

CCComponents: a JavaScript component library for abstracting web user interface development

Spano Lucio Davide, Campanile Giorgia, Cella Francesca

University of Cagliari, Italy

Mathematics and Computer Science Department

Abstract: Multimodal interfaces have been recently used in a lot of contexts and they are becoming familiar in daily life to end users. Unfortunately, is not easy and trivial to program them, especially for people that are not expert in using programming languages. CCComponents is a new JavaScript library that allows inexperienced users to create multimodal applications based on the concept of abstract components. Each element in the application is made of components of different types. In particular, CCComponents allows creating multimodal web applications in which elements can be interacted also through speech.

Keywords: components, multimodal, vocal, Artyom, JavaScript

1 INTRODUCTION

In recent years, the modes of Human-Computer interactions have been changing. In fact, traditional input devices, such as keyboard and mouse, are gradually being integrated or replaced with new interaction metaphors, such as touch, gestures and speech. Consequently, user interfaces are being re-designed to accommodate user inputs from multiple input modes. A multimodal interaction system provides a more natural and instinctive form of engagement with a computing device than traditional input based devices (such as keyboard or mouse). Multimodal interfaces then allow you to choose one or more additional modes of interaction for the same elements. However, these additional modes of interaction may have advantages or disadvantages. For example, inputs might not be recognized from the system, or the user doesn't apply the rules of the interaction system in a correct manner. Ideally the user should enjoy the experience by interacting in a convenient and user friendly manner. It is not uncommon to find multiple user interfaces on a computer system or in a web application each corresponding to a distinct interaction modality (speech-based and gesture-based interactions) because of the need to adapt systems to these new emerging technologies. A lot of researches and tools have been presented in recent years with the aim of adapting applications to these modes of interactions. Thanks to these tools, a programmer can adapt his applications so that they can be interacted with other different modes beyond the classic ones. Unfortunately, is not easy and trivial to program multimodal interfaces if the person is not expert in using programming languages; he should know a lot of concepts and programming fundamentals. CCComponents is a new JavaScript library that

allows inexperienced users to create multimodal applications based on an abstract components model. Practically, this library allows the user to build and customise widgets using components as abstraction: on the practical side, each widget declared by the user through these components, is validated and rendered using common Web technologies and mechanisms such as HTML, CSS and JavaScript. The components provided by this library are JavaScript classes that use WebComponents. We have three types of abstract components: semantic, graphic and vocal, and personalized widgets are created by coupling two or more abstract components to each other. CCComponents allows creating multimodal web applications in which final widget can be interacted also through speech, due to the use of vocal components.

In the next sections we will explain how our library is designed and how it can be used. All technical details will be presented since section 3.

2 RELATED WORK

In 2009, Paterno' et Al. [10] discuss how a novel model-based user-interface description language (*MARIA, Modelbased Language for Interactive Applications*) can provide useful support both at design and runtime for interactive applications. These interactive applications work in environments which are characterized by a wide set of interactive devices and exploit a number of functionalities developed beforehand and encapsulated in Web services. Web service annotations can also be exploited for providing hints for user interface development at design time. This presented novel model-based language for user interfaces draws on previous experiences in this area and

inherits the modular approach of TERESA XML [8].

They show how such language can be exploited in order to design and develop multidevice user interfaces, which support access to multiple pre-existing services, and how such an approach can be exploited to support migratory user interfaces, still for applications based on the use of Web services. In MARIA XML at runtime the language is exploited to support dynamic generation of user interfaces adapted to the different devices at hand during the user interface migration process. So with one language for the abstract description and then a number of platform-dependent languages that refine the abstract one depending on the interaction resources considered. Our approach is partly based on this concept of abstraction: so we have one abstract description for the library components, and two different kinds of languages and representation for the chosen interaction (graphic and vocal). With the MARIA XML approach also a definition of a data model is needed for representing the data handled by the user interface and in order to allow more control over the admissible operations that can be performed on the various interactors. So, in our approach, an interaction will be designed depending on the chosen actions and data types. The introduction of an event model allows for specifying how the user interface responds to events triggered by the user. In our work we follow this approach, so we allow the user to evaluate the possibility of insert audio and graphic feedbacks from the user interface.

In 2011, Paterno' et Al. [11] describe an original solution to make Web pages more suitable for vocal browsing by analyzing and modifying their logical structure. The proposed solution exploits intermediate logical descriptions in the MARIA framework [10] that are automatically created by reverse engineering techniques. They generate VoiceXML vocal applications, that are semantically equivalent to the originals but accessible through menu-based vocal interaction. The adaptation engine aims to identify the main logical structure of the Web page components and remove the aspects specific to the graphical modality. Then, a vocal implementation is generated to support browsing, which begins by the user's selecting from the main components.

The implementation is based on a pipeline of XSL Transformations. In order to improve the flexibility of this tool, the authors also provide designers and developers with the possibility of customizing the adaptation rules, by specifying the values of some parameters in order to better control its results. Our approach also allow users to customize some of their application parameters, but we force them using other default ones.

In year 2019 Raffailac et Al. [12] present *Polyphony*, an experimental toolkit. They introduce a new Graphical User Interface (GUI) and Interaction framework based

on the *Entity-Component-System model (ECS)*. In this model, interactive elements (Entities) are characterized only by their data (Components). Behaviors are managed by continuously running processes (Systems) which select Entities by the Components they possess. This model facilitates the handling of behaviors and promotes their reuse. It provides developers with a simple yet powerful composition pattern to build new interactive Elements with Components. It materializes interaction devices as Entities and interaction techniques as a sequence of Systems operating on them.

In *Polyphony*, the widgets (buttons, text boxes, etc.), but also the input and output devices (mouse, keyboard, screen, etc.), are Entities. Their properties are Components that can be acquired and removed dynamically. Systems are functions that run in sequence at each input/output event, each one iterating on Entities and applying a type of interactive behaviour such as border drawing, drag drop, text entry, etc. ECS (sometimes also called CES) is an architectural pattern designed to structure code and data in a program. The main elements defining the model are thus: Entities are unique identifiers (often simple integers) representing the elements of the program. They are similar to objects in Object-Oriented Programming, but have no data or methods. Components represent the program data (such as position or color), and are dynamically associated with the Entities. Depending on the interpretations of ECS, a Component alone can mean the type of data that can be bound to Entities, or one instance bound to an Entity. Systems run continuously in a fixed order, each representing a behavior of the program. Entities do not register with Systems, but acquire the necessary Components to be "seen" by them. Our library has some similar concepts of *Polyphony* system.

3 CCCOMPONENTS LIBRARY

In this section we are going to present the CCComponents library and its implementation. We will talk about the components architecture and syntax, in particular how each component works and how components are coordinated.

As previously mentioned, this library allows the user to build and customise widgets using components as abstraction: on the practical side, each widget declared by the user through these components, is validated and rendered using common Web technologies and mechanisms such as HTML, CSS and JavaScript.

The following is an example that shows how a widget can be created using this library:

```
<widget-container>
  <widget-semantic
    semantics="data-acquisition"
    type="text"
    element="textfield">
  </widget-semantic>
```

```

<widget-graphic
  background="aliceblue">
</widget-graphic>
<widget-vocal
  event="change"
  keyword="type">
</widget-vocal>
</widget-container>

```

In this case, the rendered widget, which is shown in 1, will be a text field having "aliceblue" background. The user can interact with it for typing text by saying "**type** [*] in **textfield**".



Figure 1: Widget example

3.1 Abstract Components

The components provided by this library are JavaScript classes that use WebComponents. The rendered widget takes advantage of the encapsulation by using the **shadow DOM**: in this way, each widget container cannot interfere with other.

Essentially, this library consists of three kind of abstract components:

- Semantic component: this component defines which HTML element is to be rendered and, for this reason, it is required when building the widget;
- Graphic component: this component defines the appearance of the widget;
- Vocal component: this component defines how the user can interact with the widget by voice.

An overview of these components can be seen in ???. These components are only responsible for validating their internal attributes. In defining the widget, they are wrapped inside a container element provided by the library: this container is the Widget container and aims to coordinate children, validate their composition and render the widget on the base of the user declaration.

3.2 Semantic component

Semantic components represent the meaning that is at the base of the final widget. This component is compulsory for each widget.

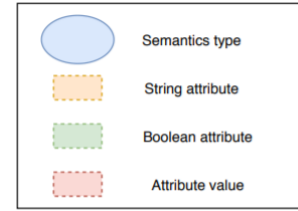


Figure 2: Legend for Abstract Components attributes

This library provides six types of semantics:

- Single choice: this kind of semantics represents a single choice, i.e. a **button**. The allowed attributes for this semantics can be viewed in 3;

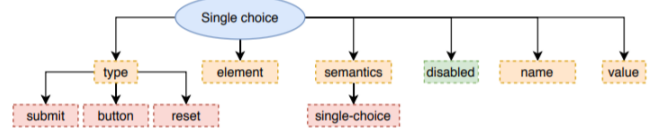


Figure 3: Single Choice

- Selection choice: this kind of semantics represents a choice that can have more than one option, i.e. a **select**, a **multiselect**, a **radio button** or a **checkbox**. This component provides the **multiple** attribute to discriminate between these types of selection choice: multiple refers to multiselect or checkbox. There is another kind of discrimination that is provided by **expanded** attribute and refers to checkbox and radio, but this is related to the graphics and will be discussed more in detail below. The allowed attributes for this semantics can be viewed in 4;

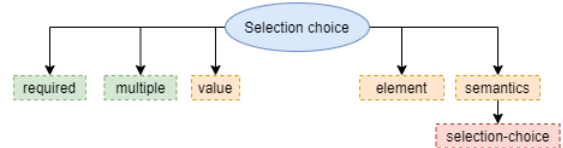


Figure 4: Selection Choice

- Navigation: this kind of semantics represents the navigation, i.e. a **link**. The allowed attributes for this semantics can be seen in 5;

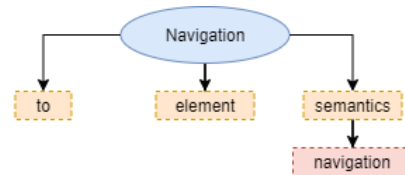


Figure 5: Navigation

- **Label**: this kind of semantics represents text, i.e. a **label** or simply **inner text**. The allowed attributes for this semantics can be seen in 6;

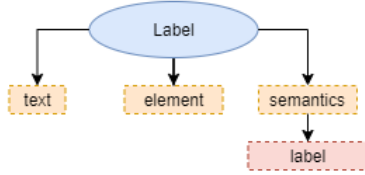


Figure 6: Label

- **Option**: this kind of semantics represents an available option for a selection choice widget. The allowed attributes for this semantics can be seen in 7;

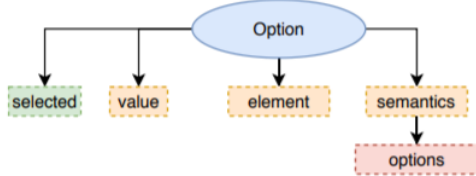


Figure 7: Option

- **Data acquisition**: this kind of semantics represents the data input, i.e. each type of **input**, except for checkbox and radio types that are treated as selection choices, and the **textarea**. This semantics allows the user to specify which type of data can be entered using the **type** attribute, while the discrimination between textarea and input can be performed using the expanded attribute for the graphic component. The allowed attributes for this semantics can be seen in 8.

These types of semantics, from a code point of view, are JavaScript prototypes: each of them maps all the attributes declared for a specific semantic component into an HTML element. The prototype that maps the component into the correct HTML element is chosen on the base of the semantics attribute.

3.2.1 Semantics attribute

These kind of semantics can be specified through the required string **semantics** attribute.

3.2.2 Element attribute

Each kind of semantics accepts the string **element** attribute, which is used for identifying and relating semantic components. In particular, in the case of selection choice, single choice, data acquisition and navigation, it is used as **id** attribute; in the case of label semantics, it is used

as **for** attribute: in the widget definition, the element attribute in label is the same as the element attribute in the semantics (single choice, selection choice, data acquisition or navigation) which the label refers to; in the case of option semantics, it is used for finding the correct selection choice in which append the option.

As far as selection choice and data acquisition are concerned, this mechanism is the same as the default HTML behaviour: this means that a label HTML element will be created and bound through the **for** attribute having a value equal to the **id**; instead, for navigation and single choice, the label is treated as inner text.

The following is an example of single choice and label semantics combination:

```

<widget-container>
  <widget-semantic
    semantic="single-choice"
    element="new_button">
  </widget-semantic>
  <widget-semantic
    semantics="label"
    element="new_button">
  </widget-semantic>
</widget-container>
  
```

3.2.3 Text attribute

The **text** attribute is used only in label semantics for defining the text contained in the label.

3.2.4 To attribute

The **to** attribute is used only in navigation semantics for defining the path for navigating.

The previous were custom attributes, the following are default HTML attributes.

3.2.5 Type attribute

The **type** attribute follows the default HTML behaviour. It is used by single choice and data acquisition semantics: as far as the single choice is concerned, the type attribute can be one of **submit**, **reset**, **button** because this kind of semantics can only be mapped to the button element; in the case of data acquisition, this attribute can be one of **text**, **number**, **email** and every value supported by default in input element.

If none is specified, the default value for single choice is **submit** while the default value for data acquisition is **text**.

3.2.6 Other default attributes

In addition, this component accepts these default attributes:

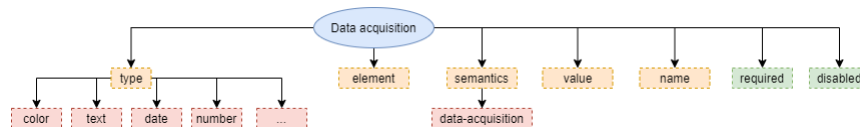


Figure 8: Data acquisition

- **disabled**, boolean;
- **required**, boolean;
- **name**, string;
- **readonly**, boolean;
- **value**, string;
- **checked**, boolean;
- **multiple**, boolean;
- **selected**, boolean.

These attributes are treated and validated using the default HTML/JavaScript behaviour.

When the container renders and validates its children, the generated HTML elements, as said before, are appended to the shadow DOM belonging to the widget container.

3.3 Graphic component

The graphic component provides a series of attributes, many of them are CSS shorthands. The allowed attributes are:

- **background**, string, CSS shorthand;
- **color**, string, CSS shorthand;
- **font**, string, CSS shorthand;
- **border**, string, CSS shorthand;
- **expanded**, boolean. It is used in combination with selection choice for discriminating between checkbox/radio, which are expanded, and select/multiselect; it is also used in combination with text type data acquisition for discriminating between textarea, which is expanded, and input of type text;
- **size**, string. It can be one of **SMALL**, **MEDIUM**, **LARGE**. It adds a little padding to the element;
- **target**, string. This attribute accept a semantic type and it is used to specify the semantic component that will be styled during the container validation and rendering phase. If none is specified, the style is applied to each (semantic) component of the widget;

- **shadow**, string. It accepts a value that is compliant with the **text-shadow** and **box-shadow**: during the container validation and rendering phase, the correct CSS property is chosen on the base of the target semantics.

Also in this case, the CSS shorthand value is validated by default and follows the usual behaviour.

As previously mentioned, the CSS is encapsulated and so related to only one widget.

3.3.1 Problems and workarounds

An HTML element that can have a state like **hover** or **disabled**, e.g. a button, when customised loses the visual feedback when its status changes. For this reason, this library provides a series of visual feedback for compensating this situation.

For example, a disabled button having a custom background is rendered changing its background to "gainsboro" and its color to "gray".

As far as the hover status is concerned, the cursor is modified in order to be a pointer; other solutions to the hover problem have been evaluated, for example the possibility to use a semi-transparent gradient background, but if the base background is defined as a gradient, the result is not very appealing.

Disabled button

Figure 9: Disabled visual feedback for customised widget

3.4 Vocal component

The vocal component provides only two attributes:

- **event**, string, required;
- **keyword**, string, required.

3.4.1 Event

The event attribute accepts one of the following events:

- **click**, pre-existing event;
- **focus**, pre-existing event;
- **blur**, pre-existing event;

- **input**, pre-existing event;
- **change**, pre-existing event;
- **open**, it is a custom event that allows to listen all the possible options for a compact selection choice semantics using the vocal library;
- **clean**, it is a custom event that allows to remove the value for a data acquisition semantics.

When the command is recognised, click, focus and blur events are triggered on the widget using the corresponding JavaScript function, while input and change are executed by changing the current widget value with the specified value and then by emitting the event. All of these events follow the default behaviour.

The case of clean event is similar to input and change events, but in this case the value assigned to the widget is an empty string.

As far as open event is concerned, when the command is recognised, all the available option will be read using the vocal library.

When using a vocal component, the widget (i.e. the semantic component) must have the element attribute defined in order to identify which is the widget for interacting.

3.4.2 Feedback

This library offers a built-in visual feedback system related to the vocal commands that is positioned on the bottom left of the interface.

When the library is ready to listen for commands, the feedback shows the "Ready!" message; when the library is listening for a command, the feedback shows the "Listening..." message with a red REC icon; when a command is recognised, the feedback shows a "Command matched!" message having a green tick, while when a command is not recognised, the feedback shows a "Command not matched." message having a red cross. When there is an error, i.e. vocal commands are not available, the feedback shows the message "Voice commands not available." with a red exclamation mark.

This feature is available thanks to the interception of the events emitted from the vocal library.

✓ Command matched!

Figure 10: Visual feedback for vocal command matched

3.5 Widget container

The widget container is responsible for validating and coordinating by rendering the composition of its children. It hasn't any custom attribute.

3.5.1 Validation

The first step of the validation performed by the container is to check if the semantic children components combination is correct. The widget container allows exactly one semantic component, except for the case where there is the label semantic component: in this case, the widget container allows exactly two semantic children.

The second step consists of validating the expanded graphic attribute, which is available only when the semantics are one of selection choice or data acquisition.

The third step of the validation is to check which are the possible semantic targets (semantic children components) and then compare them with semantic targets defined in graphic children components in order to ensure that the customisation is applied to a semantics that exists in the container.

The last step concerns the vocal component: in particular, as far as there are two custom events, in this step the container checks if the open custom event is bound to a compact selection choice semantics and if the clean custom event is bound to a data acquisition semantics.

As previously stated, the rest of the validation for CSS and default attributes, is made by default.

3.5.2 Semantics prototypes

The following are semantics prototypes that accepts a series of semantic and graphic attributes in order to map the declaration performed to the user through the library components into the effective widget, i.e. the HTML element:

- **Base element**: accepts a tag name, which is a common attribute for every prototype and the disabled attribute;
- **Single choice**: uses Base element, accepts the tag (the default is button), type (the default is submit), text, disabled, value, name, element attributes;
- **Selection choice**: uses Base element, accepts expanded, multiple, element, value, checked, disabled, required, name attributes. On the base of the input, it renders the correct widget among radio button, checkbox, select and multiselect;
- **Navigation**: uses Base element, accepts to, text, element attributes;
- **Data acquisition**: uses Base element, accepts type (the default is text), expanded, element, disabled, required, readonly, value, name attributes. On the base of the input, it renders the correct widget among textarea and various kinds of input;
- **Label**: uses Base element, accepts text and element attributes;

- Option: uses Base element, accepts value, text, disabled, selected attributes.

3.5.3 Rendering

When the validation is done and is successful, the container processes the semantic attributes contained in its children.

Based on the number of the semantics, i.e. if there is the label semantics, the component creates the correct label for the widget: as previously reported, the label can be a simple inner text in the case of navigation, option and single choice and a label HTML element in the other cases.

Then the container processes the graphic attributes and binds them to the semantic ones previously processed.

At this point, it creates an HTML element using semantic prototypes on the base of the combination of semantics and graphic attributes previously processed.

In the end, it binds the vocal commands to the widget.

3.6 Used technologies

We have designed this library with the JavaScript language. Components are built using the Web Components API [5], which are a set of web platform APIs that allow you to create new custom, reusable, encapsulated HTML tags to use in web pages and web apps. Custom components and widgets build on the Web Component standards, will work across modern browsers, and can be used with any JavaScript library or framework that works with HTML. Web components are based on existing web standards. Features to support web components are currently being added to the HTML and DOM specs, letting web developers easily extend HTML with new elements with encapsulated styling and custom behavior. In creating the effective widget after the validation, we took advantage of the concept of encapsulation for our components thorough the shadow DOM [3], which is a mechanism that allows to create a hidden separate DOM inside an element in a web UI.

During the development phase, we have used Webpack [7], which is a static module bundler for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph which maps every module your project needs and generates one or more bundles. We decided to use WebPack in order to make the inclusion of our library easier, since we use it also for the building phase: once the library is built, it can be added to a web page using a simple `<script>` tag.

As far as the vocal library is concerned, we have used *Artyom.js* [9], which is a JavaScript voice library. It is a robust but easy to use wrappers of the *speechSynthesis* [4] and *webkitSpeechRecognition* [6] APIs. Unfortunately, since it wraps *webkitSpeechRecognition* that is only available in a few browsers, our library is compatible only with browsers that supports this functionality.

Finally, the code contained in the library is documented

using JSDoc [2], which is a markup language used to annotate JavaScript source code files. Using comments containing JSDoc, programmers can add documentation describing the application programming interface of the code they're creating. Our project documentation is produced through *JSDoc*.

4 USING CCCOMPONENTS LIBRARY

In this section an example of using CComponents library will be shown.

4.1 Vocal commands building

As previously discussed, each container having a vocal component, must have the element attribute specified in semantics in order to correctly identify the widget to trigger when the command is recognised.

Vocal components have effect for each type of semantics, except for label and option.

This is the command structure:

- single-choice, navigation: **[keyword] [element]**
for example *[navigate to] [home]*, or *[press] [red button]*
- data-acquisition: **[keyword] * in [element]**
for example *[write] hello in [text field]*, or *[clean content] in [text field]*
- selection-choice: **[keyword] * from [element]**
for example *[choose] option 1 from [blue select]*, or *[read options] from [blue select]*

Using this kind of approach, element identifiers and values cannot contain characters like `"_"` or `"-"`;

4.2 Components structure example

The complete example can be seen [here](#) [1].

4.2.1 Single choice semantics

```
<widget-container>
  <widget-semantic
    value=" btn"
    semantics=" single-choice"
    type=" button"
    element=" red_button">
  </widget-semantic>
  <widget-semantic
    semantics=" label"
    text=" Large_button">
  </widget-semantic>
  <widget-vocal
    event=" click"
    keyword=" press">
  </widget-vocal>
```



```

<widget-graphic
  color="white"
  background="#d32f2f"
  size="large"
  border="none">
</widget-graphic>
<widget-graphic
  font="1rem Helvetica, sans-serif">
</widget-graphic>
</widget-container>

```

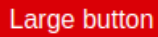


Figure 11: This button has a label and a single-choice semantics, a graphic component with red background, no border, large size and Helvetica white font, a vocal component that allows you to press the button by saying **press red button**.

4.2.2 Selection choice semantics

In this example there is a compact single selection choice having two options: for creating this widget, it is necessary to nest the container having the options into the main container.

```

<widget-container>
  <widget-semantic
    semantics="label"
    text="Select compact"
    element="select">
  </widget-semantic>
  <widget-semantic
    semantics="selection-choice"
    element="select">
  </widget-semantic>
  <widget-container>
    <widget-semantic
      semantics="label"
      text="Option 1">
    </widget-semantic>
    <widget-semantic
      semantics="option"
      value="option_1"
      element="select">
    </widget-semantic>
  </widget-container>
  <widget-container>
    <widget-semantic
      semantics="label"
      text="Option 2">
    </widget-semantic>
    <widget-semantic
      semantics="option"
      value="option_2"
      element="select">
    </widget-semantic>
  </widget-container>
  <widget-semantic
    semantics="selected"
    element="selected"
    value="option_1">
  </widget-semantic>
</widget-container>

```

```

  value="option_2"
  element="select">
</widget-semantic>
</widget-container>
<widget-vocal
  event="change"
  keyword="choose">
</widget-vocal>
<widget-vocal
  event="open"
  keyword="read_options">
</widget-vocal>
<widget-graphic
  color="#1976d2"
  target="label">
</widget-graphic>
<widget-graphic
  border="none"
  color="white"
  size="large"
  target="selection-choice"
  background="#64b5f6">
</widget-graphic>
</widget-container>

```

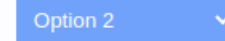



Figure 12: Large compact single selection choice semantics with label, 2 options, blue background and label and no border. The value can be changed by saying **choose option 1 from select**. The option semantics uses default selected attribute.

4.2.3 Data acquisition semantics

```

<widget-container>
  <widget-semantic
    semantics="label"
    text="Textarea"
    element="textarea">
  </widget-semantic>
  <widget-semantic
    value="Hello"
    readonly
    semantics="data-acquisition"
    type="text"
    element="textarea">
  </widget-semantic>
  <widget-graphic
    expanded
    size="large"
    target="data-acquisition">
  </widget-graphic>
</widget-container>

```


</widget-container>

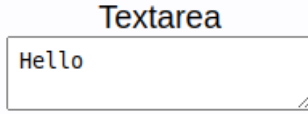


Figure 13: Large read only textarea having data acquisition and label semantics.

4.2.4 Navigation semantics

```
<widget-container>
  <widget-semantic
    semantics="navigation"
    to="/home"
    element="home">
  </widget-semantic>
  <widget-semantic
    semantics="label"
    text="This link goes to the home">
  </widget-semantic>
  <widget-graphic
    color="#115293"
    size="large">
  </widget-graphic>
  <widget-graphic
    shadow="2px 1px 5px #64b5f6">
  </widget-graphic>
  <widget-vocal
    event="click"
    keyword="go to">
  </widget-vocal>
</widget-container>
```

[This link goes to the home](#)

Figure 14: This widget has a navigation and label semantics, text shadow and colored text. The navigation can be triggered by saying **go to home**.

5 CONCLUSION

We have presented CComponents, a new JavaScript library that allows inexperienced users to create multimodal applications based on an abstract components model. This library allows the user to build and customise widgets using components as abstraction, and these widgets can be interacted with classic input modes or through speech. Similar to what happens when using applications with voice recognition, there could be some problems in using an application created with our library. For example, the vocal command might not be recognized.

We are aware that the use of our library could be quite limiting for users who would like to fully customize their application; this is due to our default attribute settings, done in order to make the library use easier. It is assumed that an experienced user who wishes to fully customize his web interface does not need to use a library like this.

References

- [1] Demo using ccomponents.
- [2] Jsdoc. <https://jsdoc.app/>.
- [3] Shadow dom. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_sh.
- [4] Speechsynthesis. <https://developer.mozilla.org/en-US/docs/Web/API/SpeechSynthesis>.
- [5] Web components. <https://www.webcomponents.org/>.
- [6] webkitspeechrecognition. <https://developers.google.com/web/updates/2013/01/Voice-Driven-Web-Apps-Introduction-to-the-Web-Speech>.
- [7] Webpack. <https://webpack.js.org/>.
- [8] Silvia Berti, Francesco Correani, Fabio Paterno, and Carmen Santoro. The teresa xml language for the description of interactive systems at multiple abstraction levels. In *Proceedings workshop on developing user interfaces with XML: advances on user interface description languages*, pages 103–110, 2004.
- [9] Carlos Delgado. Artyom.js. <https://sdkcarlos.github.io/sites/artyom.html>.
- [10] Fabio Paterno, Carmen Santoro, and Lucio Davide Spano. Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(4):1–30, 2009.
- [11] Fabio Paternò and Christian Sisti. Model-based customizable adaptation of web applications for vocal browsing. In *Proceedings of the 29th ACM international conference on Design of communication*, pages 83–90, 2011.
- [12] Thibault Raffailac and Stéphane Huot. Polyphony: Programming interfaces and interactions with the entity-component-system model. *Proc. ACM Hum.-Comput. Interact.*, 3(EICS), June 2019.