PintArq: A Visualizer of Architectural Execution Flow for Component-Based Software Architectures

Jorge Alejandro Rico García and Henry Alberto Diosa(✉)

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

Abstract. Formal effort required to specify and analyze architectures using formal languages is high. This has motivated us to build a software tool that allows the interpretation of component-based software architecture described using ρ_{arq} calculus. This tool offers the display facility to architects on a graphic way the structure and the architectural execution flow described in the formal expressions under study. For the development of this software tool some different modules were considered, altogether, they interpret expressions in accordance with the syntax and the operational semantics of the ρ_{arq} calculus; in addition, the tool maps the formal expressions to UML 2.x notation graphic elements. In this way, the application displays the architectural configuration using a visual modeling language(UML components) while showing the architectural execution flow by highlighting the provision interfaces when a ρ_{arq} calculus rewriting rule is executed. The ρ_{arq} calculus use is simplified with this. The architectural analysis tasks will be easier and the architect could focus on the architectural behavior and not on the calculus itself.

Keywords: ρ_{arq} calculus · Component-based software · Architectural execution flow · UML

1 Introduction

Software architecture has received plenty attention in the last decade because it allows better comprehension, reuse levels, control and management capabilities of software development projects [18]. This software engineering's knowledge-area emerged at end of the past century [10]. Subsequently, several architectural description languages has been proposed with less or more formal approach (See Table 1 for a not exhaustive list of related work).

Some approaches have proposed UML as the basis to describe software architectures [11,22]. Although UML has been a popular modeling language for many years [30,33], this modeling language is still semiformal. However, nowadays it supports component-based software models that include the essential concepts

Table 1. Related work about structural and dynamic modeling in several ADLs.

ADL	¿STRUCTURAL MODELLING?	¿DYNAMIC MODELLING?
WRIGHT [1]	Connector types were used to describe the interaction between components	The interaction was modelled with Hoare's CSP [16]
UNICON [36]	Compositional design of software architectures. This ADL use conector types	It doesn't propose a formal model for this aspect
RAPIDE [19]	Module specifications were used to describe the wired components, connection rules and restrictions to identify legal and illegal assembly patterns of modules	Use Partially Ordered Set of Events (POSET model)
SYNTACTIC APPROACH [4]	Set theory was used to model node types and connections at software architectures. It can model partially architectures or static perspective of sub-architectures	It doesn't propose a formal model for this aspect
UML [22]	From UML 2.x this language provides the component, provide and require interfaces as architectural abstractions. Assembly connectors and composite structures allow complex architectural configurations	The interaction diagrams and state machines provide semiformal modelling possibilities. Meta-modelling extensions can be used to support analysis of dynamic properties
AO-ADL [34]	It supports the definitions of component, connector and functional restrictions on connections. The restrictions on interfaces should be satisfied	Temporal Logic (TL) and other tools that
ACME [9]	It uses annotations to specify structural properties or additional restrictions	The last versions have the ACMELib library. It allows to programme the behavior of architectures. ACME is inherently extensible and it allows architects to associate an external formal model. This model could specify the dynamic aspects of software architectures
DARWIN [20]	It supports hierarchical models. It uses canonical textual representations. These representations describe the components and their interfaces	It uses the $\pi - calculus$ as formal tool. It models dynamic architectures (i.e. architectures changing on execution time)
xADL [3]	It defines the basic structures of prescriptive software architectures: Components, connectors, interfaces, links and groupings	It doesn't propose a formal model for this aspect
Weaves [13]	It uses directed bipartite multigraphs to model interconnected networks of components. It can be seen as a variant of the architectural style Pipe-and-Filter	It doesn't propose a formal model for this aspect
CHAM [17]	The syntax models structures and configuration. This syntax uses the analogy of chemical solutions and molecules	It has an expressions rewriting system based on chemical reaction concept
KOALA [28]	It describes structures, configuration and component interfaces within the domain of electronic devices. It inherits properties of Darwin language	Idem to Darwin language
ADML [40]	It specializes ACME with meta-properties	Analogous to ACME
ASDL [35]	It uses Z language to specify structure and static restrictions	It adds Hoare's CSP expressions to specify dynamic aspects on the interfaces
AADL [8]	Quality attributes driven design is supported. It uses compliance static analysis and data consistency	It accepts extensions to formal methodsde trabajo
π-ADL [29]	It provides graphical and textual syntax in accord with UML 2.x profile. This profile models software architectures	π-calculus typed of high order
SAM [15]	It can use a graphical and textual syntax. It allows horizontal and vertical hierarchical partitioning	It allows graphical simulation. Formal techniques as Petri nets and Temporal Linear Logical can be used

of software architecture [27]. Extending UML with a formal tool allows exploiting the analysis possibilities that formal tools offer while the software architects stays within the same design framework. We propose PintArq, a tool allowing software architects to visualize the architectural execution flow.

The PintArq tool uses a built-in interpreter that transforms ρ_{arq} calculus expressions [5,6] into Tex format [12] to visual representation of software architectures as wired components in concordance with UML 2.x notation [27]. Additionally, the tool enables users to show the execution flow in accordance with ρ_{arq} calculus rewriting rules (Operational semantics).

To begin with we show an overview of the ρ_{arq} calculus. Then, the methodology to develop the PintArq tool is explained step by step. Third section analyses results from three points of view: Transformation tools, logical architecture and technological aspects. The last sections discuss some conclusions and future work.

2 ρ_{arq} Calculus Overview

The ρ_{arq} calculus is a formalism to specify component-based software architectures; this calculus models dynamic and structural aspects with the possibility to control architectural executions based in boolean guards. As all formal calculus it comprises a syntax and semantics. Table 2 summarizes the syntactic entities with their meanings and interpretations of ρ_{arq}.

Replacements of any expression by other is governed by structural congruence rules (Table 3).

The semantic of ρ_{arq} calculus is based on rewriting rules. These operational semantic rules are shown in Table 4.

For illustration purposes, some basic examples of several architectural configurations and architectural execution flow are shown at Table 5[1].

3 Method

The first version of PintArq project involved the following steps:

3.1 The Study of the ρ_{arq} Calculus

For the purpose of representing ρ_{arq} calculus expressions using the extended BNF [25,32], the TEX format [12] was used. This activity was crucial in order to specify the formal source language that the interpreter transforms to other visual language. The Table 6 shows the equivalence between ρ_{arq} expressions against TEX expressions.

[1] For more details about syntax, semantics and examples of architectural execution control (i.e. architectural control flow) see [6,7].

Table 2. Syntax of ρ_{arq} calculus. Source: [6,23,26,38,39]

SYMBOLS		MEANING
$x, y, z, \ldots$	variables	Variables only hold names.
$a, b, c, \ldots$	names	Names and variables are named references.
$u, v, w, \ldots ::= x \mid a$	references	
EXPRESSIONS		**INTERPRETATION**
$E, F, G ::= \top$	**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 $\gamma - Calculus$ [38] [39] is a useful generalization of conventional conditional[a].
$\mid \quad x :: \overline{y}/E$	**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**	The **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 [2].
$\mid \quad \exists w E$	**Declaration**	The **Declaration** $\exists w E$ introduces a reference w with scope E.
$\mid \quad x : \overline{y}/E$	**Replication**	The 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.
$\phi, \psi ::= \dot{\top}$	**Logical truth**	Constraints as ϕ, ψ can resolve to true ($\dot{\top}$).
$\mid \quad \perp$	**Logical false**	Constraints as ϕ, ψ can resolve to false ($\perp$).
$\mid \quad x = y$	**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. [39].
$\mid \quad \phi \dot{\wedge} \psi$	**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 [26]:
$\mid \quad \dot{\exists}\phi$	**Existential quantifier**	The existential quantification over constraints is congruent to the variables declaration over constraints ($\exists x \phi$).

[a]Where $C_k ::= \exists \overline{x}(\phi_k\ then\ E_k)$ with $k = 1 \ldots n$ are arguments. Clauses $(C_1) \ \cdots \ (C_n)$ contains guards, if the guard of a clause is satisfied its body E_k is liberated for reaction; otherwise, this clause is ignored.

Table 3. Structural congruence rules of ρ_{arq} calculus. Source: [6,26]

$(\alpha - conversión)$	Cambio de referencias ligadas por referencias libres
(ACI)	$\wedge$ es asociativa, conmutativa y satisface $E \wedge \top \equiv E$
$(Interchange)$	$\exists x \exists y E \equiv \exists y \exists x E$
$(Scope)$	$\exists x\ E \wedge F \equiv \exists x (E \wedge F)\quad if\ x \notin \mathcal{FV}(F)$
$(Equivalence\ of\ Constraints)$	$\phi \equiv \psi\quad if\ \phi \models\!\!\!\dashv_{\Delta} \psi$ y $\mathcal{FV}(\phi) = \mathcal{FV}(\psi)$
$(Observable\ replication)$	$!OSO(E)\ do\ F\ else\ G \equiv OSO(E)\ do\ F\ else\ G \wedge !OSO(E)\ do\ F\ else\ G$
$(Observable\ Succesful/Failure)$	$[v/w]E^{(int)} \equiv \dagger \wedge if\ [\ (\dagger\ then\ E^{\top})\ ,\ (\dagger\ then\ E^{\perp})\ ]\ else\ (\ \top\ )$

Table 4. Rewriting rules of ρ_{arq} calculus. Source: [6,26]

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

3.2 Transformation Technology: Review and Selection

Three alternatives were evaluated in order to transform from formal textual expressions to UML graphical notation: Model Driven Architecture (MDA) tools [14,37,41], DUALLY [21] and ANTLR [31].

Based in the possibilities and the capabilities of each tool, ANTLR was the selected tool to implement PintArq because it is extensible by using a programming language that supports the tasks derived to accomplish the rewriting calculus and the transformation to UML 2.x.

3.3 Analysis, Design and Implementation of Software

Four subactivities were conducted:

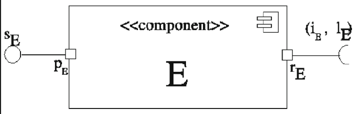
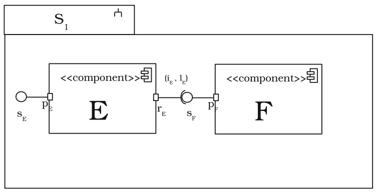
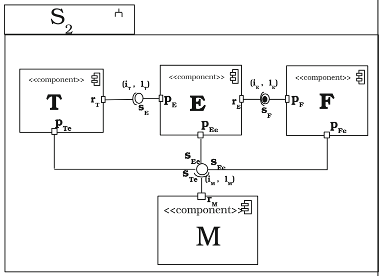
1. Definition of prescriptive architecture.
2. Analysis and design.
3. Programming of the "web enabled" application.
4. Specification of $\rho_{arq}\,Calculus$ grammar and basic architectural expressions proposed in [5,6] for the testing phase.

3.4 Concept Testing

Once the application was developed, a set of tests was executed to verify the appropriate interpretation of expressions and rewriting rules of architectures defined using ρ_{arq} calculus. The objective of these tests were:

– Verify operational semantics in action.
– Test mapping from ρ_{arq} calculus to UML's component-based diagrams.
– Review of visualization for architectural execution flows.

Table 5. ρ_{arq} calculus in action: a sample

 <<component>> E s_E p_E r_E (i_E, l_E)	**Individual component specification.** $PROV_E(p, s) \overset{def}{=} p_E : x/xs_E \equiv p_E :: x/xs_E \ \wedge \ p_E : x/xs_E$ $REQ_E(r, l, i) \overset{def}{=} \exists l_E[(r_E :: y/yl_E) \ \wedge \ (l_E :: i_E/E^{(int)})]$ then, the component is specified as: $E \overset{def}{=} PROV_E(p, s) \ \wedge \ REQ_E(r, l, i)$
 S_1 <<component>> E <<component>> F s_E p_E r_E (i_E, l_E) s_F p_F	**Components assembly.** $E \overset{def}{=} [(p_E : x/xs_E)] \ \wedge \ \exists l_E[\ (r_E :: y/yl_E) \ \wedge \ (l_E :: i_E/E^{(int)})]$ $F \overset{def}{=} (p_F : z/zs_F)$ and the connector was formally modelled as: $C_{FE} \overset{def}{=} r_E p_F$ then, components connected are: $S_1 = E \ \wedge \ F \ \wedge \ C_{FE}$ $= \{(p_E : x/xs_E) \ \wedge \ \exists l_E \ [(r_E :: y/yl_E) \ \wedge \ (l_E :: i_E/E^{(int)})]\} \ \wedge \ \{(p_F : z/zs_F)\} \ \wedge \ \{r_E p_F\}$
 S_2 <<component>> T E F M r_T (i_T, l_T) s_E p_E r_E (i_E, l_E) s_F p_F p_{Te} p_{Ee} p_{Fe} s_{Ee} s_{Fe} s_{Te} (i_M, l_M) r_M	**Architectural execution flow** Individual components are: $F \overset{def}{=} (p_F : z/zs_F) \ \wedge \ (p_{Fe} : w/ws_{Fe});$ $E \overset{def}{=} (p_E : x/xs_E) \ \wedge \ (p_{Ee} : v/vs_{Ee}) \ \wedge \ \exists l_E[(r_E :: y/yl_E \ \wedge \ (l_E :: i_E/E^{(int)})];$ $M \overset{def}{=} \exists l_M[r_M : y/yl_M \ \wedge \ (l_M : i_M/M^{(int)})];$ $T \overset{def}{=} (\exists l_T[r_T :: q/ql_T \ \wedge \ (l_T : i_T/T^{(int)})]) \wedge (p_{Te} : n/ns_{Te})$ Setting when F component is succesful in its execution: $S_2^{(0)} = F^{\top} \ \wedge \ [\ OSO(F) \ do \ F \ \wedge \ C_{FE} \ \wedge \ E \ else \ F \ \wedge \ C_{FM} \ \wedge M\} \ ] \ \wedge \ [\ OSO(E) \ do \ C_{ET} \ \wedge \ T \ else \ C_{EM} \ \wedge M \ ] \ \wedge \ [\ OSO(T) \ do \ S_2 \ = \ éxito \ else \ C_{TM} \ \wedge M \ ]$ The rewriting rules can be applied: $S_2^{(0)} \xrightarrow{Ejec_{\tau}} [\ F \ \wedge \ C_{FE} \ \wedge \ E \ ] \ \wedge \ [\ OSO(E) \ do \ C_{ET} \ \wedge \ T \ else \ C_{EM} \ \wedge M\} \wedge [\ OSO(T) \ do \ S_2 \ = \ éxito \ else \ C_{TM} \ \wedge M \ ]$ $S_2^{(1)} = \{[(p_F : z/zs_F) \ \wedge \ (p_{Fe} : w/ws_{Fe})] \ \wedge \ [r_E p_F] \ \wedge \ [(p_E : x/xs_E) \ \wedge \ (\exists l_E[r_E :: y/yl_E \ \wedge \ (l_E :: i_E/E^{(int)})]) \ \wedge \ (p_{Ee} : v/vs_{Ee})] \ \} \ \wedge \ [\ OSO(E) \ do \ C_{ET} \ \wedge \ T \ else \ C_{EM} \ \wedge M\} \ \wedge \ [\ OSO(T) \ do \ S_2 \ = \ éxito \ else \ C_{TM} \ \wedge M \ ]$ $\xrightarrow{A_{\rho_{arq}}} \{[(p_F : z/zs_F) \ \wedge \ (p_{Fe} : w/ws_{Fe})] \ \wedge \ [(p_E : x/xs_E) \ \wedge \ \exists l_E[(p_F l_E) \ \wedge \ (l_E :: i_E/E^{(int)})] \ \wedge \ (p_{Ee} : v/vs_{Ee})] \ \} \ \wedge \ [\ OSO(E) \ do \ C_{ET} \ \wedge \ T \ else \ C_{EM} \ \wedge M\} \ \wedge \ [\ OSO(T) \ do \ S_2 \ = \ éxito \ else \ C_{TM} \ \wedge M \ ]$ $\xrightarrow{A_{\rho_{arq}}} \{[(p_F : z/zs_F) \ \wedge \ (l_E s_F) \ \wedge \ (p_{Fe} : w/ws_{Fe})] \ \wedge \ [(p_E : x/xs_E) \ \wedge \ (p_{Ee} : v/vs_{Ee}) \ \wedge \ (l_E :: i_E/E^{(int)})] \ \} \ \wedge \ [\ OSO(E) \ do \ C_{ET} \ \wedge \ T \ else \ C_{EM} \ \wedge M\} \ \wedge \ [\ OSO(T) \ do \ S_2 \ = \ éxito \ else \ C_{TM} \ \wedge M \ ]$ $\xrightarrow{A_{\rho_{arq}}} \{[(p_F : z/zs_F) \ \wedge \ (p_{Fe} : w/ws_{Fe})] \ \wedge \ [(p_E : x/xs_E) \ \wedge \ (p_{Ee} : v/vs_{Ee}) \ \wedge \ ([s_F/i_E]E^{(int)})] \ \} \ \wedge \ [\ OSO(E) \ do \ C_{ET} \ \wedge \ T \ else \ C_{EM} \ \wedge M\} \ \wedge \ [\ OSO(T) \ do \ S_2 \ = \ éxito \ else \ C_{TM} \ \wedge M \ ]$ The token was located at input of E component. This situation shows the execution flow. When E component consumes the token, the $Observable \ Succesful/Failure$ rule acts accordingly. For more examples, the reader could review [6][7]

4 Results

4.1 ¿How ANTLR Was Applied?

Table 6 shows the correspondence between the ρ_{arq} calculus expressions and the TEX expressions:

Table 6. Equivalence of ρ_{arq} expressions to TEX expressions

ρ_{arq} expression	Typography	TEX
Composition	$A \wedge B$	A \wedge B
Null component	$\top$	\top
Interior of component	$A^{(int)}$	A^{(int)}
Committed choice combinator	$if(C_1...C_n)\, else\, A$	$if(C_1)(C_2)...(C_n)\, else\, A$
Abstraction	$x :: \overline{y}/E$	$x{::}y/E$
Application	$x\overline{y}/E$	$x\backslash overline\{y\}/E$
Declaration	$\exists wE$	$\backslash existswE$
Abstraction replication	$x : \overline{y}/E$	$x{::}y/E$
Successful execution	$E^{\top}$	E^{$\backslash top$}
Non-successful execution	$E^{\perp}$	E^{$\backslash bot$}
On Success Of	$OSO(E) do\, F\, else\, G$	$OSO(E)\, do\, F\, else\, G$
Replication of OSO rule	$!OSO(E)\, do\, F\, else\, G$	$!OSO(E)\, do\, F\, else\, G$
Logic truth	$\dot{\top}$	$\backslash dot\{\backslash top\}$
Logic false	$\perp$	$\backslash bot$
Equational restriction	$x = y$	$x = y$
Conjunction of constraints	$\phi\dot{\wedge}\psi$	$\phi\backslash dot\{\backslash wedge\}\psi$
Existential quantifier	$\dot{\exists}x\phi$	$\backslash dot\{\backslash exists\}x\phi$

Once the language definition was ready, the next step was to generate the API that let the manipulation and identification of each word in the architectural expressions. With this API it was possible to implement the Interpreter module by using the design patterns offered by ANTLR (Observer and Visitor) [31].

4.2 The Architecture of the Application

Figure 1 shows the architectural inception for PintArq. The component-connector diagram with stereotyped components as modules depicts the architectural configuration with assembly connectors; these lastly were labeled with the name of data-structures that flow between modules.

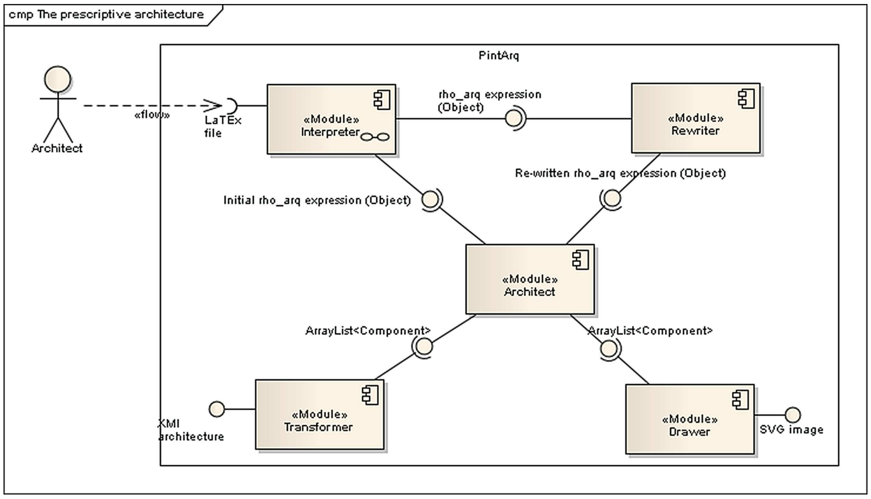


Fig. 1. The prescriptive architecture of PintArq

4.3 The Tool

The PintArq tool was built using the Java programming language, XML technologies (SVG and XMI) and ANTLR tool, the first version is web enabled. The tool is publicly available at: http://arquisoft.udistrital.edu.co/PintArq/Index.jsp. Some analyzable expressions can be downloaded from: http://arquisoft.udistrital.edu.co/documentos/EjemplosExpresionesArq.rar.

Figure 2 shows the PintArq's graphical user interface when the user loads an architectural configuration specified by ρ_{arq} calculus expressions[2]:

When the user presses the PLAY button, the visualizer shows the architectural execution flow that operational semantics in action produces and the user can see how a token passes between interfaces. In this case, the ρ_{arq} formal expressions are:

$$\begin{aligned} S =& Arch \wedge Transf \wedge Drawer \wedge Rewriter \wedge Interp \\ & \wedge (OSO\,(Interp)\,do\,(C_{IR} \wedge C_{IA} \wedge Rewriter^{\top})\,else\,\tau Interp) \wedge Interp^{\top} \\ & \wedge (OSO\,(Arch)\,do\,(C_{AG} \wedge C_{AT})\,else\,\tau Arch) \\ & \wedge OSO(Rewriter)do(C_{RA} \wedge Arch^{\top})else\tau Rewriter \end{aligned}$$

where each component was specified as:

$$\begin{aligned} Interp &= RLoad \wedge PArcInitial_{ANTLR} \wedge PArchitecture_{ANTLR} \\ RLoad &= \exists l_{Interp}[(r_{Interp} : x/x\overline{l_{Interp}}) \wedge (l_{Interp} : File/Interp^{(int)})] \\ PArcInitial_{ANTLR} &= (p1_{Interp} : y/y\overline{Architecture_{ANTLR}}) \\ PArchitecture_{ANTLR} &= (p2_{Interp} : y/y\overline{Architecture_{ANTLR}}) \end{aligned}$$

$$\begin{aligned} Rewriter &= RArchitecture \wedge PRewriter \\ RArchitecture &= \exists l_{Rewriter}[(r_{Rewriter} : x/x\overline{l_{Rewriter}}) \wedge (l_{Rewriter} : Architecture/Rewriter^{(int)})] \\ PRewriter &= (p_{Rewriter} : y/y\overline{ArchitectureWritten}) \end{aligned}$$

[2] These expressions should be written in TeX format.

$$
\begin{aligned}
Arch &= RArchitecture_{Arch} \wedge PArchitectTrans \wedge PArchitectGraf \\
RArchitecture_{Arch} &= \exists l_{Arch}[(r_{Arch} : x/x\overline{l_{Arch}}) \wedge (l_{Arch} : Architecture/Arch^{(int)})] \\
PArchitectTrans &= p1_{Arch} : y/y\overline{ObjectsArchitecture} \\
PArchitectGraf &= p2_{Arch} : y/y\overline{ObjectsArchitecture}
\end{aligned}
$$

$$
\begin{aligned}
Transf &= RArchitecture_{Transf} \wedge PTransf \\
RArchitecture_{Transf} &= \exists l_{Transf}[(r_{Transf} : x/x\overline{l_{Transf}}) \wedge (l_{Transf} : Architecture/Transf^{(int)})] \\
PTransf &= p_{Transf} : y/y\overline{ArchitectureXMDrawerI}
\end{aligned}
$$

$$
\begin{aligned}
Drawer &= RArchitecture_{Drawer} \wedge PDrawer \\
RArchitecture_{Drawer} &= \exists l_{Drawer}[(r_{Drawer} : x/x\overline{l_{Drawer}}) \wedge (l_{Drawer} : Architecture/Drawer^{(int)})] \\
PDrawer &= p_{Drawer} : y/y\overline{ArchitectureSVG}
\end{aligned}
$$

The assembly connectors are:

$C_{IR} = r_{Rewriter}\overline{p2_{Interp}}$

$C_{RA} = r_{Arch}\overline{p_{Rewriter}}$

$C_{IA} = r_{Arch}\overline{p1_{Interp}}$

$C_{AT} = r_{Transf}\overline{p1_{Arch}}$

$C_{AG} = r_{Drawer}\overline{p2_{Arch}}$

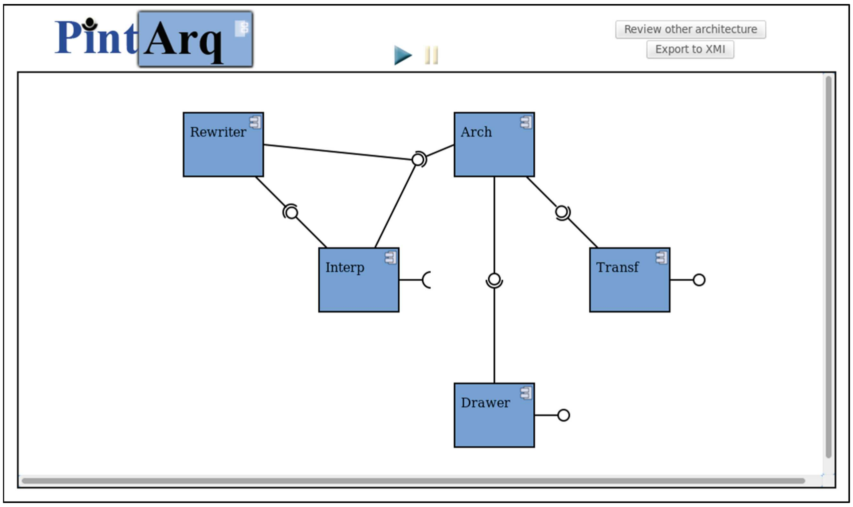


Fig. 2. The PintArq's graphic interface

5 Conclusions

In line with research works about languages that allow to simulate the execution of software architectures as Rapide [19], Archware ADL [24], Pi-ADL [29];

a visualizer of architectural execution flow for software component-based was described in this paper. In this work, the interpreter of ρ_{arq} expressions identifies the structural elements and it transforms the architectural expressions to UML component-configuration [27]. Then, the PintArq tool visualizes the execution flow according to ρ_{arq}'s operational semantics [5,6]. The interpretation engine was based in ANTLR [31] and the expressions were wrote in Tex format [12]. The prescriptive architecture was shown in the Fig. 1 and the concise description was done in the Sect. 4.

The research group ARQUISOFT is committed to the *Open Models* initiative. In consequence, the prescriptive architecture, functional models, structural models and dynamic models can be found in the ARQUISOFT's web portal: http://arquisoft.udistrital.edu.co/modelos/modelPintArqHTML/, for any interested reader.

6 Future Work

Since software architects may not know the complexities of formal calculus and prefer to concentrate their efforts on analysis tasks. A second phase of PintArq project will develop an interpreter from component-based visual models to ρ_{arq} expressions and this result will be integrated to actual version. Additionally, the correctness checking with ρ_{arq} calculus proposed in [7] must be incorporated in a new version of PintArq at a future.

Acknowledgements. Thanks to Professor Sergio Rojas. His advices were worthy.

References

1. Allen, R.J.: A formal approach to software architecture. Ph.D. thesis, Carnegie Mellon, School of computer Science (1997)
2. 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). doi:10.1007/978-3-540-39800-4_7
3. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: An infrastructure for the rapid development of XML-based architecture description languages. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), pp. 266–276. ACM, New York (2002). http://doi.acm.org/10.1145/581339.581374
4. Dean, T.R., Cordy, J.R.: A sintactic theory of software architecture. IEEE Trans. Softw. Eng. **21**(4), 302–313 (1995)
5. Diosa, H.A.: Especificación de un Modelo de Referencia Arquitectural de Software A Nivel de Configuración, Estructura y Comportamiento. Ph.D. thesis, Universidad del Valle- Escuela de Ingeniería de Sistemas y Computación, Febrero 2008
6. 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 (12) (2010)
7. Diosa, H.A., Díaz, J.F., Gaona, C.M.: Especificación formal de arquitecturas de software basadas en componentes: Chequeo de corrección con cálculo ρ_{arq}. Revista Científica. Universidad Distrital Francisco José de Caldas (12) (2010)

8. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. Addison-Wesley, Boston (2013)
9. Garlan, D., Monroe, R., Wile, D.: ACME: an architecture description interchange language. In: Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1997), p. 7. IBM Press (1997). http://portal.acm.org/citation.cfm?id=782010.782017
10. Garlan, D., Shaw, M.: An introduction to software architecture. Technical report CMU-CS-94-166. Carnegie Mellon University, Enero 1994
11. Gil, S.V.H.: Representación de la arquitectura de software usando UML. Sistemas y Telemática **1**, 63–75 (2006)
12. Goossens, M., Mittelbach, F., Samarin, A.: The LaTeX Companion. Addison-Wesley, Reading (1994)
13. Gorlick, M., Razouk, R.: Using weaves for software construction and analysis. In: 13th International Conference on Software Engineering, Proceedings, pp. 23–34, May 1991
14. Guerra, E., de Lara, J., Kolovos, D., Paige, R.: A visual specification language for model-to-model transformations. In: IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 119–126 (2010)
15. He, X., Yu, H., Shi, T., Ding, J., Deng, J.: Formally analyzing software architectural specifications using SAM. J. Syst. Softw. **71**, 11–29 (2004)
16. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**, 666–677 (1978). http://doi.acm.org/10.1145/359576.359585
17. Inverardi, P., Wolf, A.: Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE Trans. Softw. Eng. **21**(4), 373–386 (1995)
18. Bass, L., Paul Clements, R.K.: Software Architecture in Practice, Chap. 2. SEI Series in Software Engineering. Addison Wesley, Boston (2013)
19. Luckham, D.C.: Rapide: a language and toolset for simulation of distributed systems by partial orderings of events. Technical report, Stanford, CA, USA (1996)
20. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Schäfer, W., Botella, P. (eds.) ESEC 1995. LNCS, vol. 989, pp. 137–153. Springer, Heidelberg (1995). doi:10.1007/3-540-60406-5_12
21. Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D.A.: Providing architectural languages and tools interoperability through model transformation technologies. IEEE Trans. Softw. Eng. **36**(1), 119–140 (2010)
22. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the unified modeling language. ACM Trans. Softw. Eng. Methodol. **11**, 2–57 (2002). http://doi.acm.org/10.1145/504087.504088
23. Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, New York (1999)
24. Morrison, R., Kirby, G., Balasubramaniam, D., Mickan, K., Oquendo, F., Cimpan, S., Warboys, B., Snowdon, B., Greenwood, R.: Support for evolving software architectures in the ArchWare ADL. In: Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), pp. 69–78 (2004)
25. Naur, P.: Revised report on the algorithmic language ALGOL 60. Commun. ACM **6**(1), 1–17 (1963)
26. 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). doi:10.1007/3-540-60688-2_43

27. Object Management Group: OMG Unified Modeling Language (OMG UML). Version 2.5, September 2013
28. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer **33**(3), 78–85 (2000)
29. Oquendo, F.: Dynamic software architectures: formally modelling structure and behaviour with Pi-ADL. In: Software Engineering Advances (ICSEA 2008), pp. 352–359, October 2008
30. Pandey, R.K.: Architecture description languages (ADLs) vs UML: a review. SIGSOFT Softw. Eng. Notes **35**, 1–5 (2010). http://doi.acm.org/10.1145/1764810.1764828
31. Parr, T.: The Definitive ANTLR 4 Reference. The Pragmatic Bookshelf, Dallas (2012)
32. Pattis, R.E.: Extended Backus-Naur Form. Disponible en (1980). http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf
33. Robbins, J., Medvidovic, N., Redmiles, D., Rosenblum, D.: Integrating architecture description languages with a standard design method. In: Proceedings of the 1998 International Conference on Software Engineering, pp. 209–218, April 1998
34. Rong, M.: An aspect-oriented software architecture description language based on temporal logic. In: 2010 5th International Conference on Computer Science and Education (ICCSE), pp. 91–96, August 2010
35. Seidman, S.B.: Computer Science Handbook, Chap. 109. Chapman & Hall/CRC (2004)
36. Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., Zelesnik, G.: Abstractions for software architecture and tools to support them. IEEE Trans. Softw. Eng. **21**(4), 314–335 (1995)
37. Singh, Y., Sood, M.: Models and transformations in MDA. In: International Conference on Computational Intelligence, Communication Systems and Networks, pp. 253–258 (2009)
38. Smolka, G.: A calculus for higher-order concurrent constraint programming with deep guards. Technical report, Bundesminister für Forschung und Technologie (1994)
39. Smolka, G.: A foundation for higher-order concurrent constraint programming. Technical report, Bundesminister für Forschung und Technologie (1994)
40. Spencer, J.: Architecture description markup language (ADML) creating an open market for IT architecture tools. Disponible en, Septiembre 2000. http://www.opengroup.org/architecture/adml/background.htm
41. Zuo, W., Feng, J., Zhang, J.: Model transformation from xUML PIMs to AADL PSMs. In: International Conference on Computing, Control and Industrial Engineering (CCIE), pp. 54–57 (2010)