Franzé, F., Mascolo, C., 2000. Using a coordination language to specify and analyze systems containing mobile components. ACM Trans. Software Eng. Methodol. 9 (2), 167–198.

Clarke, E., Grumberg, O., Peled, D. (Eds.), 2000. Model Checking. MIT Press.

CLavele, M. et al., (Ed.), 2004. Maude Manual (Version 2.1), March.

Clements, P., 1996. A survey of architecture description languages. In: Proceedings of the Eighth International Workshop Software Specification and Design.

CMU, 1998. Acme. Carnegie Mellon University. <http://www-2.cs.cmu.edu/~acme/>.

Colvin, R., Grunske, L., Winter, K., 2007. Probabilistic timed behavior trees. In: Integrated Formal Methods, Sixth International Conference, IFM, pp. 156–175.

Colvin, R., Grunske, L., Winter, K., 2008. Timed behavior trees for failure mode and effects analysis of time-critical systems. J. Syst. Software 81 (12), 2163–2182.

Compare, D., Inverardi, P., Wolf, A.L., 1999. Uncovering architectural mismatch in component behavior. Sci. Comput. Prog. 33 (2), 101–131.

Corradini, F., Inverardi, P., 1998. Model checking cham description of software architecture. In: Proceedings of the First Working IFIP Conference on Software Architectures.

Corradini, F., Inverardi, P., Wolf, A.L., 2006. On relating functional specifications to architectural specifications: a case study. Sci. Comput. Prog. 59 (3), 171–208.

CSMV, 2000. Cadence SMV. <http://www.cis.ksu.edu/santos/smv-doc/>.

Dai, Z., He, X., Ding, J., Gao, S., 2004. Modeling and analyzing security protocols in SAM: a case study. In: IASTED Conference on Software Engineering and Applications, pp. 115–121.

Damm, W., Harel, D., 2001. LSCs: breathing life into message sequence charts. Formal Methods Syst. Des. 19 (1), 45–80.

de Lemos, R., di Giandomenico, F., Gacek, C., Muccini, H., Vieira, M., (Eds.), 2008. Architecting Dependable Systems V. Lecture Notes in Computer Science, vol. 5135, Springer, August.

Dobrica, L., Niemelä, E., 2002. A survey on software architecture analysis methods. IEEE Trans. Software Eng. 28 (7), 638–653.

Dwyer, M.B., Avrunin, G.S., Corbett, J.C., 1999. Property specification patterns for finite-state verification. In: Proceedings of the 21th International Conference on Software Engineering (ICSE1999), pp. 411–420.

Formal System Ltd., 2005. Fdr2. <http://www.fsel.com.fdr2.download.html>.

Fujaba RT Project, 2009. University of Paderborn, Software Engineering Group. <http://www.fujaba.de/projects/real-time.html>.

Garlan, D., Schmerl, B., 2006. Architecture-driven modelling and analysis. In: Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems (SCS'06), pp. 3–17.

Garlan, D., Khersonsky, S., Kim, J.S., 2003. Model checking publish-subscribe systems. In: Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN 03).

Giannakopoulou, D., 1999. Model Checking for Concurrent Software Architectures. Ph.D. Thesis, Imperial College of London, London.

Giannakopoulou, D., Magee, J., 2003. Fluent model checking for event-based systems. In: Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held Jointly with Ninth European Software Engineering Conference, ESEC/FSE 2003, pp. 257–266.

Goldsby, H., Cheng, B.H., Konrad, S., Kamdoum, S., 2006. A visualization framework for the modeling and formal analysis of high assurance systems. In: Heidelberg, S.-V.B., (Ed.), Proceedings MODELS 2006, vol. 4199, LNCS, pp. 707–21.

Grunske, L., 2008. Specification patterns for probabilistic quality properties. In: Proceedings of the 30th International Conference on Software Engineering (ICSE), pp. 31–40.

Grunske, L., Zhang, P., Monitoring probabilistic properties. In: Proceedings of the Seventh Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) ESEC/FSE. ACM Press, pp. 183–192.

Grunske, L., Colvin, R., Winter, K., 2007. Probabilistic model-checking support for fmea. In: Proceedings of the Fourth International Conference on the Quantitative Evaluation of Systems (QEST), pp. 119–128.

Hacklinger, F., 2004. Java/a – taking components into java. In: Proceedings of the 13th ISCA International Conference Intelligent and Adaptive Systems and Software Engineering (IASSE'04), pp. 163–169.

Harel, D., Marelly, R., 2003. Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine. Springer-Verlag.

He, X., 2005. A framework for ensuring system dependability from design to implementation. In: Modelling, Simulation, Verification and Validation of Enterprise Information Systems, Proceedings of the Third International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS, In Conjunction with ICEIS.

He, X., Ding, J., Deng, Y., 2002. Model checking software architecture specifications in SAM. In: Proceedings of the 14th International on Software Engineering and Knowledge Engineering. ACM Press, New York, NY, USA, pp. 271–274.

He, X., Yu, H., Shi, T., Ding, J., Deng, Y., 2004. Formally analyzing software architectural specifications using SAM. J. Syst. Software 71 (1–2), 11–29.

Henkler, S., Greenyer, J., Hirsch, M., Schäfer, M., Alhawash, K., Eckardt, T., Heinzemann, L?ffler, R., Seibel, A., Giese, H., 2009. Synthesis of timed behavior from scenarios in the Fujaba real-time tool suite. In: Proceedings of the 31th International Conference on Software Engineering (ICSE). Vancouver, Canada, May, pp. 615–618.

Holzmann, G.J., 1997. The model checker spin. IEEE Trans. Software Eng. 23 (5), 279–295.

Holzmann, G.J., 2002. The logic of bugs. In: FSE Foundations of Software Engineering.

INRIA, 2007. CADP Project: Construction and Analysis of Distributed Processes. <http://www.inrialpes.fr/vasy/cadp.html>.

Inverardi, P., Muccini, H., Pelliccione, P., 2001. Automated check of architectural models consistency using spin. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, pp. 346–349.

ITC-irst, 2005. Numv. <http://nusmv.irst.itc.it>.

Jerad, C., Barkaoui, K., 2005. On the use of rewriting logic for verification of distributed software architecture description based lfp. In: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping, pp. 202 208.

Jerad, C., Barkaoui, K., Grissa-Touzi, A., 2007. Hierarchical verification in maude of lp software architectures. In: Software Architecture, First European Conference, ECSA, pp. 156–170.

Jerad, C., Barkaoui, K., Grissa-Touzi, A., On the use of real-time maude for architecture description and verification: a case study. In: Visions of Computer Science – BCS International Academic Conference, pp. 305–317.

Kazman, R., Abowd, G., Bass, L., Clements, P., 1996. Scenario-based analysis of software architecture. IEEE Software 13 (6), 47–55.

Kramer, J., Magee, J., 1998. Analysing dynamic change in distributed software architectures. IEE Proceedings – Software 145 (5), 146–154.

Magee, J., Kramer, J., 2006. Concurrency: State Models and Java Programs, second ed. Wiley Publisher.

Magee, J., Dulay, N., Eisenbach, S., Kramer, J., 1995. Specifying distributed software architectures. In: Proceedings of the ESEC95, September.

Magee, J., Kramer, J., Giannakopoulou, D., 1999. Behavior analysis of software architectures. In: Proceedings of the First Working IFIP Conference on Software Architecture.

Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D., 2009. Providing architectural languages and tools interoperability through model transformation technologies. Transactions on Software Engineering (TSE). IEEE Computer Society.

Mateescu, R., Oquendo, F., 2006. pi-AAL: an architecture analysis language for formally specifying and verifying structural and behavioural properties of software architectures. ACM SIGSOFT Software Eng. Notes 31 (2), 1–19.

McMillan, K.L. (Ed.), 1993. Symbolic Model Checking. Kluwer Academic.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description language. IEEE Trans. Software Eng. 26 (1), 70–93.

Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E., 2002. Modeling software architectures in the unified modeling language. ACM Trans. Software Eng. Methodol. (TOSEM) 11 (1).

Medvidovic, N., Dashofy, E.M., Taylor, R.N., 2007. Moving architectural description from under the technology lamppost. Informat. Software Technol. 49, 12–31.

MunichUT, 2006. AutoFOCUS Project. <http://autofocus.in.tum.de/index-e.html>.

Mustapic, G., Wall, A., Norstrom, C., Crnkovic, I., Sandstrom, K., Andersson, J., 2004. Real world influences on software architecture – interviews with industrial system experts. In: Fourth Working IEEE/IFIP Conference on Software Architecture, WICSA, June, pp. 101–111.

Naslavsky, L., Richardson, D., Ziv, H., 2006. Scenario-based and State Machine-based Testing: An Evaluation of Automated Approaches. Technical Report, Institute for

Software Research, University of California, Irvine – ISR Technical Report UCI-ISR-06-13.
Nuseibeh, B., 2001. Weaving together requirements and architectures. IEEE Comput. 34 (3), 115–117.
Object Management Group (OMG), 2001. OMG/Model Driven Architecture – A Technical Perspective. OMG Document: ormsc/01-07-01.
Oquendo, F., 2004. pi-ADL: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Software Eng. Notes 29 (3), 1–14.
Oquendo, F., Warboys, B., Morrison, R., Dindeleux, R., Gallo, F., Garavel, H., Occhipinti, C., 2004. Archware: architecting evolvable software. In: Oquendo, F., Warboys, B., Morrison, R. (Eds.), Proceedings of the First European Workshop on Software Architecture EWSA'2004 (St. Andrews, Scotland, UK), Lecture Notes in Computer Science, vol. 3047. Springer-Verlag, May, pp. 257–271.
Peled, D., Pelliccione, P., Spoletini, P., 2008. Model checking. In: Wiley Encyclopedia of Computer Science and Engineering.
Peled, D., Pelliccione, P., Spoletini, P., 2009. Wiley Encyclopedia of Computer Science and Engineering. Chapter Model Checking, vol. 5, Wiley.
Pelliccione, P., Inverardi, P., Muccini, H., 2009. CHARMY: a framework for designing and verifying architectural specifications. IEEE Transactions on Software Engineering 35 (3), 325–346.
Perry, D.E., Wolf, A.L., 1992. Foundat. Study Software Archit. 17 (4), 40–52.
QOSA, 2005–2007. QOSA conferences on the quality of software architectures. Lecture Notes on Computer Science.
Queille, J.P., Sifakis, J., 1982. Specification and verification of concurrent systems in cesar. In: Proceedings of the Third International Symposium on Programming, Turin, pp. 337–350.
Rademaker, A., Braga, C., Sztajnberg, A., 2005. A rewriting semantics for a software architecture description language. Electron. Notes Theor. Comput. Sci. 130, 345–377.
Robby, Rodriguez, E., Dwyer, M.B., Hatcliff, J., 2004. Checking strong specifications using an extensible software model checking framework. In: Proceedings of the Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), March.
Shaw, M., Clements, P., 2006. The golden age of software architecture. IEEE Software 23 (2), 31–39.
Shaw, M., Garlan, D. (Eds.), 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall.
Shi, T., He, X., 2003. A methodology for dependability and performability analysis in SAM. In: DSN, pp. 679–688.
STRAW '03, 2003. In: Second International Workshop from Software Requirements to Architectures, May 09, Portland, Oregon, USA.
Tsai, J.J.P., Xu, K., 2000. A comparative study of formal verification techniques for software architecture specifications. Ann. Software Eng. 10 (1–4), 207–223.
Tsai, J.J.P., Sistla, A.P., Sahay, A., Paul, R., 1997. Incremental verification of architecture specification language for real-time systems. In: WORDS '97: Proceedings of the Third Workshop on Object-Oriented Real-Time Dependable Systems – (WORDS '97), IEEE Computer Society. Washington, DC, USA, p. 215.
UPPAAL, 2005. <http://www.uppaal.com>.
Vestal, S., 1993. A Cursory Overview and Comparison of Four Architecture Description Languages. Technical Report, Honeywell Technology Center.
xArch, 2007. xarch, Proposed by the University of California, Irvine. <http://www.isr.uci.edu/architecture/xarch/>.
Yu, H.Q., He, X.D., Deng, Y., Lian, M., 2004. A formal approach to designing secure software architectures. In: Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering, pp. 289–290.
Zanolin, L., Ghezzi, C., Baresi, L., 2003. An approach to model and validate publish/subscribe architectures. In: Proceedings of the SAVCBS'03 Workshop.
Zeiß, B., Vega, D., Schieferdecker, I., Neukirchen, H., Grabowski, J., 2007. Applying the ISO 9126 quality model to test specifications – exemplified for TTCN-3 test specifications. In: Software Engineering (SE 2007). Lecture Notes in Informatics (LNI) 105. Copyright Gesellschaft für Informatik, Köllen Verlag, Bonn, March, pp. 231–242.
Zhang, P., Li, B., Sun, M., 2008. A timed extension of property sequence chart. In: Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium (HASE'08), Nanjing, China. IEEE Computer Society, pp. 197–206.
Zhang, P., Grunske, L., Tang, A., Li, B., 2009. A formal syntax for probabilistic timed property sequence charts. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, ASE. IEEE Computer Society Press, pp. 500–504.
Zhang, P., Li, B., Grunske, L., 2010. Timed property sequence chart. Journal of Systems and Software 83 (3), 352–370.

Pengcheng Zhang received the PhD degree in computer science from Southeast University, in 2010. He is currently a lecturer in School of Computer and Information, Hohai University, Nanjing, China. His research interests include modeling, analysis, testing and verification of component based systems, software architectures, real-time and probabilistic systems, service-oriented systems.

Henry Muccini got is PhD degree in Computer Science from the University of Rome, La Sapienza. He is currently an Assistant Professor at the University of L'Aquila, Italy.
Henry's research interests are in the Software Engineering field, and specifically on software architecture modeling and analysis, component-based systems, model-based analysis and testing, fault tolerance, and global software engineering. He has published various conference and journal articles on these topics, co-edited two books, and co-organized various workshops on related topics. For more information, please refer to http://www.henrymuccini.com.

Bixin Li received the PhD degree in computer science from Nanjing University, in 2001. He is currently a full Professor of School of Computer Science and Engineering at the Southeast University, Nanjing, China. His research interests include: Program slicing and its application; Software evolution and maintenance; and Software modeling, analysis, testing and verification. He has published over 60 articles in refereed conferences and journals.