Universidade do Minho

André Batista Martins

Prototyping a calculus of QoS - Aware Software Components

Tese de Mestrado
Mestrado em Informática
Trabalho efectuado sob a orientação de
Prof. Dr. Luís Soares Barbosa e Prof. Dr. Nuno Miguel Rodrigues

Outubro 2011

André Batista Martins

Prototyping a calculus of QoS - Aware Software Components

Dissertação submetida à Universidade do Minho para obtenção do grau de Mestre em Informática, ramo Fundamentos da Computação.

Orientador: Luís Soares Barbosa
Co-orientador: Nuno Miguel Rodrigues

Departamento de Informática
Universidade do Minho
2011

To my parents, my sister and Cátia,
who always believed and supported me.

Acknowledgements

It is a pleasure to thank those who made this thesis possible such as my advisers who gave the moral support I needed and the insightful comments and constructive criticisms that were thought-provoking at deterrents stages of my research. They helped me focus on my ideas. I am also thankful to my parents, my sister and Cátia. I am sure it would not have been possible without their help.

To my advisor, Luís S. Barbosa, who is one of the best teachers I have ever had in my life. He sets standards for his students and he encourages and guides them to meet those standards. He introduced me to Processes and Software Architectures and his teaching inspired me to work in the article [4]. I am indebted to him for his continuous encouragement and guidance.

To my co-advisor, Nuno M. Rodrigues, who has been always there to listen and to give advice. I am grateful to him for the long discussions that helped me sort of practical details of my work.

I would like to thank to my advisers for the confidence in me, the enthusiasm, for the rigour and the interest in my research.

Finally, I would like to thank my family and friends, who often have asked when would the thesis be finished, for their support and encouragement.

Abstract

Over the last decade component-based software development arose as a promising paradigm to deal with the ever increasing complexity in software design, evolution and reuse. Such components typically encapsulate a number of services through a public interface which provides limited access to a private state space, paying tribute to the nowadays widespread object-oriented programming principles. This work is based on the calculus developed by L.S. Barbosa over several years and it aims at helping the development of formal software component solutions and to explain how they can be related, reducing their complexity. SHACC is a prototyping system for component-based systems in which components are modeled coinductively as generalized Mealy machines incorporating the ideas above. The prototype is built as a HASKELL library endowed with a graphical user interface developed in Swing.

Keywords: Software composition, Mealy machines, Prototyping.

List of Figures

List of Examples

List of Definitions

Contents

Chapter 1

Introduction

1.1 Context

A coalgebraic calculus for software components

The third party software component is a reusable piece of software developed to be distributed freely or to be sold by a retailer. The market for this kind of components is auspicious, because many of their developers believe that it improved their performance and the quality of the software. The software system that support the providers became more complex and larger, their design and verification became extremely difficult. One of the ways to cope with the growing complexity of software is to allow systems to be partially modelled, increasing the productivity, the quality and the effective way of adapting to requirements changes. Some modelling languages are particularly well-suited to model the behaviour of such system parts, since they have a mathematical background in terms of a transition system that describes exactly which steps are possible in the model in any given state.

The simulation of such system can only be done from a mathematical model and it works as follows. Given any state, we can determine which states are reachable from here by looking at the structure of the model. This way it is possible to explore the behaviour of a model by selecting an initial state, and then visiting one of the reachable states from the initial state. This selection and visiting of the next reachable states can continue indefinitely until a terminal state is reached, but the system can contain also a deadlock, i.e, the selection of the next reachable state can't continue because no other state can be reached. Simulations are important when developing software systems, since they can answer many what-if questions about the behaviour of the current system and so that helps when we need to find some unnoticed errors, asking "What if the model enters this state and where do we go from there?", "What is the condition to reach that state?", "Does that deadlock make sense?" and if perhaps we know in advance what is the desirable behaviour that system must support, we only asks "Can you support such behaviour?".

However, as time is a limited resource and manual verification is prone to errors and for that reason it is not always a real possibility for a given model. Verification and simulation are clearly nice attributes of formal behaviour languages, but another great, and perhaps more fundamental, advantage is the ability to give a precise description which is not opened to interpretation. Non-formal models have a big disadvantage they are not always precise, and the reason is partly the lack of formality, which leaves the structure of the models open to interpretation.

Formality is not always a blessing. In contrast with non-formal models, formal models tend to be difficult to understand by non-experts, and this is partly because they have not experience about the intuitive simple nature of non-formal models. Of course, one can argue that non-formal models are impossible to understand, since they are not

completely clear (i.e., formal) about each element in the model, but we do see in real life that they are somehow understood.

In this dissertation we will focus our research on *state-based components calculus* which gives us a system as a set of interconnected components, with an observable state, because studies related to component calculi are extensive. In general, state-based systems satisfy the follow criteria:

- Their behaviour depends on internal states, which are not visible to their environment
- System as reactive, interact with their environment, and are not necessarily terminating
- The interaction is performed by mutually calling services/operations declared in systems interfaces.

This favours adoption of a *behavioural* semantics: components are inherently dynamic, possess an observable behaviour, but their internal configurations remain hidden and should be identified if not distinguishable by observation. The qualificative 'state-based' is used in the sense the word 'state' has in automata theory — the internal memory of the automaton which both constrains and is constrained by the execution of component operations. Such operations are encoded in the specification of a functor which constitutes the component interface.

Since systems can be described as a set of linked state-based components, they can be seen as coalgebras. All our work follows research made by L. S. Barbosa over several years, and his PhD thesis published in 2001 entitled *Component as Coalgebras*[13]. As its title suggests, the subject emerged a model for representation of components as

co-algebras as generalized Mealy machines[5][13], with a public interface and a private state.

Our departure point is the an junction of two ideas: i) a *black-box* characterization of software components, favouring an observational semantics as a particular simple class of state-based systems having a display shows an element d of some fixed set D of data. The button t changes the inner state of the black box, in such a way that when h is pressed after t the black box shows a element d'$\in$ D on its display[8]; ii) the proposed constructions should be generic in the sense that they should not depend on a particular notion of component behaviour[13].

The language chosen to implement the component model was HASKELL as an advanced purely-functional programming language, it allows rapid development of robust, concise, correct software, with strong support for integration with other languages. Furthermore it support, the notion of monad, to encode computative non functional behaviour. Later adopted to create one graphical environment in Swing(Java), because swing gives a abstraction to APi for providing a graphical user interface(GUI) for Java programs,that we link through the work developer in a HASKELL library, becoming it easier to create environments.

The QoS challenge

Non-functional properties of software components, such as response time, availability, bandwidth requirement, memory usage, etc., cannot be ignored and become decisive in the component's selection procedure. Actually, often adaptation mechanisms have to take them into account, going far behind simple functionality wrapping to bridge between published interfaces. The expression Quality of Service (QoS) is widely

accepted to group together all these concerns [17, 25, 26]. It suggests twin notions of a level to be attained and cost to be paid, as well as point out to the design of suitable metrics to quantify such properties. Over the past few decades, several formalisms d(e.g., stochastic Petri Nets [14] and interactive Markov Chains [10]) have been proposed to capture different QoS metrics. In programming languages like Java or C#, QoS properties are often specified using meta-attributes. From a static validation perspective, these attributes can be treated like structured comments, which may be used to generate runtime monitors but their semantics is too weak to allow reasoning effectively about QoS properties. Dealing with QoS aspects in a coherent and systematic way became a main issue in component composition, which cannot be swept under the carpet in any formal account of the problem. The challenge is, then, to extend the component calculus, that is based on a coalgebraic model used to capture components with a observable behaviour and a persistence over transitions, to take into account, in an explicit way, QoS information. A possible way to express QoS properties is through (a slight generalization) of the notion of Q-algebra proposed in [7]. In brief, a Q-algebra amounts to two semirings over a common carrier, representing some form of cost domain, which allows different ways of combining and choosing between quality values. Such a perspective, which is expected to be studied in this MSc dissertation, is put forward in [18]. In any case the resulting calculus should provide a compositional approach which offers potential for complex components to be constructed systematically while satisfying QoS constraints. Although most previous laws have to be revisited in this extended model, and most of them will, most probably, turn from equalities (i.e., bisimilarity) to inequalities (i.e., refinements), proofs should still be carried on in the calculation style which is the watermark of [4, 5]. This style avoids the explicit construction of, e.g., bisimulations, when proving observational

equality, favouring an equational, essentially pointfree reasoning style as in, e.g., [6].

1.2 Objectives

1.2.1 Project aims

This MSc dissertation intends to address the following objectives:

- Develop an HASKELL prototype of a framework upon which software components can be specified, composed and animated along the lines of the calculus introduced in [4][5].
- Test this framework proof-of-concept implementation with a number of examples.
- Extend the component calculus to include QoS-aware composition mechanisms along the lines suggested in [18].

1.2.2 Institutional context

This research was carried on within the Formal Methods for High-assurance Software group, HASLab, In- formatics Department of the University of Minho. The specific context was the MONDRIAN project on Foundations for architectural design: Service certification, dynamic reconfiguration and self-adaptability, funded by F CT under contract PTDC/EIA-CCO/108302/2008. Information about this project is available at http://wiki.di.uminho.pt/twiki/bin/view/Research/MONDRIAN/WebHome.

1.3 Overview of the approach taken

As stated in 1.2, this dissertation intends to build a prototyper for the component calculus, implementing a semantical model for software components, parametric on a notion of behaviour. A component represent, a modular part of a system, that encapsulates its content and whose manifestation is replaceable within its environment. A component defines its behaviour in terms of provided and required interfaces. The calculus under this dissertation can captured well the behaviour of such components, because the formal semantics of components and their combinators is parametric on a strong monad. The monadic structure is one of the most important technologies used in the calculus, because it allows, an effective representation of e.g. non deterministic behaviour.

The research underlying this dissertation project required, first some background studies in the component calculus to get familiar with the area, and the formalisation of some architectural patterns to illustrate the calculus at work. Secondly, the implementation of a prototype for the component calculus in HASKELL. Thirdly, it was necessary to endow the prototype built as an HASKELL library with a graphical user interface developed in *Swing*. Fourthly, the extension of the prototype with a behavioural customization was considered. Fifthly, the extension of the component calculus to include QoS information along the lines of [18] and the extension of the prototype to mirror the QoS extended calculus, closed the work.

A main contribution of this work is the prototype tool, Shacc, developed as a proof-of-concept for component calculus, as published in [15]. The remainder of this dissertation is organized as follows. Chapter 2 gives a background overview of the coalgebraic component calculus which underlies our prototyper. Along chapters 3, 4

and 5, we developed a experimental prototype named SHACC based on some of the proposed calculus. Chapter 3 presents, an initial version of prototype that reflects directly the calculus defined in chapter 2. Chapter 4 documents an extension of the prototype that encompasses behavioured. At the end we extended the calculus endowed with QoS information, this is detailed in chapter 5. Finally, Section 6 provides some conclusions.

Chapter 2

Background

State-based software components are characterised as dynamic systems with a public interface and a private, encapsulated state. The behaviour can be partly characterised by a resorting to functor[1] $\mathsf{Id} \times O + \mathbf{1}$, *i.e.*, an instance of the popular *maybe* monad. Components are themselves *concrete* coalgebras[2]. For a given value of the state space — referred to as a *seed* in the sequel — a corresponding 'process', or *behaviour*, arises by computing its coinductive extension.

Other components may exhibit different *behaviour models*. For example, one can easily think of components behaving within a certain degree of non determinism or following a probability distribution. Genericity is achieved by replacing a given

[1] A functor is a mapping between algebraic structures that preserves structures, that can be thought of as homomorphisms between algebraic structures, or morphisms in the category of small algebraic structures[8]

[2] Let $F : C \to C$ be a functor. A coalgebra for F is a pair (A,a), where $a : A \to FA$ in C.

behaviour model by an arbitrary *strong monad*[3] B, leading to coalgebras for functor:

$$\mathsf{T}^{\mathsf{B}} = \mathsf{B}(\mathsf{Id} \times O)^I \tag{2.1}$$

as a possible general model for state based software components. Therefore computation of an action will not simply produce an output and a continuation state, but a B-structure of such pairs. The monadic structure provides tools to handle such computations. Unit (η) and multiplication (μ), act, respectively, as a value embedding and a 'flatten' operation to reduce nested behavioural effects. Strength, either in its right (τ_r) or left (τ_l) version, caters for context information. Finally, a strong monad is said to be *commutative* whenever δ_r and δ_l coincide.

Definition 1. A software component

Given a collection of sets I, O, ..., acting as component interfaces, a component taking input in I and producing output in O is specified by a pointed coalgebra:

$$\langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow \mathsf{B}(U_p \times O)^I \rangle \tag{2.2}$$

where u_p is the initial state, often referred to as the *seed* of the component computation, the coalgebra dynamics is captured by currying a state-transition function $a_p : U_p \times I \longrightarrow \mathsf{B}\ (U_p \times O)$.

[3] A *strong monad* is a monad $\langle \mathsf{B}, \eta, \mu \rangle$ where B is a strong functor and both η and μ are strong natural transformations. B being strong means there exist natural transformations $\tau_r^{\mathsf{T}} : \mathsf{T} \times - \Longrightarrow \mathsf{T}(\mathsf{Id} \times -)$ and $\tau_l^{\mathsf{T}} : - \times \mathsf{T} \Longrightarrow \mathsf{T}(- \times \mathsf{Id})$, called the right and left strength, respectively, subject to certain conditions. Their effect is to *distribute* the free variable values in the context "$-$" along functor B. Strength τ_r, followed by τ_l maps $\mathsf{B}I \times \mathsf{B}J$ to $\mathsf{BB}(I \times J)$, which can, then, be flattened to $\mathsf{B}(I \times J)$ via μ. In most cases, however, the *order* of application is relevant for the outcome. The Kleisli composition of the right with the left strength, gives rise to a natural transformation whose component on objects I and J is given by $\delta_r = \tau_{r_{I,J}} \bullet \tau_{l_{\mathsf{B}I,J}}$ Dually, $\delta_l = \tau_{l_{I,J}} \bullet \tau_{r_{I,\mathsf{B}J}}$. Such transformations specify how the monad distributes over product and, therefore, represent a sort of sequential composition of B-computations.

Several possibilities can be considered for B. The simplest case is, obviously, the *identity* monad, Id, whereby components behave in a totally *deterministic* way. Some of other possibility's can be considered to capturing more complex behavioural features, include:

- *Partiality*, *i.e.*, the possibility of deadlock or failure, captured by the maybe monad, $\mathsf{B} = \mathsf{Id} + \mathbf{1}$.
- *Non determinism*, introduced by the (finite) powerset monad, $\mathsf{B} = \mathcal{P}$.
- *Ordered non determinism*, based on the (finite) sequence monad, $\mathsf{B} = \mathsf{Id}^*$.
- Monoidal labelling, with $\mathsf{B} = \mathsf{Id} \times M$. Note that, for B to form a monad, parameter M should support a monoidal structure.
- *'Metric' non determinism* capturing situations in which, among the possible future evolutions of a component, some are stipulated to be more likely (cheaper, more secure, *etc*) than others.

2.1 A component calculus

We shall now look at the structure of Cp[4] by introducing an algebra of T^{B}-components parametric on a behaviour model B.

Let us start from the simple observation that functions can be regarded as particular instances of components, whose interfaces are given by their domain and codomain types.

[4] Cp is a bicategory whose objects are sets, standing for interface universes, arrows are seeded T^B-coalgebras and $2 - cells$ are the correspondent comorphisms[13].

Definition 2. The representation of a function

A function $f : A \longrightarrow B$ is represented in Cp by

$$\ulcorner f \urcorner \;=\; \langle * \in \mathbf{1}, \overline{a}_{\ulcorner f \urcorner} \rangle$$

i.e., a coalgebra over $\mathbf{1}$ whose action is given by the currying of

$$a_{\ulcorner f \urcorner} = \mathbf{1} \times A \xrightarrow{\mathsf{id} \times f} \mathbf{1} \times B \xrightarrow{\eta_{(\mathbf{1} \times B)}} \mathsf{B}(\mathbf{1} \times B)$$

Definition 3. Sequential composition

Components with compatible interfaces (for example, $p : I \longrightarrow K$ and $q : K \longrightarrow O$) can be composed sequentially[5] as

$$p \,;\, q \;=\; \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p;q} \rangle$$

where $a_{p;q} : U_p \times U_q \times I \longrightarrow \mathsf{B}(U_p \times U_q \times O)$ is detailed as follows [6]

[5] See

[6] The definition resorts to standard isomorphisms, such as associativity (a) and exchange ($\times r : A \times B \times C \to A \times C \times B$, $\times l : A \times (B \times C) \to B \times (A \times C)$), as well as to natural transformations $\tau_r : T \times - \Longrightarrow T(\mathsf{id} \times -)$ and $\tau_l : - \times T \Longrightarrow T(- \times \mathsf{id})$ denoting right and left monad strength.

$$
\begin{aligned}
a_{p;q} = {} & U_p \times U_q \times I \xrightarrow{\times r} U_p \times I \times U_q \xrightarrow{a_p \times \mathsf{id}} \\
& \mathsf{B}(U_p \times K) \times U_q \xrightarrow{\tau_r} \mathsf{B}(U_p \times K \times U_q) \xrightarrow{\mathsf{B}(a \cdot \times r)} \\
& \mathsf{B}(U_p \times (U_q \times K)) \xrightarrow{\mathsf{B}(\mathsf{id} \times a_q)} \mathsf{B}(U_p \times \mathsf{B}(U_q \times O)) \\
& \xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times (U_q \times O)) \xrightarrow{\mathsf{BB}a^\circ} \\
& \mathsf{BB}(U_p \times U_q \times O) \xrightarrow{\mu} \mathsf{B}(U_p \times U_q \times O)
\end{aligned}
$$

The pre- and post-composition of a component with Cp-lifted functions can be encapsulated into a unique combinator, called *wrapping*, which is reminiscent of the *renaming* connective found in process calculi (*e.g.*, [20]). Let $p : I \longrightarrow O$ be a component and consider functions $f : I' \longrightarrow I$ and $g : O \longrightarrow O'$. Component p wrapped by f and g, denoted by $p[f,g]$ and typed as $I' \longrightarrow O'$, is defined by input pre-composition with f and output post-composition with g. Formally,

Definition 4. Wrapping

The *wrapping* combinator is a functor

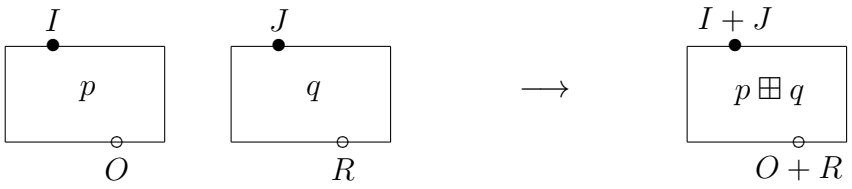
$$-[f,g] : \mathsf{Cp}(I,O) \longrightarrow \mathsf{Cp}(I',O')$$

which is the identity on morphisms and maps component $\langle u_p, \overline{a}_p \rangle$ into $\langle u_p, \overline{a}_{p[f,g]} \rangle$, where

$$
a_{p[f,g]} = U_p \times I' \xrightarrow{\mathsf{id} \times f} U_p \times I \xrightarrow{a_p} \mathsf{B}(U_p \times O) \xrightarrow{\mathsf{B}(\mathsf{id} \times g)} \mathsf{B}(U_p \times O')
$$

Components can be aggregated in a number of different ways, besides the 'pipeline' composition discussed above. Next, we introduce three other generic combinators, corresponding to *choice*, *parallel* and *concurrent* composition.

Let $p : I \longrightarrow O$ and $q : J \longrightarrow R$ be two components defined by $\langle u_p, \overline{a}_p \rangle$ and $\langle u_q, \overline{a}_q \rangle$, respectively. The first composition pattern to be considered is *external choice*, as depicted bellow:



When interacting with $p \boxplus q$, the environment is allowed to choose either to input a value of type I or one of type J, triggering the corresponding component (p or q, respectively) and producing output. Formally,

Definition 5. Choice

The *choice* combinator is defined as a lax functor $\boxplus : \mathsf{Cp} \times \mathsf{Cp} \longrightarrow \mathsf{Cp}$, which consists of an action on objects given by $I \boxplus J \ = \ I + J$ and a family of functors

$$\boxplus_{I,O,J,R} : \mathsf{Cp}(I,O) \times \mathsf{Cp}(J,R) \longrightarrow \mathsf{Cp}(I + J, O + R)$$

yielding

$$p \boxplus q \ = \ \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p \boxplus q} \rangle$$

$$
\begin{aligned}
a_{p \boxplus q} \;=\; U_p \times U_q \times (I + J) &\xrightarrow{(\mathsf{xr}+\mathsf{a})\cdot\mathsf{dr}} U_p \times I \times U_q + U_p \times (U_q \times J) \\
&\xrightarrow{a_p \times \mathsf{id} + \mathsf{id} \times a_q} \mathsf{B}\,(U_p \times O) \times U_q + U_p \times \mathsf{B}\,(U_q \times R) \\
&\xrightarrow{\tau_r + \tau_l} \mathsf{B}\,(U_p \times O \times U_q) + \mathsf{B}\,(U_p \times (U_q \times R)) \\
&\xrightarrow{\mathsf{Bx}\mathsf{r}+\mathsf{Ba}^\circ} \mathsf{B}\,(U_p \times U_q \times O) + \mathsf{B}\,(U_p \times U_q \times R) \\
&\xrightarrow{[\mathsf{B}\,(\mathsf{id}\times\iota_1),\mathsf{B}\,(\mathsf{id}\times\iota_2)]} \mathsf{B}\,(U_p \times U_q \times (O + R))
\end{aligned}
$$

and mapping pairs of arrows $\langle h_1, h_2 \rangle$ into $h_1 \times h_2$.

Definition 6. Parallel

Parallel composition, denoted by $p \boxtimes q$, corresponds to a synchronous product: both components are executed simultaneously when triggered by a pair of legal input values. Note, however, that the behaviour effect, captured by monad B, propagates. For example, if B can express component failure and one of the arguments fails, product fails as well. Formally,

The *parallel* combinator $\boxtimes$ is defined by an action $I \boxtimes J \;=\; I \times J$ on objects and a family of functors

$$\boxtimes_{IOJR} : \mathsf{Cp}(I,O) \times \mathsf{Cp}(J,R) \longrightarrow \mathsf{Cp}(I \times J, O \times R)$$

which yields

$$p \boxtimes q \;=\; \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p \boxtimes q} \rangle$$

where

$$
\begin{aligned}
a_{p \boxtimes q} = \quad & U_p \times U_q \times (I \times J) \xrightarrow{\mathsf{m}} U_p \times I \times (U_q \times J) \\
& \xrightarrow{a_p \times a_q} \mathsf{B}\,(U_p \times O) \times \mathsf{B}\,(U_q \times R) \\
& \xrightarrow{\delta_l} \mathsf{B}\,(U_p \times O \times (U_q \times R)) \\
& \xrightarrow{\mathsf{B}\,\mathsf{m}} \mathsf{B}\,(U_p \times U_q \times (O \times R))
\end{aligned}
$$

and maps every pair of arrows $\langle h_1, h_2 \rangle$ into $h_1 \times h_2$.

Finally, *concurrent* composition, denoted by $\circledast$, combines choice and parallel, in the sense that p and q can be executed independently or jointly, depending on the input supplied. Formally,

Definition 7. Concurrent

The *concurrent* combinator is defined by an action $I \circledast J \;=\; I + J + I \times J$ on objects and a family of functors

$$\circledast_{IOJR} : \mathsf{Cp}(I,O) \times \mathsf{Cp}(J,R) \longrightarrow \mathsf{Cp}(I + J + I \times J, O + R + O \times R)$$

yielding

$$p \circledast q \;=\; \langle \langle u_0, v_0 \rangle \in U_p \times U_q, \overline{a}_{p \circledast q} \rangle$$

where

$$
\begin{aligned}
a_{p \circledast q} = \quad & U_p \times U_q \times (I \circledast J) \\
& \Big\downarrow {\scriptstyle [\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)] \cdot (a_{p \boxplus q} + a_{p \boxtimes q}) \cdot \mathsf{dr}} \\
& \mathsf{B}\,(U_p \times U_q \times (O \circledast R))
\end{aligned}
$$

and maps pairs of arrows $\langle h_1, h_2 \rangle$ into $h_1 \times h_2$.

The laws of concurrent composition combine corresponding results about $\boxplus$ and $\boxtimes$.

In particular we get again permutation with sequential composition and the structure of a tensor product, which is symmetric for commutative behaviour monads.

So far component interaction was centred upon sequential composition, which is the Cp counterpart to functional composition in Set. This can be generalised to a new combinator, called *hook*, which forces *part* of the output of a component to be fed back as input. Formally,

Definition 8. Interaction

The *hook* combinator $- \leftarrow_Z$ is defined, for each tuple of objects $\langle I, O, Z\rangle$, as a functor between the (categories underlying) hom-sets $\mathsf{Cp}(I + Z, O + Z)$ and $\mathsf{Cp}(I + Z, O + Z)$ which is is the identity on arrows and maps each component $p : I + Z \longrightarrow O + Z$ to $p \leftarrow_Z : I + Z \longrightarrow O + Z$ given by

$$p \leftarrow_Z = \langle u_p \in U_p, \overline{a}_{p \leftarrow_Z} \rangle$$

where

$$\begin{aligned} a_{p \leftarrow_Z} = \quad & U_p \times (I + Z) \xrightarrow{\quad a_p \quad} \mathsf{B}(U_p \times (O + Z)) \\ & \xrightarrow{\mathsf{B}((\mathsf{id} \times \iota_1 + \mathsf{id} \times \iota_2) \cdot \mathsf{dr})} \mathsf{B}(U_p \times (O + Z) + U_p \times (I + Z)) \\ & \xrightarrow{\mathsf{B}(\eta + a_p)} \mathsf{B}(\mathsf{B}(U_p \times (O + Z)) + \mathsf{B}(U_p \times (O + Z))) \\ & \xrightarrow{\quad \mu \cdot \mathsf{B}\nabla \quad} \mathsf{B}(U_p \times (O + Z)) \end{aligned}$$

i.e., $a_{p \leftarrow_Z} = (\nabla \cdot (\eta + a_p) \cdot (\mathsf{id} \times \iota_1 + \mathsf{id} \times \iota_2) \cdot \mathsf{dr}) \bullet a_p$.

For components with the same input/output type, the *hook* combinator has a particularly simple definition as the Kleisli composition of the original dynamics. It is then

called a *feedback* and denoted by

$$p \circlearrowleft\colon Z \longrightarrow Z \;= \langle u_p \in U_p, \overline{a}_{p\circlearrowleft} \rangle$$

where

$$a_{p\circlearrowleft} \;= \qquad U_p \times Z \xrightarrow{a_p} \mathsf{B}(U_p \times Z) \xrightarrow{\mathsf{B}a_p} \mathsf{BB}(U_p \times Z) \xrightarrow{\mu} \mathsf{B}(U_p \times Z)$$

i.e., $a_{p\circlearrowleft} = a_p \bullet a_p$.

Chapter 3

A prototype for the component calculus

We resort to the programming language HASKELL to prototype the calculus referred in chapter 2. HASKELL is a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static type system based on *Hindley-Milner* type inference. As a functional programming language, the primary control construct is that of a function. The language is guided by the following criteria[1]:

"A proof is a program; The formula it proves is a type for the program"

Typically, a function in HASKELL does not have side effects, but there is a distinct type for representing side effects, orthogonal to the type of functions. The type which represents side effects is an example of a monad. Monads are a general framework which can handle different sorts of computation, the most relevant being error handling and non-determinism. The calculus detailed in chapter 2 is parametric on a monad B and therefore HASKELL turn out to be one of the most suitable tool for our purposes.

[1] Also known as Curry-Howard isomorphism

With the progress of the work, it became clear the need to abstract from code details in HASKELL and offer a graphical user interface that makes possible to create a whole system by claimed component composition in a more simple and appealing away. The range of tools for such a purpose is extensive, but the library Swing that belongs to the Java has been the one that became more attractive, because it is a flexible, stable framework that proved to be an asset. Swing is the primary Java GUI widget toolkit. It was developed to provide a more sophisticated set of GUI components than the previous version. Swing provides a native look and feel that emulates the look and feel of several platforms, and also allows applications to be unrelated to the underlying platform. On the other hand, Swing is also a component-based framework, concisely, a component is a well-behaved object with a known/specified characteristic pattern of behaviour. A new version of the prototyper arose and it became closer to the definition of integrated development environment (*IDE*). The tool is a software application that provides comprehensive facilities to computer programmers for software development. The main goal of *IDE's* is to use the technique of *RAD* (Rapid Application Development), which aims at increased productivity of developers. An *IDE* normally consists of a source code editor, a compiler or an interpreter and built in automation tools.

SHACC is a HASKELL-based prototyper for a calculus of state-based components framed as generalised Mealy machines. It was developed as a *proof-of-concept* prototype for the component calculus proposed in [2, 4]. It allows the (interactive) definition of state-based components through the set of combinators available in the calculus.

3.1 Monadic Technology

Chapter 2 introduced a small set of component combinators and studied their properties. Their implementation in SHACC is parametric on the component behaviour discipline encoded in a monad B.

As mentioned in chapter 2, the components with compatible interfaces (for example, $p : I \longrightarrow K$ and $q : K \longrightarrow O$) can be composed sequentially[2] as

$$p \,;\, q \;=\; \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p;q} \rangle$$

where $a_{p;q} : U_p \times U_q \times I \longrightarrow \mathsf{B}(U_p \times U_q \times O)$ is detailed as follows:

$$\begin{aligned} a_{p;q} = {} & U_p \times U_q \times I \xrightarrow{\times r} U_p \times I \times U_q \xrightarrow{a_p \times \mathsf{id}} \\ & \mathsf{B}(U_p \times K) \times U_q \xrightarrow{\tau_r} \mathsf{B}(U_p \times K \times U_q) \xrightarrow{\mathsf{B}(a \cdot \times r)} \\ & \mathsf{B}(U_p \times (U_q \times K)) \xrightarrow{\mathsf{B}(\mathsf{id} \times a_q)} \mathsf{B}(U_p \times \mathsf{B}(U_q \times O)) \\ & \xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times (U_q \times O)) \xrightarrow{\mathsf{BB}a^\circ} \\ & \mathsf{BB}(U_p \times U_q \times O) \xrightarrow{\mu} \mathsf{B}(U_p \times U_q \times O) \end{aligned}$$

HASKELL monadic technology provides all the ingredients for a direct implementation of this definition, suitably parametric on a strong monad `b`. Each component is represented by a monadic function from pairs of state-input values to `b`-computations of state-output pairs. The HASKELL definition of each combinator in the calculus follows closely the corresponding mathematical construction, as illustrated in figure 3.1 for

[2] For more details see definition 3

sequential composition. Computation proceeds through Kleisli composition. Note, finally, that in order to guarantee state persistence (and propagation of state values) the implementation of SHACC resorts to HASKELL state monad which is suitably combined with monad `b` capturing the underlying behavioral model.

```
seqCompostion :: Strong b =>
((u,i)-> b (u,k)) -> ((v,k)-> b (v,o))
 -> ((u,v), i) -> b ((u,v),o)

seqCompostion p q = mult . (fmap (fmap assocl)). (fmap lstr).
                    (fmap (id >< q)) . (fmap xl).
                    rstr . (p >< id) . xr
```

Figure 3.1: Implementation of sequential composition in HASKELL

3.1.1 Interfaces

A typical example of such a state-based component is the ubiquitous *stack*. Denoting by U its internal state, a stack of values of type P is handled through the usual

$$\mathsf{top}: U \longrightarrow P, \quad \mathsf{pop}: U \longrightarrow P \times U \quad \text{and} \quad \mathsf{push}: U \times P \longrightarrow U$$

operations. An alternative, 'black box' view hides U from the stack environment and regards each operation as a pair of input/output ports. For example, the top operation becomes declared as $\mathsf{top}: \mathbf{1} \longrightarrow P$, where $\mathbf{1}$ stands for the nullary (or unit) datatype. The intuition is that top is activated with the simple pushing of a 'button' (its argument being the stack private state space) whose effect is the production of a P value in the corresponding output port. Similarly typing push as $\mathsf{push}: P \longrightarrow \mathbf{1}$ means that an external argument is required on activation but no visible output is produced, but for a trivial indication of successful termination. Such 'port' signatures are grouped together

in the diagram below. Combined input type $\mathbf{1} + \mathbf{1} + P$ models the choice of three functionalities (top, pop and push in this order), of which only one takes input of type P.

$$\begin{cases} \mathsf{pop}: & \mathbf{1} \longrightarrow P \\ \mathsf{top}: & \mathbf{1} \longrightarrow P \\ \mathsf{push}: & P \longrightarrow \mathbf{1} \end{cases} \qquad \begin{array}{c} \mathbf{1}+\mathbf{1}+P \\ \hline \mathsf{Stack} \\ \hline P+P+\mathbf{1} \end{array} \tag{3.1}$$

The interface of stack are defined as $1+1+P$ for the input and $P+P+1$ for the output, that is represented in the code using the data type $Either$ and with the increasing of the complexity of this example, placing the two stacks together side by side and then redirect some of the outputs to the input, we come across to a problem - the order of the operations is important and it have some complexity, that we want to reduce. Several approaches have been tried , but we faced always with the same error:

```
Occurs check: cannot construct the infinite type: t = Either t t1
```

What makes sense because the type of generic $B = A+B$ is $B = A+A+A+...$ which it is intuitively an infinite type. In order to overcome this difficulty and because each interface are defined using either or/and split, it became necessary create an abstract tree where the order of each operation of the interface can be set, with the possibility of later, if necessary, rebuilding.

For example, if we have two components, where the interfaces - Input are defined as Figure 3.2 and Figure 3.3 and we have the intent to create a new component with the both components. We can do this, using this new approach where we create a new branch in the interface of the new component such that in the left side represents the

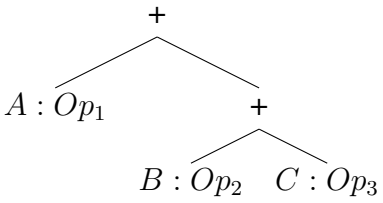


Figure 3.2: I_1 - Interface of $A : Op_1 + (B : Op_2 + C : Op_3)$

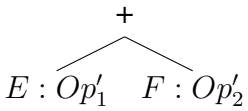


Figure 3.3: I_2 - Interface of $E : Op'_1 + F : Op'_2$

component with the interface I_1 and the right side have the interface of the components with the interface I_2. At this moment we can say that the first operation of the interface of the left side is linked to the first operation in the interface of right side, as illustrated in Figure 3.4.

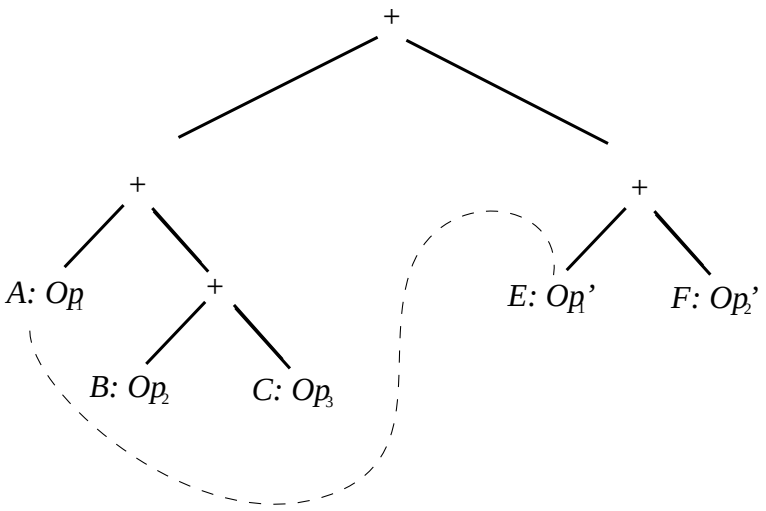


Figure 3.4: New component formed based in interface I_1 and I_2

3.1.1.1 Data Structure

The data structure created in Figure 3.5 aims to accommodate the possibility of defining the interface for each operation to which is assigned an identifier. It also provides a way to define the state of each component.

```
data Exp a  o = Val a | Branch o (Exp a o) (Exp a o)
                  deriving (Show)

data Op = Sum | Prod
            deriving (Show)

data Leave a b  = Empty
                | None Id
                | K a
                | D (a, Id)
                | State b
                  deriving (Show)
type Id = String
```

Figure 3.5: Interface data structure

Each interface may contain zero or more operations. An interface with no operation, will be represented by $Empty$. If the interface contains two operations then it could be formed using the construct $Branch$, and then we can use Sum or $Prod$ depending on idealized interface. In this approach the construct Sum represent disjoint union and the construct $Prod$ represent the split type. In each leaf it is recorded if that does not contain any operation ($Empty$) or the operation can be defined and assigned a name (using the construct $D(a, Id)$ where a is the value used in the operation and the Id is the operation name). For example, with this data structure, we can represent in the Figure 3.6 the interfaces created in Figure 3.2, where Op_1, Op_2, Op_3 are the operations names.

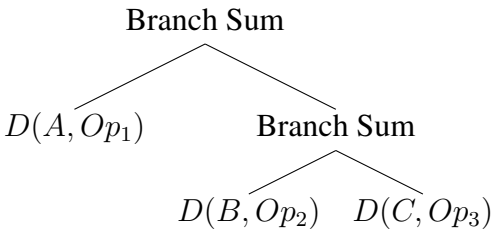


Figure 3.6: I_1' - Interface I_1 defined in our structure

3.1.1.2 Data Dynamic

While constructing the examples we found a constraint: two different components that work with different types that are attached in the various forms available, may become incompatible. The fusion of two distinct interfaces that type has to be unified in the generic type compatible with both types contained in their interfaces. Why? Because HASKELL types are limited in that the type of the state cannot change during the computation. Normally this is fine, but what if we really wanted to use the mechanics of a state monad to pass some state value that changed type, e.g, some mutually-recursive tree structure we would like to traverse?

In this sense, the need to resort to Data.Dynamic library[3]. This framework provides operations for injecting values of arbitrary type into a dynamically type value(Dynamic) and operations for converting dynamic values into a concrete(monomorphic) type.

In this library we use only two functions:

```
toDyn :: Typeable a => a -> Dynamic
```

which it converts an arbitrary value into an object of type $Dynamic$, and

```
fromDynamic :: Typeable a => Dynamic -> a -> a
```

[3] Available in http://haskell.org/ghc/docs/6.12.2/html/libraries/base-4.2.0.1/Data-Dynamic.html

which converts a $Dynamic$ object back into an ordinary HASKELL value of the correct type.

For example, the following code shows the way it is used.

```
Prelude Data.Dynamic> :t toDyn
    toDyn :: (Typeable a) => a -> Dynamic
Prelude Data.Dynamic> :t fromDynamic
    fromDynamic :: (Typeable a) => Dynamic -> Maybe a
Prelude Data.Dynamic> fromDynamic (toDyn 'c') :: Maybe Char
    Just 'c'
```

3.1.2 Defining Components

A component is defined by a set of functions that receive input and, according to a behavioural produced, return an output, which may affect or no the component's state. In the interface of the new component, it needs to be inserted @$Input$: and @$Output$:, to identify the input/output of the interface that have all operations, their names and their types. This code does not affect the behaviour of the function because it is contained in comments which the compiler will ignore, as we can see in Figure 3.7.

```
{-
@Input:  (A:Op1.:Int + B:Op2.:Int)
@Output: C:Result.:Int
-}
```

Figure 3.7: I_3: $A + B/C$

The commented code is interpreted as shown in the Figure 3.8

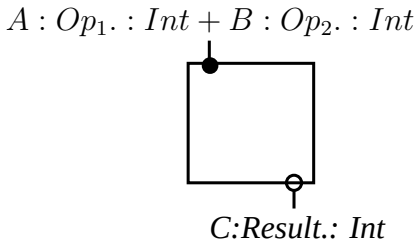


Figure 3.8: Component I_3

3.1.3 Examples

Example 1. A folder from two stacks

Component Stack encapsulates a number of services through a public *interface* providing limited access to its internal *state space*. Furthermore, it *persists* and *evolves* in time, in a way which can only be traced through observations at the interface level. One might capture these intuitions by providing an explicit semantic definition in terms of a function $[\![\mathsf{Stack}]\!] : U \times I \longrightarrow (U \times O + \mathbf{1})$, where I,O abbreviate $\mathbf{1} + \mathbf{1} + P$ and $P + P + \mathbf{1}$, respectively. The presence of $\mathbf{1}$ in its result type indicates that the overall behaviour of this component is *partial*: in a number of state configurations the execution of some operations may fail. This function describes how Stack reacts to input stimuli, produces output data (if any) and changes state. It can also be written in a curried form as

$$\overline{[\![\mathsf{Stack}]\!]} : U \longrightarrow (U \times O + \mathbf{1})^I$$

that is, as a *coalgebra* $U \longrightarrow \mathsf{T}\ U$ for functor $\mathsf{T}\ X\ =\ ((X \times O) + \mathbf{1})^I$.

The Stack example illustrates the basic elements of a semantic model for state-based

components: *a)* the presence of an *internal state space* which evolves and persists in time, and *b)* the possibility of *interaction* with other components through well-defined interfaces and during the overall computation. Components are inherently dynamic, possess an observable behaviour, but their internal configurations remain hidden and should be identified if not distinguishable by observation. The qualificative 'state-based' is used in the sense the word 'state' has in automata theory — the internal memory of the automaton which both constrains and is constrained by the execution of component operations. Such operations are encoded in a functor which constitutes the (syntax of the) component interface. On top of such a framework, reference [4] developed a calculus of component composition.

The definition of a new, base component is directly made in HASKELL . A specific strong monad B is chosen to model the envisaged behavioral effect. Figure 3.9 corresponds to a Stack component, where B is instantiated to HASKELL `Maybe` monad to capture partiality.

```
stack  (xs, ("Push", Just a)) =Just ( a:xs, ("Push", a))
stack  (xs, ("Pop", Nothing)) | xs== [] = Nothing
       | otherwise = Just ( tail xs, ("Pop", head xs))
stack  (xs, ("Top", Nothing)) | xs== [] = Nothing
       | otherwise = Just (  xs, ("Top", head xs))
```

Figure 3.9: Stack Component

In a subsequent step the component's interface is created from a suitable annotation in the source code. For this example:

```
@Input: (( 1:Pop + 1:Top) + P:Push)

@Output: (( P:Pop + P:Top) + 1:Push)
```

where `Pop`, `Top` and `Push` are introduced as labels for the component's available services.

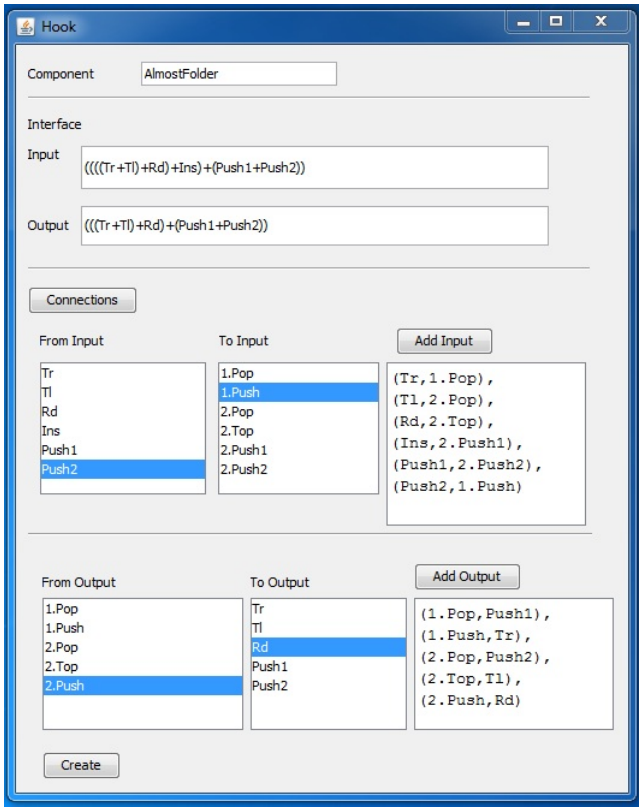


Figure 3.10: Linking ports through the *hook* combinator

Figure 3.10 refers to an example in the SHACC library in which a *folder* component is built through the combination of two stacks modelling, respectively, the folder left and right piles. The Folder component provides ports corresponding to the operations *read*, *insert* a new page, *turn a page right* and *turn a page left*. Its construction involves first that an adaptation is performed on each instance of the Stack component. This is needed, for example, to hide the *top* operation on the left stack whereas renaming the *top* on the right as the Folder *read* operation. In a second stage, both stacks are put together through the $\boxplus$ combinator and, finally, suitable feedback loops are established,

through the *hook* operator, to connect ports. This ensures, for example, that the left turn of a page is achieved through a pop action on the right stack connected to the push of the left one. Formally, this amounts to the following expression in the component calculus (see [3] for a detailed discussion)

$$\mathsf{Folder} \;=\; ((\mathsf{LeftS} \boxplus \mathsf{RightS})[wi,wp]) \;\curvearrowleft_{P+P}$$

where $\mathsf{RightS} = \mathsf{Stack}[id + \triangledown, id]$ and $\mathsf{LeftS} = \mathsf{Stack}[i_2 + Id, (id + !_{p+1}) \cdot a_+]$.

A crucial ingredient in defining Folder is to suitably wrapp the two underlying Stack components so that the intended output-input ports are effectively connected. Formally this is achieved through the *wrapping* combinator, as in the specification of LeftS and RightS. The effect is depicted in Figure 3.11. In SHACC, however, the user has the option of manually selecting the ports to be linked, as illustrated in Figure 3.10.

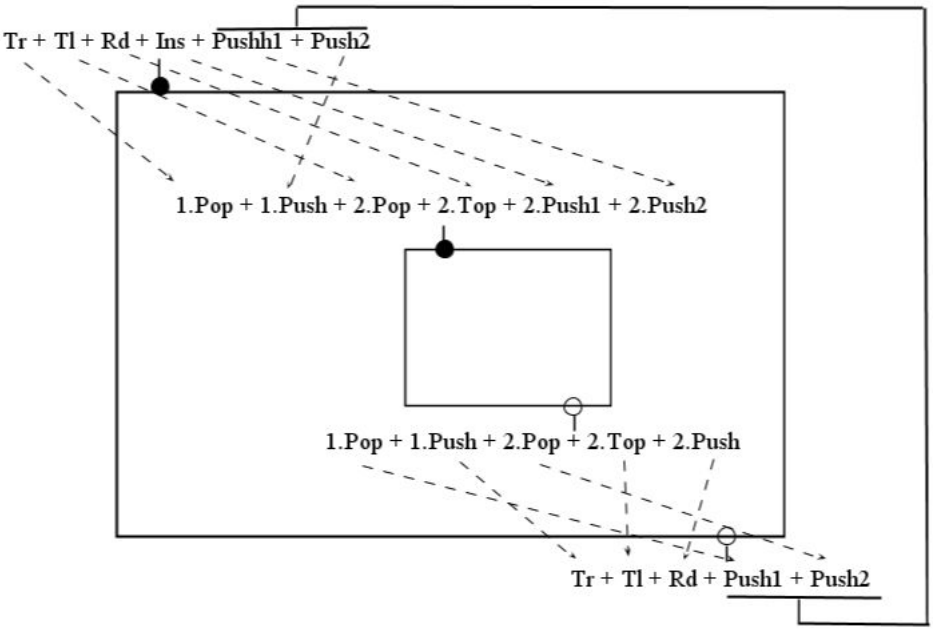


Figure 3.11: Assign ports

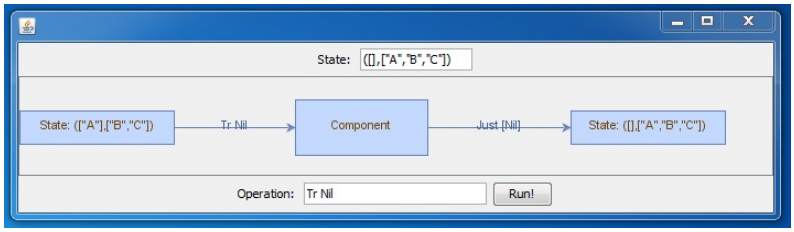


Figure 3.12: Component prototyping in SHACC

SHACC allows both the (interactive) definition of this sort of component expressions and their execution in a simulation mode. Actually, once components are defined either from scratch (*i.e.*, by providing the corresponding HASKELL code directly) or by composition of other components, SHACC offers an environment for testing by simulation. The *Run* window in the tool offers two simulation modes: a *free* mode in which, if the component's behaviour model allows, execution may lead to 'disaster' (*e.g.*, by violation of port pre-conditions on a *partial* component), and a *safe* mode in which the effect of a port operation is foreseen and eventually precluded. Component testing, on the other hand, can be made in a purely interactive way, running event by event, or by executing a whole sequence of events specified through a regular expression and supplied to the tool. Figure 3.12 illustrates the tool execution mode.

The box labelled *State* in Figure 3.12 shows the initial value of the component's state. Box *Operation*, on the other hand, accepts the component service to be called. On executing a service from the component's interface SHACC displays three boxes representing the component state before, during and after service completion.

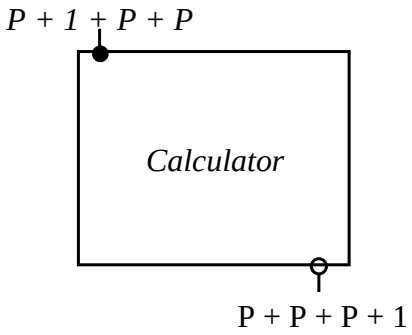
Example 2. Calculator

Our second example is a calculator which computes averages. The first step is to define a component that represents the calculator with all the intended operations. The operations defined are:

- Add: needs one argument and sum it with the value in memory
- Div: needs one argument and do division with the value in memory
- Mem: returns the value in memory of calculator
- Insert: inserts an element in the memory of the calculator

where,

$$\mathsf{Add}: P \longrightarrow P, \quad \mathsf{Mem}: 1 \longrightarrow P \quad \mathsf{Div}: P \longrightarrow P \quad \text{and} \quad \mathsf{Replace}: P \longrightarrow 1$$



Calculator protect the internal state space, and it can only be access through a public interface. This can be defined in terms of a function as

$$[\![\mathsf{Calculator}]\!] : U \times I \longrightarrow (U \times O + \mathbf{1})$$

where I,O abbreviate $P + \mathbf{1} + P + P$ and $P + P + P + \mathbf{1}$, respectively.

The interaction with Calculator can be defined in a curried form as

$$\overline{[\![\mathsf{Calculator}]\!]} : U \longrightarrow (U \times O + \mathbf{1})^I$$

that is, as a *coalgebra* $U \longrightarrow \mathsf{T}\ U$ for functor $\mathsf{T}\ X\ =\ ((X \times O) + \mathbf{1})^I$.

The code below corresponds to a $Calculator$ component, where we a strong monad B is instantiated to HASKELL Maybe monad to capture partiality.

```
calculator (m,("Add",x)) = Just (m+x,("Add", m+x))
calculator (m,("Div",x)) | m == 0.0 = Nothing
                         | otherwise = Just( (x/m), ("Div",  (x/m) )
calculator (m,("Mem",x)) = Just( m, ("Mem",   m ))
calculator (m,("Insert",x)) = Just (x,("Insert",x))
```

In a subsequent step the component's interface is created from a suitable annotation in the source code. For this example:

```
@Input: ((( P:Add.:Int + 1:Mem.:Int) + P:Div.:Int) + P:Insert.:Int )
@Output:((( P:Add.:Int + P:Mem.:Int) + P:Div.:Int) + 1:Insert.:Int )
```

where `Add`, `Mem`, `Div` and `Insert` are introduced as labels for the component's available services and Int define the type that port will support.

The calculator is formed by the junction of two components - the first that does the operation Sum and the operation Mem and the second that does the operation $Divide$ and the operation $Insert$. This is needed, for example, if we want to pass a value from the component $MSum$ to the component $MDivide$ in order to do the averages, we need two operations one to sums the values into memory and other to pass the value in

value to another component that will do the averages. So we define:

$$\mathsf{C}_1 \ : \ P + \mathbf{1} \longrightarrow P + P$$

$$\mathsf{C}_2 \ : \ P + P \longrightarrow P + \mathbf{1}$$

Then, we form the $\boxplus$ composition of both components:

$$\mathsf{C}_1 \boxplus \mathsf{C}_2 \ : \ P + \mathbf{1} + (P + P) \longrightarrow P + P + (P + \mathbf{1})$$

The nex step builds the desirable connections using hook over ths composite, which requires a previous wrapping by a pair of suitable isomorphisms:

$$\mathsf{AlmostCalculator} \ = \ ((\mathsf{C}_1 \boxplus \mathsf{C}_2)[\mathsf{wi}, \mathsf{wo}]) \curvearrowleft_{P+P}$$

where, wi and wo redirect the output to input, this connection and the final interface of the system can be seen in the Figure 3.13.

Example 3. Bank

In this section we will introduce a small example that represents a $Bank$. A typical $Bank$ provides the possibility to create a new account, withdraw some quantity of money from the account and do deposits of any quantity of money. In this example we define the three operations, that will receive a pair of values: the first element is the id

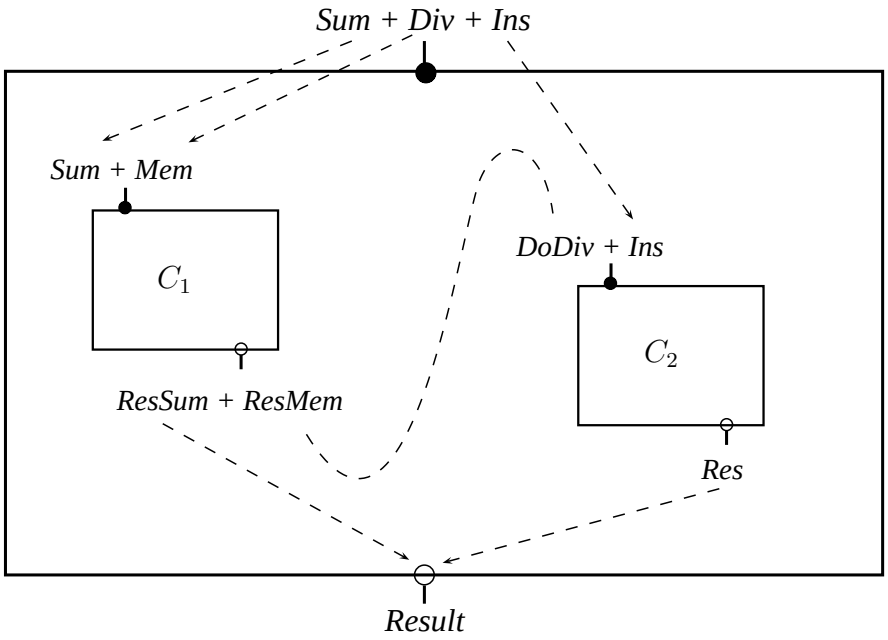
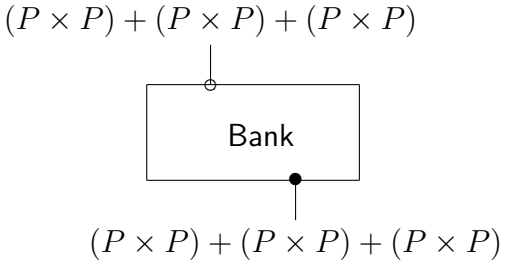


Figure 3.13: Calculator - Assign ports

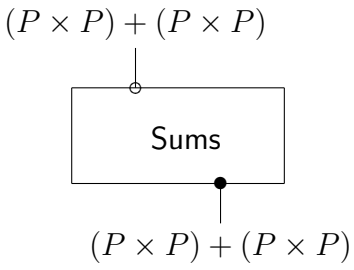
from the account and the second element is a value, i.e.:

$$\begin{cases} \mathsf{NewAcc}: & P \times P \longrightarrow P \times P \\ \mathsf{Withdraw}: & P \times P \longrightarrow P \times P \\ \mathsf{Deposit}: & P \times P \longrightarrow P \times P \end{cases}$$

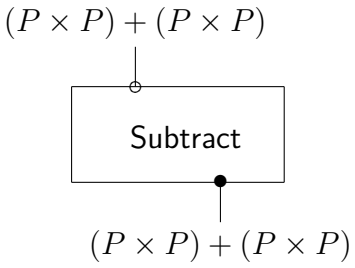


In order to make this a more interesting example, we added two component to the bank that will do every arithmetical operations. The first component will do $Sums$ and the second will do $Subtractions$. Both of them will receives the same types, i.e, an account identifier and the amount of money - $(\mathsf{id}, quantity)$.

$$\begin{cases} \mathsf{SumDep}: & P \times P \longrightarrow P \times P \\ \mathsf{DoSums}: & P \times P \longrightarrow P \times P \end{cases}$$



$$\begin{cases} \mathsf{SubWith}: & P \times P \longrightarrow P \times P \\ \mathsf{DoSubtrac}: & P \times P \longrightarrow P \times P \end{cases}$$



Component Bank can be specifies with using a function $[\![\mathsf{Bank}]\!] : U \times I \longrightarrow (U \times O + \mathbf{1})$, where I abbreviate $(P \times P) + (P \times P) + (P \times P)$ and O abbreviate $(P \times P) + (P \times P) + (P \times P)$.

Figure 3.14 corresponds to a Bank component, where B is instantiated to HASKELL Maybe monad to capture partiality.

```
bank  (xs,("NewAcc", x)) = Just (x:xs, ("NewAcc",x))
bank (xs,("Withdraw", x)) = case getAcc xs x of
                Nothing -> Nothing
                Just val -> case (snd val) >= (snd x) of
                     True -> Just ( remove xs x, ("Withdraw", val))
                     False -> Nothing
bank (xs,("Deposit", x)) =case getAcc xs x of
                Nothing -> Nothing
                Just val -> Just ( remove xs x, ("Deposit", val))
```

Figure 3.14: Bank component

In a subsequent step the component's interface is created from a suitable annotation in the source code. For this example:

```
@Input: ((P:NewAcc.:((Int,Int)) +P:Withdraw.:((Int,Int))) + P:Deposit.: ((I
@Output: ((P:NewAcc.:((Int,Int)) +P:Withdraw.:((Int,Int))) + P:Deposit.: ((
```

where `NewAcc`, `Withdraw` and `Deposit` are introduced as labels for the component's available services and the type that port support, is (Int, Int).

The definition of component, which does $Sums$ is directly made in HASKELL, as :

```
sums (xs,("SumsDep", x)) =Just ( snd x,("SumsDep", x))
sums (xs,("DoSums",x)) =Just ( xs, ("DoSums", (fst x, (snd x)+xs)))
```

where the interface is created from annotation in the source code, as:

```
@Input: (P:SumsDep.:((Int,Int)) + P:DoSums.:((Int,Int)))
@Output: (P:SumsDep.:((Int,Int)) + P:DoSums.:((Int,Int)))
```

The following code describes the component that will support the operation $Withdraw$ from the $Bank$.

```
subtrac (xs,("SubWith", x)) =Just ( snd x,("SubWith", x))
subtrac (xs,("DoSubtrac",x)) =Just ( xs, ("DoSubtrac", (fst x, (snd x)-xs))
```

whose interface is specified as:

```
@Input: (P:SubWith.:((Int,Int)) + P:DoSubtrac.:((Int,Int)))
@Output: (P:SubWith.:((Int,Int)) + P:DoSubtrac.:((Int,Int)))
```

The $Bank$, the Sum and the $Subtract$ components, together form a system that typically supported a $Bank_F$. The $Bank_F$ component provides ports corresponding to

the operations new account, withdraw and deposit some money. This component can not do operations as $sums$ or $subtract$ accounts, so we need connect with the components $Sums$ and $Subtract$, in order to support the operation $Withdraw$ and $Deposit$. In the second stage, we put component Sum and component $Subtract$ together through the $\boxplus$ combinator that we named as $Calculator$. In the third stage we composed with the component $Bank$ using the same combinator - $\boxplus$ and, finally, suitable feedback loops are established, through the $hook$ operator, $\curvearrowleft$, to connect ports. Formally, this can be expressed as follows.

$$\mathsf{Bank_F} \;=\; ((\mathsf{Bank} \boxplus (\mathsf{Sums} \boxplus \mathsf{Subtrac}))[wi,wo]) \curvearrowleft_{P+P+P+P+P}$$

where $\curvearrowleft_{P+P+P+P+P}$ represent $DoGetAccSub$, $DoSub$, $DoGetAccSums$, $DoSums$ and $DoNewAcc$ respectively, that will do some loops and feed again the input of the component $Bank_F$.

A crucial ingredient in defining $\mathsf{Bank_F}$ is to suitably wrap the two underlying Bank and $Calculator$ components so that the intended output-input ports are effectively connected. The effect is depicted in Figure 3.15.

In order to provide the final interface to the user we hide the operations needed for the $hook$ combinator, because this operations only concern to the computation of such a combinator. For this purpose, the final interface can be seen in the Figure 3.16.

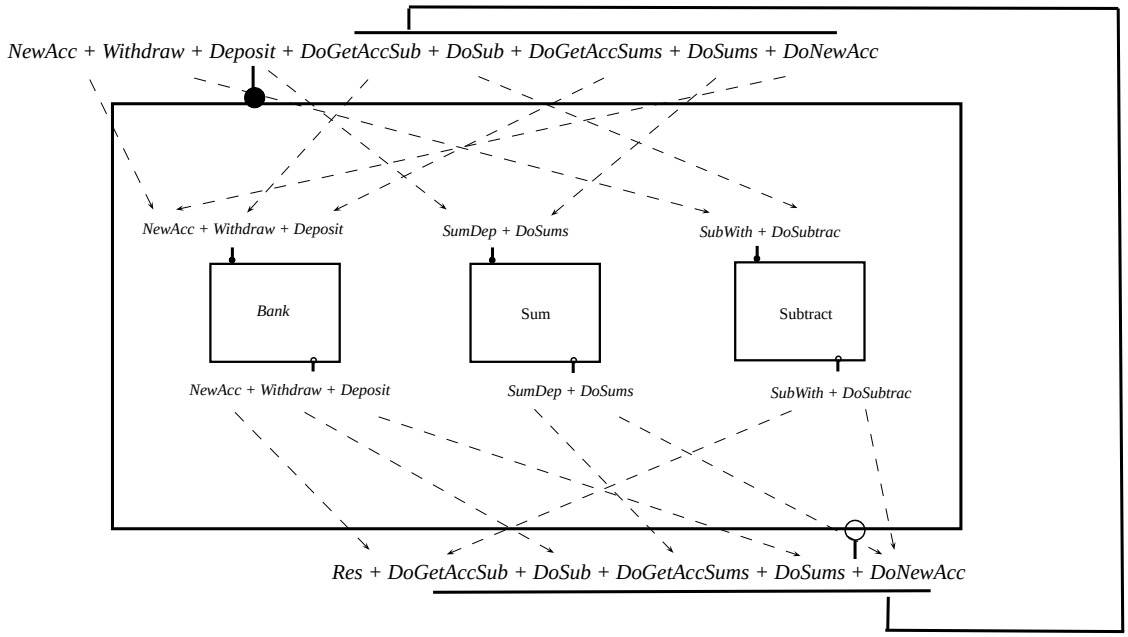


Figure 3.15: Bank - Assign ports

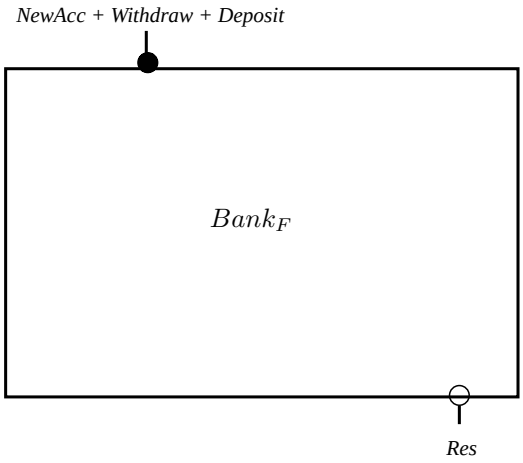


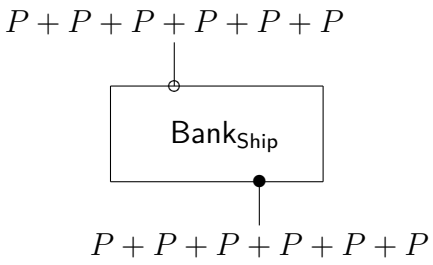
Figure 3.16: Bank - Final interface

Example 4. Shipments between banks

In this example, we aim to join two banks in order to provide the possibility of transference between them. Thus the bank allows to make deposits, withdraws and transfers.

While the transactions deposits and withdraws belong to the internal operations of each bank, the operation transfer connect, the two banks, and allows for the exchange of funds. Formally, we define the $Bank_{Ship}$, as:

$$\begin{cases} \mathsf{Ins}_1 : & P \longrightarrow P \\ \mathsf{With}_1 : & P \longrightarrow P \\ \mathsf{Trans}_1 : & P \longrightarrow P \\ \mathsf{Ins}_2 : & P \longrightarrow P \\ \mathsf{With}_2 : & P \longrightarrow P \\ \mathsf{Trans}_2 : & P \longrightarrow P \end{cases}$$



Component $\mathsf{Bank}_{\mathsf{Ship}}$ gives services that provides limited access to its internal *state space*. It is describes using the follow definition:

$$[\![\mathsf{Bank}_{\mathsf{Ship}}]\!] : U \times I \longrightarrow (U \times O + \mathbf{1})$$

where I,O abbreviate $P+P+P+P+P+P$ and $P+P+P+P+P+P$, respectively.

Function $[\![\mathsf{Bank}_{\mathsf{Ship}}]\!]$ describes how $\mathsf{Bank}_{\mathsf{Ship}}$ reacts to input stimuli, produces output data (if any) and changes state. It can also be written in a curried form as

$$\overline{[\![\mathsf{Bank}_{\mathsf{Ship}}]\!]} : U \longrightarrow (U \times O + \mathbf{1})^I$$

that is, as a *coalgebra* $U \longrightarrow \mathsf{T}\ U$ for functor $\mathsf{T}\ X\ =\ ((X \times O) + \mathbf{1})^I$.

The definition of a new, base component is directly made in HASKELL . A specific strong monad B is chosen to model the envisaged behavioral effect. The code below

corresponds to a Bank component, where B is instantiated to HASKELL Maybe monad to capture partiality.

```
bank (m,("Ins",x)) = Just (m+x,("Ins", m+x))
bank (m,("With",x)) | (m-x)>=0 =  Just( m-x, ("With",  m-x ))
            | otherwise = Nothing
bank (m,("Trans",x)) | (m-x)>=0 =  Just( m-x, ("Trans",  x ))
                     | otherwise = Nothing
```

In a subsequent step the component's interface is created from a suitable annotation in the source code. For this example:

```
@Input: (( P:Ins.:Int + P:With.:Int) + P:Trans.:Int)
@Output:(( P:Ins.:Int + P:With.:Int) + P:Trans.:Int)
```

where `Ins`, `With` and `Trans` are introduced as labels for the component's available services.

The component $Bank_{Ship}$, permits six operations where the first three belongs to the $Bank_1$ and the last three belongs to the $Bank_2$. In the first stage, we put component $Bank_1$ side by side with component $Bank_2$ through the $\boxplus$ combinator and, finally, suitable feedback loops are established, through the hook operator, to connect ports. Formally, we can express this as follows.

$$\mathsf{AlmostBank_{Ship}} \;=\; ((\mathsf{Bank_1} \boxplus \mathsf{Bank_2})[wi,wo]) \circlearrowleft_{P+P}$$

The final interface provide to the user is describe in Figure 3.17, where we show how the inner components will behaving and the the ports $TransB_2$ is connect to the operation

$TransB_2$ of the component $Bank_2$ that will produce a output named $TransB_2$ that will be redirected to operation Ins_1 of the component $Bank_1$ and then we produces a output that will be show to the user by the port $Result$.

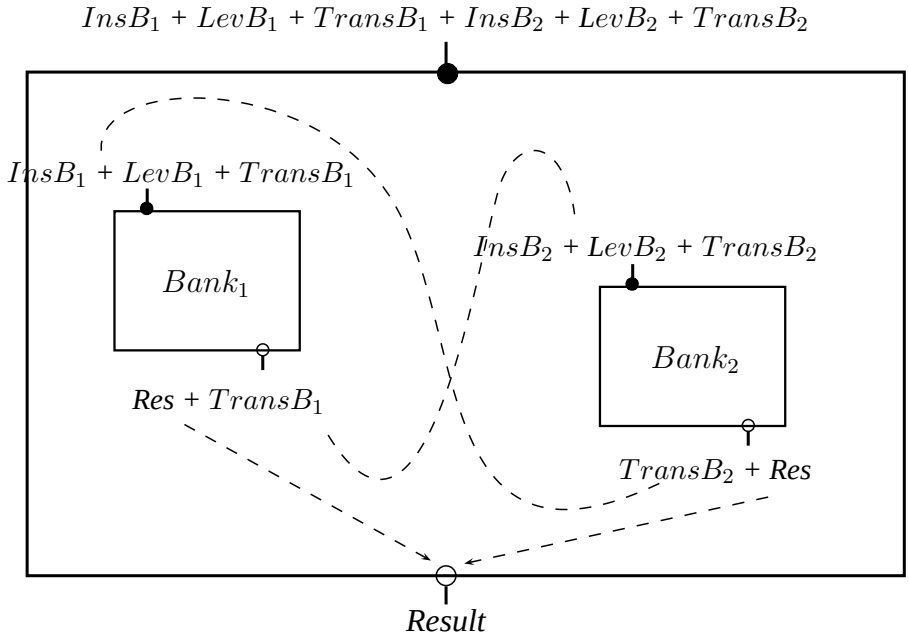


Figure 3.17: Bank Shipment - Final interface

Chapter 4

Behavioural customization

Executions may lead to 'disaster' (e.g., by violation of port pre-conditions on a partial component), and a safe mode in which the effect of a port operation is foreseen and eventually precluded. Component testing, on the other hand, can be made in purely interactive way, running event by event, or by executing a whole sequence of events specified through a regulars expression.

4.0.4 Regular Expressions

Regular expressions describe regular languages in formal language theory. They have thus the same expressive power as regular grammars. Regular expressions consist of constants and operators that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found in most textbooks on formal language theory[12, 23].

Definition: Given a finite alphabet $\sum$. A regular expression over $\sum$ is a word in the language $ER(A)$ over the alphabet $\sum \cup \{\emptyset, \epsilon, (,), |, ., *\}$ inductively defined by:

- $\emptyset$, denotes the empty set
- ϵ, denotes the set containing the empty string
- a, denotes literal character contained in $\sum$
- alternation - if e_1, $e_2 \in ER(A)$, then $(e_1|e_2) \in ER(A)$
- concatenation - if e_1, $e_2 \in ER(A)$, then $(e_1.e_2) \in ER(A)$
- kleene closure - if e $\in ER(A)$, then (e*) $\in ER(A)$

Explaining them with the use of an example is perhaps the best way to understand regular expressions and their use.

Example 5. Good or bad person

Let the alphabet $\sum$ be the 26 letters {a, b, ..., z}. If language $A = \{good, bad\}$ and language $B = \{boy, girl\}$, then:

- A | B = {good, bad, boy, girl}
- A . B = {goodboy, goodgirl, badboy, badgirl}
- A* = {ϵ, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, ...}

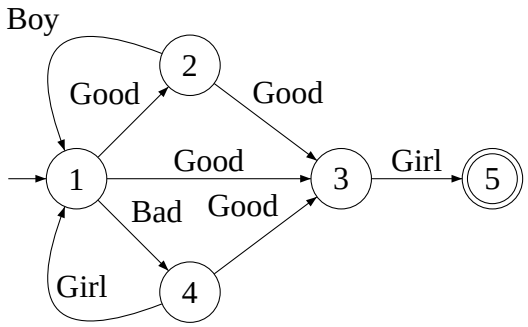
Based on the example 5, we can define that all the boys are good and in a set of girls only one is good, the pattern described is $(good.boy|bad.girl)^*.good.girl$.

The Kleene's Theorem[21] says if an language which can be be defined by either:

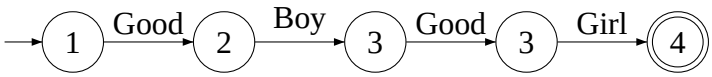
- Regular Expressions
- Finite Automaton
- Non-deterministic Finite Automaton(NFA)

it can also be defined by any of the other models. So for each regular expression it is possible to build an finite automata that recognizes the language it specifies.

This pattern can be converted into an automata and it is given by:

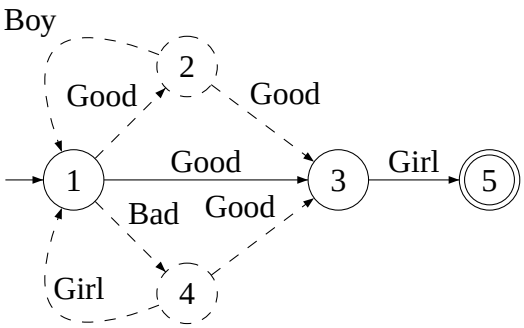


The automata says that it recognises an element if a move can be made from one state to another state, and this move can be done if there is a transition which a recognized symbol. An element will be accepted by the automata if there is a sequence of moves through states of the automata starting at the start state and terminating at one of the terminal states. For instance, the element $goodboygoodgirl$ will produce the following path:



Imagine that our world is described by the previous expression. Let's call the population of $\sum$ that has elements like: {goodboy, goodgirl, badgirl, goodboybadgirl, ...}, where $goodboybadgirl$ represent two different people, one good boy and one bad girl. Our goal is to find a good girl to lead our civilization for a new balanced Karma.

Initially, all people are apt to be the ideal candidate, but we have restricted the candidates to only one a good girl. According to the automata that describes our population, we only want one trace of it that represent the requirements. The trace is given by the following automata, where the trace is not dashed:



Our goal is submit all population to that requirement, which will produce a subset that contains all possible candidates, from that sub-set we are able to select the perfect candidate and singleton good girl.

4.0.5 Prototype with customization

To support modelling, manipulating and animating regular languages in HASKELL, we resort to the $HaLeX$ library [22]. Using this library, the construction of regular expressions and their conversion to the automata has been shown concisely and easily modelled in HASKELL.

This library was developed in the context of a course for undergraduate students. I was one of the students covered by this course, where Professor João Saraiva introduced the basic concepts of regular expressions, finite automata and context-free languages. At the end of the course we had developed a complete $HaLeX$ library.

Figure 4.1 shows the graphical representation of the non-deterministic finite automaton(NDFA) induced by the previous example 5, produced by HaLeX.

Formally, we describe the definition of Non-deterministic Finite Automata, as:

Definition 9. Non-deterministic Finite Automata

- A finite state space X

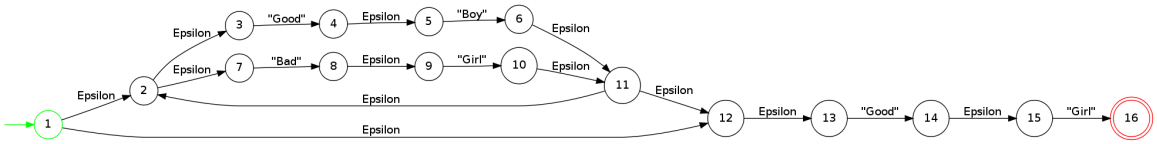


Figure 4.1: Non-deterministic Good or bad person Automaton

- A finite alphabet $\sum$ which represents the possible input symbols. Let $\sum_\epsilon = \sum \cup \{\epsilon\}$
- A transition function, $\delta : X \times \sum_\epsilon \to P(X)$. For each state and symbol, a set of outgoing edges is specified by indicating the states that are reached.
- A start state $x_0 \in X$
- A set $A \subseteq X$ of accept states

We could convert all non-deterministic finite automata into deterministic finite automata(DFA) in order to reduce the number of states, by eliminating the transition labelled by the symbol ϵ, to draw the automaton. In deterministic automata, every state has exactly one transition for each possible input. In non-deterministic automata, an input can lead to one, more than one or no transition(ϵ) for a given state. There are algorithms to convert from any NFA into a DFA with identical functionality, in the majority of the cases a equivalent DFA has the same number of states that a NFA, but with more transitions[1]. So on, let us restrict to the use of NFA.

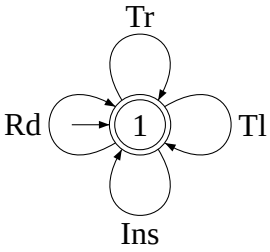
Let's see now how automata and regular expression can be used to specify the constrained behaviour of a component in our prototyper.

Consider again the folder example in chapter 3.1.3. The last interface provided to the final user, had four operations:

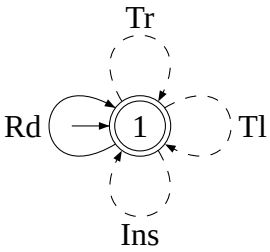
- Turn right page - Tr

- Turn left page - Tl
- Read a page - Rd
- Insert a page - Ins

So the folder needs to support every operation and we express the language $L_1 = (Tr|Tl|Rd|Ins)^*$ with regular expression and then turn it in a automaton:



Now for each operation the user tries, the prototyper runs the automaton to find out whether there is a transition from the initial state to a possible reachable state which is labelled with a possible operation. For example, if the folder already contains the elements "A" and "B" on the left and the elements "C" and "D" on the right side of the folder. Putting side by side the automaton that emerges from the L_1 and the folder. If user chooses operation Read then iff there exists a transition with the symbol $Read$, this operationis allowed to be executed. The result should be element A.



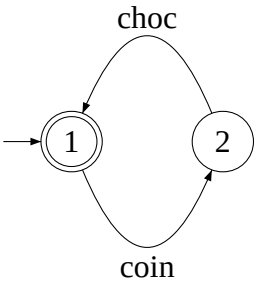
If the user enters a transaction not previously defined, the automaton will not recognize it as a valid path. This customization is used to constrain the way the user can interact with the component.

All the simulations that can be done in the system can be described and exemplified through a sequence trace of the automaton. In this way we create a free mode in which something can possibly go wrong, or a secure mode in which is based on constrained behaviour: only a number of traces are allowed.

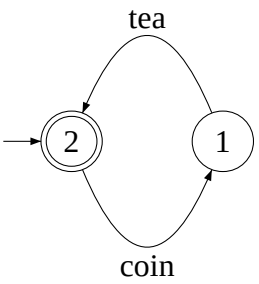
4.0.6 Extension of the regular expressions

The usual regular expression have not the necessary expressive power to express all possible behaviours. In particular we do not have a way to express behaviour generated by the concurrent combinator. So it is necessary to extend regular expressions to model a concurrent computation.

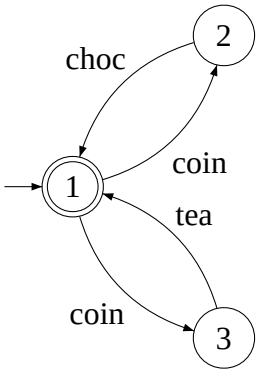
To define concurrent operations, it is especially useful to be able to specify the interleaving of two sequences. Consider for example the waiting room, where a system that has two vending machines exists VM_1 and VM_2. The behaviour of VM_1 can be defined as $(coin.choc)^*$ and we can visualize it as a automaton:



The behaviour of VM_2 as $(coin.tea)^*$ is given by automaton:



With a normal regular expression, the system only work in sequential mode, i.e, first, we can put a coin and get the tea and second we can put a coin and get the chocolate or vice-versa or non of them or even only one of them. We can express that as $(coin.choc|coin.tea)^*$ and with the following automaton.



As expected the previous automaton have not sufficient expressive power to express the behaviour of such system as a real one, where we are able to do it simultaneously. The behaviour of the entire system would be defined as a interleaving of VM_1 and VM_2.

To achieve these objectives, we define an operator called interleaving, denoted by $||$, and with it we are able to define systems with a multiple autonomous process in order to achieve a common goal.

Interleaving is formally defined as follows[9]:

- $a||\epsilon = \epsilon||a = \{a\}, \forall a \in \sum$
- $a.s||b.t = a.(s||bt) \cup b.(a.s||t), \forall a,b \in \sum, s,t \in \sum^*$

For example, if we consider two sets A and B as follows $A = \{ab\}$ and $B = \{ba\}$ then $A||B = \{abac, aaba, abab, bacb, baba\}$

This operator does not increase the modelling power of regular expressions with the interleaving operator, because any expression that uses $||$ can be reduced to a regular expression without $||$.

The automaton that accepts the language represented by the extension of regular expressions is called Parallel Finite Automata(PFA) where it is capable of directly express interleaving forms of parallelism without having it encoded into the meaning of state. The formal definition of PFA is slightly modified to express parallel activity but it is still similar to that commonly used for deterministic and non-deterministic finite automata.

Definition 10. Parallel finite automaton

A PFA can be formally defined as a $7-tuple\ M = (N, Q, \sum, \gamma, \delta, q_0, F)$ in which

- N is a finite set of nodes
- $Q \subseteq 2^N$ is a finite set of states
- $\sum$ is a finite input alphabet
- $\gamma : 2^N \times (\sum \cup \gamma) \longrightarrow 2^{2^N}$ is the node transition function
- $\delta : Q \times (\sum \cup \gamma) \longrightarrow 2^Q$is the state transition function
- $q_0 \in Q$ is the start state
- $F \subseteq N$ is the set of final nodes

and where γ and δ are partial functions.

The node transition function γ is used to generalize the notion, where a element of γ can be defined like $((A,B,a),C,D,E)$, where the transition labelled a exists with sources nodes A and B, and with target nodes C, D, and E.

Initially, the set of active nodes for M is exactly q_0, the initial state. During execution of M, observing the input symbol c in state q, the set of active nodes constituting the next state for M is any one of the sets in $\delta(q,c)$[1].

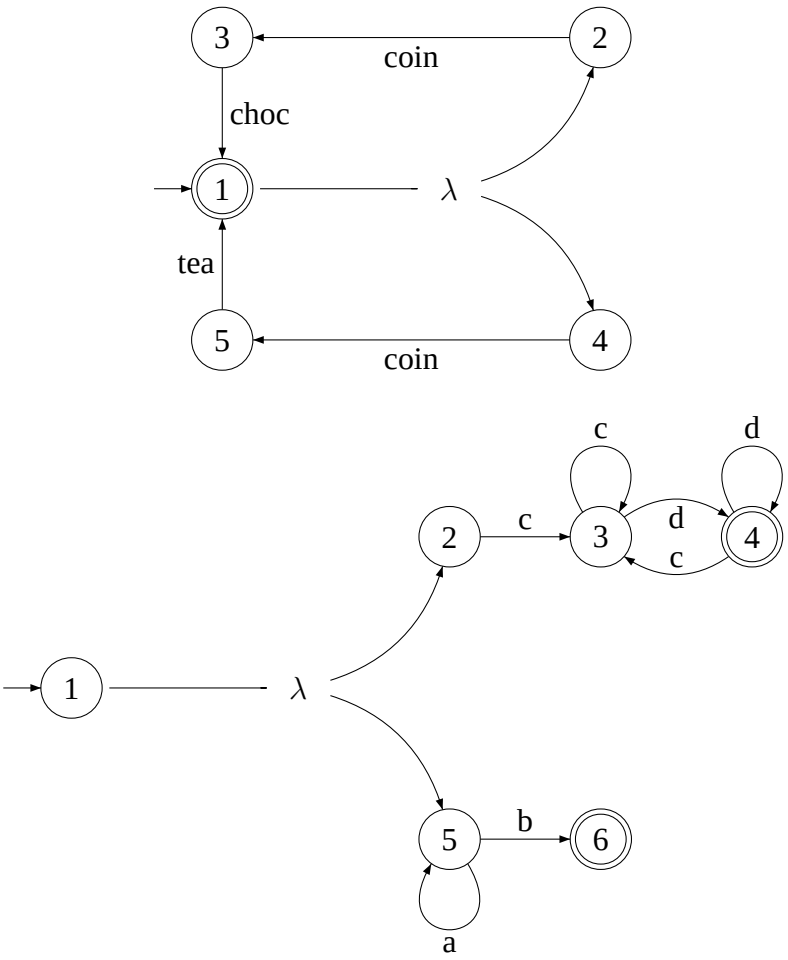


Figure 4.2: PFA - $a^*b||(c^+d^+)^+$

Figure 4.2 is a representation of PFA, where the nodes 4 and 6 are the final nodes, and the initial node is the node 1. This PFA represent the automaton that accept the language:

$$a^*b||(c^+d^+)^+ \tag{4.1}$$

[1]For more details please see the reference [24]

Operator || represents interleaving of languages.

Figure 4.3 specifies the DFA a equivalent to PFA in figure 4.2, ie. that accepts the same language.

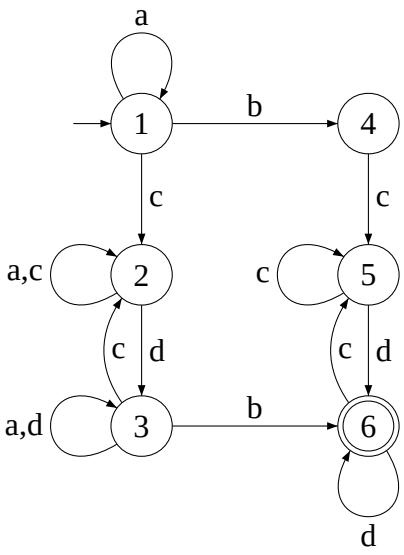


Figure 4.3: DFA - $a^*b||(c^+d^+)^+$

We can do some executions, where we show how the execution sequence was obtained while accepting the word. In this example we will see if the word $accbd$ is accepted.

$$\{1\} \xrightarrow{\lambda} \{2,5\} \xrightarrow{a} \{2,5\} \xrightarrow{c} \{3,5\} \xrightarrow{c}$$
$$\{3,5\} \xrightarrow{b} \{3,6\} \xrightarrow{d} \{3,4\}$$
$$\longrightarrow accept$$

If we execute some word that the automaton must not accept, the trace must be finalized with the word $fail$. The following sequence is one of the possible executions that must

fail.

$$\{1\} \xrightarrow{\lambda} \{2,5\} \xrightarrow{c} \{3,5\} \xrightarrow{a} \{3,5\} \xrightarrow{b}$$
$$\{3,6\} \xrightarrow{a} \{3,6\} \longrightarrow fail$$

The example of PFA shown in the figure 4.2 has an equivalent and minimal DFA shown in the figure 4.3. So we can translate every language with the operator $||$ into a $PFA - automaton$ and then in a $DFA - automaton$. This customization have been added to $Shacc$, producing a new simpler, and more direct way to express the component calculus.

Chapter 5

QoS information

Systems nowadays are typically heterogeneous and geographically distributed, usually exploit communication infrastructures whose topology frequently varies and components can, at any moment, connect to or detach from. The underlying system and the communication resources are constantly changing for a several reasons, including equipment failures, competition from other consumers and security attacks.

Providing a possible hostile computing environment, that requires a dynamic adaptation to changes in quality of service is essential to the survival of the system. There is no shared agreement on what QoS is and what it is not, but generally the service quality is a measure of the non-functional properties of services along multiple dimensions, such as reliability, security, scalability, response time, reputation, and it is often confused with performance level or achieved service quality. The properties of such components cannot be ignored and become decisive in the selection procedures. In brief, QoS is the acceptable cumulative effect on subscriber satisfaction of all implementation and imperfections that are affecting the service.

Quality of service (QoS) can capture different QoS metrics using constraint semir-

ings, that provide a suitable level of abstraction for QoS values. The c-semirings provide an algebraic structure with operations for combining values into a new QoS value.

5.1 Extension of the component calculus

The component calculus in chapter 2 was extended to be taken into account, in an explicit way, as QoS information. The extension of the calculus was discussed in reference [19] where the authors introduced QoS information represented as a Q-algebra where $R = (C, \oplus, \otimes, \otimes, 0, 1)$ is an algebraic structure, with $R_{\oplus} = (C, \oplus, \otimes, 0, 1)$ and $R_{\otimes} = (C, \oplus, \otimes, 0, 1)$, both c-semirings, where:

1. C, represents the QoS domain
2. $\oplus$, represents a choice between two QoS values
3. $\otimes$, compose two QoS values sequentially

The definition of c-semirings[7] entails the following laws:

$$a \oplus a = a \tag{5.1}$$

$$a \oplus b = b \oplus a \tag{5.2}$$

$$a \otimes \mathbf{0} = \mathbf{0} \tag{5.3}$$

$$a \oplus \mathbf{0} = a \tag{5.4}$$

$$a \otimes \mathbf{1} = a \tag{5.5}$$

$$a \oplus \mathbf{1} = a \tag{5.6}$$

$$a \oplus b = b \otimes a \tag{5.7}$$

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c) \tag{5.8}$$

$$(a \oplus b) \otimes (a \oplus c) = a \oplus (b \otimes c) \tag{5.9}$$

We can derive the result of (5.9) using the distribution law already established, *i.e.*

$$
\begin{aligned}
& (a \oplus b) \otimes (a \oplus c) \\
\Leftrightarrow \quad & \{ \text{(5.8)} \} \\
& (a \otimes a) \oplus (a \otimes c) \oplus (b \otimes a) \oplus (b \otimes c) \\
\Leftrightarrow \quad & \{ \text{(5.8)} \} \\
& (a \otimes a) \oplus (a \otimes b) \oplus (c \otimes a) \oplus (c \otimes b) \\
\Leftrightarrow \quad & \{ \text{(5.1),(5.8) and (5.2)} \} \\
& a \otimes (\mathbf{1} \oplus b \oplus c) \oplus (c \otimes b) \\
\Leftrightarrow \quad & \{ \text{(5.6)} \} \\
& (a \otimes \mathbf{1}) \oplus (c \otimes b) \\
\Leftrightarrow \quad & \{ \text{(5.5)} \} \\
& a \oplus (c \otimes b)
\end{aligned}
$$

The operation $\oplus$ defines a partial order $\leq$ on C defined by $a \leq b$, iff $a \oplus b = b$. That is a important rule required to establish the QoS values of each component, meaning that a is worse than b. For example,

$$
\begin{aligned}
& a \oplus b \leq a \oplus b \\
\Leftrightarrow \quad & \{ \text{definition of partial order} \} \\
& (a \oplus b) \oplus (a \oplus b) = a \oplus b \\
\Leftrightarrow \quad & \{ \text{(5.1)} \} \\
& a \oplus b = a \oplus b
\end{aligned}
$$

And,

$$
\begin{aligned}
& a \otimes b \leq a \otimes b \\
\Leftrightarrow \quad & \{ \text{definition of partial order} \} \\
& (a \otimes b) \oplus (a \otimes b) = a \otimes b \\
\Leftrightarrow \quad & \{ (5.8) \} \\
& (a \oplus a) \otimes b = a \otimes b \\
\Leftrightarrow \quad & \{ (5.1) \} \\
& a \otimes b = a \otimes b
\end{aligned}
$$

This sort of representation of QoS informations allows for different ways of combining and choosing between quality values. A new attribute, that represents the QoS information, is included in each operator of the component calculus. On the other hand its execution generates a QoS value which is observable. New definitions appears, that go through an evolution of definitions already defined previously in chapter 2. For example, the first definition 2.2 that appears in this document, was changed into the following definition that contains an additional QoS attribute- C.

Definition 11. A software component with QoS is specified by a pointed coalgebra

$$\langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow \mathsf{B}(U_p \times C \times O)^I \rangle \tag{5.10}$$

where C is the domain of some Q-algebra $R = (C, \oplus, \otimes, \otimes, 0, 1)$. Component calculus changes to take the observed QoS levels of their parameters into account. Most of the component combinators need to be changed to take this into account. An example is

the sequential combinator that becomes:

Definition 12. Sequential composition with QoS information

$$
\begin{aligned}
a_{p;q} &= U_p \times U_q \times I \xrightarrow{\mathsf{xr}} U_p \times I \times U_q \\
&\xrightarrow{a_p \times \mathsf{id}} \mathsf{B}(U_p \times C \times K) \times U_q \\
&\xrightarrow{\tau_r} \mathsf{B}(U_p \times C \times K \times U_q) \\
&\xrightarrow{\mathsf{B}(\mathsf{id} \times a_q)} \mathsf{B}((U_p \times C) \times \mathsf{B}(U_q \times C \times O)) \\
&\xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times C \times (U_q \times C \times O)) \\
&\xrightarrow{\mathsf{BBa}^\circ} \mathsf{BB}((U_p \times C \times (U_q \times C)) \times O) \\
&\xrightarrow{\mu} \mathsf{B}((U_p \times C \times (U_q \times C)) \times O) \\
&\xrightarrow{\mathsf{B}(\mathsf{m} \times \mathsf{id})} \mathsf{B}((U_p \times U_q \times (C \times C)) \times O) \\
&\xrightarrow{\mathsf{B}(\mathsf{id} \times \otimes \times \mathsf{id})} \mathsf{B}(U_p \times U_q \times C \times O)
\end{aligned}
$$

where the use of $\otimes$ denotes the sequential composition of QoS levels.

The same happens with the $hook-\ {}^{\backprime}\!\!\urcorner_Z$, which is essentially a generalization of sequential composition and becomes:

Definition 13. Hook combinator with QoS information

$$
\begin{aligned}
a_{p\urcorner_Z} &= U_p \times (I \times Z) \xrightarrow{a_p} \mathsf{B}(U_p \times C \times (O \times Z)) \\
&\xrightarrow{\mathsf{B}(\mathsf{id} \times \iota_1 + \mathsf{id} \times \iota_2) \cdot \mathsf{dr}} \mathsf{B}(U_p \times C \times (O \times Z) + U_p \times C \times (I \times Z)) \\
&\xrightarrow{\mathsf{B}(\eta + a_p \times \mathsf{id})} \mathsf{B}(\mathsf{B}((U_p \times C \times (O + Z))) + \mathsf{B}((U_p \times C \times (I + Z))))
\end{aligned}
$$

$$\xrightarrow{\mathsf{B}(\mathsf{id}+\mathsf{B}(xl\times\mathsf{id}))} \mathsf{B}(\mathsf{B}((U_p \times C \times (O+Z))) + \mathsf{B}(C \times U_p \times (I+Z)))$$

$$\xrightarrow{\mathsf{B}(\mathsf{id}+\mathsf{B}(id\times a_p))} \mathsf{B}(\mathsf{B}((U_p \times C \times (O+Z))) + \mathsf{B}(C \times \mathsf{B}(U_p \times C \times (O+Z))))$$

$$\xrightarrow{\mathsf{B}(\mathsf{id}+\mathsf{B}(\tau_r\times\mathsf{xl}))} \mathsf{B}(\mathsf{B}((U_p \times C \times (O+Z))) + \mathsf{BB}(C \times C \times U_p \times (O+Z)))$$

$$\xrightarrow{\mathsf{B}(\mathsf{id}+\mu)} \mathsf{B}(\mathsf{B}((U_p \times C \times (O+Z))) + \mathsf{B}(C \times C \times U_p \times (O+Z)))$$

$$\xrightarrow{\mathsf{B}(\mathsf{id}+\mathsf{B}(\mathsf{xl}\cdot(\otimes\times id)))} \mathsf{B}(\mathsf{B}((U_p \times C \times (O+Z))) + \mathsf{B}(U_p \times C \times (O+Z)))$$

$$\xrightarrow{[\mathsf{B}(\mathsf{id}\times\iota_1),\mathsf{B}(\mathsf{id}\times\iota_2)]} \mathsf{B}(U_p \times (C \times C) \times (O+Z))$$

$$\xrightarrow{\mathsf{B}(\mathsf{id}\times\otimes\times\mathsf{id})} \mathsf{B}(U_p \times (C \times C) \times (O+Z))$$

The redefinition of parallel composition, on its turn, resorts to definition of $\otimes$ where:

Definition 14. Parallel composition with QoS information

$$\begin{aligned}
\alpha_{p\boxtimes q} = U_p \times U_q \times (I \times J) &\xrightarrow{\mathsf{m}} (U_p \times I) \times (U_q \times J) \\
&\xrightarrow{\alpha_p\times\alpha_q} \mathsf{B}(U_p \times C \times O) \times \mathsf{B}(U_q \times C \times K) \\
&\xrightarrow{\delta_l} \mathsf{B}((U_p \times C \times O) \times (U_q \times C \times K)) \\
&\xrightarrow{\mathsf{Bm}} \mathsf{B}((U_p \times C) \times (U_q \times C) \times (O \times K)) \\
&\xrightarrow{\mathsf{B}(\mathsf{m}\times\mathsf{id})} \mathsf{B}((U_p \times U_q) \times (C \times C) \times (O \times K)) \\
&\xrightarrow{\mathsf{B}(\mathsf{id}\times\otimes\times\mathsf{id})} \mathsf{B}(U_p \times U_q \times C \times (O \times K))
\end{aligned}$$

The combinator choice, $\boxplus$, in turn uses the operator - $\oplus$. The QoS level of $p \boxplus q$ is computed as $c_1 \oplus c_2$ where $\oplus$ is the glb of order $\leqslant$. Formally, the combinator choice becomes:

Definition 15. Combinator choice with QoS information

$$\begin{aligned}
\alpha_{p \boxplus q} &= U_p \times U_q \times (I \times J) \xrightarrow{(\mathsf{xr}+a)\cdot\mathsf{dr}} U_p \times I \times U_q + U_p \times (U_q \times J) \\
&\xrightarrow{a_p \times \mathsf{id} + \mathsf{id} \times a_q} \mathsf{B}(U_p \times C \times O) \times U_q + U_p \times \mathsf{B}(U_q \times C \times R) \\
&\xrightarrow{\tau_r + \tau_l} \mathsf{B}(U_p \times C \times O \times U_q) \times \mathsf{B}(U_p \times U_q \times C \times K)) \\
&\xrightarrow{\mathsf{B}(\mathsf{id} \times \mathsf{xr}) + \mathsf{Ba}^\circ} \mathsf{B}(U_p \times U_q \times C \times O) + \mathsf{B}(U_p \times U_q \times C \times R) \\
&\xrightarrow{[\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)]} \mathsf{B}(U_p \times U_q \times (C \times C) \times (O + R)) \\
&\xrightarrow{\mathsf{B}(\mathsf{id} \times \oplus \times \mathsf{id})} \mathsf{B}(U_p \times U_q \times C \times (O + R))
\end{aligned}$$

Finally, the concurrent composition is defined using the operator $\circledast$ as:

Definition 16. Combinator concurrent with QoS information

$$\begin{aligned}
\alpha_{p \circledast q} &= U_p \times U_q \times (I \times J) \xrightarrow{\circledast} U_p \times U_q \times (I + J + I \times J) \\
&\xrightarrow{\circledast} U_p \times U_q \times (I + J) + U_p \times U_q \times (I \times J) \\
&\xrightarrow{a_{p \boxplus q} + a_{p \boxtimes q}} \mathsf{B}(U_p \times U_q \times C \times (O + R)) + \mathsf{B}(U_p \times U_q \times C \times (O + R)) \\
&\xrightarrow{[\mathsf{B}(\mathsf{id} \times \iota_1), \mathsf{B}(\mathsf{id} \times \iota_2)]} \mathsf{B}(U_p \times U_q \times (C \times C) \times (O + R)) \\
&\xrightarrow{\mathsf{B}(\mathsf{id} \times \otimes \times \mathsf{id})} \mathsf{B}(U_p \times U_q \times C \times (O + R))
\end{aligned}$$

5.2 Extension of the prototype to mirror the QoS extended calculus

Few changes need to be designed to attach QoS infrastructure into the prototype. The prototype was developed already to facilitate the integration of new functionalities, since it is divided in three layers - Components, Combiners and Final system interface. This modification will change mostly the part of component definition. It must contain a new attribute to represent the value of QoS, and change the definition of the interfaces in the Combiners layer. The algebraic structure $R = (C, \oplus, \otimes, \otimes, 0, 1)$ mentioned in chapter 5.1, will be represented in the HASKELL implementation with a datatype defined, as:

```
data QualityOfService v b a'=
                            QoS   {
                                    value      :: v,
                                    choice     :: b-> b-> a',
                                    sequential :: b-> b-> a',
                                    concurrent :: b-> b-> a'
                                  }
                                   deriving ( Typeable)
```

The $value$ is the initial QoS variable, the $choice$, the $sequencial$ and the $concurrent$ operators are functions that will transform QoS variables in a new QoS variable. The definition of such functions should be provided by the user depending on what sort of QoS measures he is interested in. The QoS information defined in all components will be propagated and perhaps modified through the applied combinators and its effect will

be shown at the end of the system. For example, when we want to know how long the system takes to process a operation, i.e. the round-trip delay time, we need to establish the functions that will support such behaviour.

The action $choice$ can be defined by the function max:

```
max :: (Ord a) => a -> a -> a
```

because the function is applied to two components, but only one is actually executed at a time.

The operation $sequencial$ can be defined by the function sum':

```
sum' :: (Num a) => a -> a -> a
```

as one component is executed after the other, we need to increment the time in each execution.

Finally the operation $concurrent$ can be defined by the function max:

```
max :: (Ord a) => a -> a -> a
```

where the components were executed and the component that takes less time waits for the slower component to finish execution.

Example 6. Publisher subscribe

In this section we will introduce a small example that represents the Publish/subscribe architectural schema wit QoS propagation. Publish/subscribe is a messaging pattern where publishers do not send messages directly to specific receivers (subscribers). Subscribers express the interest in one or more publishers, and they only receive

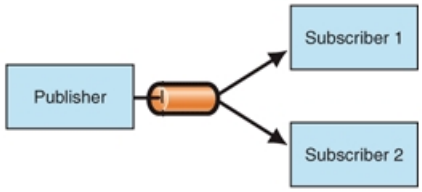
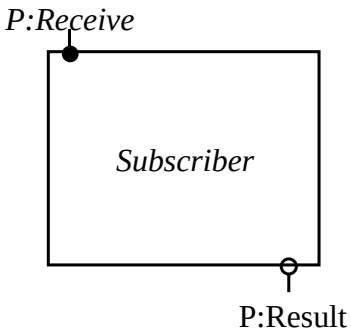


Figure 5.1: Publish/subscribe

messages that are of their interest. This mechanism of publishers and subscribers can allow scalability and a more dynamic network topology.

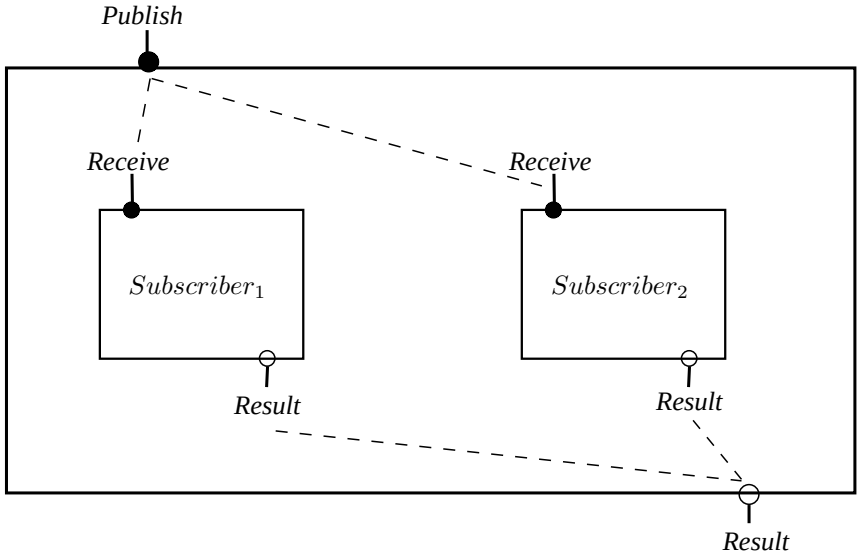
Figure 5.1 shows an integration solution where the publisher write a message in the channel and two subscribers that had subscribed to these topics, receive the messages from the subscribed channel.

The first step is to define a component that represents the subscriber with the operation $Receive$ that represents the messages received through the channel and produces a behaviour represented by the port $Result$, i.e.:



In this example, we have two subscribers - $Subscriber_1$ e $Subscriber_2$; who are registered to a specific topic written by the publisher. Therefore the system should be

in charge of forwarding all messages from the subscribed channel. The system can be represented by the following diagram, in which there are two components inside the system that represent the two existing subscribers. The publisher writes the message, the system duplicates the message and sends it to the two subscribers simultaneously.



The definition of a new, base component is directly made in HASKELL . A specific strong monad B is chosen to model the envisaged behavioral effect. Figure 5.2 corresponds to a $Subscriber$ component, where B is instantiated to HASKELL Maybe monad to capture partiality. where, the $qosValue$ is described by figure 5.3. In a

```
subscriber1  (xs, x) =Just ( x:xs,(qosValue 1, x))
```

Figure 5.2: Component subscriber

subsequent step the component's interface is created from a suitable annotation in the source code. For example:

```
@Input: P:Receive
@Output: P:Result
```

```
qosValue x = QoS {
                   value = x,
                   choice = max ,
                   sequential= sum',
                   concurrent = max
                  }
```

Figure 5.3: Definition of the subscriber qos value

where `Receive` and `Result` are introduced as labels for the component's available services.

The component $Publisher$ permits one operation which will be duplicated to be sent to subscribers. In the first stage, we put component $Subscriber_1$ side by side with component $Subscriber_2$ through the $\boxtimes$ combinator. After this step we need to duplicate the message and for this we use the diagonal operator, i.e.:

$$\triangle \; : \; \mathsf{C} \longrightarrow \mathsf{C} \times \mathsf{C}$$

So, at this stage we need to join the definitions given above with the combiner ;. Formally, we can express this as follows.

$$\mathsf{Publisher} \; = \; \triangle ; (\mathsf{Subscriber}_1 \circledast \mathsf{Subscriber}_2)$$

SHACC allows both the (interactive) activation of this sort of component expressions and their execution in a simulation mode. Actually, once components are defined either from scratch (*i.e.*, by providing the corresponding HASKELL code directly) or by composition of other components, SHACC offers an environment for simulation testing. If the component's behaviour model allows the *Run* window in the tool offers two simulation modes: a *free* mode in which, execution may lead to 'disaster' (*e.g.*, by

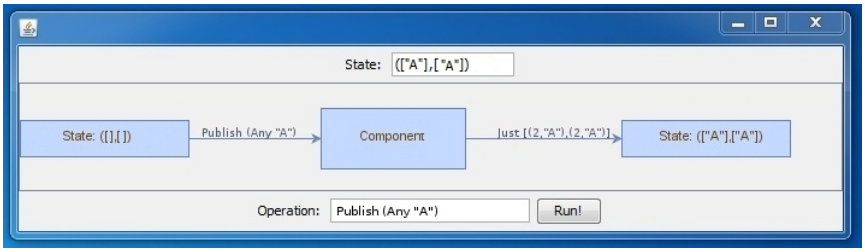


Figure 5.4: Publish/subscribe

violation of port pre-conditions on a *partial* component), and a *safe* mode in which the effect of a port operation is foreseen and eventually precluded. Component testing, on the other hand, can be made in a purely interactive way, running event by event, or by executing a whole sequence of events specified through a regular expression and supplied to the tool. Figure 5.4 illustrates the tool execution mode.

The box labelled *State* in Figure 5.4 shows the initial value of the component's state. Box *Operation*, on the other hand, accepts the component service to receive the inputs from the environment. Executing a service from the component's interface SHACC displays three boxes that representing the component state before, during and after service completion.

Chapter 6

Conclusion

This dissertation invested in the study of a component calculus. Several aspects remain to be explored. So in this document we started by joining two ideas that favour an observational semantics for state-based systems The thesis developed an Haskell library,and a prototype with a graphical user interface with the intent of helping the design of systems based on components.

The prototype is the implementation of the calculus. It provides a framework to specify, compose and animate software components. This calculus, which generalizes the algebra of Mealy machines, acts as a glue code for wiring autonomous components. A component's interface is, basically, a collection of ports through which values flow. Several operators are given to produce new components from old ones. Co-algebra proved to be helpful in the calculus since it is a convenient way to observe a component's behaviour upon a status change caused by some trigger, and the use of monads to encode behaviour models.

SHACC is the name of the prototyping framework developed. It includes:

- full animation of the component calculus

- a simulation mode guided by a customization expression
- the incorporation of QoS reasoning in the definition of the component combinators

This framework was tested with a wide number of the examples. The examples that we have presented in this dissertation proved to be useful in improving the tool, in particular by refining of the way that the user specifies the software components, and how this can be captured in an efficient Haskell implementation. We believe this technology will help create better structured components, through the implementation and design in SHACC.

In this work we show how such combinators can be neatly and effectively implemented in Haskell by exploring some techniques that we have implemented in our work. This provides not only a smooth way to directly incorporate component-ware in Haskell, but also a testable method for prototyping software patterns.

We have also seen how we can specify some behaviour customization, resorting to regular expressions to help us in the definition of the customization behaviour. But this choice was just one of the possibles paths that we could have chosen. Instead of the extension of regular expressions we could have resort to a formalism for specifying and verifying concurrent system for example CCS(Milner, 1980). The Calculus of Communicating Systems (CCS) is a process calculus introduced by Robin Milner around 1980. The formal language includes primitives for describing parallel composition, choice between actions and scope restriction. CCS is useful for evaluating the qualitative correctness of properties of a system such as deadlock or livelock.

In this document we show how we can extend the calculation of components to deal with measures of QoS with the help of the generalization of the notion of Q-algebra.

This is a smooth extension of the original calculus. As mentioned in the bibliography [QoS] it is feasible and it opens many paths to be followed, providing a new way to guarantee not only the functional simulation relation given by the behaviour of components but also a higher service quality.

With this work, we have not only gained a new method to implement software components, but we have also gained quality and organization. We have seen with examples how simple it is to implement systems in SHACC and how the based-components implementation is well organized with simple primitives and code organization that allows the fast use of this technology. The simplicity of our language hides, however, the powerful mechanism and strategies that we support.

State-based component calculus, is an area with great potential in software architecture. Such a study would constitute an excellent future project and it would quantify the true gains on specifying a calculus of qos-aware software components through the SHACC prototype. Our language captures a fair amount of information and allows us, as we have illustrated in several examples, to perform relevant architectural transformations.

Availability. SHACC is available from shacc.wetpaint.com. It is documented in publication [15].

Bibliography

[1] James A. Anderson. *Automata Theory with Modern Applications*. Cambridge University Press, New York, NY, USA, 2006.

[2] L. S. Barbosa and J. N. Oliveira. State-based components made generic. In H. Peter Gumm, editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1. Elsevier, 2003.

[3] L. S. Barbosa, M. Sun, B. K. Aichernig, and N. Rodrigues. On the semantics of componentware: A coalgebraic perspective. In Jifeng He and Zhiming Liu, editors, *Mathematical Frameworks for Component Software*, pages 69–117. World Scientific, 2006.

[4] Luís S. Barbosa. Towards a calculus of state-based software components. *Journal of Universal Computer Science*, 9:891–909, 2003.

[5] Luís Soares Barbosa and José Nuno Oliveira. State-based components made generic. *Electr. Notes Theor. Comput. Sci.*, 82(1), 2003.

[6] Richard Bird and Oege de Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[7] Tom Chothia and Jetty Kleijn. Q-automata: Modelling the resource usage of concurrent components. *Electron. Notes Theor. Comput. Sci.*, 175:153–167, June 2007.

[8] Klaus Denecke and Shelly L. Wismath. *Universal Algebra and Coalgebra*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2009.

[9] Vijay K. Garg and M. T. Ragunath. Concurrent regular expressions and their relationship to petri nets. *Theor. Comput. Sci.*, 96:285–304, April 1992.

[10] H. Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality.*, volume 2428 of *LNCS*. Springer, 2002.

[11] Holger Hermanns. *Interactive Markov chains: and the quest for quantified quality*. Springer-Verlag, Berlin, Heidelberg, 2002.

[12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[13] L.S.Barbosa. *Component as Coalgebra*. PhD thesis, Universidade do Minho, PT, 2001.

[14] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of generalised stochastic petri nets for the performance evaluation of multiprocessor systems. In *Proceedings of the 1983 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '83, pages 198–199, New York, NY, USA, 1983. ACM.

[15] André Martins, Luís Barbosa, and Nuno Rodrigues. Shacc: A functional prototyper for a component calculus. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 413–419. Springer Berlin / Heidelberg, 2011.

[16] Alain J. Mayer and Larry J. Stockmeyer. The complexity of word problems - this time with interleaving. *Inf. Comput.*, 115:293–311, December 1994.

[17] Daniel A. Menasce. Composing web services: A qos view. *IEEE Internet Computing*, 8:88–90, November 2004.

[18] Sun Meng and Luis S. Barbosa. Towards the introduction of qos information in a component model. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2045–2046, New York, NY, USA, 2010. ACM.

[19] Sun Meng and Luís Soares Barbosa. Towards the introduction of qos information in a component model. In *SAC'10*, pages 2045–2046, 2010.

[20] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.

[21] Jacques Sakarovitch. Kleene's theorem revisited. In *Selected contributions on Trends, techniques, and problems in theoretical computer science. 4th International Meeting of Young Computer Scientists*, pages 39–50, London, UK, 1987. Springer-Verlag.

[22] Joao Saraiva. HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. In M. Hanus, S. Krishnamurthi and S. Thompson, editor, *Pro-*

ceedings of the ACM Workshop on Functional and Declarative Programming in Education, University of Kiel Technical Report 0210, pages 133–140, September 2002.

[23] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[24] P. David Stotts and William Pugh. Parallel finite automata for modeling concurrent software systems. *Journal of Systems and Software*, 27:27–43, 1994.

[25] Tao Yu and Kwei jay Lin. K.: Service selection algorithms for composing complex services with multiple qos constraints. In *In: ICSOCâĂŹ05: 3rd Int. Conf. on Service Oriented Computing*, pages 130–143, 2005.

[26] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30:311–327, May 2004.

Appendix A

User's Guide of *Shacc*

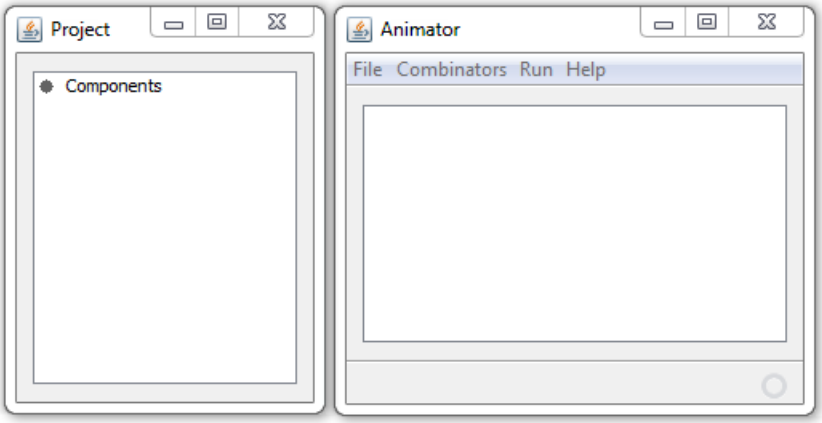


Figure A.1: Main windows

In our framework the first window (figure A.1) shows a compiler for the functional language Haskell, where one can see if all components were successfully created. Along with this first window, the tool provides another window named Project, which gives us an overview of the entire system of components being developed, by showing the components created or added. The menu bar present in window Animator has three sub-menus: Menu File that permits some basic operations as:

- Import - Add existing components directly, they will appear in the window Component
- Open - Open existing file Haskell in a new window named Open File
- Load - Load existing file Haskell, run a compiler and show the result in the window Animator
- Exit - Quits from application

Menu Combinators includes the following combinators:

- External Choice - Produces an interface through the amalgamation of two disjoint interfaces
- Hook - Part of the output is redirected to the input ports of the same component
- Wrap - Adds or modifies on interface to a component

Menu Run is provided to test the system being developed. Menu Help contains some information about the context of this framework.

In window Opened File in the figure A.2, we can reload the current component, open a new component, save the current component and exit from the current window. All these operations are available from the sub-menu File. In sub-menu Extra we are allowed to create components and add them to the window Project automatically.

Note: In order to do this correctly, it is necessary to annotate the code, the input and the output, for each interface port. If the component takes some argument, we add P:, if it receives nothing, we add 1:. E.g. ((1: A + P: B) + 1: C), means that A and C do not receive any arguments and B receives an argument.

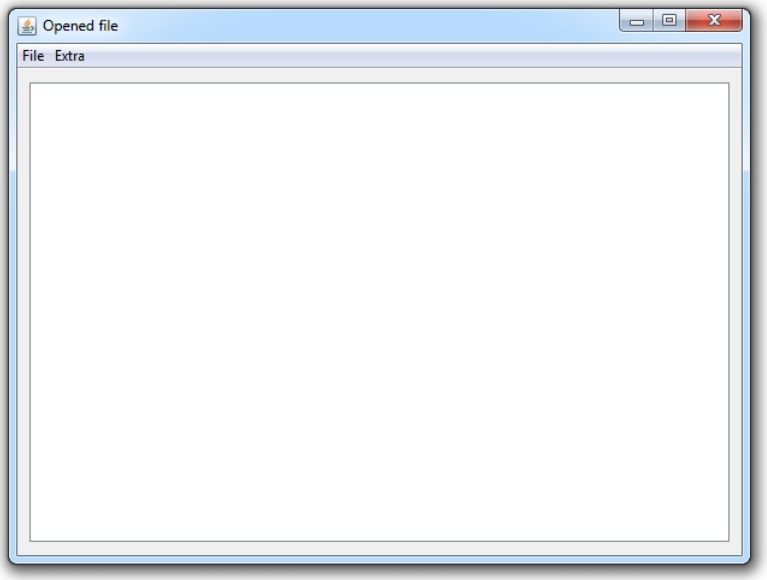


Figure A.2: Window where we can choose to create or import components

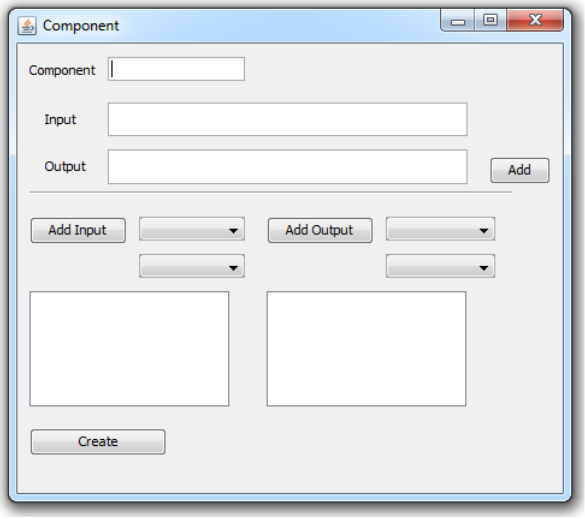


Figure A.3: Create components

Now, we explain each combinator window usage. The window Component in the figure A.3 belongs to the sub-menu Wrap, consisting of one label where we can put the name of new component, one label where to define the input interface and another to define the output interface. Subsequently we will show how an interface can be defined.

As an example, suppose we want to specify an interface with three ports. We can put them as ((A+B)+C) or (A+(B+C)) without any behaviour change. The step is achieved by pressing the Add button, used to connect the interface defined with the component. At the end of this process we can press the Create button that will generate the code which represents component.

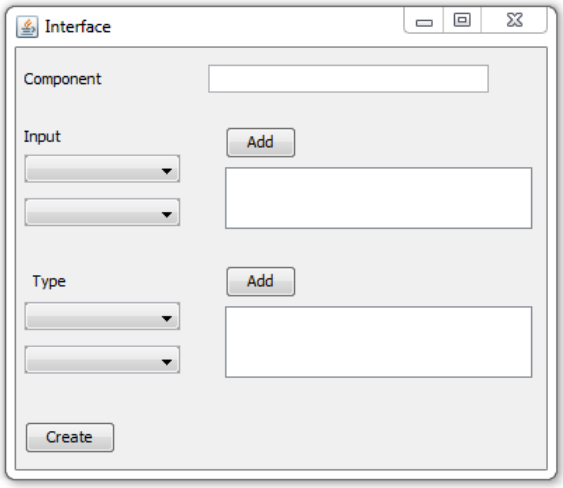


Figure A.4: Main windows

Window Interface - figure A.4, is the second part of the sub-menu Wrap, where we name the new component and connect the several ports, in order to produce the desired effect on the component.

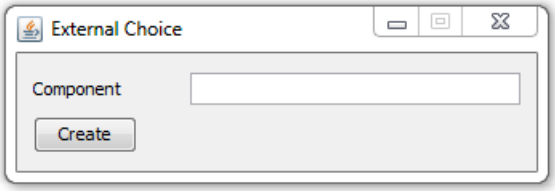


Figure A.5: External choice

In window External Choice in the figure A.5 , all we need is to put a name of the new component and press button Create.

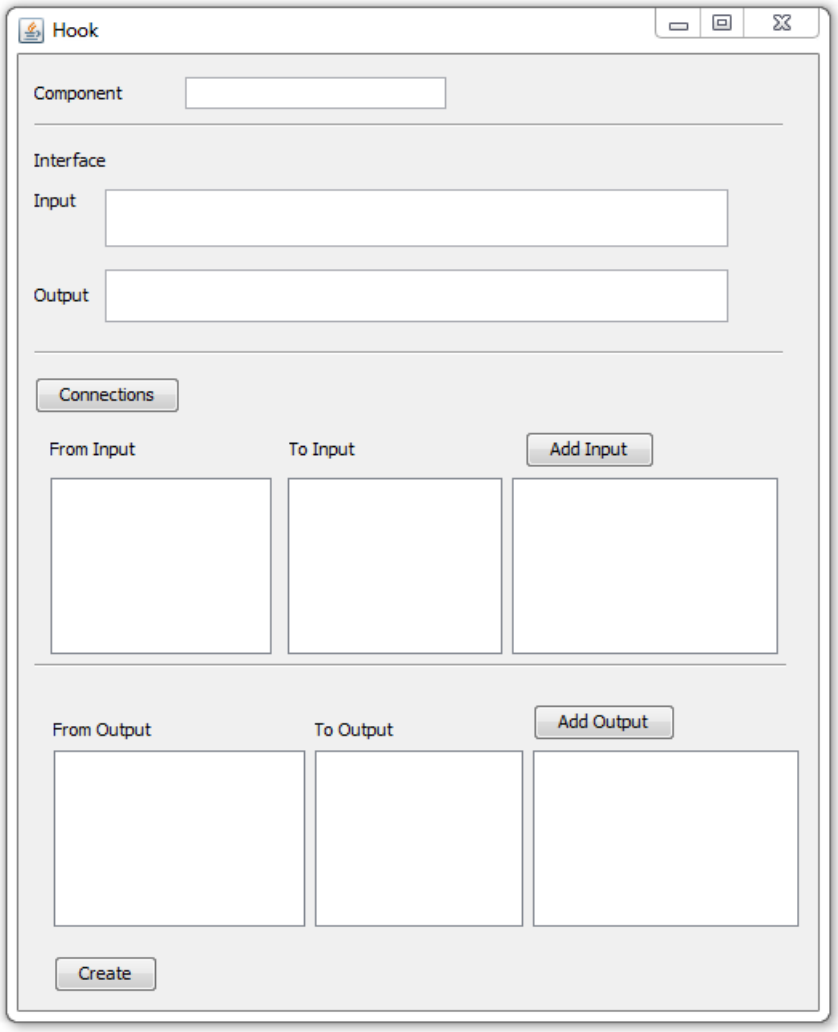


Figure A.6: Hook window

Window Hook(see in figure A.6) allows one to define the name of the new component and its supporter interface. In this step we need to press the button Connections which allows to perform all connections between the interface defined previously and the internal component interface. To finish this component definition we must press button Create.

Finally, Run window in figure A.7, we can be accessed through menu Run, by

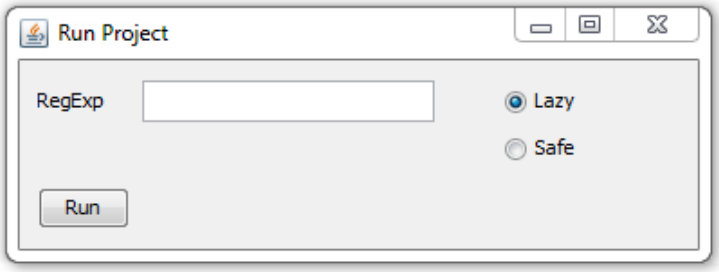


Figure A.7: Running tests

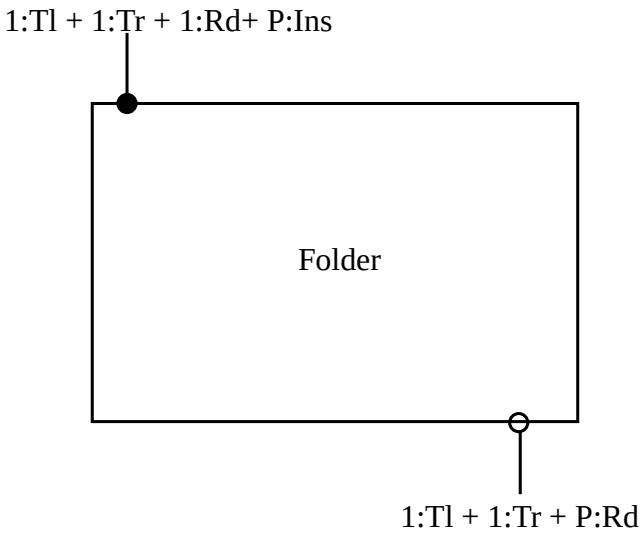
selecting Run Project. This window allows one to test the composed system built into the tool. Here, we can choose between a lazy execution, where the animation allows the crash of the system, and a safe execution, where the animation is based on previously defined regular expressions[1].

Folder

A Folder component is built through the combination of two stacks modelling, respectively, the folder left and right piles. The Folder component provides ports corresponding to the operations: read, insert, turn page right and turn page left.

[1] **Important Note:** In order to perform each actions it is necessary to choose at least one of the components defined in the Project window, the number of components needed to select is describe as:

- Component - Select 1 Component
- Interface - Select 1 Component
- External Choice - Select 2 Components
- Hook - Select 1 Component
- Run Project - Select 1 Component



In a second stage, we will use the stacks, to helping in the computation, where it is characterized by three fundamental operations, pop, top, push. The pop operation removes an item from the top of the stack, and returns this value to the caller. The top operation return the current top element of the stack without removing it. The push operation adds an item to the top of the stack, hiding any items already on the stack, or initializing the stack if it is empty.

Denoting by U its internal state, a stack of values of type P is handled through the usual

$$\mathsf{Top}: \ U \longrightarrow P, \quad \mathsf{Pop}: \ U \longrightarrow P \times U \quad and \ \mathsf{Push}: \ U \times P \longrightarrow U$$

operations. An alternative, 'black box' view hides U from the stack environment and regards each operation as a pair of input/output ports. For example, the top operation becomes declared as:

$$\mathsf{Top}: \ 1 \longrightarrow P,$$

where 1 stands for the nullary(or unit) datatype. The intuition is that top is activated with the simple pushing of a 'button' (its argument being the stack private state space) whose effect is the production of a P value in the corresponding output port. Similarly typing push as:

$$\mathsf{Push}: \ P \longrightarrow 1,$$

means that an external argument is required on activation but no visible output is produced, but for a trivial indication of successful termination. Such 'port' signatures are grouped together in the diagram below. Combined input type $1 + 1 + P$ models the choice of three functionalities (top, pop and push in this order), of which only one takes input of type P.

Representation in Shacc: Open Shacc go to the menu File and click Open, and in the directory of the tool we will find a file named "Stack.hs", press open and we will see a windows as the follow image: In the number 1 we defined the input and output port

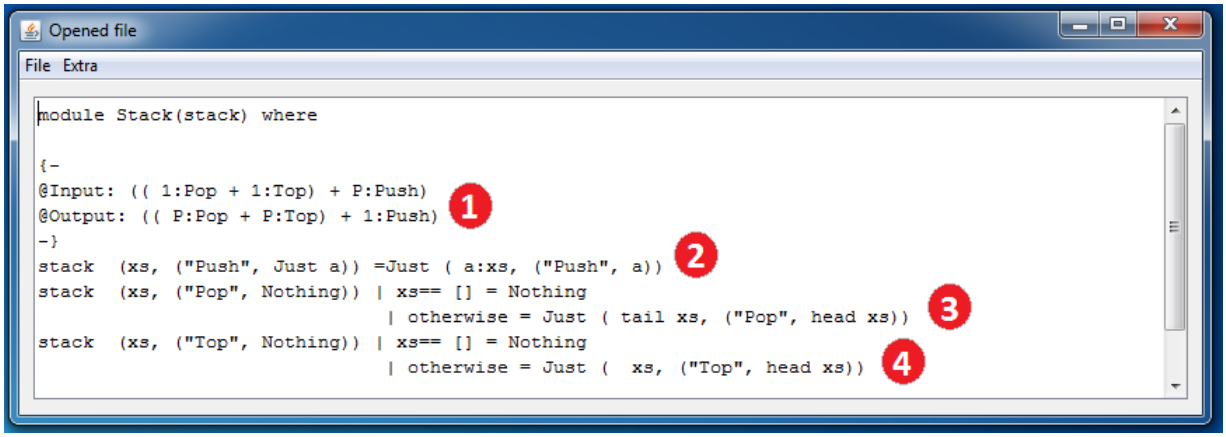


Figure A.8: Stack

as well if it receive and produce a value (1 represents that port does not receive any type in argument, and P represents a value type). The number 2, 3 and 4 is the function

of stack who receives a pair in argument, and in first element in this pair is the inner state of stack, the second element is as well other pair who in the first element identifies what kind of operation and the second element is the argument of this operation.

In next stage, suitable feedback loops are established, through the hook operator, to connect ports. This ensures, form example, that the left turn of a page is achieved through a pop action on the right stack connected to the push of the left one. Formally, this amounts to the following expression in the component calculus:

$$\mathsf{Folder} \;=\; ((\mathsf{LeftS} \boxplus \mathsf{RightS})[wi,wo]) \;{}^{\backprime}\!\!\urcorner_{P+P}$$

where $\mathsf{RightS} = \mathsf{Stack}[id + \triangledown, id]$ and $\mathsf{LeftS} = \mathsf{Stack}[i_2 + Id, (id{+}!_{p+1}) \cdot a_+]$.

Representation in Shacc: $LeftS$ is allowed to make two operations (pop and push), these operations are linked to the operations on $Stack$. In the first stage, we need select the interface of stack, who is found in the Project Window(see Figure A.9 where the interface is marked with 1) Now, fill the fields like the Figure A.10 Finally, when we

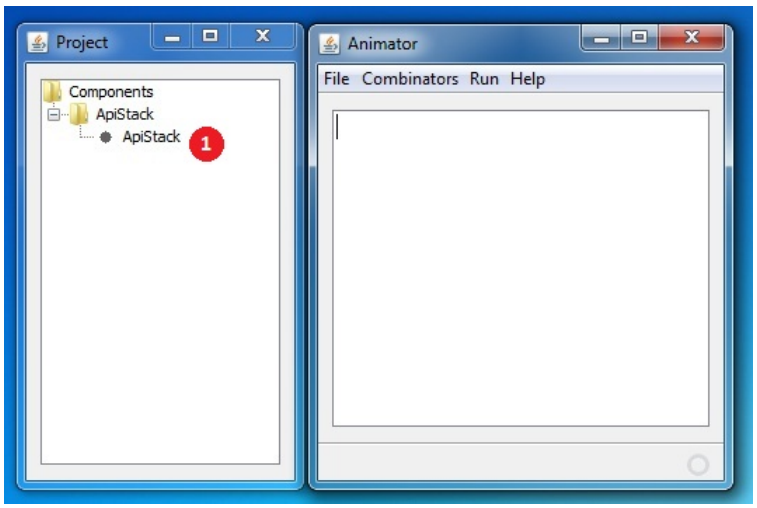


Figure A.9: Select Stack

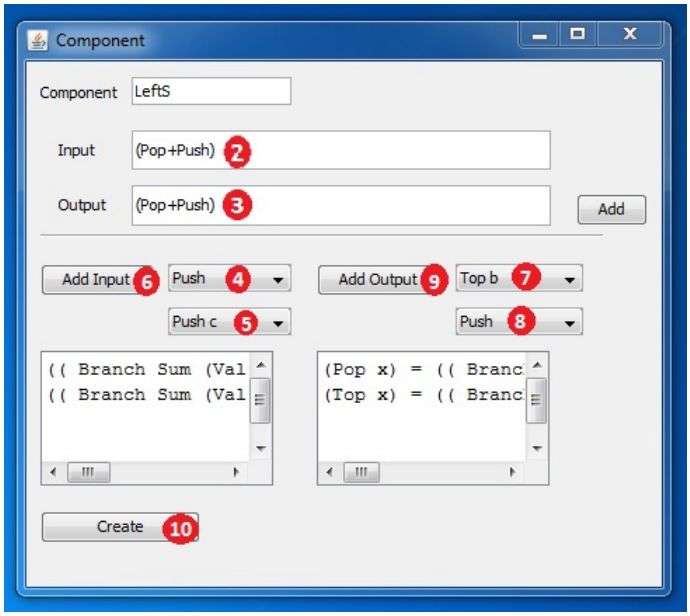


Figure A.10: LeftS component

press button $Create$ a new item will appear on window $Project$ in main window - $LeftS$.

Do the same steps for $RigthS$, and in the windows $Component$ first we need named the new component with RightS, in the second and third stage we need label operations in input and output with $((Pop+Top)+(Push1+Push2))$ and $((Pop+Top)+Push)$ respectively, and then press Add button who permits linked this new labels with the operations in the Stack, this step corresponds to the number 4, 5 and 6 to the interface input, and 7,8 and 9 to the interface output. Follow the Figure A.11 to fill all fields, and press button $Create$ a new item will appear on window $Project$ in main window - $RigthS$.

Now go to the main windows (Figure A.12). As we can see number 1 show $LeftS$ and number 2 show $RightS$, the next step is necessary select first the $LeftS$ and then $RightS$ in order to combine with external choice. With the to components selects go

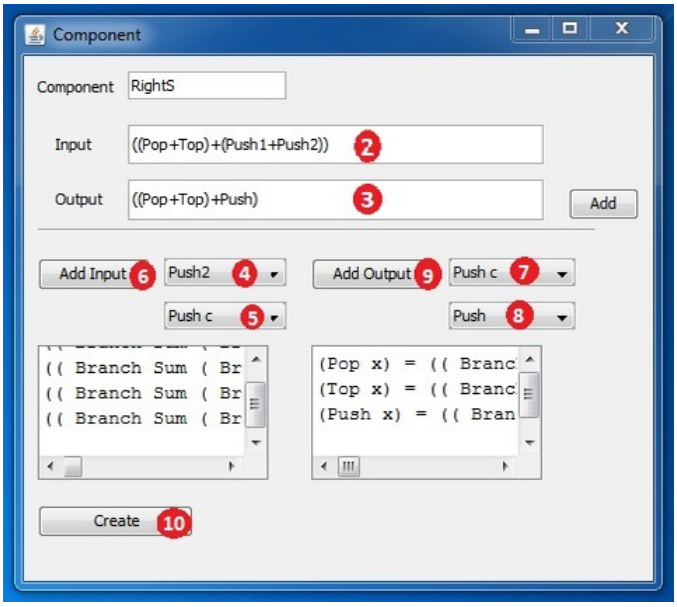


Figure A.11: RightS component

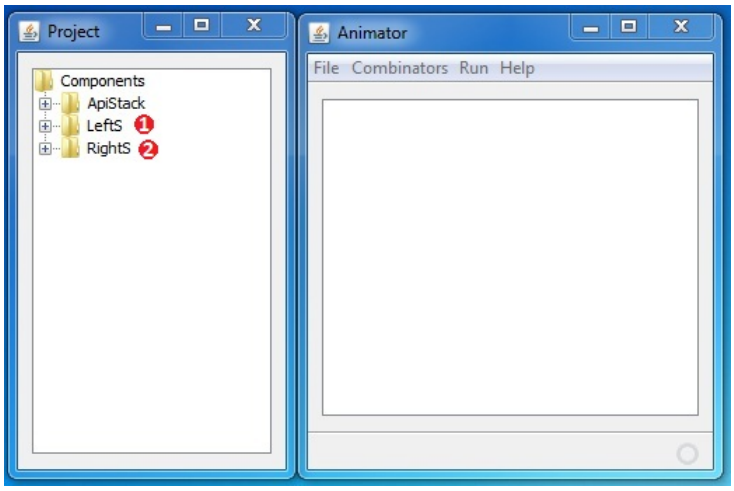


Figure A.12: Main window

to menu $Combinators$ and click on item $Externalchoice$, and a new window will appear. Insert the new name - $LeftSAndRightS$, then press the button create and a new item will be added on $Project$ window, Figure A.13.

In this stage, we select the component $LeftSAndRightS$ (number 1 Figure A.13)

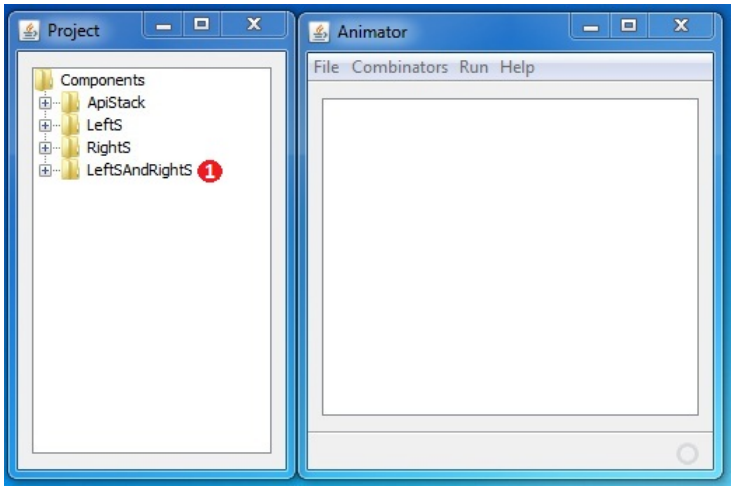


Figure A.13: External choice

and go through menu Combinators and select item Hook, and the window Hook will appear. To complete this step is necessary fill the name, the interface and how we will connect wires of the interface. In the field Component is necessary introduce the new name of component - $AlmostFolder$. In the section interface we have a label for input where we introduce $((((Tr + Tl) + Rd) + Ins) + (Push1 + Push2))$ and for the output - $(((Tr + Tl) + Rd) + (Push1 + Push2))$, matches to numbers 1 and 2 next image. To pass to next phase is required press the button$Connections$, then will be appear all ports (see numbers 4, 5 , 7 and 8 in Figure A.14) corresponds to the interface of this component and the interface of component that will be used by this component. Select each item in the list From Input (4) with the corresponding list To Input (5), and then click on button Add Input (6) in order to make up the list of links below the button Add Input. The same should be done in list $FromOutput$ and $ToOutput$. Finally we able to press the button $Create$ and a new item will appear in the window $Project$ named $AlmostFolder$.

In this last step we will provide the interface that will provide a front end so they can

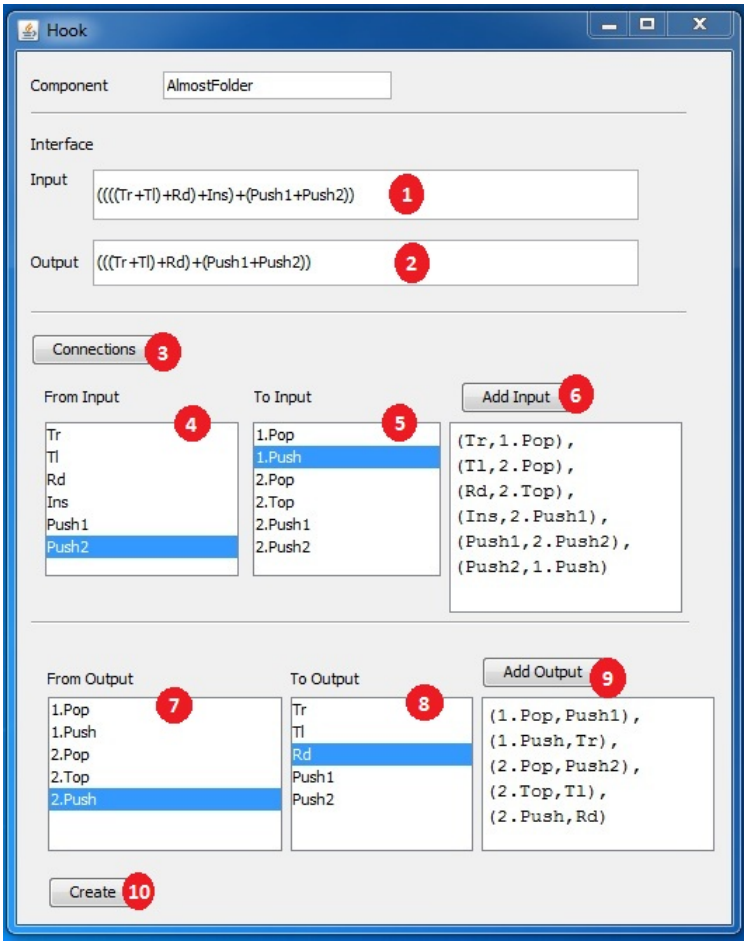


Figure A.14: Applying hook

interact with the system based on component. First name the component with $Folder$, in the Figure A.15 on number 1 and 2 corresponds to the interface of the component used by this and the final $datatype$ that this component will be used, connect them and press the button Add. In number 4 and 5 is where we can filter some of arguments of component, if a port return one element and that will not interest to end-system then we can return a Nil that correspond to nothing. After this phase click on $Create$ button, and we are able to do stage of tests. In the test phase, this component allows us to do four operations:

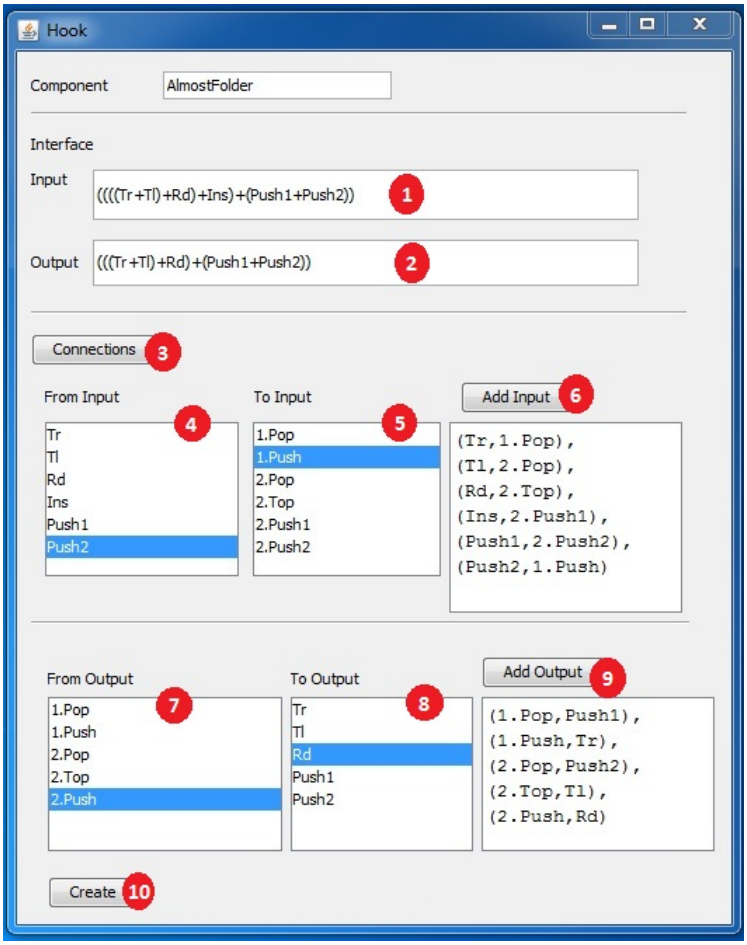


Figure A.15: Final interface

- Tl
- Tr
- Rd
- Ins

To introduce one of the operations allowed, go through the menu Run in window Animator, select Run Project and choose Lazy mode and press Run. In number 1 in Figure A.16 is where we introduce the state of component, in this case is a pair of

empty list - ([],[]). In number 2 is where we introduce the operations chosen. Aiming to show how it makes those operations, we will illustrate them all as a template:

- Tl (Nil) - Turn left do not receive any argument
- Tr (Nil) - Turn right do not receive any argument
- Rd (Nil) - Read do not receive any argument
- Ins (Any "A") - Insert a element on system

interest to end-system then we can return a $Nill$ that correspond to nothing. After this phase click on $Create$ button, and we are able to do stage of tests. For example suppose

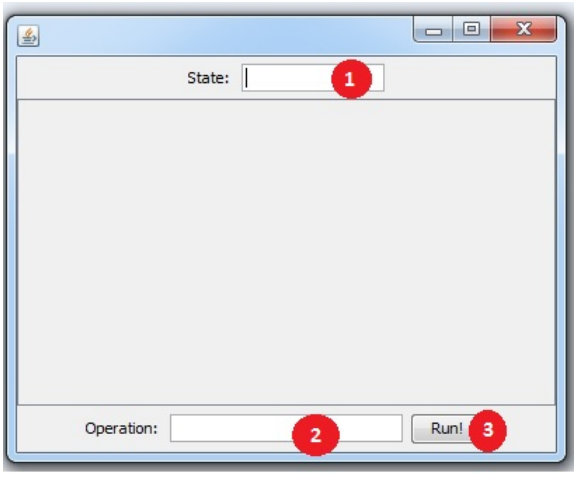


Figure A.16: Test window

we choose to do a turn right the output is in Figure A.17, where the previous stage of component have a element A in left stack and after we do turn right, the same element will pass to the right stack.

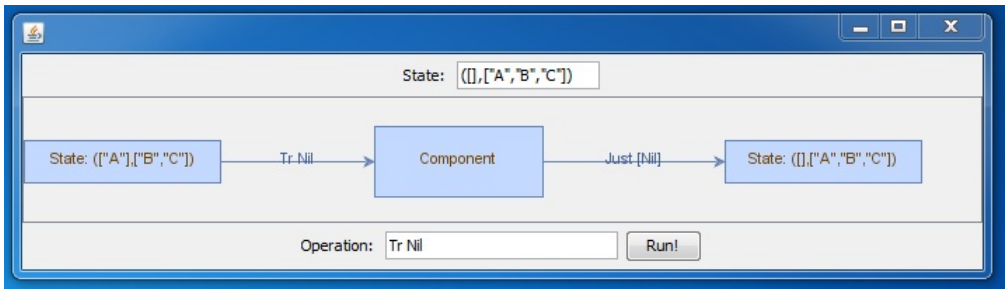


Figure A.17: Testing turn right page