Leveraging High-Level Petri Nets for Cyber-Physical Systems Development

Nuno Fernandes[1], João-Paulo Barros[1,2,3,*] and Rogério Campos-Rebelo[1,2,3,4]

[1] *Polytechnic Institute of Beja, Beja, Portugal*
[2] *Center of Technology and Systems (CTS), UNINOVA, Portugal*
[3] *Intelligent Systems Associate Laboratory (LASI), Portugal*
[4] *Lisbon School of Engineering (ISEL), Polytechnic University of Lisbon, Lisbon, Portugal*

Abstract

Petri nets are a powerful graphical formalism that is well-suited for modeling and simulating concurrent systems. However, existing solutions for integrating high-level Petri net models with external systems and physical devices are limited, restricting their applicability for developing cyber-physical systems. This paper presents a middleware implementation that bridges this gap by enabling reference nets, a high-level Petri net class supported by the Renew tool, to interface with physical devices. The proposed middleware implementation facilitates Petri net-driven development, where the graphical net model serves as the core specification driving the system's behavior using a low-code, top-down approach. This approach transforms the traditionally autonomous Petri net model into an event-driven, reactive model capable of bidirectional communication with external physical components. The paper demonstrates the middleware's usage through a case study involving a gate and button as physical devices, exemplifying how their states and events can be seamlessly integrated into the high-level Petri net model's execution semantics. This integration empowers the development of sophisticated cyber-physical systems with Petri nets as the central, formal modeling paradigm.

Keywords

Reference nets, RENEW, Cyber-Physical Systems, Low-Code, Model-Based Engineering, Internet of Things

1. Introduction

Model-based engineering leverages low-code platforms that employ precise diagrammatic models to bridge requirements and automated code generation. The rise of user-friendly, low-code tools not only eases software development but can accelerate time-to-market by streamlining the model-driven development lifecycle, allowing developers to focus on core system logic rather than low-level coding. Petri nets offer a powerful visual formalism with precise semantics, supporting key concepts adopted by numerous graphical modeling languages. Unlike some graphical specification languages, Petri nets' formal rigor enhances effective communication, system validation, and automated code generation throughout the development

PNSE'24, International Workshop on Petri Nets and Software Engineering, 2024
*Corresponding author.
nuno.fernandes@ieee.org (N. Fernandes); joao.barros@ipbeja.pt (J. Barros); rogerio.rebelo@ipbeja.pt (R. Campos-Rebelo)
0009-0009-2011-472X (N. Fernandes); 0000-0002-0097-9883 (J. Barros); 0000-0001-7177-7966 (R. Campos-Rebelo)

lifecycle. This aligns with low-code approaches (e.g., [1]) but requires models that can interact with external entities.

In this paper, we leverage Reference nets [2, 3], a class of high-level Petri nets supported by a freely available tool, by presenting a middleware for communication between net models and physical devices. The middleware includes some net models that act as the first communication level between the net domain and the physical devices. The presented net models and middleware are freely available at https://doi.org/10.5281/zenodo.11555310.

The following section surveys relevant related work. Section 3 uses a running example to describe the proposed system's components and their behavior. Section 4 delves into the example Petri net models, including the implementation of the Observer pattern. Finally, Section 5 concludes the paper and outlines future directions.

2. Related Work

Graphical modeling formalisms, such as Petri nets, are often used to design the system, but also to simulate it, as in the work presented in [4]. The authors present an approach to integrate a Petri net simulator, using the Renew tool, into a service-oriented simulation architecture in order to provide on-demand simulation as a service. However, the proposal does not provide an explicit support for communication between the model and physical devices. The focus is primarily on the integration of Petri net simulation into a service-oriented architecture, reflective simulation, remote simulation capabilities, and model manipulation within a distributed simulation environment.

The work [5] presents a set of tools, based on the IOPT-Nets class, an non-autonomous low-level net class. The tools allow modeling, simulation, verification, debugging, and code generation for discrete-event controllers.

The work presented in [6] presents a symmetric Petri net model with publish-subscribe middleware and a generic component abstraction that captures the flow of events, without modeling event data, providing a useful abstraction for viewing a system composed of distributed components. Hanisch and Lüder [7] present a signal extension for low-level Petri Nets emphasizing the separation between the controller and the plant and a modular approach. They conclude that the resulting models are valid, but the models are no longer Petri Nets.

Although there have been many studies and projects involving Petri nets, it is rare to come across the use of high-level Petri nets as a foundation for implementing systems using a low-code, model-driven approach. The use of Petri nets in systems implementation seems mostly restricted to the areas of design and simulation. The following section, presents the use of the developed middleware through an example of a simple cyber-physical system.

3. System components and behaviour

As a running example, we considered a prototype for a Cyber-Physical System (CPS) comprising two physical devices: a gate and a button. The system is loosely based on the one presented by Jackson [8]. In our system, a gate can open or close based on an external timer that commands it.

If a human operator presses a button, the gate stops moving and can then resume its movement or returned to its previous position based on the command received from the external timer.

Each communication between the Physical Device and the net model is supported by two main components: (1) one implemented in C++ and running on each of the external devices (Arduinos, in our prototype), and (2) a logical component, developed in Kotlin and reference nets models, running on the Renew tool. These two components communicate bidirectionally via the MQTT (Message Queuing Telemetry Transport) protocol, a well-known protocol that requires minimal resources [9]. The prototype uses the MQTT Quality of Service Level 0 (QoS0) with default specifications.

The gate behavior and the interaction between the model and the Physical devices (the gate and button) take place on a reference net named `Controller`, later presented in Section 4.1. This net models the gate behavior and can be seen as its controller model. It creates instances of `Gate` nets, presented in Section 4.2, that handle the MQTT messages to and from the physical devices.

3.1. Physical Component

The *Physical Component* is supported by C++ code running on, or directly connected to, a physical device, and has two tasks: (1) receive, read, and execute the actions from the logical component (the reference net); (2) send messages to the logical component allowing its update and state synchronization. The messages are exchanged using the MQTT protocol.

3.2. Logical Component (Kotlin and Petri Nets)

The *Logical Component* is the software that initiates communication with a physical device. It is written in the Kotlin programming language, chosen for its seamless integration with Java, which the Renew tool uses. The Logical Component also encompasses Petri Nets-based modeling to represent and control the system's states and transitions.

3.3. The MQTT protocol

The MQTT protocol supports the bidirectional communication between each physical device and the Logical Component. MQTT enables the two system parts to communicate effectively and dependably, even in unstable network conditions or with constrained bandwidth.

3.4. Model description

The MQTT server acts as a communication intermediary between each physical device and the Logical Component. It is hosted in the cloud. It allows the connected devices (in this case, the physical device in an Arduino and the server running the Logical Component) to post messages on specific topics and subscribe to receive messages of interest. This cloud server offers scalability, availability and security for communication between the system's components, allowing them to communicate reliably on a global scale.

3.5. Asynchronous execution environment

The Logical Component operates in a computing environment distinct from that of the Physical Component. This means that the Logical Component can be on a device such as a personal computer or a server. At the same time, in our prototype, each Physical Device, is embedded in an Arduino.

3.6. System behavior

Several components need to interact asynchronously and synchronously to allow communication between each physical device and the logical component. The sequence diagram in Figure 1 shows two message flows between the various elements, with all the elements to the right of the MQTT Server forming part of the Logical Component. Message sequence 1 to 5 starts on the Physical Device, while message sequence 6 to 9 start on the Controller net. Note that messages up to the Communicator element (1 to 3) are asynchronous, as the communication between the Java MQTTClient class and the Communicator net uses the Observer Pattern [10]. This design pattern was used to avoid blocking the execution of the entire model while it waits for messages. The net model reacts accordingly to the arrival of each different message (see bottom of Fig. 3a).

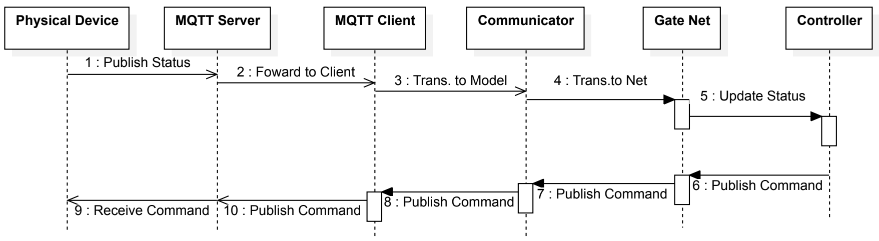

Figure 1: Sequence diagram

The observer pattern is implemented using a net as an observer. This net uses synchronization channels (between transitions) in different net instances to synchronously pass the message. The Renew tool provides a specific tool, called *stubs* [2, 3], to ease the integration of nets and Java code. The integration with Kotlin code was also achieved using this tool.

In summary, the middleware prototype demonstrates bidirectional communication between an embedded device (Arduino) and a computing environment (PC/server) using MQTT. The Logical Component (based on Kotlin and Petri Nets) reflects Physical Devices' status and sends commands. The Physical Device's C++ code executes received commands and provides results to the Logical Component. The cloud MQTT server facilitates message exchange between the components.

4. Net models

In this section, we present the different nets that make up the Logical Component. In addition to demonstrating the system state, we also model the interaction between the gate and button devices. The observer pattern implementation is presented in Subsection 4.4.

4.1. Net `Controller`

The net `Controller` consists of two subnets (see Fig. 2): (1) the one on the left represents what state the gate is in, and (2) the one on the right how a stop button interacts with the model to pause the gate when it is moving: the gate up and down movement is gain modeled, and a virtual place is used to duplicate place `In Move`. A *virtual place* is represented by a double circle or ellipse and is a reference to a place, thus allowing multiple occurrences of the same place. On the left, the net state is updated with messages that are sent by physical devices to instances of the net gate, where each instance models a physical gate.

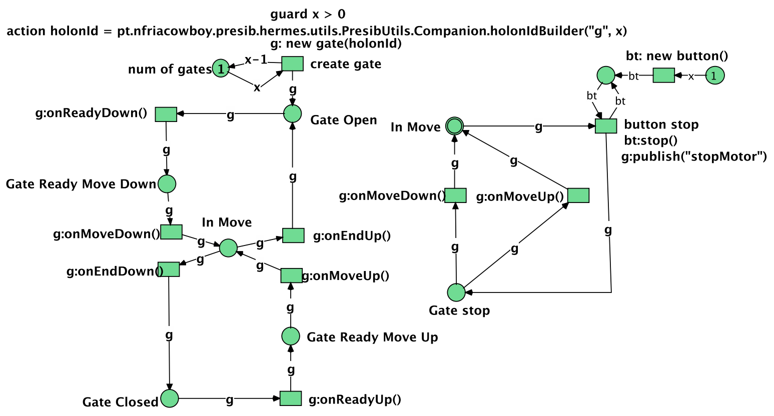

Figure 2: Net `Controller`.

As usual in high-level nets, tokens in Reference nets can have distinct values. The place `num of gates` uses that possibility as the token has de initial value `1` and transition `create gate` can remove that token, bind it to variable `x` and create a new token with value `x + 1` using an output arc expression. Additionally, and very significantly, Reference nets implement the nets-within-nets paradigm [11] where a token can be a net. More precisely, a token can be a reference to a net instance mimicking class-based object-oriented programming languages: nets are classes from which executable net instances are created. In net `Controller`, transition `create gate` creates an instance of net `Gate`, with an identifier as an argument, and assigns the instance to variable `g` (`g: new gate(holonid)`, which becomes a token in place `Gate Open`. Hence, a running net `Gate` instance (see Fig. 3a) is created in net `Controller` and the `g` token is added to the place that represents the state of the physical gate.

Like the net `Gate`, an instance of net `Button` is also created, represented by variable `bt` (top right in Fig 2. If there is a `g` token in the `In Move` *virtual place*, the transition `button stop` is enabled. The stop button sends a message to the gate instance (`g:publish("stopMotor")`) to make the physical system stop, but only if an `"s"` message has been received in net `Button` allowing the bottom transition in Fig. 3 to fire. This is achieved using *synchronous channels*, a type of transition fusion inspired by method calling in object-oriented programming languages: one side uses the notation `netInstance:message()` and the other `:message()`. When transition `button stop` fires, the `g` token is placed in place `Gate stop`. After, the gate movement can continue up or down.

4.2. Net `Gate` and net `Button`

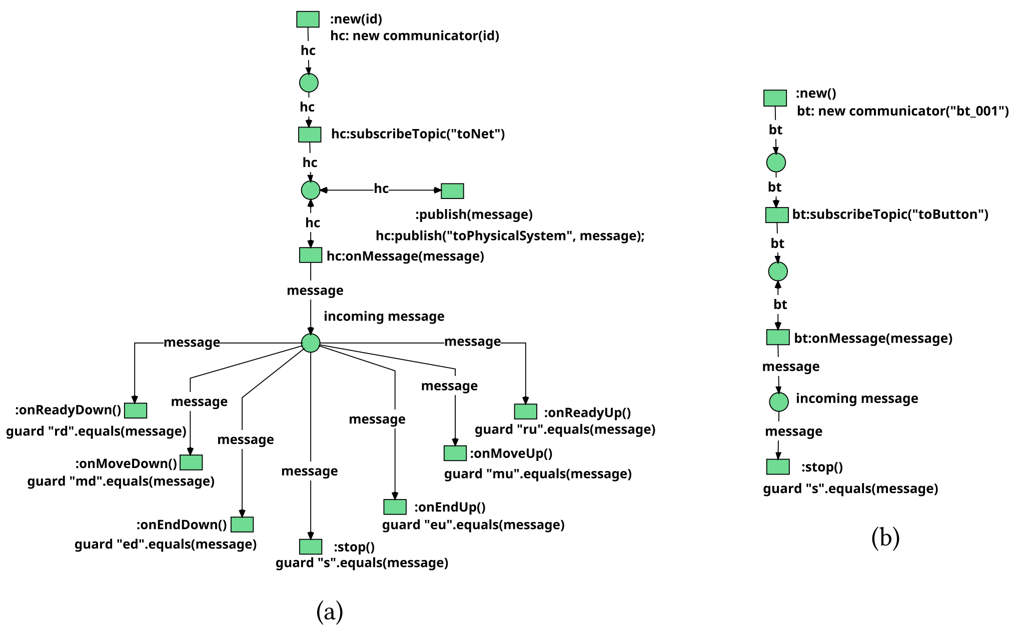
:new(id)
hc: new communicator(id)
hc
hc
hc:subscribeTopic("toNet")
hc
hc
:publish(message)
hc:publish("toPhysicalSystem", message);
hc
hc:onMessage(message)
message
incoming message
message
message
message
message
message
message
message
:onReadyDown()
guard "rd".equals(message)
:onMoveDown()
guard "md".equals(message)
:onEndDown()
guard "ed".equals(message)
:stop()
guard "s".equals(message)
:onEndUp()
guard "eu".equals(message)
:onMoveUp()
guard "mu".equals(message)
:onReadyUp()
guard "ru".equals(message)
(a)
:new()
bt: new communicator("bt_001")
bt
bt
bt:subscribeTopic("toButton")
bt
bt
bt:onMessage(message)
message
incoming message
message
:stop()
guard "s".equals(message)
(b)

Figure 3: Nets (a) `Gate` and (b) `Button`.

When the net `Controller` is instantiated, a unique net identifier is obtained (`holonId`). This identifier is sent to the created net `Gate` instance (`g: new gate(holonId)`), which passes it to the new instance of net `Communicator`, that is stored in variable `hc` (`:new(id); hc: new communicator(id)`) (see top of Fig. 3a). The following transition (in net `Gate`) subscribes to the "toNet" topic, receiving messages sent by the physical device. This net allows messages to be published to the "toPhysicalSystem" topic, which is subscribed to by the physical counterpart, and to receive messages. Whenever the `Communicator` net instance, in `hc`, receives a message (`hc:onMessage(message)`), the transition fires: the `hc` token is not consumed, and the message is put in place `incoming message` to be consumed by one of the following transitions. Guards are used to translate from the text message to the respective transition to be fired, thus establishing synchronous communication between this net and the net where the message is relevant to fire the intended transition.

The net `Button` is analogous to the net `Gate`, with the primary distinction being its inability to publish messages. Instead, it only receives a message indicating the intention to stop.

4.3. Net `Communicator`

The net `Communicator` (see Fig. 4a) is responsible for instantiating the Kotlin class `MqttClient`, responsible for communication via the MQTT protocol and identified as "client", and for creating an instance of net `MqttMessageReceiver` (see Fig. 4b), identified as "receiver", which is then passed on to the "client".

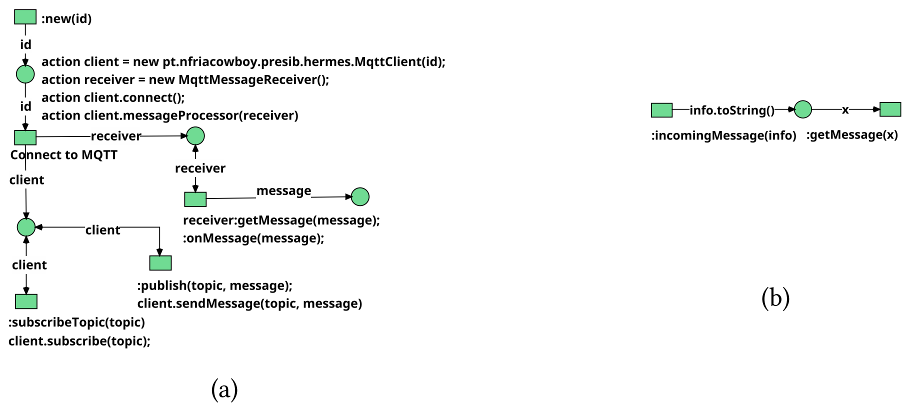
:new(id)
id
action client = new pt.nfriacowboy.presib.hermes.MqttClient(id);
action receiver = new MqttMessageReceiver();
action client.connect();
action client.messageProcessor(receiver)
id
receiver
Connect to MQTT
receiver
client
message
receiver:getMessage(message);
:onMessage(message);
client
client
:publish(topic, message);
client.sendMessage(topic, message)
:subscribeTopic(topic)
client.subscribe(topic);
(a)
info.toString()
x
:incomingMessage(info)
:getMessage(x)
(b)

Figure 4: Nets (a) `Communicator` and (b) `mqtt_message_receiver`.

Whenever the "client" instance receives a message, it is passed to the "receiver" net instance by a "stub" code built according to the Renew tool requirements [2]. The "receiver" instance uses transition synchronization to pass the message to the net `Communicator`. The net `Communicator` also uses synchronization between transitions to enable the message to be transmitted to the network in which it was instantiated: the `Gate` or `Button` network.

The listing below shows the stub code used to enable communication between the mqtt_message_receiver network, represented in Figure 4b, and the Kotlin MqttClient class.

```
class MqttMessageReceiver for net mqtt_message_receiver
implements pt.nfriacowboy.presib.hermes.communication.IMessageProcessor
{
  break void messageReceived(org.eclipse.paho.mqttv5.common.MqttMessage arg0) {
    this:incomingMessage(arg0);
  }
}
```

Although the code is very similar, it is worth noting the line in the class declaration with the reserved word sequence "for net". Renew uses these words to identify which network the stub is applied to. The stub is used by the Java class as if it were another Java class. When it receives a message, the stub's "message received" method is invoked, and a MqttMessage object is passed. The method `MessageReceived` contains code that is interpreted by renew: the "this" refers to the net instance defined by the "for net" in the stub's class declaration followed by the transition that is to be fired, in this case, `incomingMessage`. The MQTT message is passed from one net instance to the other through *synchronous channels* between transitions.

4.4. Observer Pattern

The `MqttClient` class implements the observer pattern [10]. The respective code is partially presented in the listing below. When a topic is subscribed to, an observer is created. When that topic receives a message, the message received method of the stub is executed, and the message is passed as a parameter to the method. This triggers the sequence that sends the messages to the net model. The Kotlin code used in this model is provided by the PRESIB architecture [12].

```
class MqttClient(netID: String) : ICommunicationClient {
    ...
    override fun connect() {...}
    override fun sendMessage(topic: String, message: String) {
        mqttService.sendMesssage(topic, message)
    }
    override fun subscribe(topic: String) {
        subscribersObservers.getOrPut(topic, ::mutableListOf).add {
            result: ReceivedMessage -> this.processor.messageReceived(result.message)
        }
        mqttService.subscribe(topic)
    }
    override fun messageProcessor(processor: IMessageProcessor) { ... }
}
```

5. Conclusions

The presented functional example provides a method and software to support the use of a high-level Petri Net model as the central component of a CPS implementation. This provides a low-code approach, where the Petri Net models' precise semantics is used to define the systems' behavior and the interaction between the different devices. This approach offers practical benefits, as the nets used for simulation and analyzed for correctness can be directly applied in implementation. In fact, the nets become part of the implementation, which not only reduces development time but also minimizes errors, enhancing the efficiency and reliability of the CPS implementation process.

The use of bidirectional asynchronous communication provides added value as it makes the system reactive and dependent on incoming messages. This makes the net model evolution non-autonomous, as needed for proper synchronization with external devices. Moreover, the proposed approach facilitates the integration of external systems with minimal modifications to the physical components themselves. For instance, it paves the way for seamless integration with artificial intelligence-based systems, which can be leveraged to dynamically condition the execution semantics of the Petri net models. Another future step will be the net model deployment outside the Renew tool, which will allow for autonomous and lighter execution. However, using Java with garbage collection still deserves further testing as it will still be a limitation in critical systems.

Acknowledgments

This work was financed by Portuguese Agency FCT – Fundação para a Ciência e Tecnologia, in the framework of project UIDB/00066/2020.

References

[1] A. Bucaioni, A. Cicchetti, F. Ciccozzi, Modelling in low-code development: a multi-vocal systematic review, Software and Systems Modeling 21 (2022) 1959–1981. URL: https://doi.org/10.1007/s10270-021-00964-0. doi:10.1007/s10270-021-00964-0.

[2] O. Kummer, F. Wienberg, M. Duvigneau, L. Cabac, M. Haustermann, D. Mosteller, D. M. Arbeitsbereich, Renew User Guide, Technical Report, University of Hamburg,Department for Informatics, Theoretical Foundations Group, 2022. Available online: https://www2.informatik.uni-hamburg.de/TGI/renew/4.0/renew4.0.pdf (accessed on 2022-11-23).

[3] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, R. Valk, An extensible editor and simulation engine for Petri nets: Renew, in: J. Cortadella, W. Reisig (Eds.), Applications and Theory of Petri Nets 2004. 25th International Conference, ICATPN 2004, Bologna, Italy, June 2004. Proceedings, volume 3099 of *LNCS*, Springer, 2004, pp. 484–493. URL: http://dx.doi.org/10.1007/978-3-540-27793-4_29. doi:10.1007/978-3-540-27793-4_29.

[4] P. Polasek, V. Janousek, M. Ceska, Petri net simulation as a service., PNSE@ Petri Nets 1160 (2014).

[5] F. Pereira, F. Moutinho, A. Costa, J.-P. Barros, R. Campos-Rebelo, L. Gomes, Iopt-tools – from executable models to automatic code generation for embedded controllers development, in: L. Bernardinello, L. Petrucci (Eds.), Application and Theory of Petri Nets and Concurrency, Springer International Publishing, Cham, 2022, pp. 127–138.

[6] T. Meyer, A symmetric petri net model of generic publish-subscribe systems for verification and business process conformance checking, in: PNSE'23: International Workshop on Petri Nets and Software Engineering, June 26–27, 2023, Lisbon, Portugal, volume 3430, 2023, pp. 88–109.

[7] H.-M. Hanisch, A. Lüder, A signal extension for petri nets and its use in controller design, Fundamenta informaticae 41 (2000) 415–431.

[8] M. Jackson, Problem frames: analyzing and structuring software development problems, Addison-Wesley Longman Publishing Co., Inc., USA, 2000.

[9] MQTT.org, MQTT Specification, 2024. URL: https://mqtt.org/mqtt-specification. Accessed on 2024/04/30.

[10] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1 ed., Addison-Wesley Professional, 1994.

[11] R. Valk, Object Petri Nets – Using the Nets-within-Nets Paradigm, in: J. Desel, W. Reisig, G. Rozenberg (Eds.), Advances in Petri Nets: Lectures on Concurrency and Petri Nets, volume 3098 of *LNCS*, Springer, 2004, pp. 819–848. URL: http://dx.doi.org/10.1007/978-3-540-27755-2_23.

[12] N. Fernandes, Modelo gráfico para simulação e controlo do chão de fábrica no contexto da indústria 5.0, 2022. Master's thesis (in portuguese), Polythecnic Institute of Beja. URL: https://hdl.handle.net/20.500.12207/5792.