Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 109C (2017) 1140–1145

Procedia Computer Science

www.elsevier.com/locate/procedia

International Workshop on Adaptive Technology (WAT 2017)

Hierarchical Design of Adaptive Automata Models

Ricardo Luis de Azevedo da Rocha[a,*]

[a]*Department of Computer Engineering, Escola Politécnica, Universidade de São Paulo*
Av. Luciano Gualberto, travessa 3, 380, 05508-900, Sao Paulo, SP, Brazil

Abstract

The purpose of this paper is to devise a method for the design of adaptive automaton models using functional decomposition. One of the biggest challenges faced by researchers is to restrain side effects asserting design features. We defined a formal notation to structure the process; this allows the developer to build an adaptive automaton in a hierarchical way. As a result, we proved that the design method preserves some properties (determinism). Furthermore, we present an example of an evaluation of the process.

Peer-review under responsibility of the Conference Program Chairs.

Keywords: Adaptive Automata, Computation Theory, Hierarchical formulation.

1. Introduction

This paper aims at an abstraction and a method using that concept in the design of adaptive automaton-based models [1,2], using hierarchical functional decomposition. A formal notation associated with a visualization perspective of the process allows developers to design and refine models based on adaptive automata hierarchically, but it is not entirely within the scope of this paper. As usual in successive refinement-based developing methods, the process may be refined at any suitable level to accommodate not yet considered users' needs [3,4,5].

One of the biggest challenges is to restrain side effects asserting design features. Since we deal with refining adaptive devices, we must take into account adaptive actions beside topological elements. Hence, to preserve hierarchy in the structure of the hierarchically-defined automaton being refined, whenever adaptive actions are used, they must refer strictly to topological elements that are present in the current refinement level in progress. This way we can restrain side effects.

We show an application that illustrates how to apply the suggested method in practical situations. Such example is also used for evaluating the proposal. The main results are presented in this paper.

The paper has six sections, namely: this Introduction; the Main Concepts (foundations); the Definitions and Proper Containment; the proposal; the Example Application; conclusions.

* Corresponding author
E-mail address: rlarocha@usp.br (Ricardo Luis de Azevedo da Rocha).

Peer-review under responsibility of the Conference Program Chairs.
10.1016/j.procs.2017.05.434

1.1. Objectives

The primary goal of this paper is to propose and define a design process for models based on adaptive automata. We set a hierarchical notation for abstracting such a process, which supports functional decomposition of the models, in a similar way to that employed at refining statecharts[3,4,6]. We prove the method is free of non-deterministic structures.

2. Main Concepts

In this section, all needed concepts are introduced and in connection with the literature. The first part defines the foundations of the hierarchical graphs that will structure the development process. In the second part of section 3 we apply such definitions to adaptive automata[1,2] stated hierarchically in a novel formulation, more practical than the traditional one in use so far. We must bear in mind that the graphs here are meant to be automata.

2.1. Description of hierarchical graphs

At the start of the process, there is a simple graph[7,6], which has a single final macrostate (which is also an exit macrostate). This macrostate can then be refined into a single input and single exit subgraph, using as components two macrostates and a clear connection. The components (MS, and CN) can be defined as

MS: A macrostate has an entry point, an exit point, and a refinement flag; the flag has two possible values:

- **false:** in this case, the macrostate represents an automaton state; whenever a new macrostate is created its flag assumes the value false until it is changed to true, thus, allowing its edition;
- **true:** in this case, the macrostate has one start macrostate and one exit macrostate connected by an ϵ-transition (empty); the only editable part is the connection between the macrostates. Edition operations comprehend the replacement of the empty transition by another single non-empty transition, or any subgraph having one single entry point and a single exit point.

CN: A connection is a labeled transition between two macrostates. Replacement can refine any connection: its label may be replaced; or the connection can be replaced by a subgraph. In the particular case that a connection is established between a pair of macrostates that do not allow further refinements, then such connection stands for an actual transition of the automaton. Connection refinements are allowed that change the connection into:

- **insert:** An already defined subgraph, representing a sub-automaton[7];
- **edit:** A tailor-made automaton model, edited according to such specification descriptions. In this case, it is possible to create macrostates and connections between them, without changing the current macrostates. It is also not possible to connect any macrostate to the start macrostate. However, it is feasible to connect any macrostate to another macrostate other than the start state, including the exit macrostate.

Those descriptions allow the formal definitions 1 and 2 on section 3.

3. Definitions and Proper Containment of the Deterministic-Feature

Definition 1 (Macrostate)**.** *A macrostate is a labeled refinable and editable graph vertex that stands for a graph-unit and has the following attributes/features:*

1. *A single Entry point: a single connector that enables any macrostate in the graph to establish a directed connection that uses as its destination the current macrostate;*
2. *A single Exit point: a single connector that enables the current macrostate to establish a directed connection having the entry point of any macrostate in the graph as its destination;*
3. *Refinement flag: a Boolean-valued flag, indicating the existence of further refinements for the current macrostate.*
4. *Label: a user-defined string-valued attribute that names the current macrostate.*

5. *Level: a non-negative integer-valued attribute that represents the hierarchical level of the macrostate.*
6. *Unit type: a non-negative integer-valued flag indicating the type of graph unit the macrostate stands for: main graph (level* 0*), named sub-graph (level* $i > 0$*), named macro (level* $i \geq 0$*)*

Definition 2 (Connection)**.** *A connection is an editable, labeled, directed graph edge that is used for establishing connections from the exit point of some macrostate to the entry point of any macrostate in the graph.*

Definition 3 (Macrostate Refinement Process)**.** *The macrostate refinement maps the macrostate under refinement into a subgraph. Refining a macrostate replaces it with that subgraph. There is a mapping function* $\underline{nam} : \mathcal{L} \rightarrow \mathcal{L}$*, where again* $\mathcal{L}$ *represents a set of labels, and an injection* $\overline{refin} : \mathcal{L} \times \mathbb{G} \rightarrow \mathcal{G}$ *that creates an extension to the labeled macrostate as a newly introduced subgraph with the same properties (a single entry point, a single exit point, and a refinement flag).*

Definition 4 (Connection Editing Process)**.** *The connection refinement is performed by:*

1. *Replacement: the label of the connection is replaced by another through the mapping function* $\underline{rep} : \mathcal{L} \rightarrow \mathcal{L}$*, where* $\mathcal{L}$ *represents a finite set of predefined values, in this case (edges) corresponding to the alphabet of the automaton under development;*
2. *Insertion: the connection is deleted, and in its place (between the exit and entry points of the connecting macrostates respectively) is included a subgraph. The injection* $\underline{ins} : \mathbb{G} \rightarrow \mathcal{G}$ *makes the changes, where* $\mathbb{G}$ *represents the set of all possible subgraphs and* $\mathcal{G}$ *represents the current automaton graph under construction;*
3. *Edition: happens precisely the same way the insertion does, but it accounts for a construction process, meaning that the injection only occurs after the version is completed (this is a development process).*

Definition 5 (Path-Deterministic Graph)**.** *Any graph is considered to be path-deterministic when there is at most one path for each pair* $(Vertex, Label)$*, where Label represents an alphabet symbol, or the empty string* ϵ*, for the automaton. Which means that for every vertex in which there is an empty transition, there can be no other transition departing from the same vertex.*

Fact 6 (ϵ-transitions)**.** *According to definition 5, for any automaton that does not have more than one pair* $(state, symbol)$ *for each state and each symbol. Moreover, also does not have any* ϵ*-transition where there is at least one pair* $(state, symbol)$*; or where there is only one* ϵ*-transition for a state and no other transition, then the automaton is path-deterministic.*

In this work, the <Property> under study is the path-deterministic one. But the Proposition 7, and Theorem 8, and Corollary 9 below are more generic than path-deterministic only.

Proposition 7 (Propagation to Upper and Lower Levels)**.** *Consider an editing process of an automaton on a specified level; if this level is <Property>-safe then at any refinement process inside such level, there is a <Property>-safe propagation to the upper or lower level being defined.*

Proof. By definitions 1 and 3, there is only a single entry point and a single exit point at each refinement level of each macrostate, and also by definitions 2 and 4, there are only one entry and one exit points to a connection refinement. So we have two cases:

$\overset{Lower}{\Rightarrow}$ **:** As the only points of refinement are the macrostates or the connections; there is only one possible refinement to a lower level, the macrostate refinement. There is no other information from the current level flowing to a lower level but the entry and exit points. Therefore the <Property> is contained inside the level where defined.

$\overset{Upper}{\Rightarrow}$ **:** The refinement process, when finished, can put some properties on the automaton graph, including some non-<Property>-safe connections. However, as there is a single connection at the top level, there is no information passing from the lower level to the upper level of refinement.

□

Theorem 8 (Non-<Property>-safe Containment)**.** *At any refinement level each non-<Property>-safe path is contained at this level, which means that it does not propagate to lower levels, nor upper levels of refinement.*

Proof. By propositions 7, there is only <Property>-safe propagation from the editing process. Hence non-<Property>-safe automata can be handled inside each non-<Property>-safe level of the designed model since there will be no non-<Property>-safe propagation from a level to another. □

This theorem shows that the development is properly contained, meaning that the problems may be solved inside a refinement level without having to take into account the whole process. Moreover, it can be easily refined to any other graph property.

Corollary 9 (Graph <Property>-safe Level-Containment)**.** *At any refinement level if each graph <Property> is self-contained, then it does not propagate to lower levels, nor upper levels of refinement.*

Proof. Immediate from Theorem 8 □

Corollary 9 allows the hierarchical construction of adaptive automata models if we can assure containment of the adaptive actions effects. Therefore there is only one issue to be tackled, the adaptive actions.

From this point onward this works considers <Property> as path-determinism.

3.1. Deterministic Adaptive Automata – (properties extracted from (Castro et al)[8])

Property 10 (Necessity - Definition 6 of (Castro et al)[8])**.** *An adaptive automaton model is said deterministic if and only if its underlying device is deterministic at each and every computational step.*

Property 11 (Strong Determinism - Definition 7 of (Castro et al)[8])**.** *An adaptive automaton model is said strongly deterministic when its underlying device remains deterministic at each and every adaptive action applied for an input string $\omega \in \Sigma^*$.*

4. Proposal

Properties 10 and 11 together with Theorem 8 and Corollary 9 assures containment even for adaptive actions. A hierarchical development process based on adaptive automata models should follow the process defined by Harel [3,4]. Our proposal is to extend the development process of Wagner[5] introducing adaptive actions and containment at each level of design as stated by Theorem12.

Theorem 12 (Deterministic Adaptive Automata Model)**.** *If at any refinement level the adaptive automaton follows properties 10 and 11 and each graph property is self-contained, then it is a strong deterministic adaptive automaton.*

Proof. Immediate from Properties 10 and 11, Theorem 8 and Corollary 9. □

Consider Example 13 below where an adaptive automaton is built from a finite-state automaton recognizing the input string "*abc*". That is an example of the application of the Properties and Theorems of section 3. The adaptive action η_1 is defined to change the underlying finite-state model without introducing non-deterministic paths.

Example 13 (Adaptive Automaton)**.** *Consider the basic model for a language $L = a^n b^n c^n, n > 0$ as the automaton described in the upper part of Figure 1. By this model, if there is no input string (ϵ), then it will not be accepted. After the first symbol a to each other consecutive symbol a received the automaton will create another path using action η_1 inserting states and transitions to the current model. The adaptive action η_1 is defined in Algorithm 1 as:*

If the input string is "*abc*" then it will be accepted as shown in figure 1. If the input string is "*aabbcc*" then there is another *a*, and the model is changed by action η_1 to accept the string. This process continues for any $n > 2$.

Algorithm 1 Adaptive Action $\eta_1(r, s)$

$\eta_1(r, s) = \{*r_1, *s_1$ // '$*$' denotes generator, adding new states to the model in this case

1: $-(q_1, a, q_1) : \bullet\eta_1(r, s)$ // Remove the adaptive transition
2: $-(p, c, q)$ // Remove the transition for symbol c
3: $+(p, b, r_1)$ // Add a new transition for symbol b
4: $+(r_1, c, s_1)$ // Add a new transition for symbol c
5: $+(s_1, c, q)$ // Add another transition for symbol c reconnecting to the Final state
6: $+(q_1, a, q_1) : \bullet\eta_1(r_1, s_1)\}$ // Recreate the adaptive transition properly, as executed after the transition happens (denoted after $\bullet$)

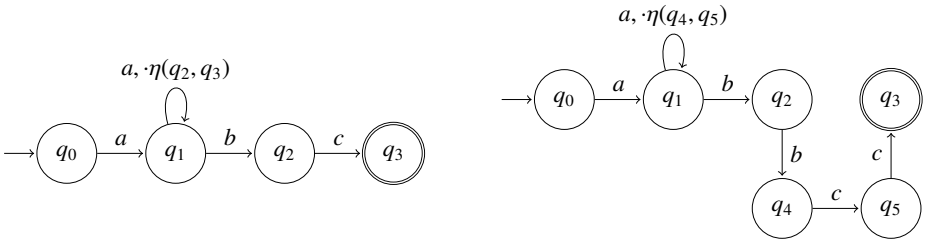

Figure 1. From Example 13– Basic Automaton (first) and one change after "*aa*" (second).

5. Example Application

5.1. Presentation of a Problem extracted from (Wagner, 2006)[5] pages 134–135

"First of all we have to define (input) control values and (output) actions. The first input is obvious - it is the control value representing the button; let us call it Di-Request. A traffic light control uses several timers to measure the required time period. In the discussed case, the state machine needs three timers:

Ti-Yellow : to measure the car yellow period, and also used for a corresponding red–red overlap period later in the cycle.

Ti-Green : to measure the pedestrian green period.

Ti-Disabled : to measure the disabled pedestrian period.

Note that the state machine does not need a timer to measure the car red period as it is defined by the pedestrian green period. Hence, we have the following control values for the state machine:

Di-Request

Ti-Yellow-OVER

Ti-Green-OVER

Ti-Disabled-OVER

The state machine has to set traffic lights. Thus, we define the following actions for the car traffic lights:

C-Red-On

C-Yellow-On

C-Green-On

and for the pedestrian's traffic lights:

P-Red-On

P-Green-On

By these definitions we have assumed that the electrical part of the traffic lights has some 'intelligence' switching on the required lamps and switching off the other ones. In other words, the actions are just commands to a control port, which handles the lamps directly."

5.2. Proposed Solution

Figure 2 describes one model for the original problem as shown in (Wagner, 2006)[5] figure 8.7 on page 135.

To create a more complex example, we add a small feature to detect whether a disabled pedestrian has completed the traversal. If the timer is set to switch the state of the automaton, but the pedestrian has not completed the traversal, we could call an adaptive action to solve it. Nevertheless, there is another way to deal with the new situation.

At the Green-Red state (#7) we create a hierarchical level; within this state, we create a finite-state machine that has an input from a visual sensor able to identify a human being traversing the street. This machine has one entry state a one exit state, following the definitions of section 3.

These two states are initially disconnected, and there are two auto-transitions at the entry state, and one of them is an adaptive transition. Both transitions are applied considering the sensor output; if the sensor produces a *TRUE* (T) signal, it indicates the pedestrian is still traversing, and the machine continues its work. At the time the sensor produces a *FALSE* (F) signal the second transition takes place and the adaptive action $\mathcal{B}()$ removes this transition

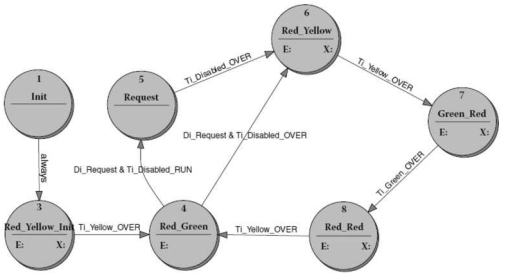

Figure 2. Representation of the Finite-State Automaton – Extracted from (Wagner, 2006)[5] figure 8.7 on page 135

before it is performed and inserts a new adaptive transition connecting the entry state to the exit state. After this, the transition is performed and the adaptive action $\mathcal{A}()$ undoes what action $\mathcal{B}()$ have built; afterward the machine finishes and exits returning to the upper level.

In figure 3 the dimmed edge connecting q_0 to q_1 shows a transition that is built upon execution of action $\mathcal{B}()$; action $\mathcal{A}()$ will redo the original situation of the model. Hence we showed our design process assuring, by construction, the deterministic feature.

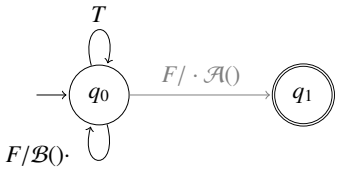

Figure 3. Lower Level Machine.

6. Conclusion

We proposed an abstraction and a method using that concept to develop an adaptive automaton-based model. The process ensures that the desired feature of preserving deterministic models under development if the design follows the definitions herein. Example 13 and Section 5.2 illustrates the design process.

As future works, we expect to associate a visualization perspective of the process allows developers to develop and refine models based on adaptive automata hierarchically. Also, we want to accommodate other features.

References

1. José Neto, J.. Adaptive automata for context-dependent languages. *ACM SIGPLAN Notices* 1994;**29**(9):115–124.
2. José Neto, J., Pariente, C.A.B.. Adaptive automata - a revisited proposal. In: *CIAA 2002: Proceedings of the 7th International Conference on Implementation and Application of Automata - Lecture Notes in Computer Science*; vol. 2608. Springer-Verlag; July, 2002, p. 158–168.
3. Harel, D.. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 1987;**8**(3):231 – 274.
4. Harel, D., Pnueli, A.. *On the development of reactive systems*. Springer, New York; 1985, p. 477–498.
5. Wagner, F., Schmuki, R., Wagner, T., Wolstenholme, P.. *Modeling Software with Finite State Machines: A Practical Approach*. Hoboken, NJ: Taylor and Francis; 2006. URL `http://cds.cern.ch/record/1985457`.
6. Lewis, H., Papadimitriou, C.. *Elements of the Theory of Computation*. New Jersey: Prentice-Hall; 2nd ed.; 1998.
7. Denecke, K., Wismath, S.L.. *Universal algebra and applications in theoretical computer science*. Boca Raton, Florida: Chapman and Hall/CRC; 1st ed.; 2002.
8. Castro JR, A., José Neto, J., Pistori, H.. Determinism on Finite Adaptive Automata – in Portuguese. *IEEE Latin America Transactions* 2007; **5**(7):515–521.