Guidelines for Modelling Reactive Systems with Coloured Petri Nets*

Madalena Gonçalves and João M. Fernandes

Centro Algoritmi — Universidade do Minho
Braga, Portugal
pg18396@alunos.uminho.pt, jmf@di.uminho.pt

Abstract. This paper focus on the modelling of reactive systems, more particularly, control systems. A set of guidelines is proposed in order to build models that support analysis, simulation and prototyping. The guidelines are split in two parts; the analysis of a problem is addressed first, followed by the design with Coloured Petri Nets (CPNs). A smart library example is used as case study. The models developed under this approach turn out to be modular, parameterisable, configurable and executable.

Keywords: Reactive Systems, Coloured Petri Nets, Modelling.

1 Introduction

A reactive system is "a system that is able to create desired effects in its environment by enabling, enforcing, or preventing events in the environment" [18]. This characterization implies that, in requirements engineering for reactive systems, it is necessary to describe not only the system itself, but also the environment in which the system is expected to operate [9].

In this paper, we are particularly interested in controllers, i.e., reactive systems that control, guide or direct their external environments. This work assumes that a controller (to be developed) and its surrounding environment are linked by a set of physical entities, as depicted in fig. 1. This structure clearly highlights two interfaces A and B that are relevant to two different groups of stakeholders, users and developers, during the task of requirements analysis.

From the user's or client's point of view, the system is composed of the controller and the physical entities. Typically, the users are not aware of this separation; they see a device and they need only to follow the rules imposed by interface B to use it. In fact, they may not even know that there is a computer-based system controlling the system they interact with.

* This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-015095.

R.J. Machado et al. (Eds.): MOMPES 2012, LNCS 7706, pp. 126–137, 2013.

From the developer's perspective, the environment is also divided in two parts with different behavioural properties; the physical entities have predictable behaviour, while the human actors may exhibit, for example, disobedience with respect to their expected behaviour. Thus, the description of the behaviour of the environment must consider the physical entities (usually called sensors and actuators) which the system interacts with through interface A. In some cases, these physical entities are given, and software engineers cannot change or affect them during the development process, but they need to know how they operate. Additionally, some relevant behaviour of the human users that interact with the system through interface B must be considered and actually reflected in the models.

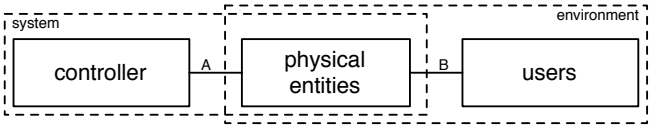

Fig. 1. A controller and its environment

In this paper, we suggest a set of guidelines for modelling reactive systems; although some of the proposed guidelines may be applicable to other formalisms and languages, we specifically targeted them for the Coloured Petri Nets (CPN) modelling language. We assume that the readers are familiar with the CPN common practices, thereby we do not provide details on the subject. For an introduction, please refer to [10], [12] and to the website `http://cpntools.org`.

We focus on the study of reactive systems, finding which common properties emerge among different systems of this kind. We also study how these properties may be described by building specification models that support executability, modularity, configurability and parameterisation. Our approach is illustrated in the development of a Smart Library.

This work has been developed under the scope of the APEX project, which aims to study the user experience within ubiquitous systems. Within this project, a framework was developed [15,16] to facilitate the connection between CPN models and simulation engines, allowing the creation of virtual prototypes, with which users may interact. Such prototypes shall support the study of the behaviour of users facing computer systems, which may be a source of requirements for the system specification.

With our guidelines, we intend to systematise the process of modelling; this is our main contribution. We hope that, by following these guidelines, the resulting models are accurate enough, for the development of reliable prototypes, and structured enough to allow the models to be more easily modified in case requirements need to be changed or new ones introduced.

The paper is structured as follows. Sect. 2 discusses related literature. In sect. 3, the Smart Library case study is introduced. In sects. 4 and 5, we present several guidelines for modelling reactive systems with CPNs. Some conclusions are drawn in sect. 6.

2 Related Work

There is no "official" recipe book describing how to model software, in general. However, many modelling approaches have been proposed for specific contexts, targeting different types of systems and different modelling languages.

In [1], Coad and Yourdon give strong emphasis to problem analysis, before start modelling the problem. They state that both analysis and design must be performed using the same underlying representation, in order to avoid great differences between those two tasks. Their suggestion resides in an object-oriented approach.

In [8], the Statemate approach is explained, addressing the modelling of reactive systems. A system specification is organised in three views - *functional*, *behavioural*, and *structural* - and each of these describes a different perspective of the system under consideration.

Wieringa also discusses the modelling of reactive systems in [18], providing some new outlooks on the topic. Similarly, in [17], some overall modelling guidelines are given, with a great focus on the modelling of real-time systems.

A lot of studies combine different modelling languages, trying to gain some leverage from the particularities of each language. For example, the Unified Modelling Language (UML) and Petri Nets (PNs) can complement each other, since the UML is a standard modelling language, world-wide known and used, and PNs are formal and executable, and provide good analysis and validation techniques [2]. Examples of systems that are modelled using UML diagrams and then transformed into a PN model can be found in [4], [13] and [14]. The UML diagrams that are more commonly used for this purpose are Use Cases, Statecharts and Scenario Diagrams (Sequence or Colaboration Diagrams). The results are presented in different variants of PNs, like CPNs or Object-Oriented Petri Nets (OOPNs).

Girault and Valk [7] also refer some techniques to model with PNs; following a bottom-up approach, they suggest to make a list of every possible states of an object class, then find an event for each state change and connecting those events to the corresponding states. Later, different object classes are combined to model a greater (target) system.

The guidelines proposed in sections 4 and 5 of this paper use a collection of ideas and concepts from the literature mentioned above, but they are mainly inspired on the works of Coad&Yourdon, Wieringa, and Girault&Valk. In both our analysis and design parts, we structure our approach according to ideas that follow. First, we deal with entities (users and physical entities) and then we define the structure that connects the entire system (similarly to Coad&Yourdon). Afterwards, we narrow the entities, by identifying states and events and binding these properly, in order to achieve the desired behaviour (similarly to Girault&Valk). Wieringa's work contributed with some definitions that are used in this paper.

3 Case Study: Smart Library

The guidelines presented in this paper were developed along with a few CPN diagrams for a smart library example. In our example, there are some gates for entering and other gates for exiting the library; each gate allows the entry/exit from multiple users, at the same time. The problem described in [16] explains what happens when a user is looking for books he had previously requested (e.g., via a web interface). This library recognizes registered users and creates different light paths to guide them from the entry gate to their requested books. Unregistered users can not be guided by the smart library.

Every time a registered user approaches an entry or exit gate, a screen near that gate displays that user's information (a list of his/her requested and returned books). Presence sensors, scattered throughout the library, are responsible for real-time recognition of registered users and books locations (inside or near the library). A registered user carries an id card or device (like a PDA) that makes him/her recognizable by the sensors; as he/she approaches a book that he/she requested, lights of a specific and unique colour are turned on, showing the bookshelf where the book stands, and highlighting the actual book. Different light colours are used to distinguish the requests of different users.

4 Guidelines for Analysis

Before start sketching actual diagrams, one must take a moment to analyse the problem in hands. The first thing to do when modelling any system is to study it and understand its purpose. So, it is necessary to identify every single entity within the system and within its environment - the Controller, all Physical Entities, and all Users - the roles they play and how they do it.

4.1 Identify the Physical Entities

The sensors are the physical objects that observe events from the system environment [18], and that warn the system when any relevant external event has occurred. Such warnings are presented to the system as stimuli and can be responded to. To describe a sensor one must state which type of sensor is needed, what does the sensor do, and how does it do it. Since not all changes in the environment are relevant to the system under development, it is also important to state which events must be reported to the system.

The actuators are the other physical objects that comprise the system. One must analyse their behaviours to ascertain which ones have active roles within the system (i.e., which ones have behaviour of their own that is relevant to the system) and which ones are just passive actuators (i.e., do not have behaviour of their own that is relevant to the system) [3].

In the smart library example, five physical entities were identified: books, presence sensors, lights, gates, and displays.

4.2 Identify the Users

Each user represents a category of human actors that interact with the system. Users may have different privileges and perform different actions; hence, one must tell users categories apart, and identify all possible actions for each category.

In the smart library example, we assume that registered users can perform two actions: moving between areas of the library (i.e., areas that are sensed by the library presence sensors) and picking up books.

4.3 Identify the Global Structure for Communications

Both the controller and the users exchange data between each other. Recalling fig. 1, the communication structure is defined by two interfaces, A and B. To describe each of these interfaces, one must identify: (1) which physical entities are directly connected to the controller and (2) which physical entities are directly connected with each category of users, respectively.

In the smart library example, both the controller and the users interact with all physical entities.

4.4 Identify the Functions of the Controller

The controller is the *brain* of the system: by connecting the physical entities and making sequences of actions of those entities, the controller creates a new behaviour that responds in a desired fashion to the environment in which the system is embedded. The events that are external to the system are observed by the sensors and sent to the controller as stimuli. The controller then chooses an appropriate answer and enforces particular behaviours to the actuators.

To describe the controller, one must find all the functions, validations and decisions that must be performed by the system under consideration, but cannot be assigned to users or physical entities.

In the smart library example, the main responsibilities of the controller are updating the user information in the displays, identifying the books by reading their RFID tags, turning the lights on and off, opening and closing the gates, and reading users' positions from the presence sensors.

4.5 Identify Private Phenomena of the Physical Entities

The private phenomena of an entity are its states and internal events. Regarding physical entities with active roles, one can start describing their behaviour by determining what are the stares of those entities, and then identifying which possible actions could be performed to toggle among those states.

The identification of the internal events strongly depends on the choices of the modeller. In some cases, it is possible to assign a different event for every single state, but it is also possible to create only one event to toggle between all the

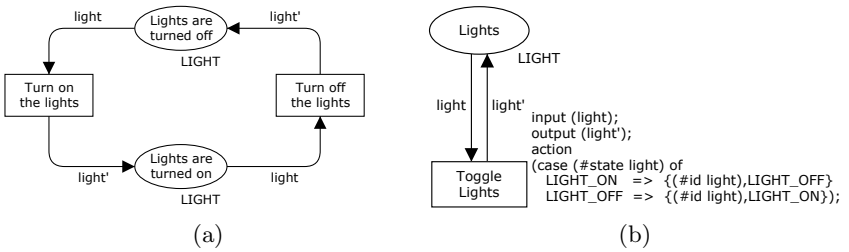

Fig. 2. (a) Each state responds to a particular event; (b) Both states respond to the same event

states. As an example, imagine a lightbulb that can be either *turned on* or *turned off* (hence, it has two different states); the behaviour of that lightbulb can be described by two events, *Turn the lights on* and *Turn the lights off* (fig. 2(a)); or by a single one, *Toggle lights* (fig. 2(b)).

The occurrence of an internal event may depend on internal and external conditions [18]. Examples of internal conditions may include time (i.e., an event that only occurs after a given amount of time) or the current state of the entity itself (i.e., an event that only happens when/if the entity finds itself in a specific state). Examples of external conditions may include commands sent by the controller and actions performed by users.

To describe the private phenomena of a physical entity one must identify its states, its internal events, and those internal events restrictions. Relevant attributes, like id, name, or others, must also be identified.

In the smart library example, the lights are modelled as it can be seen in fig. 2(b): two possible states (*on* and *off*) and one event to toggle between them (*toggle lights*). This event depends on an external condition: it can only occur if the controller issues a command to toggle the lights.

4.6 Identify Phenomena Shared between Physical Entities

The shared phenomena [5] are the states and events shared within the interfaces A and B (fig. 1). For each pair of communicating entities that were identified in the guideline 4.3, one must decide what type of data is shared between them. To help in this decision, one can face a state as data that is always available for others to read, and although it can be changed, it does not disappear; while an event can be seen as data that is only available for the first who "gets" it; once it is consumed by some entity, it ceases to exists.

In the smart library example, the commands sent by the controller to *toggle lights* (recall the example explained in the previous guideline) are an example of shared phenomena. Each command from the controller to the lights is a shared event, since each command can only be executed once.

5 Guidelines for Modelling

After studying the problem carefully, one can move on to drawing CPN diagrams. By using this language one can guarantee that models can be executed, and thus simulated.

The following guidelines explain how to use these diagrams to model reactive systems within *CPN Tools*.

5.1 Create a Petri Net

The first thing to do is to open *CPN Tools* and create a new CPN model and a few new pages: one page for each sensor; one page for each active actuator; one page for each user; one page for the controller. Each page works as an individual module. By following this guideline, one guarantees model modularity.

5.2 Draw the Physical Entities

In the previous analysis, active and passive actuators have been distinguished. This knowledge helps to decide which actuators shall be modelled as individual modules (the ones that have an active role) and which ones shall be modelled as simple tokens or not be modelled at all (the ones that have a passive role). This guideline focuses on sensors and active actuators, and for each of these entities one must do the following five tasks:

Create Colour Sets. In a first approach, the color set must be as simple as possible, only describing the states and imperative attributes.

The color set of an entity can be either simple or compound. If one wishes to describe more than just the states of the entity, then a compound color set must be written.

To help in the comprehension of the model, this color set should not be a list; however this is not mandatory. The reason for this suggestion is because, when using a list, one has to analyse its content to realize how many tokens it holds; for other color sets, the number of tokens is presented directly in the graphical interface of the model, and that makes it faster to read it.

Create the Main Place for Each Module. Every CPN module needs a place to hold the instances of the entity it represents. Draw one place, name it, and give it the proper color set. For example, a module for a *door* must have a place to hold *door instances*. From now on, such place is called as the *main place*.

The place *Gates* in fig. 3 is an example of a main place. This place holds tokens with the colour set *GATE* which has three attributes: the id, the position and the state of the gate.

The main place should always be a *state place*, i.e., a place that holds tokens with information about the state of an entity. This means that it always holds the same number of tokens that it initially possesses. Any transition connected

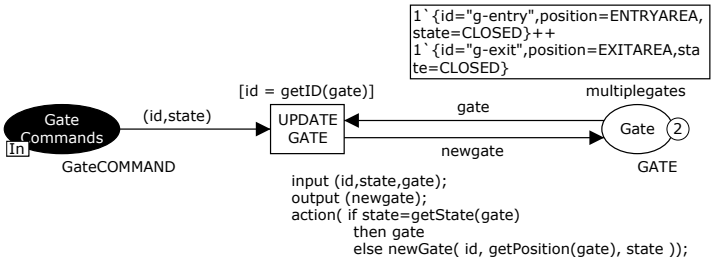

Fig. 3. The Gate module

to this kind of place can "peek" on the data that is inside it; the occurrence of such a transition can consume tokens, in order to change their values, but those tokens have to be restored right away. Consuming and restoring a token from a state place can be seen as a single, instantaneous action. In an *event place*, i.e., a place that holds tokens with information about events, the data within that place is stored by some transitions, and consumed by others (which translates into two different actions).

Create Internal Events. Draw a transition for every event and name each properly.

The transition *UPDATE GATE* in fig. 3 exemplifies the representation of an internal events.

Create CPN ML Primitives. Some CPN ML variables and other coding must now be implemented. Any transition has four types of inscriptions: name, guards, time delays and code segments. One must now write the CPN ML primitives that validate the conditions imposed to the occurrence of internal events (i.e., implement the guards); and assign a behaviour to those events (i.e., implement the code segments). A more experienced modeller may deal with time inscriptions right away, but these can also be dealt with later, once the first version of the model is functional and stable.

In fig. 3 we can find examples of variables, like *state* and *gate*, functions like *getID* and *newGate*, and also an example of a value *multiplegates*.

Set the Data Flowing Direction. Draw the arcs that connect the internal events to the main place and assign them the proper variables (which were created in the previous task).

The two arcs between the transition and the main place, in fig. 3, show how the tokens travel in the gate module. The transition *UPDATE GATE* "grabs" a *gate* token from the main place, complaint with the condition that is enforced by the guard of that transition, processes that token (which means the token may be change, or not), and returns the token to the main place as *newgate*.

5.3 Draw Interfaces for Shared Phenomena

The CPN modelling language allows two representations of shared places: sockets and fusion places. A *fusion place* is like two places that were merged, and are accessed by two different entities, that share their contents. A *port place* is a place that can communicate with another port place through a private communication channel (a socket). It can be either an unidirectional or a bidirectional socket. The doubt comes when one has to decide wether use fusion places or port places, for each kind of data (states or events).

In CPN Tools, a socket can only be shared between two entities, and it is not possible to share port places. Since a states place may be accessed by several entities, port places are not suitable to depict such places. In contrast, fusion places are perfectly fit to share a states places; in CPN Tools, in order to make a fusion place from a common place, one has to tag that common place with the desired fusion tag.

We advice the use of fusion places for shared states and the use of port places for shared events.

The black place *Gate Commands* in fig. 3 is a port place of a socket shared with the controller module. The controller adds tokens in this place, and the gate module consumes those tokens.

5.4 Draw Scenarios

Last but not least, one must create scenarios to depict the desired behaviours of both the controller and the users. Two tasks must be carried out to do so:

Create Initial Values. To test a CPN model, it is necessary to create instances of the physical entities and initialize the main places with those values. By doing so, the models become parameterisable.

The value *multiplegates* in fig. 3 is the initial value of the main place *Gates*.

Create the Desired Behaviours of the Controller and the Users. This is the most creative task of all these guidelines, because it depends a lot on the nature of the problem under consideration and how the modeller faces it.

The shared places must already be drawn for these modules (guideline 5.3) and that is a starting point; but, from now on, the modeller has to decide how to process the data that arrives in the input shared places and to whom send the results of that processing, i.e., to which output shared places should tokens be added.

Usually, there are many possible behaviours that can be depicted, for both the controller and the users; therefore, one must use scenarios to illustrate those behaviours. Scenarios are useful for depicting different courses of action, but also to consider paths that may or may not happen. For example, consider the example of a user, walking inside the smart library, following the path the library suggests, when at a certain moment he ignores that suggestion and starts wandering inside the library at his will. In this situation, there are two different

behaviours, which means, two different courses of action: first, the action we want to be executed, where the user follows the suggested path; and second, an alternative action, where the user does not follow the suggested path. These actions are equally important and must be represented in the users scenario model. Since the CPN modelling language provides formal analysis of its models, one can evaluate if a particular action (or set of actions) will be executed or not.

The key action behind any scenario is the decision-making; scenarios are made to consider many decision points and many solutions. Variation points (VPs) can be used to depict paths of alternative execution. Please refer to [6] for a complete explanation on VPs with CPNs.

Scenarios are models prone to change; the conditions that support a given scenario may change rapidly, and when that happens, the scenario needs to be adapted accordingly to the new conditions. If the problem being modelled is quite simple and is not subject to major changes, then the modeller can choose to create more graphical scenarios, which can be quickly read and easily understood. On the other hand, if the problem is likely to change over time, requiring new scenarios to be considered or old ones to be disregarded, then the modeller should opt to draw less graphics and write more code, because code can be easily modified and is much more scalable. Thereby, it is advisable to build simple scenarios that are easily configurable; and because of that, the modeller must sometimes choose code over graphics.

Fig. 4 depicts the users module for the smart library, and is an example of how a scenario can look like. *Users inside or near the Library* can perform the action *MOVE BETWEEN AREAS*, by choosing one path from the *Possible Paths* place. This choice is restricted by two conditions: a user can only move to another area, if there is a path from that user's position to the area he wants to go to, and that path is not blocked by a closed gate. For example, a user u can go from *inside* the library to the *exit area*, but the user cannot exit the library, because the path *exit area* to *outside* is blocked by a gate in the *exit area*.

The use of scenarios ensures that the models can be easily configured.

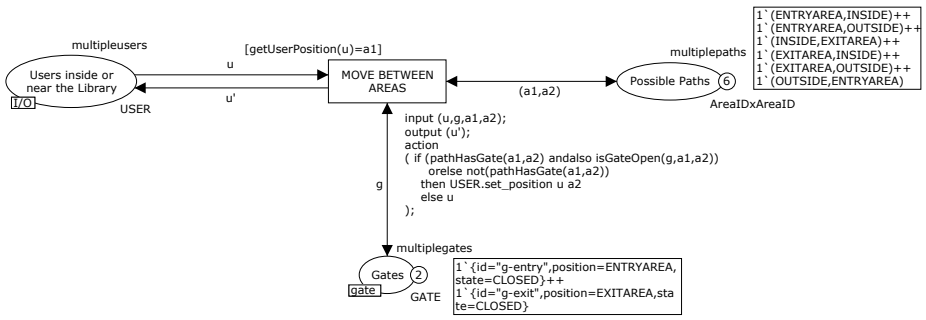

Fig. 4. The users scenario for the Smart Library

6 Conclusions and Future Work

This work is a first step in the development of a modelling methodology, that focus on model simulation, formal analysis and prototyping. Here, we propose a set of guidelines for modelling reactive systems, more particularly control systems. The guidelines are suitable for modelling with Coloured Petri Nets (CPNs), and cover both problem analysis and design. With these guidelines, we intend to help modellers develop simple and structured models, that can be used as executable prototypes of a final product. Since the prototypes are executable, users are able to interact with an abstraction of the real system, which makes these prototypes a good source of requirements for the system specification. A smart library example is used to illustrate the guidelines.

Our approach supports models that benefit from the features that follow next. (1) *Modularity*: setting modules apart, makes it easier to add or remove involved actors (both people and devices); (2) *Parameterisation*: storing parameters in tokens, makes the models expandable; (3) *Configurability*: using scenarios, makes it possible to depict a great number of possible behaviours; (4) *Executability*: being executable, makes the models suitable for simulation, formal analysis and prototyping.

These guidelines were developed within the APEX framework, which aims the study of human behaviour towards software systems. As future work, we intend to apply these guidelines in other case studies, developing other models that can be used as tests for the framework. We expect those experiments to help us on adding new guidelines and tuning the existing ones.

References

1. Coad, P., Yourdon, E.: Object-oriented analysis, 2nd edn. Yourdon Press (1990)
2. Denaro, G., Pezzé, M.: Petri nets and software engineering. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 439–466. Springer, Heidelberg (2004)
3. Douglass, B.P.: Real-time UML: Developing efficient objects for embedded systems. Addison-Wesley (2000)
4. Elkoutbi, M., Keller, R.K.: Modeling interactive systems with hierarchical colored Petri nets. In: Proceedings of the 1998 Advanced Simulation Technologies Conference, pp. 432–437 (1997)
5. Fernandes, J.M., Jørgensen, J.B., Tjell, S., Baek, J.: Requirements engineering for reactive systems: Coloured petri nets for an elevator controller. In: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 2007), pp. 294–301. IEEE Computer Society (2007), doi:10.1007/s11334-009-0075-6
6. Fernandes, J.M., Tjell, S., Jørgensen, J.B., Ribeiro, O.: Designing tool support for translating use cases and UML 2.0 sequence diagrams into a coloured Petri net. In: Proceedings of the 6th International Workshop on Scenarios and State Machines (SCESM 2007). IEEE Computer Society Press (2007), doi:10.1109/SCESM.2007.1
7. Girault, C., Valk, R.: Petri nets for system engineering: A guide to modeling, verification, and applications. Springer (2001)

8. Harel, D., Politi, M.: Modeling reactive systems with Statecharts: The Statemate approach, 1st edn. McGraw-Hill (1998)
9. Jackson, M.: Problem frames analyzing and structuring software development problems. Addison-Wesley (2001)
10. Jensen, K.: Coloured Petri nets basic concepts, analysis methods and practical use. Monographs in Theoretical Computer Science, vol. 1. Springer (1992)
11. Jensen, K., Kristensen, L.M.: Coloured Petri nets: Modelling and validation of concurrent systems. Springer (2009)
12. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. Software Tools for Technology Transfer 9(3-4), 213–254 (2007), doi:10.1007/s10009-007-0038-x
13. Jørgensen, J.B., Tjell, S., Fernandes, J.M.: Formal requirements modelling with executable use cases and coloured Petri nets. Innovations in Systems and Software Engineering 5(1), 13–25 (2009), doi:10.1007/s11334-009-0075-6
14. Saldhana, J.A., Shatz, S.M.: UML Diagrams to object Petri net models: An approach for modeling and analysis. In: Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE 2000), pp. 103–110 (2000)
15. Silva, J.L., Campos, J.C., Harrison, M.D.: An infrastructure for experience centered agile prototyping of ambient intelligence. In: Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2009), pp. 79–84. ACM (2009), doi:10.1145/1570433.1570450
16. Silva, J.L., Ribeiro, O.R., Fernandes, J.M., Campos, J.C., Harrison, M.D.: The APEX framework: Prototyping of ubiquitous environments based on Petri nets. In: Proceedings of the 3rd International Conference on Human-Centred Software Engineering (HCSE 2010), pp. 6–21. Springer (2010), doi:10.1007/978-3-642-16488-0_2
17. Ward, P.T., Mellor, S.J.: Structured development for real-time systems. Essential modeling techniques, vol. II. Pearson Education (1986)
18. Wieringa, R.J.: Design methods for reactive systems - Yourdon, Statemate, and the UML. Morgan Kaufmann (2003)