Towards a User Interface Description Language based on bigraphs

Nicolas Nalpon, Cyril Allignol[0000−0001−7528−5512], and Celia Picard[0000−0002−8715−4365]

ENAC, Toulouse University, Toulouse, France
{name.surname}@enac.fr

Abstract. User interface description languages (UIDL) are high-level languages allowing to model user interfaces (UI). Their purpose is to ease the design of UIs. They are widely used, including to develop critical interactive systems. Nevertheless, the problem of verifying systems developed with UIDLs is barely addressed in the literature. The first step is to provide a formal semantics using an appropriate theory. We claim that the bigraphs theory is a good candidate theory. In this short paper, presenting a work in progress, we introduce the common features of UIDLs and show how bigraphs could be used to define UIDLs semantics and help with UI verification.

Keywords: User Interface Description Language · Graphical User Interfaces · formalisation · bigraphs.

1 Introduction

User interface description languages (UIDL) [10] are high-level languages allowing to model user interfaces (UI). Their purpose is to make the design of the UIs independent and avoid all the difficulties related to their programming such as spaghetti code due to callbacks [17] and the maintenance of an event loop.

Nowadays, UIDLs are widely used to design UIs for interactive systems, including critical systems [13, 19]. This emphasizes the need of formal verification for the UIDLs [19], both on the language and program aspects e.g. formal semantics, verified compiler and properties verification on written programs. We focus on the UIDLs specialised on graphical user interfaces (GUI). These UIDLs allow to describe a scene graph and how it evolves over time according to user interactions. In safety-critical systems, the specifications of the GUIs expressed through the UIDLs need to be consistent during all the program lifetime. The purpose of this article is to present a new idea of formal foundations for UIDLs with the same objective. Generally, the formal aspect of UIDLs is little studied. This includes their semantics, except for a few works such as [9]. So, one of the first questions to tackle is the formalisation of their semantics.

The semantics of UIDLs specialised on GUIs description present two common features: the representation of the scene graph and the representation of the control flow. These features relate to the spatial and non-spatial aspects of the

GUI and we need a suitable formalism to represent these aspects of the semantics. Bigraphs [16], a diagrammatic framework allowing to represent agents, their locality, their interactions and how they evolve over time, have the potential to represent appropriately these features and thus verify GUIs.

This short article, describing a work in progress, aims to motivate the use of the bigraphs theory to formalise and verify UIDLs semantics and UIs.

We first detail the definition and common mechanisms of the UIDLs in Section 2, before briefly presenting the bigraphs theory in Section 3. Then, in Section 4, we show on a QML example, how we can model the GUI mechanisms common to all UIDLs using the bigraphs theory. Furthermore, Section 5 gives a glimpse of how verification can be done on a bigraphs model. Finally, in Section 6, we give an overview of the existing work on GUIs verification and discuss the concrete benefit to use bigraphs to verify GUIs.

2 UIDLs

UIDLs are programming languages generally used to design UIs. Programming GUIs has been a tedious task for a long time: they had to be described by sequential code, and callbacks were used to handle events. These practices were criticised [14, 17], in particular because of the causal relationships between the different program entities that were not clearly represented. UIDLs are the solution to this problem. Firstly, they make a clear distinction between the design of the UIs and the rest of the application to be developed. Secondly, they only focus on the design of the UIs and propose a suitable syntax to express graphical entities location and interactions (as QML signals and slots [8]).

UIDLs often express GUIs through a tree structure and interactions through special operators, variables affectation or scripting code. There are many UIDLs, but for illustration sake, here are some details about three of them:two popular UIDLs widely used in large projects, FXML and QML, and another one, smala, used to develop critical systems. A more detailed comparison of UIDLs based on XML syntax can be found in this survey [10]. FXML [5] based on Java, describes the graph scene through an XML syntax and represents interactions by variable affectation (bindings) or scripting code (event handler). QML[7], a UIDL based on C++ and Python, describes the interactions in the same way as FXML but JSON is favored to describe the graph scene. Lastly Smala [13], based on C++, uses a bracket syntax along with the graphical entities definition order to describe the scene graph and special operators to describe the interactions.

Despite the diversity of UIDLs, two features [20] are shared by all their semantics: 1) the representation of the graph scene, giving explicit information on the location of each graphical entity; 2) the interactions present in the GUI. To implement these features, UIDLs always provide the following kind of mechanisms: 1) an encapsulation mechanism, related to the scene graph criteria and allowing to create a hierarchy among the graphical entities; 2) event handlers and bindings (stream), allowing to handle GUIs interaction aspect. The hierarchy induced by the encapsulation and the entities dependencies induced by the

event handlers and bindings can be respectively represented by a forest and a graph. This double graph structure is very similar to Milner's bigraphs one.

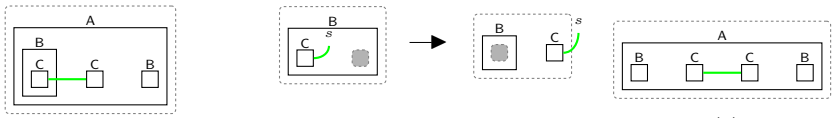
3 Bigraphs

Bigraphs [16] are a diagrammatic framework introduced by Robin Milner allowing to model systems that evolve over time and space. They consist of a set of entites (nodes) shared by two orthogonal graph structures. The place graph, which is a forest, represents the spatial aspect (by mean of nesting) of a system and the link graph, which is a hypergraph, represents the interactions (by means of hyperedges) present in the system.

3.1 Structural aspect and rewriting rules

Bigraphs, illustrated by Fig. 1a and Fig. 1c, are composed of entities, to which we can associate a control (similar to a type), that in turn associates an arity (number of links we can connect) to the entity. For example, the control C in Fig. 1a has arity one and the controls A and B have arity zero. An entity has a fixed arity. Entities can be nested into other entities (place graph) and can be linked (through green hyperedges) to other entities (link graph). An entity that cannot contain another entity is called atomic and is a leaf in the place graph.

Special structures allow bigraphs to be built and decomposed compositionally as regions (dashed rectangle) and sites (grey rectangle). Regions are the root container of a bigraph. Sites abstract away a bigraph part. Sites contain an unspecified bigraph, even possibly the empty bigraph, contained in a region. So, it is possible to build a bigger bigraph by placing regions into sites. In the same way, the links allow composition by using names. For instance in Fig. 1b, the link tagged s can be connected to another link tagged s from another bigraph. Two types of link can be found in bigraphs: the open links, used to compose bigraphs, and the close one as in Fig. 1a



(a) (b) (c)

Fig. 1: (a) inital bigraph, (b) reaction rule and (c) bigraph after reaction

3.2 Bigraphical reactive systems (BRS)

A bigraph corresponds to a state of a system at a certain time. A BRS describes how bigraphs evole over time using reaction rules, as shown in Fig. 1b. If the left

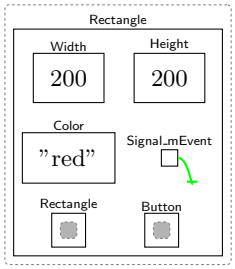
hand side of a reaction rule is matched in a larger bigraph then we can replace the matched part by the right hand side of the rule. Fig. 1b states that whenever a control B contains a control C we can rewrite it by removing C from B. Fig. 1c is the result of applying the rule in Fig. 1b to the bigraph in Fig. 1a.

4 Representation of mechanisms with bigraphs

In this section, we present, using QML as an example, the graphical user interfaces mechanisms that most of UIDL specialised on GUI can express. The encapsulation in QML, is encoded via the type ITEM, inherited by all the graphical entities of the language, and is represented by records from the JSON syntax. About the event handler and the bindings, the former is encoded by the QML signals and slots and the latter by affectations of record fields. A bigraphical representation is provided for each mechanism, to give an idea on how bigraphs could model a GUI.

```
Rectangle {
  width: 200
  height: 200
  color: "red"
  signal mEvent()
  Rectangle {
    width: 100
    height: parent.height
    color: "blue"
  }
  Button {
    onClicked:parent.mEvent()
  }
}
```

(a) QML program example



(b) Scene graph representation

Fig. 2: QML program example and its partial bigraph representation

4.1 Representation of the scene graph

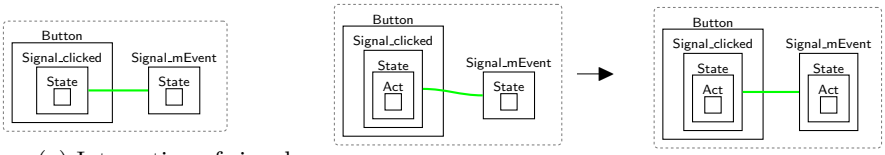
The scene graph is an abstract representation of the program GUI and controls. Often, UIDLs are based on markup-languages (e.g. XML, JSON) because a tree structure is easily induced from their syntax and it also makes the UI design more intuitive for the developers [10]. Hence, reading the QML program from Fig. 2a, we understand that its scene graph root is the red rectangle and the root children are `width:200`, `height:200`, `color:"red"`, `signal mEvent()`, the blue rectangle and the button. Moreover, the induced tree gives information about

the positioning of the entities on the actual interface. Since the blue rectangle and the button are children of the red rectangle, the GUI presents them on top of the red rectangle.

Fig. 2b represents a part of the program scene graph. The nesting of bigraphs helps representing the hierarchical aspect of a GUI scene graph and therefore catching all the needed information.

4.2 GUI interactions

UIDLs allow to describe interactions taking place in the GUI. For instance, two interactions are described in Fig. 2a. The first one, defined at line 8, relates the red rectangle height and the blue rectangle height and implies the update of the blue rectangle height each time the red rectangle height is updated. The second interaction, defined at line 12, relates the implicit `clicked` signal from the button and the signal `mEvent` defined at line 5. It implies that each time the `clicked` signal of the button is emitted (i.e. when the user clicks on the button) then the signal `myEvent` is also emitted.



(a) Interaction of signal

(b) Signal activation

Fig. 3: QML Interactions

Generally, interactions are represented by links in a bigraph. Fig. 3a, represents the interaction from Fig. 2a involving the signals. In this diagram, we define two linked entities corresponding to both the signals from the program. An entity State is nested into each signal, corresponding to its emission state i.e. emitted or not. The activation of a signal by another one can be represented by a reaction rule (Fig. 3b). This rule matches the signal Signal_clicked (implicit signal of the button) activated and linked to the signal mEvent. Then, it activates the signal mEvent by nesting an entity Act into its entity State.

4.3 Bigraphs expressiveness

This section gives a glimpse of bigraphs expressiveness through two examples. The first case deals with an activation condition on entities. If an entity is encapsulated, then it can only be activated if its parent is activated. For instance, only the entities having their parent activated are rendered in a GUI. In other words, a GUI should never be in a state shown by Fig. 4. This property which defines semantics dynamic aspect can be covered by bigraphs thanks to reaction rules.

It could be formalised by a rule similar to Fig. 6. Here, the bigraphs expressiveness allows to ensure parent activation (spatial aspect) and signal activation (non-spatial aspect). This is an original features of bigraphs compared to other process algebra theories.

The other case deals with types of entity activation. For instance, we could associate a type to the graphical entities and another one to the signal entities as shown in Fig. 5. This kind of typing eases the formalisation of the entities activation process. On the one hand, once activated, a graphical entity remains activated until the end of the program or until another entity deacticates it (depending on the UIDL expressive power). On the other hand, a signal entity, once emitted, is deactivated. Hence, bigraphs allow to define, via reaction rules, a general signal emission mechanism according to a typing defined on entities.

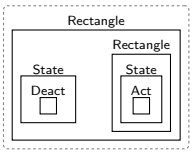


Fig. 4: Inconsistent GUI state

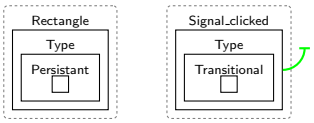


Fig. 5: Process type

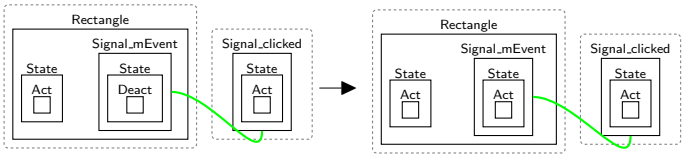


Fig. 6: Activation entity only if parent activated

5 Bigraphs verification

Bigraphs develop a general theory which unifies and represents existing calculi for concurrent communication and mobility. One of the key benefits to formalise UI using bigraph is the possibility to check properties related by their spatial and communication aspects. Indeed, BigraphER [2] an open-source framework for working with bigraphs, allows a transition system, built from a BRS representing an UI states and update sequences, to be exported. The transition system can then be used by existing model checkers to check properties on the given UI. Here, the model checker PRISM [12], that allows specifying temporal properties in the PCTL specification language, is used to check properties. As our model is not probabilistic, we restrict ourselves to the non-probabilistic fragment. In the following, we show through a small example how bigraphs allow us to automatically check properties on UIs models.

5.1 Predicats

To check properties on the generated transition system we require labels on its states. In BigraphER, labels are defined as bigraph patterns $l = B$ that specify that a state should be labelled with l if there is a match of B in that state. In other words, you can think of these patterns as the left-hand-side of a reaction rule. For our analysis we specify two predicates : 1) *signal_clicked* that label states when `Signal_clicked` is activated and `Signal_mEvent` is not ; 2) *signal_mEvent* that label states when both `Signal_clicked` and `Signal_mEvent` are activated.

5.2 Properties verification on UI

To show how properties can be checked on a given program we encode the example of Fig. 2a when the signal `signal_clicked` is triggered i.e. when the button has been clicked. Once the button is clicked,the signal `signal_clicked` is emitted then the rule from Fig. 3b is applied to the model to trigger the signal `signal_mEvent`. We can write a formula in PCTL ensuring that the signal `signal_mEvent` is really activated : $\mathbf{A}[\, signal\_clicked \implies \mathbf{F}\ signal\_mEvent\,]$.

This states that *forall* paths ($\mathbf{A}$) if *signal_clicked* is activated then *eventually* (at some point in the future; $\mathbf{F}$) *signal_mEvent* must be active.

This feature of bigraphs, can be useful for developers to check the soundness of the UI described. Indeed, the size of the UI makes the debugging much more harder [15] which can be eased my automatic verification. This feature could also be used to check that an UI satisfy semantics properties of the UIDL used.

6 Related and future work

This article gives a glimpse of the bigraphs theory, shows how it could be used to model common features of the semantics of GUI specialised UIDLs and how to automatically verify properties on the model. We provide [18] the formalisation of the example in Fig. 2a and a setup to run the verification from Section 4.

Currently, several works exist on the verification of UIs but none concerns UIDLs semantics. Verified react [1] is a project offering the possibility to check logic properties and explore state on react programs. Some work [4, 6] exists on the framework Djnn/Smala addressing the verification of interactive and graphic properties by static analysis. Related to UIDLs, Interactive Cooperative Objects [19] (ICO) is a formalism aimed at describing UIs. It borrows concepts from the object-oriented approach (i.e. inheritance, encapsulation, dynamic instantiation) to describe the structural aspect of a system and uses a high-level Petri nets [11] to describe its dynamic aspect. To reason on this formalism, PetShop [3] will allow to simulate the model and all verification tools for Petri nets can be used.

Our purpose is more related to the programming language aspect. We aim to define a generic UIDL based on the bigraphs theory which covering all common features of GUI specialised UIDLs. The idea is to use this UIDL as an intermediate representation for other UIDLs. This would allow to model mobility and concurrent aspects of a GUI in a unique framework and enable the use of any tool relates to bigraphs, e.g. BigraphER, to formally verify the GUI.

References

[1] Dave Aitken. *Introducing Verified React*. Jan. 2019. URL: https://medium.com/imandra/introducing-verified-react-9c2ef03f821b.

[2] Blair Archibald, Muffy Calder, and Michele Sevegnani. "Conditional bigraphs". In: *Springer International Publishing* (), pp. 3–19.

[3] Eric Barboni et al. "Bridging the gap between a behavioural formal description technique and a user interface description language: Enhancing ICO with a graphical user interface markup language". en. In: *Science of Computer Programming* 86 (June 2014), pp. 3–29. ISSN: 01676423. DOI: 10.1016/j.scico.2013.04.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642313000993 (visited on 06/17/2022).

[4] Pascal Béger. "Vérification formelle des propriétés graphiques des systèmes informatiques interactifs". en. In: (), p. 195.

[5] Gail Chappell and Nancy Hildebrandt. *Using FXML to create a User Interface*. English. Sept. 2013. URL: https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm.

[6] Stéphane Chatty, Mathieu Magnaudet, and Daniel Prun. "Verification of properties of interactive components from their executable code". en. In: *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. Duisburg Germany: ACM, June 2015, pp. 276–285. ISBN: 978-1-4503-3646-8. DOI: 10.1145/2774225.2774848. URL: https://dl.acm.org/doi/10.1145/2774225.2774848 (visited on 06/30/2022).

[7] Qt Company. *QML Tutorial*. 2022. URL: https://doc.qt.io/qt-5/qml-tutorial.html.

[8] Qt Company. *Signals & Slots*. 2022. URL: https://doc.qt.io/qt-6/signalsandslots.html.

[9] Calvary Gaëlle et al. *User Interface eXtensible Markup Language SIG*. en. Ed. by Pedro Campos et al. Vol. 6949. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 693–695. DOI: 10.1007/978-3-642-23768-3. URL: http://link.springer.com/10.1007/978-3-642-23768-3 (visited on 06/24/2022).

[10] Josefina Guerrero-Garcia et al. "A Theoretical Survey of User Interface Description Languages: Preliminary Results". en. In: *2009 Latin American Web Congress*. Merida, Yucatan, Mexico: IEEE, Nov. 2009, pp. 36–43. ISBN: 978-0-7695-3856-3. DOI: 10.1109/LA-WEB.2009.40. URL: http://ieeexplore.ieee.org/document/5341626/ (visited on 06/17/2022).

[11] Kurt Jensen and Grzegorz Rozenberg. *High-level Petri Nets : Theory and Application*. Springer Berlin, Heidelberg. 1991.

[12] M. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[13] Mathieu Magnaudet et al. "Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming". en. In: *Proceedings of*

the ACM on Human-Computer Interaction 2.EICS (June 2018), pp. 1–27. ISSN: 2573-0142. DOI: 10.1145/3229094. URL: https://dl.acm.org/doi/10.1145/3229094 (visited on 06/29/2022).

[14] Ingo Maier, Tiark Rompf, and Martin Odersky. "Deprecating the Observer Pattern". en. In: (), p. 18.

[15] Alice Martin, Mathieu Magnaudet, and Stéphane Conversy. "Causette: User-Controlled Rearrangement of Causal Constructs in a Code Editor". In: *Proceedings of the 30th IEEE/ACM Conference on Program Comprehension* (2022). DOI: 10.1145/3524610.3527885. URL: https://hal-enac.archives-ouvertes.fr/hal-03659579.

[16] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press. USA, 2009.

[17] Brad A. Myers. "Separating application code from toolkits: eliminating the spaghetti of call-backs". en. In: *Proceedings of the 4th annual ACM symposium on User interface software and technology - UIST '91*. Hilton Head, South Carolina, United States: ACM Press, 1991, pp. 211–220. ISBN: 978-0-89791-451-2. DOI: 10.1145/120782.120805. URL: http://portal.acm.org/citation.cfm?doid=120782.120805 (visited on 06/29/2022).

[18] Nicolas Nalpon, Cyril Allignol, and Célia Picard. "Toward a User Interface Description Language based on bigraphs (model files, supplemental material)". In: (Aug. 2022). URL: https://hal.archives-ouvertes.fr/hal-03754387.

[19] David Navarre et al. "ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability". en. In: *ACM Transactions on Computer-Human Interaction* 16.4 (Nov. 2009), pp. 1–56. ISSN: 1073-0516, 1557-7325. DOI: 10.1145/1614390.1614393. URL: https://dl.acm.org/doi/10.1145/1614390.1614393 (visited on 06/24/2022).

[20] Carlos Eduardo Silva and José Creissac Campos. "Can GUI Implementation Markup Languages Be Used for Modelling?" en. In: *Human-Centered Software Engineering*. Ed. by David Hutchison et al. Vol. 7623. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 112–129. DOI: 10.1007/978-3-642-34347-6_7. URL: http://link.springer.com/10.1007/978-3-642-34347-6_7 (visited on 06/30/2022).