Model Checking of CHAM Descriptions of Software Architectures [*]

Flavio Corradini Paola Inverardi

Dipartimento di Matematica Pura ed Applicata
Università degli Studi di L'Aquila
Via Vetoio, Loc. Coppito, L'Aquila, Italy
{flavio,inverard}@univaq.it

Abstract

In this paper we show how to define and prove different properties of a software architecture description based on the CHAM. We consider both *structural* properties, that is properties of the system which are connected with its structure either static and dynamic, and *functional* properties, that is properties on how the system manipulates its data. We will use a logic approach to describe the properties and a model checking approach to verify them. As case study we take the software architecture of a sender-receiver system.

1 Introduction

An important feature of software architectures is the ability to allow reasoning on interesting properties of the described system. This would allow an early in the software development analysis of relevant features of the system, and it would help in evaluating the suitability of a software architecture. In the literature there are many examples of analysis of software architecture properties, including the usual liveness and safety properties [1, 2, 6, 9].

In this paper we show how to define and prove different properties of a software architecture description based on the CHAM [8]. We consider both *structural* properties, that is properties of the system which are connected to its structure either static and dynamic, and *functional* properties, that is properties on how the system manipulates its data.

Our goal is to show how it is possible to formalize the intuitive reasoning which a designer carries out when he tries to evaluate, from different viewpoints, the proposed architecture. Thus the properties we address can be on one side specific of the given architecture, and on the other specific of the system to be modeled, that is part of its requirements. As a case study we use the software architecture description of a Sender-Receiver System.

For expressing the properties we use the CTL* logic [5, 4] and to prove them we use a model checking approach based on the ability of deriving a Kripke structure out of a CHAM specification, as described in Section 2. To define the properties, we first reason at the CHAM level, in order to understand in terms of the system behavior what is the property we want to state, then we formally translate this reasoning in a logic formula. In Section 3 we show how to identify interesting properties and how these can be formalized in CTL* .

[*]This work has been partially funded by CNR, Progetto:Metodologie, e strumenti di analisi, verifica e validazione per sistemi software affidabili

2 Specifying CHAM Descriptions Properties via CTL* Logic

2.1 The CHAM Model

In this section we briefly introduce the CHAM formalism and its use for the description of Software Architectures as introduced in [8]. We refer to that paper for more details.

A Chemical Abstract Machine is specified by defining *molecules* m, m', m'',... defined as terms of a syntactic algebra that derive from a set of constants and a set of operations and *solutions* S, S', S'', ... of molecules. Molecules constitute the basic elements of a CHAM, while solutions are multisets of molecules interpreted as defining the *states* of a CHAM. A CHAM specification contains *reaction rules* T, T',... (of the form $m_1, m_2, ..., m_k \longrightarrow m'_1, m'_2, ..., m'_k$) that define (via an inference rule of the form $S \longrightarrow S'$ implies $S \uplus S'' \longrightarrow S' \uplus S''$) a transformation relation $S \longrightarrow S'$, dictating the way solutions can evolve (i.e., states can change) in the CHAM. At any given point, a CHAM can apply as many rules as possible to a solution, provided that their premises do not conflict that is, no molecule is involved in more than one rule. In this way it is possible to model parallel behaviors by performing parallel reactions. When more than one rule can apply to the same (set of) molecule the CHAM makes a nondeterministic choice as to which reaction to perform. The CHAM description of a software architecture [8] consists of a syntactic description of the static components of the architecture (the *molecule*), a solution representing the initial state of the architecture (the *initial solution*), and of a set of reaction rules which describe how the system dynamically evolves through reaction steps (the *reaction rules*).

2.2 Modeling CHAM Descriptions Behaviors via *Kripke Structures*

As already investigated in [8], the CHAM formalism allows for different analysis and verification techniques. This can be very convenient since depending on the kind of property, one can choose the most adequate technique. In particular, one can either exploit the algebraic and equational nature of CHAM or take advantage of its operational flavor to derive a transition system or a Kripke structure out of a CHAM description and then reason at this abstraction level.

Let us show how to derive a Kripke structure out of a CHAM description. We derive it from the operational semantics, by considering that our reaction rules are the operational semantics rules.

Definition 2.1 *(Operational semantics induced by $\mathcal{R}$) Let $\mathcal{R}$ be the set of reaction rules of a CHAM C. Then $\mathcal{R}$ defines a relation $\rightarrow_{\mathcal{R}} \subseteq Solution \times Solution$. The relation is the least relation satisfying the rules.*

Definition 2.2 *(Derivative) Given a set of reaction rules $\mathcal{R}$, an $\mathcal{R}$-derivation from a solution S to a solution S_n is a sequence $\{S_i, 1 \leq i \leq n,\ n > 1\}$ such that $S = S_1$ and for any $1 \leq i \leq n-1$, $S_i \rightarrow_{\mathcal{R}} S_{i+1}$. A solution S is called an $\mathcal{R}$-*derivative *of S' if an $\mathcal{R}$-derivation exists from S' to S. The set of derivatives of S is denoted by $D_{\mathcal{R}}(S)$ while $M_{\mathcal{R}}(S)$ denotes the set of molecules within solutions in $D_{\mathcal{R}}(S)$.*

Definition 2.3 *(Kripke Structure) A* Kripke Structure *(or KS) is a 5-tuple $\mathcal{K} = (\mathcal{S}, \mathcal{AP}, \mathcal{L}, \mathcal{D}, s_0)$ where*

- *$\mathcal{S}$ is a set of states;*
- *$\mathcal{AP}$ is a non-empty set of* atomic proposition names *ranged over by $p, p_1, \ldots$;*
- *$\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{AP}}$ is a function that assigns to each state a set of atomic propositions true in that state;*
- *$\mathcal{D} \subseteq \mathcal{S} \times \mathcal{S}$ is the* transition relation*;*

- *s_0 is the initial state.*

We can now show how, given a CHAM and a solution, we can derive a Kripke structure which represents the whole set of possible derivations. If the number of derivable solutions is finite also the Kripke structure is finite.

Definition 2.4 *(Kripke Structure corresponding to a solution) Given a solution S and a set of reaction rules $\mathcal{R}$, $\mathcal{R}(S)$ is the Kripke structure $(D_{\mathcal{R}}(S) \cup \{S\}, M_{\mathcal{R}}(S), \mathcal{L}, \rightarrow_{\mathcal{R}}, S)$, where for every $S' \in D_{\mathcal{R}}(S) \cup \{S\}$, $\mathcal{L}(S')$ is the set of molecules in S' and $\rightarrow_{\mathcal{R}}$ is the relation defined by $\mathcal{R}$.*

2.3 The CTL* Logic

We shortly introduce the branching temporal logic CTL* defined in [5, 4]. CTL* is suitable to express properties of reactive systems defined by means of transition systems or Kripke structure. In this paper, we concentrate on some structural and functional properties that our architectures have to possess. Although our main focus in the analysis will be centered around these properties, we can use CTL* to easily define and verify usual safety and liveness properties. Before defining syntax and semantics of the CTL* operators, let us introduce some definitions which will be used in the sequel. A notion of *paths* (or runs) is needed:

- σ is a path from $r_0 \in S$ if either $\sigma = r_0$ (the empty path from r_0) or σ is a sequence (possibly infinite) $(r_0, r_1)(r_1, r_2) \ldots$ such that $(r_i, r_{i+1}) \in \rightarrow$ for each $i \geq 0$. σ^i, for $i \geq 0$, denotes the suffix path $(r_i, r_{i+1})(r_{i+1}, r_{i+2}) \ldots$ while $\sigma(i)$ the i^{th} state in the sequence, i.e. r_i.
- A path σ is called maximal if either it is infinite or it is finite and its last state r has no successor states. The set of maximal paths from r_0 will be denoted by $\Pi(r_0)$.
- If σ is infinite, then $|\sigma| = \omega$ ($|\sigma|$ denotes the length of σ). If $\sigma = r_0$, then $|\sigma| = 0$.
  If $\sigma = (r_0, r_1)(r_1, r_2) \ldots (r_n, r_{n+1})$, $n \geq 0$, then $|\sigma| = n + 1$.

The syntax of CTL* is defined by state and path formulae. They are generated by the following grammar; ϕ ranges over state formulae, γ ranges over path formulae and p over AP:

$$\begin{array}{lcl} \phi & ::= & p \mid \phi \wedge \phi \mid \neg\phi \mid \exists\gamma \mid \forall\gamma \\ \gamma & ::= & \phi \mid \gamma \wedge \gamma \mid \neg\gamma \mid X\gamma \mid F\gamma \mid \gamma U \gamma\,. \end{array}$$

Operator $\exists$ (for some path) and $\forall$ (for all paths) are path quantifiers while X, U and F are the *next*, the *until* and the *sometimes* operators respectively. Intuitively, a path σ holds the *next* modality $X\gamma$ if and only if σ^1 (the ''next moment") holds γ. σ holds the *sometimes* modality $F\gamma$ if and only if there exists $i \geq 0$ (some ''future moment") such that σ^i holds γ. Finally, σ holds the *until* modality $\gamma U \gamma'$ if and only if there exists $i \geq 0$ such that σ^i holds γ' (ϕ' does eventually holds) and for all j such that $0 \leq j < i$, σ^j holds γ (γ holds ''everywhere prior" to γ').

Let $\mathcal{K} = (\mathcal{S}, \mathcal{AP}, \mathcal{L}, \rightarrow, s_0)$ be a Kripke Structure. *Satisfaction* of a state formula ϕ (path formula γ) by a state s (path σ), notation $s \models_{\mathcal{K}} \phi$ ($\sigma \models_{\mathcal{K}} \gamma$) is given inductively by :

$$\begin{array}{lll} s \models_{\mathcal{K}} p & \text{iff} & p \in \mathcal{L}(s) \\ s \models_{\mathcal{K}} \phi \wedge \phi' & \text{iff} & s \models_{\mathcal{K}} \phi \text{ and } s \models_{\mathcal{K}} \phi' \\ s \models_{\mathcal{K}} \neg\phi & \text{iff} & \text{It is not the case that } s \models_{\mathcal{K}} \phi \\ s \models_{\mathcal{K}} \exists\gamma & \text{iff} & \exists\sigma \in \Pi(s) \text{ such that } \sigma \models_{\mathcal{K}} \gamma \\ s \models_{\mathcal{K}} \forall\gamma & \text{iff} & \forall\sigma \in \Pi(s),\ \sigma \models_{\mathcal{K}} \gamma \\ \sigma \models_{\mathcal{K}} \phi & \text{iff} & \sigma(0) \models_{\mathcal{K}} \phi \\ \sigma \models_{\mathcal{K}} \gamma \wedge \gamma' & \text{iff} & \sigma \models_{\mathcal{K}} \gamma \text{ and } \sigma \models_{\mathcal{K}} \gamma' \\ \sigma \models_{\mathcal{K}} \neg\gamma & \text{iff} & \text{It is not the case that } \sigma \models_{\mathcal{K}} \gamma \\ \sigma \models_{\mathcal{K}} F\gamma & \text{iff} & \exists i \geq 0\,.\, \sigma^i \models_{\mathcal{K}} \gamma \\ \sigma \models_{\mathcal{K}} X\gamma & \text{iff} & |\sigma| \geq 1 \text{ and } \sigma^1 \models_{\mathcal{K}} \gamma \\ \sigma \models_{\mathcal{K}} \gamma U \gamma' & \text{iff} & \exists i \geq 0 \text{ such that } \sigma^i \models_{\mathcal{K}} \gamma', \text{ and } \forall j < i,\ \sigma^j \models_{\mathcal{K}} \gamma \end{array}$$

3 On Using CTL* to Analyze and Verify System Properties

As already said, the CTL* logic can be used to specify and verify system properties. Here, we use this logic to specify and verify properties of software architectures. As case study we use a Sender-Receiver System. In the following we briefly describe the case study, then we show its CHAM formalization and how relevant properties can be expressed in CTL* .

3.1 The Sender-Receiver System

A sender, SENDER, asks for services, namely $iserv_1$, ...,$iserv_n$, to a receiver, RECEIVER. The communication between SENDER and RECEIVER is asynchronous; i.e., the former sends services to the latter by pushing them in a buffer, BUFFER, and then proceeds autonomously. RECEIVER, on its side, pops up services from the buffer which then independently performs.

A brief description of the architecture of the Sender-Receiver System is now in order. The syntax for the architecture is given in Table 1-(1). Following Perry and Wolf [10], P represents the processing elements, D represents the data elements (service is the type of the service instances $iserv_1$, ...,$iserv_n$) and C represents the connecting elements. The initial solution is given in Table 1-(2). This solution consists of three molecules, one for the initial state of each of the three major processing elements in the architecture. Finally, the reaction rules are given in Table 1-(3). They represent how the system can dynamically evolve. The operator + denotes the non deterministic composition, $\star$ a sort of Kleene star (it is used for recursive definitions), and $\diamond$ denotes the sequential composition (where the molecule is consumed from left to right). To make the description as shorter as possible, we used $iserv$ to denote one of the service instances $iserv_1$, ...,$iserv_n$. Hence T_6, for instance, being for n rules; those obtained by replacing $iserv$ with one of the services $iserv_1$, ...,$iserv_n$.

3.2 Expressing Structural Properties of the Sender-Receiver System

In this section we show how properties related to the structure of the system can be specified and verified. In these kind of systems we are often interested in the amount of concurrency-parallelism exploited by the components acting as data resources. The architecture of the system previously presented describes, indeed, a parallel system and, some of the specified components, are actually data resources. SENDER, for instance, provides data (actually, services) to RECEIVER via BUFFER.

Concurrency in the buffer between SENDER and RECEIVER may arise if SENDER decides to push a new service while RECEIVER decides, at the same time, to pop up a service to be performed.

Property 3.1 *It is possible that* SENDER *is ready to push a service in the buffer when* RECEIVER *is ready to pop up a service from the buffer.*

In other words, we say that in a certain solution, SENDER is ready to push a new service into the buffer if molecule

$\phi_1 = \mathsf{request(iserv,BUFFER)} \diamond (m)^\star \diamond \mathsf{SENDER}$

is available in the current solution. Furthermore, RECEIVER can pop up a service from the buffer if (i) the buffer is not empty and, hence,

$\phi_2 = (\mathsf{buffer(iserv,RECEIVER)} \diamond \mathsf{BUFFER} \vee \mathsf{buffer(iserv,RECEIVER)} \diamond m_2 \diamond \mathsf{BUFFER})$

is in the actual solution, and (ii) molecule

$\phi_3 = \mathsf{input(iserv,BUFFER)} \diamond m_2 \diamond \mathsf{RECEIVER}$

requiring a service from the buffer is also in the current solution.

The CTL* formula expressing Property 3.1 is hence

$$\varphi_1 = \exists(F(\phi_1 \wedge \phi_2 \wedge \phi_3)).$$

M	::=	$P \mid C \mid M \diamond M \mid M^\star$
P	::=	SENDER $\mid$ RECEIVER $\mid$ BUFFER
D	::=	service $\mid$ $iserv_1$ $\mid$ ... $\mid$ $iserv_n$
C	::=	request(D,P) $\mid$ input(D,P) $\mid$ make(D) $\mid$ buffer(D,P) $\mid$ $C + C$
		(1)-Syntax for the Architecture.
S_1	::=	(request($iserv_1$,BUFFER) + ... + request($iserv_n$,BUFFER))* $\diamond$ SENDER, BUFFER, (input(service,BUFFER) $\diamond$ make(service))* $\diamond$ RECEIVER
		(2)-Initial Solution.
T_1	$\equiv$	$(c_1 + c_2) \diamond m \longrightarrow c_1 \diamond m$
T_2	$\equiv$	$(c_1 + c_2) \diamond m \longrightarrow c_2 \diamond m$
T_3	$\equiv$	$(m)^\star \diamond$ SENDER $\longrightarrow m \diamond (m)^\star \diamond$ SENDER
T_4	$\equiv$	$(m)^\star \diamond$ SENDER $\longrightarrow$ SENDER
T_5	$\equiv$	$(m)^\star \diamond$ RECEIVER $\longrightarrow m \diamond (m)^\star \diamond$ RECEIVER
T_6	$\equiv$	request(iserv,BUFFER) $\diamond (m)^\star \diamond$ SENDER, BUFFER $\longrightarrow$ $(m)^\star \diamond$ SENDER, buffer(iserv,RECEIVER) $\diamond$ BUFFER
T_7	$\equiv$	request(iserv,BUFFER) $\diamond (m)^\star \diamond$ SENDER, $m_1 \diamond$ BUFFER $\longrightarrow$ $(m)^\star \diamond$ SENDER, $m_1 \diamond$ buffer(iserv,RECEIVER) $\diamond$ BUFFER
T_8	$\equiv$	buffer(iserv,RECEIVER) $\diamond$ BUFFER, input(service,BUFFER) $\diamond$ make(service) $\diamond m_2 \diamond$ RECEIVER $\longrightarrow$ BUFFER, make(iserv) $\diamond m_2 \diamond$ RECEIVER
T_9	$\equiv$	buffer(iserv,RECEIVER) $\diamond m_1 \diamond$ BUFFER, input(service,BUFFER) $\diamond$ make(service) $\diamond m_2 \diamond$ RECEIVER $\longrightarrow$ $m_1 \diamond$ BUFFER, make(iserv) $\diamond m_2 \diamond$ RECEIVER
T_{10}	$\equiv$	make(iserv) $\diamond m \diamond$ RECEIVER $\longrightarrow m \diamond$ RECEIVER
		(3)-Reaction Rules.

Table 1: The Software Architecture of the Sender-Receiver System.

By examining the paths out of S_1 we can prove that the initial solution holds φ_1. The rule T_6 (T_7) applied to ϕ_1 and buffer(iserv,RECEIVER) $\diamond$ BUFFER (buffer(iserv,RECEIVER) $\diamond m_2 \diamond$ BUFFER), to push a new service, and rule T_8 (T_9) applied to buffer(iserv,RECEIVER) $\diamond$ BUFFER (buffer(iserv,RECEIVER) $\diamond m_2 \diamond$ BUFFER) and ϕ_3, to pop up a service, can be both active in the current solution at the same time. In other words, the operations of pushing a service by SENDER and popping a service by RECEIVER can be concurrent.

3.3 Expressing Functional Properties of the Sender-Receiver System

Here we prove that our architecture satisfies a particular integrity constraint. More in detail , we show that every service that RECEIVER is going to perform has been previously required by SENDER. To this aim, we show that if RECEIVER is ready to perform a service in a given state of the system, then there has been a state, along the same computation (i.e.path), where SENDER bufferized the service in BUFFER. Moreover, we show that the service is removed by the buffer before it is performed by RECEIVER and that RECEIVER does not perform any other service in the meantime.

A CTL* formula expressing this property is:

$$\varphi \;=\; \neg\, F\, (\neg\gamma)$$

where $\gamma = \gamma_1 \vee \gamma_2$ and γ_1, γ_2 are defined by:

$$\begin{aligned}
\gamma_1 &= (\mathsf{buffer(iserv,RECEIVER)} \diamond \mathsf{BUFFER} \wedge \neg\mathsf{make(iserv)} \diamond m \diamond \mathsf{RECEIVER})\ U \\
&\quad (\mathsf{BUFFER} \wedge \mathsf{make(iserv)} \diamond m \diamond \mathsf{RECEIVER}) \\
\gamma_2 &= (\mathsf{buffer(iserv,RECEIVER)} \diamond m_1 \diamond \mathsf{BUFFER} \wedge \neg\mathsf{make(iserv)} \diamond m \diamond \mathsf{RECEIVER})\ U \\
&\quad (m_1 \diamond \mathsf{BUFFER} \wedge \mathsf{make(iserv)} \diamond m \diamond \mathsf{RECEIVER}).
\end{aligned}$$

We can then prove that our architecture actually satisfies the above property.

Acknowledgments: Alex Wolf is thanked for common work on the topic of this paper.

References

1. S.C. Cheung and J. Kramer: Checking Subsystem Safety Properties in Compositional Reachability Analysis. Proc. of the 18th International Conference on Software Engineering, 1996.

2. S.C. Cheung, D. Giannakopoulou, and J. Kramer: Verification of Liveness Properties using Compositional Reachability Analysis. Proc. of ESEC/FSE 97, 1997.

3. D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. *Science of Computer Programming*, 1998. To appear.

4. E.A. Emerson: Temporal and Modal Logic. *Handbook of Theoretical Computer Science*, volume B, chapter 16. Elsevier Science Publishers B.V., 1990.

5. E.A. Emerson, J.Y. Halpern: “Sometimes” and “Not Never” revisited: on branching time versus linear time temporal logic. *Journal of ACM* **33** (1), pp.151-178, 1986.

6. D. Giannakopoulou, J. Kramer, and S.C. Cheung: Analyzing the Behavior of Distributed Systems using Tracta. *Journal of Automated Software Engineering*, Special Issue on Automated Analysis of Software (R. Cleaveland and D. Jackson, Eds.). To appear.

7. D. Garlan, W. Tichy and F. Paulisch: Summary of the Dagstuhl Workshop on Software Architecture. *SIGSOFT Software Engineering Notes* **20**(3), pp. 63-83, 1995.

8. P. Inverardi, A. Wolf: Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering* **21** (4) pp.373-386, 1995.

9. D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, **21** (4) pp. 336-355, 1995.

10. D.E. Perry e A.L. Wolf. Foundations for the Study of Software Architecture. ACM SigSoft *Software Engineering Notes*, **17** (4), pp. 40-52, 1992.