Scientific Research Publishing

Journal of Software Engineering and Applications, 2018, 11, 552-567
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

A Grammatical Approach of Multi-Localization of GUIs: Application to the Multi-Localization of the Checkers Game GUI

Maurice Tchoupé Tchendji, Freddy-Viany Tatou Ahoukeng

Department of Mathematics and Computer Science, Faculty of Sciences, University of Dschang, Dschang, Cameroon
Email: ttchoupe@yahoo.fr, maurice.tchoupe@univ-dschang.org, freddyviany@gmail.fr

How to cite this paper: Tchendji, M.T. and Ahoukeng, F.-V.T. (2018) A Grammatical Approach of Multi-Localization of GUIs: Application to the Multi-Localization of the Checkers Game GUI. *Journal of Software Engineering and Applications*, **11**, 552-567.
https://doi.org/10.4236/jsea.2018.1111033

Received: October 23, 2018
Accepted: November 27, 2018
Published: November 30, 2018

Abstract

By proposing tools that help for the accomplishment of tasks in almost all sectors of activities, computer science has revolutionized the world in a general way. Nowadays, it addresses the peculiarities of peoples through their culture in order to produce increasingly easy-to-use software for end users: This is the aim of software localization. Localizing a software consists among other things, in adapting its GUI according to the end user culture. We propose in this paper a generic approach allowing accomplishing this adaptation, even for multi-user applications like gaming applications, collaborative editors, etc. Techniques of functional interpretations of abstracts structures parameterized by algebras, constitute the formal base of our approach.

Keywords

GUIs, Context-Free Grammars, Multi-Localization, Abstract Syntax Tree and Its Interpretations, XML, GUI Description Language, Haskell

1. Introduction

Regarded as the fourth pillar for sustainable development [1], according to the UNESCO[1] Convention on the Protection and Promotion of the Diversity of Cultural Expressions, cultural diversity is a "world heritage" [2] and constitutes a great wealth for the people. Therefore, the protection, promotion and maintenance of cultural diversity are essentials for sustainable development for the benefit of present and future generations. Moreover, as far as computer science is concerns, one can easily agree that, a given user (a human), will use much more

[1]UNESCO: United Nations Educational, Scientific and Cultural Organization.

intuitively a software if he finds in the latter, elements that are culturally close to him as language, iconography, and the color palette used, etc.

From the observations above, it is clear that it is imperative to take into account the various aspects related to cultural diversity during the development of a software. This is not only to ensure that it is built in respect of the culture of its various end users, but also to produce increasingly easy-to-use software for end users. In order to achieve these objectives, cultural computing[2] is the paradigm that should be used if we want to efficiently take into account the possible cultural diversity of end users, when producing a software tool. With this in mind, software publishers, are now using knowledge from cultural computing to improve the user's experience of each of the end users of their tools: The tool must be constructed in such a way that it is the GUI (Graphical User Interface) that adapts to the user and not the other way around.

In order to produce several (localized) versions of the same software product, publishers use two techniques: internationalization (i18n) and localization (l10n) [4] [5]. Internationalization is the process of generalization which, in software design, allows abstracting it from the peculiarities of a given culture by representing intentionally the objects which it manipulates. According to Schäler [6], localisation is "*the linguistic and cultural adaptation of digital content to the requirements and locale of a foreign market, and the provision of services and technologies for the management of multilingualism across the digital global information flow*".

In order to perform a linguistic and cultural adaptation of an application, among other activities, its GUIs must be translated to the signifiers, the habits, etc of targets cultures. **Figure 1** presents two located examples of George Weah's bibliography in Wikipedia for Francophone culture[3] (**Figure 1(a)**) and for Arab culture[4] (**Figure 1(b)**). On these figures, we can notice some differences concerning the language used, the position of images and menus, colors, etc.

Most single-user software used around the world are not located. It's even worse when you consider the case of those running on a network[5], because for such applications, it might be ideal to offer to each application user at a given time, a GUI who is consistent with its own culture. For example, it's about designing a multiplayer game or a collaborative editing application so that, each participant interacts on a located GUI in their culture. This is what we call

[2]The goal of cultural computing is to "*addresse underlying and almost unconscious cultural determinants that have since ancient times a strong influence on our way of thinking, feeling and worldview in general*" [3]. As far as GUI is concerns, it allows the end user of a given software to experience an interaction closely related to the fundamental aspects of his culture.

[3]https://fr.wikipedia.org/wiki/George_Weah

[4]https://ar.wikipedia.org/wiki/George_Weah

[5]This is easily verifiable on popular internet multi-player game applications as *Le Seigneur des Anneaux Online*: *les Ombres d'Angmar*, *Conan Exiles*, *Street fighter*, etc. or for collaborative online text editors such as: *etherpad*, *Mediawiki*, *FidusWriter* etc. for which all the GUIs of the different participants in the game or in the edition are all identical, and are really suitable only for those users who master the language of the manufacturer. Even though they are located, only the linguistic aspect (English translation) is addressed.

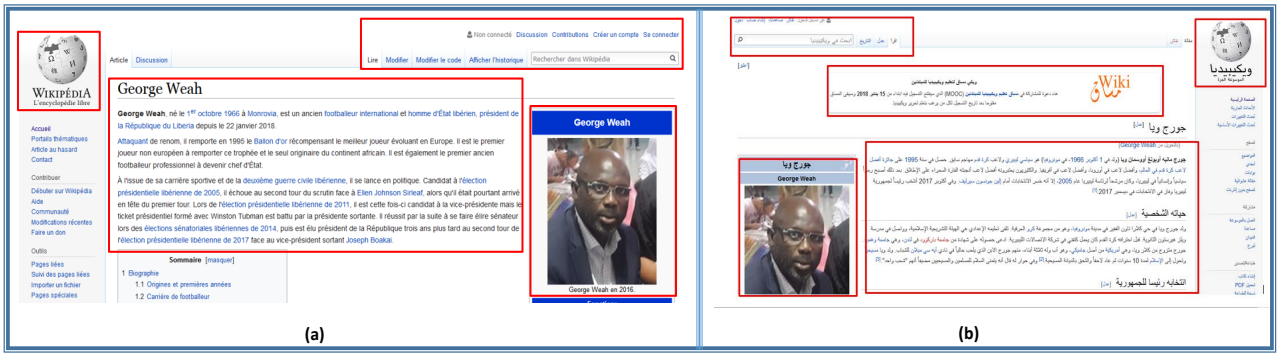

Figure 1. Bibliography of George Weah in Wikipedia: (a) for French culture, (b) for Arab culture.

multi-localization or *simultaneous localization* and this is what constitutes the main objective of this paper: to propose a new approach of external multi-localization of GUIs of collaborative and interactive software.

Starting from the observation that the tree structure of a GUI can be represented intentionally by an abstract syntax tree (AST) for a given context-free grammar (see Section 2.1), we borrow techniques and tools from the domains of language theory, compilation, and functional programming, to show that, a particular localization of a GUI represented intentionally by an AST, is in fact only a particular interpretation of this AST according to a given algebra: this is the formal base on which our multi-localization approach is based.

More precisely, from an abstract grammar (AST's model of GUIs of a given application), we deduce: 1) Haskell data types[6] [7] [8] for each of its grammatical symbols, 2) types for algebras that will be use to write as many interpreters as wanted, in as many GUI description languages (FXML[7] [9], UIML [10], etc.), 3) evaluation functions, each parameterized by both an algebra and a *localization file* which will make it possible to locate the AST by interpretation. A synoptic view of the proposed approach is sketched in **Figure 2**.

Organization of the manuscript: Section 2 introduces some concepts related to software localization and GUI modeling using context-free grammars (CFG). Our multi-localization approach, followed by its experimentation (derivation of multi-located GUIs for a network checkers game) is presented in Section 3, while Section 4 is devoted to the conclusion.

2. Preliminaries

2.1. Context-Free Grammars as GUI's Models

A graphical window (GUI) consists of a set of basic components[8] (Image, Menu, Table, Text Box, etc.), arranged in relation to each other (horizontally, vertically,

[6]Haskell is a functional programming language who is used (without prejudice to the generality) as the supporting functional language of this presentation.

[7]FXML: JavaFX features a language known as FXML, which is a HTML like declarative markup language. The purpose of this language is to define a user interface.

[8]These components are also called *Graphical user interface elements*; they are components used by GUIs to visually represent information stored in computers: They are visible on the GUI.

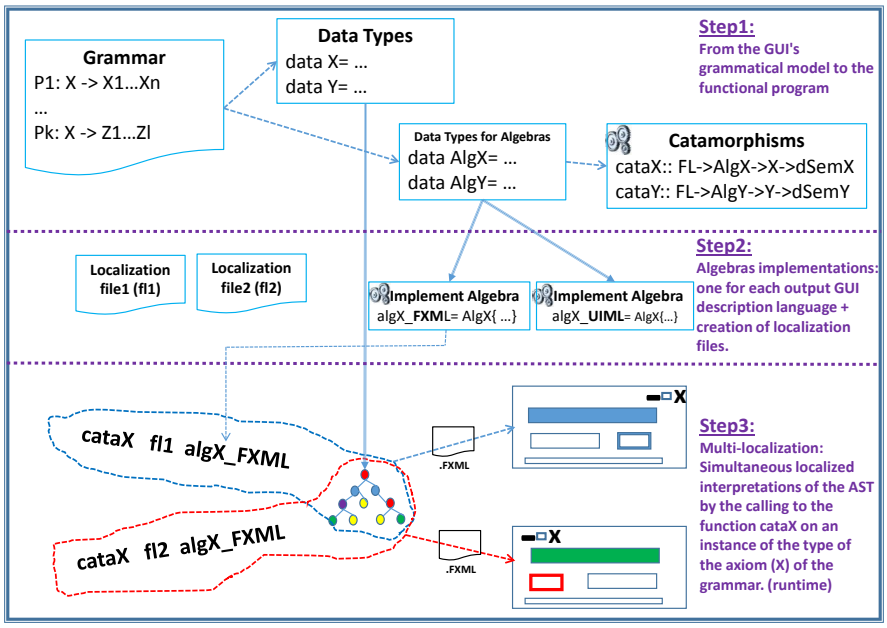

Figure 2. A synoptic view of the proposed multi-localization approach.

etc.) to a precise positions on the window by means of the so called *structural components*[9] [11] [12]. Presented this way, one can easily realize that, the rules governing the grouping of the (basics and/or structural) components of a GUI can be described using the production rules of a CFG. Therefore, a GUI (tree structure) can be intentionally represented by an AST of a CFG. In fact, for a given graphical application, we can construct a CFG in such a way that, any of its GUIs is intentionally represented by an AST of the latter (see Section 3.2.1): CFG can be used as models of the GUIs of an application. This is the use made of it in this paper.

Recall that a CFG defines the structure of its instances (AST) by means of productions. A production, generally denoted $p: X_0 \to X_1 \cdots X_n$ is assimilated in the GUIs context, to a structuring rule showing how the component X_0, located on the left hand side of the production, allows structuring/grouping the other components $X_1 \cdots X_n$ located on the right hand side. More formally, we have the following definition:

Definition 1 An abstract context free grammar (CFG) is given by $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ composed of a finite set $\mathcal{S}$ of grammatical symbols or sorts corresponding to the different syntactic categories involved, a particular grammatical symbol $A \in \mathcal{S}$ called axiom, and a finite set $\mathcal{P} \subseteq \mathcal{S} \times \mathcal{S}^*$ of productions. A production $P = \left(X_{P(0)}, X_{P(1)} \cdots X_{P(|P|)}\right)$ is denoted $P: X_{P(0)} \to X_{P(1)} \cdots X_{P(|P|)}$, $|P|$ denotes the length of its right hand side and $lhs(P)$ (resp. $rhs(P)$ return its left (resp. right) hand side. A production with the symbol X as left part ie. $lhs(P) = X$, is called a X-production.

An AST is a tree whose nodes are labeled by the grammatical symbols of the

[9]These components are also called *layout components*; they are invisible on the (rendering of) GUI and are used to specify the arrangement of the basic components on which the user can interact.

grammar and is such that, for any internal node labeled X, having n sons labeled $X_1, \cdots, X_n$, rule $X \to X_1 \cdots X_n$ must be a grammar production. Moreover, the leaf nodes must be associated with ε *-productions*.

Definition 2 The set $AST(\mathbb{G}, X)$ of **abstract syntax trees** according to the grammar $\mathbb{G}$ associated with grammatical symbol X consists of trees in the form $X[t_1, \cdots, t_n]$ where X is the label of the root node[10] of the tree, $t_1 \cdots t_n$ are subtrees of the root node and there is a production $P : X \to X_1 \cdots X_n$ such that $X = X_{P(0)}$, $n = |P|$ and $t_i \in AST(\mathbb{G}, X_i)$ for all $1 \le i \le n$.

AST can be interpreted as evidence of the conformity of the GUI (tree structure) with the grammar.

2.2. On Software Localization: External vs. Inner Localization

The localization of a software does not only concern its GUIs. This is an activity that also includes the technical documentation provided with the software (installation guides, user manuals, etc.), online help and so on [13].

When the need to locate softwares was felt, the first answer given to this new challenge by the software publishers was purely linguistic and recommended working directly on the source code of the application: this is what we call *inner software localization* [4]. Since the strings to be translated are generally scattered throughout the application's code, the localizer must have access to it. The latter must then be either a translator with programming knowledge, or a programmer with translation knowledge, or both must work in perfect intelligence.

Granting free access to the source code to a third party who is not the editor poses at least the problem of security and confidentiality. Indeed, since translators do not generally have a great programming knowledge, the risk is great that they inadvertently modify, or copy for unconfessed purposes the source code available to them for translation [4].

In the early nineties, a new form of localization using *resource files*[11] and called *external software localization* [14] has been created. It allows translators to process the text contained in the GUIs without need to have any particular programming knowledge, or to constantly need the assistance of the programmers. They only intervene on the *resource file* (containing localizations of localizable elements such as text, etc.) which is subsequently delivered with the software product. In production, the choice of the localization to use for a particular running of the software is done either interactively at the start of the software, or is previously set in a configuration file. By doing so, the software developed is not only completely independent of the end user culture, but is also highly extensible from a localization perspective. Indeed, it is enough to create a new localization file containing data relating to a new culture, for it to be taken into account by the tool.

[10]In the following, as long as there is no ambiguity, we will not differentiate between a node and its label.

[11]A resource refers to a structural or constant element of software that can be referenced at any time in the body of the main program.

Recall that the work presented in this manuscript focuses exclusively on the external localization of GUIs.

3. A grammatical Approach of External Multi-Localization of GUIs

3.1. Derivation of a (Parameterized) Functional Interpreter of ASTs

We have seen (section 2.1) that the logical structure of a GUI can be represented by an abstract CFG.

Let $\mathbb{G}_1 = (\mathcal{S}, \mathcal{P}, A)$ with $\mathcal{P} = \{P_1 : X \to X_1 \cdots X_n, P_2 : X \to Y_1 \cdots Y_m, \cdots, P_k : X \to Z_1 \cdots Z_l\}$, be a grammatical model of GUIs of a software. We present in this subsection how to encode $\mathbb{G}_1$, its ASTs as well as their interpreters in the functional language Haskell.

Well-known techniques describing how to carry out a functional implementation of a CFG, and how to write interpreters of its ASTs exist [15] [16]. We briefly present below how we proceed. Note that, our way of doing things is not fundamentally different from others but has the advantage of being modular. Indeed, whereas generally a single algebra structure is associated with a grammar, we suitably associate an algebra structure with each syntactic category of the grammar and consider in fine that, the algebra structure associated with the grammar is the one associated with its axiom.

Abstract grammar allows to specify syntactic structures that can be associated in Haskell to a set of algebraic data types[12] describing the different syntactic categories used in grammar. Terms of a data structure are generally subject to several interpretations, all of them following the same recursion pattern. This is why they are generally specified by means of algebras[13] formally defined as follows:

Definition 3 Let $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ be a CFG. An algebra $\mathcal{A}_X$ associated with a grammar symbol $X \in \mathcal{S}$ is given by: 1) an interpretation domain D_Y associated with each grammar symbol Y appearing in a X-production of $\mathcal{P}$, 2) a reference to the $\mathcal{A}_Z$ algebra of each grammar symbol Z appearing on the right-hand side of a X-production. 3) an application $P^{\mathcal{A}} : D_{X_{P(1)}} \times \cdots \times D_{X_{P(|P|)}} \to D_X$ associated with each X-production $P : X \to X_{P(1)} \cdots X_{P_{P(|P|)}}$. The algebra $\mathcal{A}_{\mathbb{G}}$ associated with the grammar $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ is the one associated with his axiom A: $\mathcal{A}_{\mathbb{G}} = \mathcal{A}_A$.

Given an abstract grammar $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$, we deduce from systematic way the Haskell data types, the associated algebra data types, and the evaluation functions (catamorphisms[14]) as follows:

[12]An algebraic data type is a kind of composite type, whose values are data of another type wrapped in one of its data constructors.

[13]Intuitively, an algebra is the homologue of *interfaces* in java language. For a given algebraic data type, it encapsulates the types of various interpretation functions of this one: there is as much interpretation functions as of *data constructors* of the given data type.

[14]In functional programming, catamorphisms provide generalizations of folds of lists to arbitrary algebraic data types. It effectively computes a "simple value" from a "container like" structure and a computation mechanism to compose the values in it.

1) A data type is created for each grammar symbol.

2) For each created data type, an associated *algebra data type* is created. It consists of two groups of selectors:

- The group formed by types of *interpretation functions* of algebraic data type: there is one for each data constructor of algebraic data type,
- The group containing references to algebras associated with the different algebraic data types used in the definition of the type of which the current algebra is associated.

3) For each data type created, an *evaluation function* encapsulating the recursion pattern is created; it is parameterized by the algebra associated with this data type.

Application of the approach to the grammar $\mathbb{G}_1$

1) Creating data types:

```
1  data X = P1 {selX1 :: X1, ..., selXn :: Xn}
2         | P2 {selY1 :: Y1, ..., selYm :: Ym}
3           ...
4         | Pk {selZ1 :: Z1, ..., selZl :: Yl}
5  data X1 = ...
6  ...
7  data Zl = ...
8  ...
```

2) Creation of data types for algebras: D_{X_i} denotes the type variable representing the interpretation domain of the type X_i.

```
1  data AlgX DX DX1 ... = AlgX {
2       -- group of interpretation functions
3       interP1 :: DX1 -> DX2 -> ... -> DXn -> DX,
4       interP2 :: DY1 -> DY2 -> ... -> DYm -> DX,
5       ...
6       -- group of references on used algebras
7       interAlgX1 :: AlgX1, ..., interAlgZl :: AlgZl }
8  data AlgX1 ...
9  ...
```

3) Evaluation function parameterized by algebra data type associated to X data type[15].

```
1  cataX :: AlgX -> X -> DX
2  cataX algX = evalX where
3    evalX (P1 x1 ... xn) = interP1 algX valx1 ... valxn
4         where
5              valx1 = cataX1 (interAlgX1 algX) x1
6              valx2 = cataX2 (interAlgX2 algX) x2
7              ...
8    evalX (P2 y1 ... xm) = interP2 algX valy1 ... valym
9         where
10             valy1 = cataY1 (interAlgY1 algX) y1
11             valy2 = cataY2 (interAlgY2 algX) y2
12             ...
13   ...
```

[15]Note that in line 5 of this function, we evaluate the component x_1 by using the algebra dedicated to it; it is contained in the *interAlgX*$_1$ component of *algX*.

3.2. A Multi-Localization Approach: The Steps

In order to be intensively localizable, GUIs have to be internationalised to abstract any information directly related to a culture. This justifies the decomposition of our multi-localization approach into two phases: an internationalization phase whose objective is to produce a grammatical model (an abstract grammar) of all the GUIs of the software; followed by a phase of multi-localization, whose objective is to produce on the fly, located GUIs for each of the participants in the collaboration.

3.2.1. The Internationalization Phase

The internationalization phase can be considered as an analysis-design phase, allowing not only to identify the localizable elements of the GUIs for the abstraction purpose, but also to highlight the structural relations which exist between them. Only a summary description of this phase is given below, because it is not the main object of this study. It takes place in four steps:

1) *Sketching*: produce a sketch of all the GUIs of the software. This can be done because, this set is finished.

2) *Identification*: identify for each GUI the localizable elements for their abstraction; *symbolic names* will be found and only one occurrence of similar elements appearing in more than one GUI is retained.

3) *Production of ASTs*: identify the structural relationships between localizable elements of the GUIs and for each relation identified, study constraints related to the relationship; for example, the meaning of the relation that can be different from one culture to another, etc. This step ends with the production of an AST draft for each GUI: this is its intentional representation.

4) *Derivation of the grammar model of the application's GUIs*: from the different ASTs produced in the previous step, derive their CFG by considering that, the set of AST's labels form the set of grammatical symbols and that, a label of a inner node and those of its directs children form a production.

3.2.2. The Multi-Localization Phase

This phase starts after the previous one and takes as input the grammar produced. It takes place in three steps (see **Figure 2**):

1) *Generation of data structures:* generate from the grammar data structures, algebras data structures and evaluation functions for each grammatical symbol (**Figure 2**-Step1) as described in section (see section 3.1).

2) *The choice of GUIs description language*(*s*) *and cultures to be located*: the produced GUIs are described in a (textual) language like FXML [9], UIML [10], etc. An implementation of algebra must therefore be provided for each of the target GUI (textual) description languages. Likewise, a *localization file* must be produced for each target culture (**Figure 2**-Step2).

3) *Multi-localization:* at the running time, the located GUI is obtained by invoking the evaluation function associated with the axiom of the grammar. An instance of algebra corresponding to the desired description language, and the

localization file related to the target culture are provided as effective parameters during this invocation (**Figure 2**-Step3).

External localization of software uses resource files that can be in the binary (.DLL, .EXE, etc.) or text format. In the following, we will call *localization file* a text resource file, containing in addition to strings to display in the GUI (linguistic aspect), other cultural information (color codes, disposition order of components, etc.) whose suitable use by the ASTs interpretation functions will provide a located GUI in which, not only linguistic concerns are addressed.

A localization file is structured in sections, and each of them contain information about a specific cultural concern addressed (language, color, components layout, etc.). Each section is structured as an associative list, ie. a list of pairs *(key, value)* where *key* is the identifier of a resource (its internationalized form), and *value* its concrete representation in the current target culture. We have on the listing 6 an example of a localization file.

3.3. Experimentation: Application to the Multi-Localization of the Checkers Game GUI

Recall that checkers game consists of a set of pieces that moves according to precise rules, on a checkerboard made up of 100 squares. It is played by at least 2 players and each player has 20 pieces of single color, which is distinct from the color of pieces of the other player. The goal of checkers game is to take or block the largest number of opposing pieces.

Considering that a checkers game can only be played by at least 2 players, it can be implemented and played in a network: it is therefore a kind of cooperative application. One can easily imagine a checkers game played by participants belonging to different cultural areas. As mentioned in the introduction (Section 1), the game's GUI can be multi-located in order to improve for each player his application's use comfort: each of the participants will then interact on a located GUI relating to his culture.

The rest of this subsection presents how this goal can be achieved by applying the multi-localization approach described in Section 3.2. Note that our goal here is not to build a fully functional checkers game, but, to present how our approach can easily derive a multi-located GUI for the latter. For a better understanding of the example, we presented on **Figure 3**, the orchestration diagram summarizing the interaction between different players during a game: any action of a player (moving a piece) on his GUI is raised at the AST level and the GUI of the other player is immediately updated accordingly; bidirectional *binding* [17] is implemented between each localized GUI and the AST.

3.3.1. Localizable Elements and AST of the Checkers Game

For illustration purposes, we have retained in addition to the GUI components related to the linguistic aspect, some others whose interpretation may vary from one culture to another. Finally, the localizable elements selected for our toy application are: strings (the linguistic aspect), colors, images of the pieces, layout

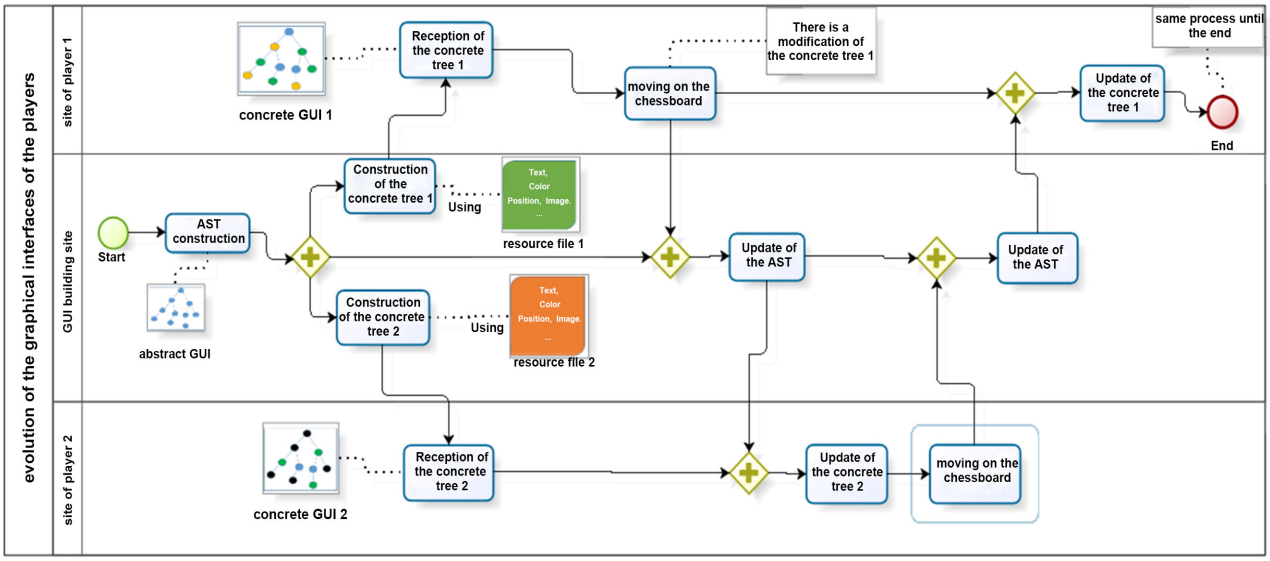

Figure 3. Orchestration diagram modeling the interaction between two checkers players.

components of the GUI and the chronometer displaying the remaining time at the end of which the player must have moved a piece: his color will have to change as soon as this time reaches a certain threshold.

From the abstraction of its elements and the structural relations that can be identified between them, we have the AST of **Figure 4** in which, the components *Label*, *Button*, *ImageView*, *TextField* (resp. *Vbox*, *Hbox*, *Gridpane*) are abstractions of basic (resp. structuring/layout) components that names are related to those that we generally have in the GUI's build libraries.

3.3.2. GUI's Grammar for the Checkers Game

From the abstract representation of checkers game (**Figure 4**) one can derive abstract grammar whose productions (an extract) are presented on the listing 1.

Listing 1: GUI's grammar for the checkers game

```
Ast -> Vbox
     | Hbox
Vbox -> BaseCompOrGroup*
Hbox -> BaseCompOrGroup*
BaseCompOrGroup -> BaseComp
     | Ast
BaseComp -> Label
     | Button | ImageView
     | GridPane
Label -> title.game
     |namePlayer1.game
     |namePlayer2.game
     | time.game | ...
Button -> btnBegin.game
     | btnFullScreen.game
     | btnQuit.game
ImageView -> piece1.game
     | piece2.game
```

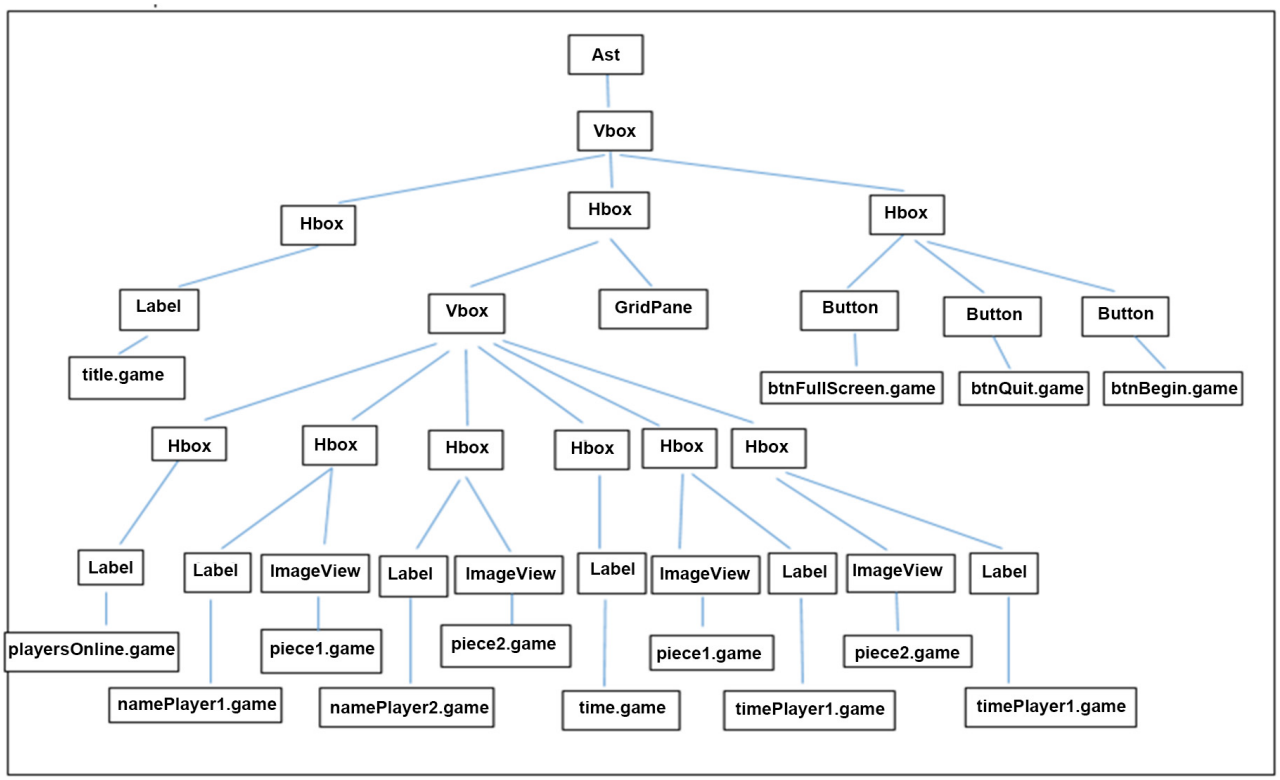

Figure 4. An example of a simplified AST representing the GUI of a checkers game.

In the listing 1, grammar symbols that do not appear on the left hand side of any production are assumed to be associated with ε-productions (not shown in this listing). In addition, in line 3, the notation "*" simply means that the grammar symbol *Vbox* is used to structure zero or more *BaseCompOrGroup* components.

3.3.3. Data Structures Derived from the Grammar of the Listing 1

An extract is given in the listing 2. Note that, a flat must be made to what have been said in Section 3.1 with regard to the data types to create. In fact, we will not create neither a Haskell data type nor algebra for grammatical symbols that we call *linguistic symbols*; these are *symbolic names* whose concrete value in the language of target culture is given in the localization file. In the listing 1, these are the grammatical symbols whose names have "*.game*" as extension; they are represented by the type *String* (their domain of interpretation) in the listing 2, lines 7 and 8.

Listing 2: Data structures derived from the grammar of the listing 1

```
1  --structuring components
2  data Ast = Vbox_ {vboxAst::Vbox}
3           | Hbox_ {hboxAst::Hbox} deriving Show
4  data Vbox = Vbox {childrenVbox::[BaseCompOrGroup]} deriving Show
5  ...
6  -- basic components (elementary)
7  data  Button = Button {buttonLabel::String} deriving Show
8  data Label = Label {label::String} deriving Show
```

Listing 3 presents the Haskell code of the algebraic structures associated with the data types of the listing 2. In this one, the types variables D_{Vbox}, D_{Hbox}, D_{Ast}, etc. represent the interpretation domains of the respective components, *Vbox*, *Hbox*, *Ast*, etc. Note that, the interpretation functions of derived algebraic structures, use data contained in the localization file. The latter is therefore provided as a parameter (listing 3, lines 7 and 10), and treated as an inherited attribute[16] associated with all the syntactic categories of the grammar.

During the scanning of a linearization of the AST, the checkers board is generated when *GridPane* is met. The syntactic categories *BaseCompOrgroup*, *BaseComp* and *Ast* are used only for factorisation purposes (reducing the number of productions), the interpretation functions of algebra that are associated do not use data contained in the localization file: they do not take it as a parameter as it is the case for other algebras.

Listing 3: Some algebras structures derived from the grammar of the listing 1

```
1  data AlgAst D_Vbox D_Hbox D_Ast = AlgAst{
2          interAstVbox_ :: D_Vbox -> D_Ast,
3          interAstHbox_ :: D_Hbox -> D_Ast,
4          interAlgAstVbox_ :: AlgVbox,
5          interAlgAstHbox_ :: AlgHbox}
6  data AlgVbox D_BaseCompOgroup D_Vbox= AlgVbox{
7          interVbox_ :: LocalFile -> [D_BaseCompOgroup] -> D_Vbox,
8          interAlgBCVbox_ :: AlgBaseCompOgroup }
9  data AlgGridPane D_GridPane = AlgGridPane{
10         interGridPane_ :: LocalFile ->  D_GridPane }
11 data AlgLabel D_Label = AlgLabel{
12         interLabel_ :: LocalFile -> NomSymbolique -> D_Label}
13 ...
```

As for the interpretation functions of algebras, the evaluation functions parameterized by algebras (listing 4) use data contained in the localization file; it is therefore provided as a parameter to these functions, and they must propagate it in the AST, by making them available at the level of each of their sons (listing 4, line 4).

Listing 4: The catamorphisms associated with the algebras of the listing 3

```
1  cataAst :: LocalFile -> AlgAst -> Ast -> D_Ast
2  cataAst locFil algAst = f where
3       f (Vbox_ vbox ) = interAstVbox_ algAst eval_vbox where
4          eval_vbox = cataVbox locFil (interAlgAstVbox_ algAst) vbox
5       f (Hbox_ hbox ) = interAstHbox_ algAst eval_hbox where
6          eval_hbox = cataHbox locFil (interAlgAstHbox_ algAst) hbox
7  cataVbox :: LocalFile -> AlgVbox -> Vbox-> D_Vbox
8  cataVbox lf algVb (Vbox bc0G) = interVbox_  algVb lf eval_BG where
9       eval_BG = map ((evalBC0grp lf) (interAlgBCVbox_ algVbox)) bc0G
10 cataLabel :: LocalFile -> AlgLabel -> Label-> D_Label
11 cataLabel lf algLabel (Label lab) = interLabel_  algLabel lf lab
```

3.3.4. An Implementation of Algebra for FXML

For the checkers game, we chose *FXML* as the GUI description language. All syntactic categories, except those qualified as *linguistic symbols*, therefore have

[16]In the attributed grammars, the inherited attributes allow among other things to propagate information from the root to the leaves of a tree.

FXML as interpretation domain. A particular implementation of the algebra of the listing 3, of which an extract is given in the listing 5, makes it possible to encode the located GUIs in *FXML*. It basically contains instructions for creating (insertion of *tags*) a well-formed and valid *FXML* file.

Listing 5: An algebra for the localized interpretation of AST in FXML

```
--interAST::AlgAst FXML FXML FXML
interAST = AlgAst  vbFXML hbFXML  gPFXML  intVbFXML intHbFXML
 where
  prelude_ = "<?xml version=\"1.0\" encoding=\"UTF-8\"?> \
     \<?import javafx.scene.control.*?>\
     \<?import javafx.scene.layout.*?>" ...
  begin_ = "< maxHeight=\"-Infinity\" ... \<children>"
  end_ = "   </children> ...
  vboxFXML = \x -> prelude_ ++ begin_  ++ x ++ end_
  hboxFXML = \x -> begin_  ++ x ++ end_
  gridPaneFXML = \x -> begin_  ++ x ++ end_
  intVbFXML = interVbox
  intHbFXML = interHbox
--interVbox ::  AlgVbox FXML FXML
interVbox = AlgVbox intVbFXML algBCVbFXML where
  intVbFXML = \lf xs -> begin_++(concat_xs lf xs)++end_
   where
     begin_ = "<VBox layoutX=\"15.0 ...
     end_   = "</children>\ ... \</VBox>"
     concat_xs res_ xs_= ...
   algBCVbFXML = interBaseCompOgroup
--interLabel :: AlgLabel FXML
interLabel =  AlgLabel  (\lf texteInt -> "<Label text=...
...
```

3.3.5. Localization Files

For illustrative purposes and without detracting from the generality, we have choose to multi-locate the GUI of the checkers game according to two cultures: "Western like culture" and "African like culture". Their respective localization files are given in the listings 6 and 7. They have been save in files named respectively *fichLocalWestern* and *fichLocalAfrican*, which are used in listing 8.

Listing 6: Localization file for a "Western like culture"

```
ressourceFR = Ressource {
  language =
      [("btnBegin.game", "Start"),
       ("btnQuit.game","Exit"),
       ("btnFullScreen.game","Full screen"),
       ("title.game", "Welcome to this Checkers Game!"),
       ("playersOnline.game", "players online"),
       ("namePlayer1.game","player 1 "),
       ("namePlayer2.game","player 2 "),
       ("time.game","Stopwatch"),
       ("timePlayer1.game","00:00:00"),
       ("timePlayer2.game","00:00:00")],
  colors = [("backgroundColor1", "green"),
            ("backgroundColor2", "red"),
            ("backgroundHBOX","pink")],
  dispositions = [("Hbox", Invert), ("Vbox", Direct)],
  images = [("piece1", "file:ressource/piece1.PNG"),
            ("piece2", "file:ressource/piece2.PNG")] }
```

Listing 7: Localization file for an "African like culture"

```
ressourceFR = Ressource {
  language =
   [("btnBegin.game", "kuanza"),
    ("btnQuit.game","kuondoka"),
    ("btnFullScreen.game","dirisha kubwa"),
   ("title.game", "Karibu kwenye mchezo huu wa kuchoma!"),
    ("playersOnline.game", "Wachezaji wa sasa"),
    ("namePlayer1.game","mchezaji a"),
    ("namePlayer2.game","mchezaji mbili"),
    ("time.game","wakati"),
    ("timePlayer1.game","00:00:00"),
    ("timePlayer2.game","00:00:00")],
  colors = [("backgroundColor1", "yellow"),
            ("backgroundColor2", "white"),
            ("backgroundHBOX","orange")],
  dispositions = [("Hbox", Direct), ("Vbox", Invert)],
  images = [("piece1", "file:ressource/piece3.PNG"),
            ("piece2", "file:ressource/piece4.PNG")] }
```

3.3.6. Located GUIs

The *FXML* codes of the located GUIs of the AST of the **Figure 4**, in each of the two target cultures, are obtained by invoking the function *cataAST* (listing 8) as described in the subsection 3.2.2. Once the *FXML* files are created, they can be viewed by using a *FXML* interpreter such as *sceneBuilder*[17] to have the outputs given by **Figure 5**.

Listing 8: FXML localizations of the AST of the **Figure 4**

```
cataAST fichLocalWestern algFXML  anAst                    1
cataAST fichLocalAfrican algFXML  anAst                    2
```

4. Conclusions

In this paper, we presented an external multi-localization approach of GUIs of interactive software. It can be used for the implementation of both collaborative and standalone applications, in order to adapt on the fly the end user's GUI to its own culture: The user experience is therefore improved.

The approach has been experimented with great satisfaction for the development of the multi-located GUI of checkers game, whose main lines of its implementation have been presented in this manuscript.

The proposed approach is based on the use of functional techniques for the interpretation of abstract structures by the means of algebras. The use of algebras gives to the programmer the latitude to offer with a lower intellectual investment the same localizations in various GUI description languages. Since the kind of localization explored in this paper uses localization files, the proposed

[17]JavaFX provides an application named Scene Builder. On integrating this application in IDE's such as Eclipse and NetBeans, users can access a drag and drop design interface, which is used to develop FXML applications.
https://docs.oracle.com/javase/8/scene-builder-2/get-started-tutorial/overview.htm#JSBGS164

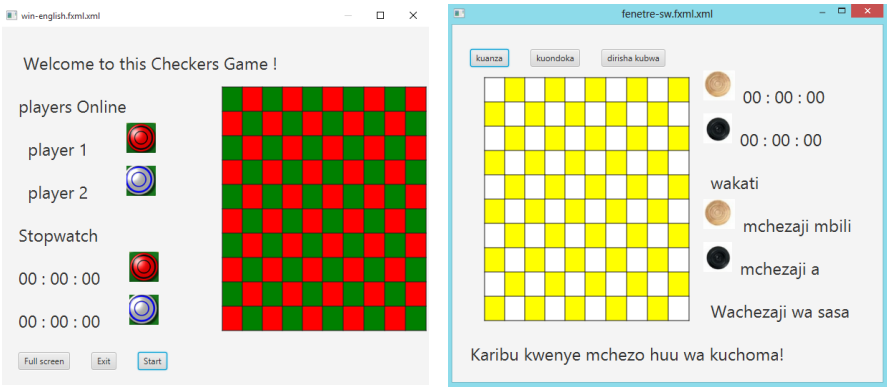

Figure 5. Multi-localization of the AST of **Figure 4**: (a) localization in a "Western like culture", (b) localization in an "African like culture".

approach may allow even a non-computer specialist to increment the localization degree (adding a new culture) of a software as soon as it has already been located for at least one culture: It is enough for that to adequately create a new localization file for the new target culture. Note however that, the creation of the new file will be easier if one can do it by the mean of a DSL (Domain Specific Language) [18]; this is one of the immediate perspectives of this work: that is, to investigate about an implementation of the proposed method as a software system, that generate code from high-level description, in order to not leave too much manual low-level writing of boilerplate code (like the one in the listing 5) to the software developer.

Acknowledgements

Authors warmly thank professor *Tayou Djameni Clémentin* for enriching discussions they had, during both laboratory seminars and the defense of Master's thesis of Mr. Tatou Freddy.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Sommet mondial des dirigeants locaux et régionaux - 3ème congrès mondial de cglu, mexico, 2010. https://www.uclg-cisdp.org/en/node/1078

[2] UNESCO (2002) Déclaration universelle de l'unesco sur la diversité culturelle. http://unesdoc.unesco.org/images/0012/001271/127162f.pdf

[3] Rauterberg, G.W.M. (2015) From Personal to Cultural Computing: How to Assess a Cultural Experience. Pabst Science Publisher, 13-21.

[4] Fraisse, A. (2010) Internal and in Context Localisation of Commercial and Free Software. Theses, Université de Grenoble.

[5] Esselink, B. (2000) A Practical Guide to Localization. Benjamins, John Publishing

Company. https://doi.org/10.1075/liwd.4

[6] Schäler, R. (2007) Localization. In: Baker, M. and Saldanha, G., Eds., *Encyclopedia of Translation Studies*, 2nd Edition, 157-161.

[7] Davie, A.J.T. (1992) An Introduction to Functional Programming Systems Using Haskell. Cambridge University Press, New York.

[8] Doets, K. and Van, J.E. (2004) The Haskell Road To Logic, Maths and Programming. King's College Publications.

[9] Schildt, H. (2015) Introducing JavaFX 8 Programming. 1st Edition, McGraw-Hill Education Group.

[10] Abrams, M., Phanouriou, C., Batongbacal, A.L., Williams, S.M. and Shuster, J.E. (1999) Uiml: An Appliance-Independent XML User Interface Language. *Computer Networks*, **31**, 1695-1708. https://doi.org/10.1016/S1389-1286(99)00044-4

[11] Ganneau, V., Calvary, G. and Demumieux, R. (2007) Métamodèle de règles d'adaptation pour la plasticité des interfaces homme-machine. *Proceedings of the 19th Conference on L'Interaction Homme-Machine*, *IHM* '07, Paris, 13-15 November 2007, 91-98. https://doi.org/10.1145/1541436.1541454

[12] Gabillon, Y. (2011) Composition d'interfaces homme-machine par planification automatique. In Interface Homme-Machine [cs.HC]. Université de Grenoble, Grenoble.

[13] Reina, L.A., Robles, G. and González-Barahona, J.M. (2013) A Preliminary Analysis of Localization in Free Software: How Translations Are Performed. In: Petrinja, E., Succi, G., Ioini, N. and Sillitti, A., Eds., *9th Open Source Software* (*OSS*), *volume AICT*-404 *of Open Source Software*: *Quality Verification*, Part 1: Full Papers—Practices and Methods, Springer, Koper-Capodistria, 153-167.

[14] Dirk, S.K. (2005) Internationalisierung und lokalisierung von software. In: *Reineke and Schmitz Klaus*, Einführung in die Softwarelokalisierung, Gunter NarrVerlag, Tübingen, 11-26.

[15] Backhouse, K. (2002) A Functional Semantics of Attribute Grammars. *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, *TACAS* '02, Springer-Verlag, Berlin, 142-157. https://doi.org/10.1007/3-540-46002-0_11

[16] Fokkinga, M., Jeuring, J., Meertens, L. and Meijer, E. (1991) A Translation from Attribute Grammars to Catamorphisms. *The Squiggolist*, **2**, 20-26.

[17] Hu, Z.J., Mu, S.-C. and Takeichi, M. (2008) A Programmable Editor for Developing Structured Documents Based on Bidirectional Transformations. *Higher-Order and Symbolic Computation*, **21**, 89-118. https://doi.org/10.1007/s10990-008-9025-5

[18] Van Deursen, A., Klint, P. and Visser, J. (2000) Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, **35**, 26-36. https://doi.org/10.1145/352029.352035