Linear Temporal Logic Applied to Component-Based Software Architectural Models Specified Through ρ_{arq} Calculus

Oscar Javier Puentes(✉) and Henry Alberto Diosa

Research Group ARQUISOFT, Faculty of Engineering, Universidad Distrital Francisco José de Caldas, Bogotá, Colombia
ojpuentesp@correo.udistrital.edu.co, hdiosa@udistrital.edu.co
http://arquisoft.udistrital.edu.co

Abstract. This paper reports a mechanism to incorporate Linear Temporal Logic (LTL) for a component-based software architectural configuration specified by the ρ_{arq}-calculus. This process was made through the translation of the system definition, structure and behavior, to Atomic Propositions Transition System (APTS), upon which, the verification of one property was performed using LTL. The PintArq software application was extended to support this mechanism. One example ilustrates the verification of responsiveness, a subtype of liveness property.

Keywords: ρ_{arq} calculus · Component-based software
Architectural execution flow · Linear temporal logic · Model checking

1 Introduction

On software architectures, a challenging area of growing interest has been to find mathematical tools that allow checking properties and quality aspects. There are some developments about formal methods [14], i.e., λ-calculus [4] for sequential processes, π-calculus [13,18] for concurrent processes, and recently, ρ-calculus [7] for object oriented paradigms [22]. ρ-calculus provides a foundation to model object oriented paradigm through Unified Modeling Language (UML) [12,16].

The ρ_{arq} calculus [9] was proposed to specify structural and behavioral aspects about component-based software architectures; furthermore, it's a tool to check desirable properties such as correctness [10]. Currently, a software application allows to visualize the software architectural execution flow specified by this calculus. This application receives a set of formulas as input and the software shows each stage of the execution [19].

The approach used in this project aims to apply (LTL) to specify properties and check models that satisfies them [3]. This work addresses on a subtype liveness property called "responsiveness".

At first, a conceptual frame is presented, then a translation method to Atomic Proposition Transition System using LTL operators is ilustrated, subsequently

J. C. Figueroa-García et al. (Eds.): WEA 2018, CCIS 915, pp. 393–405, 2018.
https://doi.org/10.1007/978-3-030-00350-0_33

an example is executed to show the checking process and by last, the new version of software application is described.

2 Conceptual Context

2.1 $\rho_{\mathbf{arq}}$ Calculus

ρ_{arq} calculus is an architectural description language (ADL) with formal notation to specify structural and dynamic aspects of component-based software architectures. Table 1 describes ρ_{arq}-calculus syntax. A more detailed description can be obtained in [9–11,19].

Table 1. Syntax of ρ_{arq} calculus. Source: [9,13,15,19–21]

SYMBOLS	MEANING
$x, y, z, \ldots$ variables $a, b, c, \ldots$ names $u, v, w, \ldots ::= x \mid a$ references	Variables only hold names. Names and variables are named references.
EXPRESSIONS	**INTERPRETATION**
$E, F, G ::= \quad \top$	**Null component** Component that doesn't execute any action.
$\mid \quad E \wedge F$	**Composition** It represents concurrent execution of E and F.
$\mid \quad E^{(int)}$	**Interior of component** E No observable part of E
$\mid \quad if(C_1 \cdots C_n)\ else\ G$	**Committed choice combinator** This representation of components with alternative executions in the $\rho_{arq}-Calculus$ is a derivation of the **Guarded Disjunction** proposed in the early extended versions of $\gamma - Calculus$ [20] [21] is a useful generalization of conventional conditional[1].
$\mid \quad x :: \overline{y}/E$	**Abstraction** It represents receiving a symbolic entity by means of x, it can replace $\overline{y}$ in E, as long as this entity is free in the scope of component E.
$\mid \quad x\overline{y}/E$	**Application** $x\overline{y}/E$ expresses sending $\overline{y}$ by means of x and continuing with the execution of E.
$\mid \quad \tau/E$	**Internal reaction** It is represented with τ/E, this term doesn't have its explicit counterpart in the original $\rho-Calculus$. It might demand specifying many transitions as internal reactions to limit the quantity of observations [11].
$\mid \quad \exists w E$	**Declaration** $\exists w E$ introduces a reference w with scope E.
$\mid \quad x : \overline{y}/E$	**Replication** $x : \overline{y}/E$ can be expressed as: $x : \overline{y}/E \equiv x :: \overline{y}/E \ \wedge \ x : \overline{y}/E$ It produces a new abstraction, ready for reaction and it allows of replicating another when necessary.
$\mid \quad E^{\top}$	E**'s succesful execution** Observable succesful execution of E
$\mid \quad E^{\perp}$	E**'s non succesful execution** Observable non succesful execution of E
$\mid \quad OSO(E)\ do\ F\ else\ G$	**On Success Of** If E executes with succes then it redirects to execute architectural expression F else it redirects to execute the architectural expression G.
$\mid \quad !OSO(E)\ do\ F\ else\ G$	**Replication of OSO rule** Consecutives observations of "On Succes Of " rule on the same component.
$\phi, \psi ::= \quad \dot{\top}$	**Logical truth** Constraints as ϕ, ψ can resolve to true ($\dot{\top}$).
$\mid \quad \dot{\perp}$	**Logical false** Constraints as ϕ, ψ can resolve to false ($\dot{\perp}$).
$\mid \quad x = y$	**Equational restriction** Constraints can correspond to equational constraints ($x = y$) with logical variables. The information about values of variables can be determined by means of equations that can be seen as constraints. The equations can be expressed as total information (i.e.: $x = a$) or partial information(i.e.: $x = y$); taking into account that the names are only values loaded to variables. [21].
$\mid \quad \phi \dot{\wedge} \psi$	**Conjunction of constraints** Constraints can correspond to conjunction ($\phi \dot{\wedge} \psi$); the conjunction is congruent to constraints' composition. This leads to constraints that must be explicitly combined by means of reduction [15]:
$\mid \quad \dot{\exists} \phi$	**Existential quantifier** The existential quantification over constraints is congruent to the variables declaration over constraints ($\exists x \phi$).

It uses structural congruence ($\equiv$) from ρ calculus, that holds for the least congruence (least logical relationship of equivalence) of the axioms and the reduction rules that represent the semantics (See Table 2). About new axioms showed on Table 2: *Observable replication*, it allows to do successive observations in the component execution and, *Observable Successful/Failure* that allows to do

Table 2. Structural congruence rules of ρ_{arq} calculus. Source: [9,15,19]

$(\alpha - conversión)$	Change of bounding references by free references
(ACI)	$\wedge$ It's associative, conmutative and satisfies $E \wedge \top \equiv E$
$(Interchange)$	$\exists x \exists y E \equiv \exists y \exists x E$
$(Scope)$	$\exists x\ E \wedge F \equiv \exists x (E \wedge F) \quad if\ x \notin \mathcal{FV}(F)$
$(Equivalence\ of\ Constraints)$	$\phi \equiv \psi \quad if\ \phi \models\!\!\models_{\Delta} \psi\ y\ \mathcal{FV}(\phi) = \mathcal{FV}(\psi)$
$(Observable\ replication)$	$!OSO(E)\ do\ F\ else\ G \equiv OSO(E)\ do\ F\ else\ G \wedge !OSO(E)\ do\ F\ else\ G$
$(Observable\ Succesful/Failure)$	$[v/w]E^{(int)} \equiv \dagger \wedge if\ [\ (\dagger\ then\ E^{\top})\ ,\ (\dagger\ then\ E^{\bot})\ ]\ else\ (\top)$

replacements in a component inputs and execute it. A successful/failure observation can be represented by $(E^{\top}, E^{\bot})$ respectively.

Calculus models behavior of component-based architectures through its operational semantic. So then, the calculus uses labeled transition systems (LTS) to show the evolution in the execution of the architecture through the operational semantic defined for this purpose. Additionally, it has a graphical notation based on stereotyped extension of UML 2.x that translates to calculus [9].

Table 3. Rewriting rules of ρ_{arq} calculus. Source: [9,15,19]

$(A_{\rho arq})$	$\phi \wedge x : \overline{y}/E \wedge x' \overline{z}/F \longrightarrow \phi \wedge x : \overline{y}/E \wedge [\overline{z}/\overline{y}]E^{(int)} \wedge F \quad si\ \phi \models_{\Delta} x = x', \mathcal{V}(\overline{z}) \cap \mathcal{BV}(E^{(int)}) = \emptyset$
$(C_{\rho arq})$	$\phi_1 \wedge \phi_2 \longrightarrow \psi \quad if\ \phi_1 \wedge \phi_2 \models\!\!\models_{\Delta} \psi$
$(Comb_{\rho arq})$	$\phi \wedge if\ (C_1) \ldots (C_n)\ else\ F\ fi \longrightarrow \begin{cases} E_k, & if\ \phi \models_{\Delta} \psi_k \\ F, & if\ \phi \models_{\Delta} \neg\psi_k\ \forall k = 1, 2, \ldots, n \end{cases}$ $Donde\ C_k ::= \exists \overline{x}(\psi_k\ Then\ E_k)\ ;\ k = 1, 2, \ldots, n$
$(Ejec_{\tau})$	(a) $[OSO(E)\ do\ F\ else\ G] \wedge E^{\top} \longrightarrow F, Because\ of\ succesful\ execution\ of\ E\ component$ (b) $[OSO(E)\ do\ F\ else\ G] \wedge E^{\bot} \longrightarrow G, Because\ of\ non\ succesful\ execution\ of\ E\ component$

The rules at Table 3 specifying formally the progress in the execution of an architecture and they can be interpreted in this way:

- $A_{\rho arq}$: *Application*, executes a concurrent combination of an abstraction with a replication that instances another application. This rule models remote procedures calls passing parameters within a component.
- $C_{\rho arq}$: *Constraint combination* allows to combine restrictions with the purpose to extend or to simplify the rule set in the repository.
- $Comb_{\rho arq}$: *Committed choice combinator* triggers the execution of an E_k component, if the context constraint is enough strong and it allows to deduce from ϕ, the guard ψ_k in the conditional. This rule chooses a component E_k within a group, as long as it holds the defined guard.
- $Ejec_{\tau}$: It sets the observational success/failure execution of a component. This is made with the purpose to represent a component as a black box, where the relevant part is the final behavior in its execution but not the internal processing, that is nos visible to an external observer.

2.2 Linear Temporal Logic

Linear Temporal Logic is built from the syntax and semantic described in CTL* [8] with the constraint that it does not use quantifiers. Its formulas are uniquely path-oriented and the representation does not generates a tree structure but one unique path.

This logic has been used to model synchronous systems whose components act step by step. This means, a transition progress in discrete time; the present moment is defined as the actual state and the next moment is the successor. The system is observable in 0, 1, 2, ... moments. A graphical representation over some of the LTL operators is shown in Fig. 1.

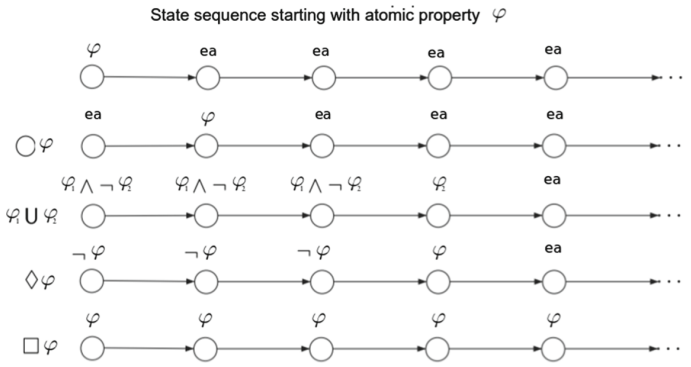

Fig. 1. Graphical representation of LTL operators. (Adapted from [3,8])

Syntax and Semantic to Check Temporal Properties. Syntax and semantic used to specify temporal properties of an software architectural model described through calculus were based on [5] to specify properties in a reactive system. A formula was built to specify LTL properties; this formula was composed by atomic propositions represented by $a_i \in AP$ where a_i is a state label (or an alphabet **letter**) in the system, the basic boolean connectors $\wedge, \vee, \neg$ (*and*, *or*, *not*), the basic temporal modalities $\bigcirc, \bigcup$ (*next*, *until*) and φ, φ_1, φ_2 which are LTL formulas. Thus, a LTL formula can be expressed through Bakus-Naur notation:

$$\varphi ::= true \,|\, a_i \,|\, \varphi_1 \wedge \varphi_2 \,|\, \neg\varphi \,|\, \bigcirc \varphi \,|\, \varphi_1 \bigcup \varphi_2$$

Rules of Logical Equivalence. The implication and double implication logical connectors can be expressed through the basic operators ($\wedge$, $\vee$). Likewise, the compound temporal modalities as $\Diamond$ (*eventually*), $\Box$ (*globally*), $\Diamond\Box$ (*eventually forever*) and $\Box\Diamond$ (*infinitely often*) can be rewritten through the basic operators. These rules are:

$$\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2) \quad (1) \qquad \varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2 \quad (2)$$

$$\varphi_1 \leftrightarrow \varphi_2 = (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \quad (3) \qquad \Diamond\varphi = true \bigcup \varphi \quad (4)$$

$$\Box\varphi = \neg\Diamond\neg\varphi \quad (5) \qquad \Box\Diamond\varphi = \Box(true \bigcup \varphi) \quad (6)$$

$$\Diamond\Box\varphi = true \bigcup (\Box\varphi) \quad (7)$$

2.3 Definitions

Some useful definitions were established by [3,5]:

- AP is the atomic proposition set of the system, it means: $AP = \{a_0, a_1, a_2, \ldots, a_n\}$
- $\mathcal{P}(AP)$ is the power set over AP (set made up of all subsets over AP), it means:

$$\mathcal{P}(AP) = \{\{\}, \{a_0\}, \{a_1\}, \{a_2\}, ..., \{a_n\}, ..., \{a_0, a_1\}, \{a_0, a_2\}, \ldots, \{a_0, .., a_n\} \ldots \{a_1, a_2\} \ldots\}$$

- A *word* is a sequence of elements over a set. For example: a *word* over AP would be a_0a_2 or over $\mathcal{P}(AP)$ would be $\{a_0\}\{a_1\}\{a_0, a_1\}\{a_2\}$
- AP_{INF} is the infinite set composed by all *words* over the power set $\mathcal{P}(AP)$. For example:

$$AP_{INF} = \{\{a_0\}, \{a_0\}\{a_1\}, \{a_0\}\{a_1\}\{a_0\}, \{a_0\}\{a_1\}\{a_0, a_1\}, \{a_0\}\{a_1\}\{a_0, a_1\}\{a_0\}\{a_2\}\{a_0, a_2\} \ldots\}$$

- A property defined over AP is a subset of AP_{INF}
- $Traces(a_i)$ is the path set which initial state is a_i. $Traces(a_i) \subseteq AP_{INF}$
- $Traces(ST)$ is the path set which all initial states in the transition system. $Traces(ST) \subseteq AP_{INF}$

With these definitions, an example of a property specification over a model can be expressed as: a_1 is always true.

This property can be represented as $\{A_0A_1A_2 \ldots A_n \in AP_{INF}\}$ where each A_i contains $\{a_1\}$, in this case, a set of *words* that satisfies the property consists in: $\{\{a_1\}, \{a_1\}\{a_0, a_1\}, \{a_1\}\{a_1\}\{a_1, a_2\}, \{a_1\}\{a_0, a_1, a_2\}, \ldots\}$. In this way, a formula specified in LTL describes subsets of AP_{INF}, it means that, a given formula LTL φ, can be associated with a *words* set identified with the expression $Words(\varphi)$ whose elements belong to the sequence of states reached in each transition. If φ is a LTL formula:
$\varphi \longrightarrow Words(\varphi) \subseteq AP_{INF}$; where $Words(\varphi)$ is the set that satisfies the formula:
φ: $Words(\varphi) = \{\sigma \in AP_{INF} \mid \sigma \text{ satisfies } \varphi\}$

Verification Rules. To determine if a word satisfies a formula, the next rules are applied:

With the word σ established: $Word\,\sigma : A_0A_1A_2 \ldots A_n \in AP_{INF}$

- Each word in AP_{INF} satisfies $true$.

$$Words(true) = AP_{INF} \quad (8)$$

- σ satisfies a_i, if $a_i \in A_0$.

$$Words(a_i) = \{A_0A_1A_2 \ldots \mid a_i \in A_0\} \quad (9)$$

– σ satisfies $\varphi_1 \wedge \varphi_2$, if σ satisfies φ_1 and σ satisfies φ_2.

$$Words(\varphi_1 \wedge \varphi_2) = Words(\varphi_1) \cap Words(\varphi_2) \tag{10}$$

– σ satisfies $\varphi_1 \vee \varphi_2$, if σ satisfies φ_1 or σ satisfies φ_2.

$$Words(\varphi_1 \vee \varphi_2) = Words(\varphi_1) \cup Words(\varphi_2) \tag{11}$$

– σ satisfies $\neg\varphi$, if σ not satisfies φ.

$$Words(\neg\varphi) = Words(\varphi)' \tag{12}$$

– σ satisfies $\bigcirc\varphi$, if $A_1A_2\ldots$ satisfies φ.

$$Words(\bigcirc\varphi) = \{A_0A_1A_2\ldots \mid A_1, A_2 \in Words(\varphi)\} \tag{13}$$

– σ satisfies $\varphi_1 \bigcup \varphi_2$, there is j such as $A_jA_{j+1}\ldots$ satisfy φ_2 and for all $0 \leq i < j$ $A_iA_{i+1}\ldots$ satisfy φ_1.

$$Words(\varphi_1 \bigcup \varphi_2) = \{A_0A_1A_2\ldots \mid \exists\, j.A_jA_{j+1}\ldots \in Words(\varphi_2)$$
$$\text{and } \forall\, 0 \leq i < j,\ A_iA_{i+1}\ldots \in Words(\varphi_1)\} \tag{14}$$

– σ satisfies $\Diamond\varphi = true \bigcup \varphi$, if there is a j such as $A_jA_{j+1}\ldots$ satisfies φ and for all $0 \leq i < j$ $A_iA_{i+1}\ldots$ satisfies $true$.

$$Words(\Diamond\varphi) = \{A_0A_1A_2\ldots \mid \exists\, j.A_jA_{j+1}\ldots \in Words(\varphi)$$
$$\text{for } \forall\, 0 \leq i < j.A_iA_{i+1}\ldots \in Words(\varphi) \tag{15}$$

– σ satisfies $\Box\varphi = \neg\Diamond\neg\varphi$, for this case it decomposes in:
  1. σ satisfies $\Diamond\neg\varphi$, if there is a j such as $A_jA_{j+1}\ldots$ satisfies $\neg\varphi$
  2. σ satisfies $\neg\Diamond\neg\varphi$, if σ not satisfies $\Diamond\neg\varphi$.
  3. σ satisfies $\neg\Diamond\neg\varphi$, if for all j such as $A_jA_{j+1}\ldots$ satisfies φ

$$Words(\Box\varphi) = \{A_0A_1A_2\ldots \mid \forall\, j.A_jA_{j+1}\ldots \in Words(\varphi) \tag{16}$$

3 The Method

The architectural configuration showed in the Fig. 2 was used to specify and verify the "responsiveness" property.

Formulas that specifies this configuration through the ρ_{arq}-calculus are:

$$E \stackrel{def}{=} [(p_E : x/xs_E)] \wedge \exists l_E[(r_E :: y/yl_E) \wedge (l_E :: i_E/E^{(int)})] \tag{17}$$

$$F \stackrel{def}{=} (p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe}) \tag{18}$$

$$M \stackrel{def}{=} \exists l_M[(r_M :: y/yl_M) \wedge (l_M :: i_M/M^{(int)})] \tag{19}$$

$$T \stackrel{def}{=} [(p_{Te} : n/ns_{Te})] \wedge \exists l_T[(r_T :: q/ql_T) \wedge (l_T :: i_T/T^{(int)})] \tag{20}$$

$$C_FE = r_E\overline{p_F} \tag{21}$$

$$C_FM = r_M\overline{p_Fe} \tag{22}$$

$$C_ET = r_T\overline{p_E} \tag{23}$$

$$C_EM = r_M\overline{p_Ee} \tag{24}$$

$$C_TM = r_M\overline{p_Te} \tag{25}$$

The system's initial configuration is:

$$\begin{aligned} S = [F \ \wedge\ OSO(F)\ do\ C_FE \ \wedge\ E\ else \ \wedge\ C_FM \ \wedge\ M\ ] \\ \wedge\ [\ OSO(E)\ do\ C_ET \ \wedge\ T\ else\ C_EM \ \wedge\ M\ ] \\ \wedge\ [\ OSO(T)\ do\ T\ else\ C_TM \ \wedge\ M\ ] \end{aligned} \tag{26}$$

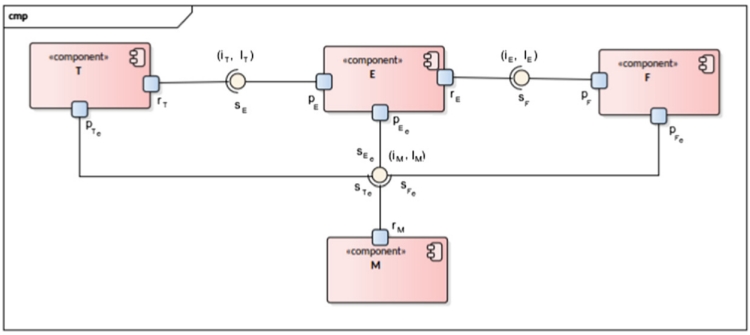

Fig. 2. Complex assembly of components. (Source [9])

3.1 APTS Generation

Atomic Propositions Transition System (APTS) represents the system states associated to atomic propositions. This model was generated by the architecture definition (components and connectors) and execution rules provided by the ρ_{arq}-calculus.

First Step: Identify the Source Components. Only they can start the execution. Each component inside the model represents a state: its execution (i.e. E). Transitions that they can take represent the successful or failure execution states (i.e. $E^{\top}$ and $E^{\perp}$ respectively), this behavior is shown in the Fig. 3.

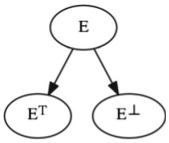

Fig. 3. APTS representation of a component execution.

Second Step: Obtain Transitions Between States. It is done capturing system's behavior through evaluation of observation rules disposed in the formula. An observation rule is described as:

$$OSO(F)\, do\, [C_F E \wedge E]\, else\, [C_F M \wedge M] \tag{27}$$

From this rule it can be built links between states. If F executes successfully it communicates with E, else it links with M. In the arrival of new components, the first step is repeated and the APTS is developed until there are no more components to analyze.

Third Step: Close Process. In the moment to achieve end states (terminal states) they indicate a global state of the system. It proceeds to set if the system as a whole is executed or not successfully. This state only can be obtained from the terminal nodes whose execution represents one of the success or failure global final states. When it reaches one of these states, it connects again with the nodes that represent the source components for evaluating a new execution (Fig. 4).

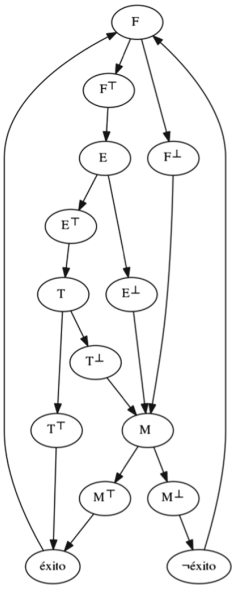

Fig. 4. Full APTS representation

3.2 Verification of a Temporal Property

Atomic properties were made up from the system states and the defined operators were used for this purpose. One property was specified in order to check if the system could fulfill it. For example (with previous APTS):

$$\varphi = F \rightarrow \Diamond(T^{\top} \vee M^{\top}) \tag{28}$$

This property defines if F executes, eventually in the future M or T will execute successfully.

A temporal property specifies paths (states sequences) that a transition system should expose (states that can be observable); these properties specify desired or allowable behavior expected from the system.

Formally, a property P is a subset of AP_{INF} where AP is the set of atomic propositions and AP_{INF} represents the words set that come from infinite words concatenation in $\mathcal{P}(AP)$. Therefore, a property is an infinite words set over

$\mathcal{P}(AP)$ [3] or in another way, if the execution of a program is a σ infinite states sequence, it is said that property P is true in σ if the sequences set defined by the program are contained in the property P [1].

One of these properties is named *liveness*. This property declares intuitively that "something good" eventually will happen or that a program eventually will reach a desired state [2,6,17]. Specifically, the *liveness* property shows one of the next behaviors: *Starvation freedom*, *Termination* and *Guaranteed service/Responsiveness*.

For this work was proposed the verification of the "Responsiveness" property, a subtype of liveness property. To check this property, the experiment proceeded from the simplest case until to reach the most complex case.

For $\varphi = F^{\top} \rightarrow \Diamond M^{\top}$

The following sets are by definition:

$$AP = \{F, F^{\top}, F^{\perp}, E, E^{\top}, E^{\perp}, T, T^{\top}, T^{\perp}, M, M^{\top}, M^{\perp}, success, failure\}$$
$$AP_{INF} = \{\{\}, \{F\}, \{F^{\top}\}, \{F^{\perp}\}, \{E\}, \{E^{\top}\}, \{E^{\perp}\}, \{T\}, \{T^{\top}\}, \{T^{\perp}\}, \{M\}, \{M^{\top}\}, \{M^{\perp}\}, \{F\}\{F^{\top}\}, \{F\}\{F^{\perp}\}, \{F\}\{F^{\top}\}\{E\}, ...\}$$

The system complies the formula φ if:

$$Words(\varphi) = Words(F^{\top} \rightarrow \Diamond M^{\top}) \text{ and } Traces(TS) \cap Words(\varphi)$$

Applying the rules of composition, the formula can be expressed in subformulas that can be evaluated in the following sequence:

$$Words(\varphi) = Words(F^{\top} \rightarrow \Diamond M^{\top}); \quad \text{applying(2)}$$
$$Words(\varphi) = Words(\neg F^{\top} \vee \Diamond M^{\top}); \quad \text{applying (11)}$$
$$Words(\varphi) = Words(\neg F^{\top}) \cup Words(\Diamond M^{\top}); \quad \text{applying(12)}$$
$$Words(\varphi) = Words(F^{\top})' \cup Words(\Diamond M^{\top}); \quad \text{applying(4)}$$
$$Words(\varphi) = Words(F^{\top})' \cup Words(true \bigcup M^{\top})$$

the first terms set was described by extension:

$$Words(\varphi) = \{\{F\}, \{F\}\{F^{\perp}\}, \{F\}\{F^{\perp}\}\{M\}, \{F\}\{F^{\perp}\}\{M\}\{M^{\top}\},$$
$$\{F\}\{F^{\perp}\}\{M\}\{M^{\top}\}\{success\}, \{F\}\{F^{\perp}\}\{M\}\{M^{\perp}\},$$
$$\{F\}\{F^{\perp}\}\{M\}\{M^{\perp}\}\{\neg success\}\} \cup Words(true \bigcup M^{\top}); \quad \text{applying(14)}$$
$$\ldots$$

The process continues recursively until sets of states sequence that satisfy the specified property are found.

Likewise, the sequences that must be delimited inside the set of possible paths offered by the APTS must be filtered, therefore must be satisfied that $Traces(TS) \cap Words(\varphi)$, where $Traces(TS)$ is the set of possible paths in the transition system that are determined for the initial states in the system.

In this way, only paths that make true the temporal property for the APTS with the paths that begin with the initial states are filtered. The Fig. 5 shows the relationship between these elements; the intersection area represents the elements that satisfy the property.

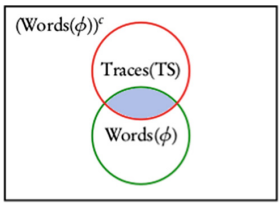

Fig. 5. Relationship between $Traces(TS)$ and $Words(\varphi)$ Source [5]

In this system it can be seen there are paths that move to successful execution of M $(M^{\top})$: $\{F\}\{F^{\top}\}\{E\}\{E^{\top}\}\{T\}\{T^{\perp}\}\{M\}\{M^{\top}\}$ and thus, property is satisfied for the system. On the other hand, a property that the system does not satisfy is the following: $\varphi =!(F \rightarrow \Diamond \neg success)$. It can be identified that not all paths lead to the successful execution and there is at least one path that flows towards an unwanted state of failure.

4 Results and Discussion

4.1 Mapping Architectural Specification to APTS

One of the worthy results in this job is the achieved method to map structural and behavioral architectural configurations using the ρ_{arq}-calculus to an ATPS that allows checking properties using LTL formulas. This effort is pertinent for academic communities that work and research about model checking possibilities. This result indicates that ρ_{arq}-calculus expressions are translatable to LTL formulas and the other properties could be analyzed.

4.2 PintArq Extension to Check LTL Properties

In the first version of the software, an architectural execution flow visualizer was achieved with the work described in [19] and it was named PintArq. It was built to support the method described in Sect. 3. To implement this solution, the software process development phases were carried out with their functional, structural an dynamic models that are completely documented and for free access in the ARQUISOFT Research Group portal. The prescriptive architecture of the application is illustrated in Fig. 6. This architecture is composed for the next modules:

Original modules:

Interpreter: It identifies structural elements and it transforms architectural expressions to UML component-configuration.
Rewriter: It obtains original expressions and rewrites them which represents reactions in the system.
Architect: It generates components and connectors from interpreted structural elements.
Drawer: It shows on screen, component-based architecture through an UML Component diagram (in SVG image format).

Transformer: It transforms architecture in an XMI file for exchange with other systems.

Modules added in this project:

APTS Generator: It creates the APTS based on the architecture (components and connectors) and initial calculus expressions.
Property Checker: It takes property provided for the user and identifies whether it is satisfied by the system. If property is not satisfied, it creates a counter-example path that represents this behavior.

Model checking viewer: It shows on screen, the generated APTS and the possible path in case the property is not satisfied.

To extend the PintArq tool with new possibilities to analyze software component-based architectures from model checking perspective is very interesting and it opens new research motivations to apply this proposal to real software architectures. Nowadays, the Pintarq tool must be extended to graphical modelling of real software systems or the tool could be extended with a models interchange module. This module should import real software component-based models and it should translate these models to ρ_{arq}-calculus. With this translation the software architect could use the analysis capabilities of PintArq tool.

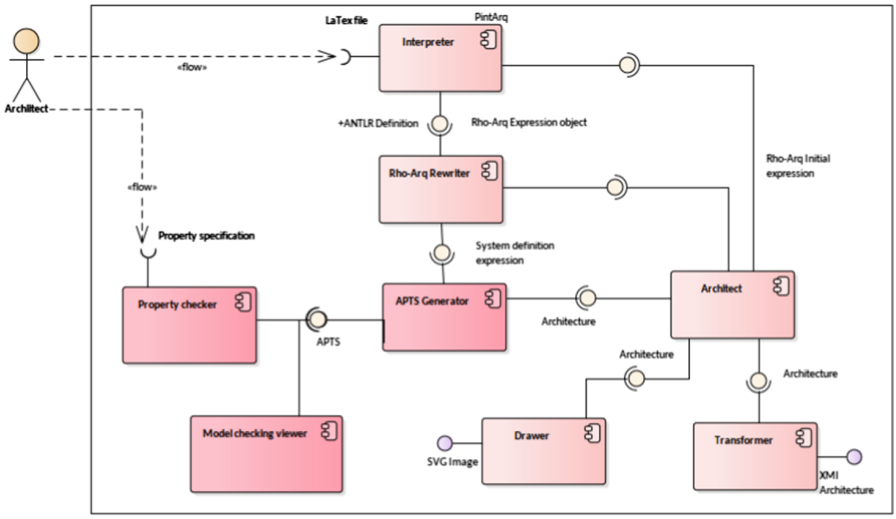

Fig. 6. PintArq's prescriptive architecture with additional components.

To obtain the detailed documentation about this project you can visit http://arquisoft.udistrital.edu.co, option: "Proyectos finalizados".

5 Future Work

Several scenarios on which it can start or continue related work are:

1. Expand the scope of the mechanism to verify temporal properties as *Safety*, *Deadlock detection* or *Starvation freedom*.
2. Optimize the implementations to find the shortest path that the counterexample illustrate when the model does not satisfy the specified property.
3. Improve the implementation of the extension in PintArq to visualize the counterexample in the syntax of the formal ρ_{arq}-calculus and not in the LaTeX format.
4. Expand the analysis of possibilities to other temporal logics: Branch temporal logic, intuitionistic temporal logic and temporal multi-valued logic.

References

1. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, London (2008)
4. Barendregt, H., Barendsen, E.: Introduction to lambda calculus. Nieuw archief voor wisenkunde **4**(Mar), 337–372 (2000). http://homepages.nyu.edu/~cb125/Lambda/barendregt.94.pdf
5. Biswas, S., Deka, J.K.: NPTEL: Computer Science and Engineering - Design Verification and Test of Digital VLSI Circuits (2014). http://nptel.ac.in/courses/106103116/19
6. Cheung, S.C., Giannakopoulou, D., Kramer, J.: Verification of liveness properties using compositional reachability analysis. In: Jazayeri, M., Schauer, H. (eds.) ESEC/SIGSOFT FSE -1997. LNCS, vol. 1301, pp. 227–243. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63531-9_17
7. Cirstea, H., Kirchner, C.: ρ-Calculus. Its Syntax and Basic Properties, vol. 53, no. 9 (1998)
8. Clarke, E.: Model Checking. MIT Press, London (2000)
9. Diosa, H., Díaz Frías, J.F., Gaona, C.M.: Especificación formal de arquitecturas de software basadas en componentes: chequeo de corrección con cálculo rho-arq, no. 12 (2010)
10. Diosa, H.A., Díaz, J.F., Gaona C.M.: Cálculo para el modelado formal de arquitecturas de software basadas en componentes: cálculo ρ_{arq}. Revista Científica. Universidad Distrital Francisco José de Caldas, no. 12 (2010)
11. Bertolino, A., Inverardi, P., Muccini, H.: Formal methods in testing software architectures. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 122–147. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39800-4_7
12. Micskei, Z., Waeselynck, H.: The many meanings of UML 2 sequence diagrams: a survey. Softw. Syst. Model. **10**(4), 489–514 (2010). https://doi.org/10.1007/s10270-010-0157-9
13. Milner, R.: Communicating and Mobile Systems: The Pi Calculus. Cambridge University Press, Cambridge (1999)

14. Montoya Serna, E.: Métodos formales e Ingeniería de Software. Revista Virtual Universidad Católica del Norte, no. 30, 1–26 (2011). http://revistavirtual.ucn.edu.co/index.php/RevistaUCN/article/view/62
15. Niehren, J., Müller, M.: Constraints for free in concurrent computation. In: Kanchanasut, K., Lévy, J.-J. (eds.) ACSC 1995. LNCS, vol. 1023, pp. 171–186. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60688-2_43
16. Object Management Group: OMG Unified Modeling Language (OMG UML), version 2.5, March 2015
17. Owicki, S., Lamport, L.: Proving liveness properties of concurrent programs. ACM Trans. Program. Lang. Syst. **4**(3), 455–495 (1982)
18. Parrow, J.: An Introduction to the pi-Calculus (2001). http://homepages.nyu.edu/~cb125/Lambda/barendregt.94.pdf
19. Rico, J.A.: Representación visual de la ejecución de una arquitectura de software basada en componentes con especificación formal en cálculo ρarq (2015)
20. Smolka, G.: A calculus for higher-order concurrent constraint programming with deep guards. Technical report, Bundesminister für Forschung und Technologie (1994)
21. Smolka, G.: A Foundation for Higher-order Concurrent Constraint Programming. Tech. rep., Bundesminister für Forschung und Technologie (1994)
22. Wing, J.M.: FAQ on π-Calculus, pp. 1–8, December 2002