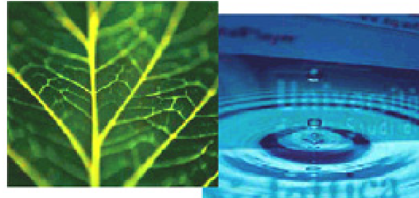
PhD Dissertation



International Doctorate School in Information and Communication Technologies

DISI - University of Trento

MODELS AND SYSTEMS FOR MANAGING SENSOR AND CROWD-ORIENTED PROCESSES

Stefano Tranquillini

Advisors:

Prof. Fabio Casati and Dr. Florian Daniel

Università degli Studi di Trento

March 2014

Acknowledgments

Since I don't want to forget anyone, I'll not write any names here. I'm sure that if you have to be thanked you will find your place.

I would like to thank the people who gave me the opportunity to do the PhD and guided me until the end, losing their time but not wasting it. I will be always grateful to those who gave me the possibility to do a period abroad for the experiences it gave me. I want to thank all the beautiful people, colleagues and friends, who I met during these years: the *mates*, the *freunde*, and all the *amici*. I would like to thank all my friends because they are my friends. I want to thank the people who bear with me and support me every day, one in particular. I want to say *grazie* to my family, especially to my parents that never asked me to find a real job and always supported me in whatever choice I made.

And I would like to thank you.

Stefano

Abstract

Business process modeling refers to the design of business process models, using business processes languages, to orchestrate the work executed by employees, their interaction with external entities, and work items that are necessary to achieve a predefined goal. Model-driven development allows people, generally called modelers, to design also sophisticated application logic using high-level abstractions. Process modeling is typically connected with business, hence, existing process languages focus principally on the support and orchestration of activities executed by employees, or by external entities like web services. However, there is a wide range of other application logics that are process-driven and that can benefit from high-level abstractions to model low-level details.

Our initial research focuses on distributed UIs, which are a distributed type of actors, and then particularly concentrated on Wireless Sensor Networks (WSNs) and crowdsourcing, which are distributed and also autonomous types of actors (they can execute a part of an application logic in an autonomous and isolated fashion). Developing applications in these areas requires a deep knowledge of the field and a non-trivial programming effort; domain experts have to code and orchestrate the logic executed by these actors. Since these applications are highly process-driven, domain experts could take advantage of high-level, process-oriented modeling conventions to design the internal logic of these kinds of applications. However, the intrinsic complexity of these domains and the current state of the art of modeling paradigms make the design and execution of processes for these new actors challenging.

In this dissertation we analyze, design, and present modeling formalisms and systems for managing processes in these contexts. We tackle the challenges of the three areas with an approach that analyzes and extends existing process modeling languages, to enable the design of the processes, and with an architecture, similar for the three focuses, to support the development and execution of processes. Starting from our initial work on the orchestration of distributed UIs, for which we present a modeling language with a set of modeling constructs specific for the UIs, we then present our contribution to WSNs and crowdsourcing domains, which are: a modeling convention for the development of WSN applications, with high-level modeling constructs that abstract the low-level details of the networks; and a modeling paradigm to design processes that are partially executed by a crowd of people. These languages are all equipped with prototypes that contain a modeling

tool to design processes and a runtime environment to support the execution. The impact of this work is not only to the domains we focused on but also to the business process domain as we demonstrate how a process modeling is a flexible and suitable formalism to design processes with very diverging, domain-specific requirements.

Keywords

Process modeling, process languages, process language extensions, BPEL, BPMN, distributed UIs, mashup, Wireless Sensor Networks, crowdsourcing.

Contents

1 Introduction **1**

1.1 Challenges 5

1.2 Methodology 8

1.3 Contributions and Results 12

1.4 Structure of the thesis 17

2 Distributed Orchestration of User Interfaces **19**

2.1 Introduction 20

2.2 State of the Art in Orchestrating Services, People and UIs 23

2.3 Distributed User Interface Orchestration: Definitions, Requirements, and Architecture 27

2.3.1 Requirements and approach 28

2.3.2 Architecture 31

2.4 The Building Blocks: Web Services and UI Components . 35

2.5 The UI Orchestration Meta-Model 41

2.6 Modeling Distributed UI Orchestrations 46

2.6.1 Core UI orchestration design patterns 47

2.6.2 Data transformations 49

2.6.3 Message correlation 49

2.6.4 Graphical layout 51

2.7 Types of UI orchestrations 51

2.7.1 Pure UI synchronizations 52

2.7.2 Pure service orchestrations 55
2.7.3 UI-driven UI orchestrations 56
2.7.4 Process-driven UI orchestrations 58
2.7.5 Complex UI orchestrations 59
2.8 Implementing and Running UI Orchestrations 59
2.9 Lessons Learned 63
2.10 Conclusion 65

3 Process-Based Design and Integration of Wireless Sensor Network Applications 67
3.1 Introduction 68
3.2 Scenario: Convention Center HVAC Management 70
3.3 Relevant Properties of Wireless Sensor Networks 71
3.4 Requirements and Approach 73
3.5 BPMN4WSN 78
3.5.1 Process Logic 79
3.5.2 WSN Task Specification 81
3.6 Prototype 84
3.7 Evaluation of the approach 90
3.8 Discussion and Lessons Learned 94
3.9 Related work 95
3.10 Conclusion 97

4 Modeling and Enacting Flexible Crowdsourcing Processes 99
4.1 Introduction 100
4.2 Crowdsourcing: Concepts and State of the Art 102
4.2.1 Core concepts 102
4.2.2 Crowdsourcing tactics 104
4.2.3 Crowdsourcing processes 106
4.2.4 Problem statement 107

4.3 Modeling and enacting advanced crowdsourcing processes . 109
4.3.1 Requirments 110
4.3.2 Approach 112
4.4 Modeling Crowdsourcing Processes: BPMN4Crowd 116
4.4.1 Crowd task 116
4.4.2 Data transformation 118
4.4.3 Modeling a crowdsourcing process with BPMN4Crowd 123
4.5 The crowd computer 124
4.6 Modeling crowdsourcing tactics 129
4.6.1 Designing Tactics 129
4.6.2 Tactic configurations 138
4.7 Prototype implementation 142
4.7.1 Model 143
4.7.2 Compile 146
4.7.3 Execute 147
4.8 Evaluation 148
4.8.1 Scenario: crowd-based pattern mining 148
4.8.2 Implementation 151
4.8.3 Analysis 154
4.9 Discussion and Lessons Learned 156
4.10 Related work 158
4.11 Conclusion 162

5 Conclusion 165
5.1 Lessons Learned and Limitations 165
5.2 Future work 169
5.3 Final Remarks 172

Bibliography **173**

List of Figures

1.1 The process model that summarizes our work methodology. 10

2.1 A home assistance application integrating both web services and UI components into a process-like orchestration logic. 21

2.2 From design time to runtime: overall system architecture of MarcoFlow. 30

2.3 Graphical rendering and internal logic of a UI component . 36

2.4 Simplified WSDL4UI meta-model (inspired by [23] and extended – via the gray boxes – toward UI components). . . 37

2.5 Example of WSDL/UI description of a UI component. . . 39

2.6 Simplified BPEL4UI meta-model in UML. White classes correspond to standard BPEL constructs [89]; gray classes correspond to constructs for UI and user management. . . 42

2.7 Excerpt of the BPEL4UI home assistance process (new constructs in bold) 45

2.8 Part of the BPEL4UI model of the home assistance process as modeled in the extended Eclipse BPEL editor (the dashed and dotted lines/arrows have been overlaid as a means to explain the model). 46

2.9 The HTML template of the assistant's web page highlighting the empty place holders for UI components. 52

2.10 The four types of (UI) orchestration supported by BPEL4UI and the MarcoFlow system. 53

2.11 The extended Eclipse BPEL editor for developing UI orchestrations at work. 61
2.12 The management console for developers and users allowing them to deploy, instantiate, and participate in UI orchestrations. 62

3.1 Integration of a convention center's BP engine with a WSN for HVAC. 70
3.2 Conceptual model of WSN operations 76
3.3 Architecture 78
3.4 WSN-specific modeling constructs in BPMN4WSN. 79
3.5 The Startup page for the configuration of the scenario. . . 84
3.6 The editor for the creation of the BPMN process. 85
3.7 The editor for the meta abstraction composition. 85
3.8 The HVAC process of the convention center. 86
3.9 The CO_2 and ventilation operations modeled as a script task 87
3.10 The CO_2 and ventilation operations modeled with the parametrized WSN logic constructs [81] 88
3.11 Deployment of the ventilation scenario in Cadiz, Spain. On the left part an overview of the setup and on the right part an actuator with the flap. [25] 90
3.12 Task Completion [33] 93

4.1 The high-level steps of crowdsourcing and the respective actors 103
4.2 The most prominent tactics to crowdsource work 104
4.3 A simple crowdsourcing process in BPMN [67]: the text recognition task is iterated automatically until there are no doubts left about the correct wording 106

4.4 A crowdsourcing process involving different actors (humans, machines and the crowd) and possibly different crowdsourcing tactics 108
4.5 The architecture of our approach. 113
4.6 The visual representation of the layer approach. 115
4.7 The crowd task. 118
4.8 The data transformation notations. 118
4.9 The data and object operations. 119
4.10 The different split data set operations. 120
4.11 The process of the photo scenario modeled with BPMN4Crowd. 123
4.12 Functional architecture of the crowd computer. 125
4.13 The crowd computer task to interact with the API. 129
4.14 The process of the marketplace tactic. 130
4.15 The process of the contest tactic. 132
4.16 The process that models the collection of answers during a contest. 133
4.17 The process of the auction tactic. 135
4.18 The process that models the selection of the winner, in this case it is the worker that first bids less than the bid level. . 136
4.19 The process of the mailing list tactic. 137
4.20 On the left the process that assign to each worker an instance. On the right the process that collects the worker's result and that validates and reward workers. 138
4.21 Processes of four validation configurations. 139
4.22 The implemented architecture of BPMN4Crowd solution. . 143
4.23 The crowdsourcing process of the crowd-based pattern mining pattern mining scenario. 151

4.24 The visual modeling editor with the process of the crowd-based pattern mining scenario. The Figure has the BPMN4Crowd palette and the properties window zoomed. 153

List of Tables

3.1 Exercise steps [33] 92

4.1 List of the API. 128

Chapter 1

Introduction

This dissertation gathers the outcomes of more than three years of research in the context of business process management. During this time the focus of the research evolved following a path driven by new possibilities and applications that emerged in the field of business processes. The research presented in this dissertation builds on background knowledge acquired with the orchestration of distributed UIs, an investigation we started already before enrolling in the PhD program and that inspired the further work on WSNs and crowdsourcing.

Creating applications that interact with distributed UIs, or a WSN, or the crowd today requires a manual and non-trivial effort. Applications have to be programmed to integrate these computational "resources", which we also call *new actors*, and exploit their capabilities. Each resource has its own **characteristics**, yet they share a set of features that make them similar from a technical and conceptual prospective. In particular:

- User interfaces are the means by which people interact with machines. Recently mashups [26] have focused on the integration and coordination of pieces of user interfaces, e.g., a Google map, inside simple web pages. Thus, we have the possibility to combine pieces of UIs from different sources and create new applications. Yet, mashups typically

only focus on single-user applications. With *distributed UIs* we refer to the possibility to orchestrate different pieces of UIs that are deployed and executed in a distribute fashion. This means having the possibility to create an application that supports the collaboration of multiple users via the orchestration of various pieces of UIs deployed on separated pages. Yet, UIs can interact in various fashions. UIs can be synchronized within the same page, as in a conventional mashup, for example, a map that changes the displayed location when the user selects a new address in a list. UIs can be synchronized among different machines, for example, a chat application used by two users. UIs can also trigger part of the process, for example, a user that sends a form for a loan that triggers the process to validate the request.

- A *WSN* is a network of sensors and actuators able to interact with the physical world. WSN applications are used in domains like building automation, control system, remote healthcare, and similar. For example, a WSN can be used to sense the temperature in a room and to open a window. Nodes inside a network have different capabilities, for example a node can sense the temperature while another node can sense the level of CO_2. Sensors and actuators may join or leave a network dynamically, for example, creating a sub set of the network depending on the type of capabilities each node has. Nodes may change their role over time, a node can become a getaway, which is a node in charge of controlling the network and forwarding data to a server, depending on characteristics evaluated at runtime (e.g., charge of the battery). Most importantly, WSNs may form temporary, ad-hoc self-collaborations for specific tasks, such as computing an average temperature out of a large number of sensors.

- *Crowdsourcing* is an activity and business model that is based on

the outsourcing, i.e., externalization, of a unit of work to a crowd of people via an open call for contributions [40]. Crowdsourcing can be used to outsource tasks that require a huge amount of people to be solved. For example, a person can divide a set of one million pictures in small subsets and ask people from a crowd to tag each set of photos. Crowdsourcing shares some characteristics of the WSNs, thus similar problematics. Both are networks of executors: on one side we have sensors and actuators, while on the other there are workers, which are members of the crowd that perform of crowdsourced work. A worker of a task is not known a priori when the application is created, he is selected based on his capabilities and availability (thus if he is willing to execute the task). Crowdsourcing brings also the need for supporting the management of data that each worker has to process. For example, if the task is to tag one million pictures we have to support the splitting of pictures into sets that each worker can process (e.g., set of 10 pictures), and later collect all the results. Additionally, tasks for the crowd can have various execution logics, for example a task can have many instances each of which executed by a single worker (e.g., the tagging of picture where each worker tags 10 pictures), or workers may compete to solve a task (e.g., creating the logo for a company) where workers submit results but only one is selected as winner. A crowd task logic also specifies how the workers are rewarded (most of the worker perform tasks in return of a monetary reward) and how their work is validated, thus how results are checked by the person who crowdsources the work.

With these three new actors we have now distributed actors that have their interactions logic, different capabilities, and that are able to execute a task of work in an rather autonomous fashion (especially WSNs and the crowd), such as a WSN that is able to report the average temperature of a room

by querying various sensors and by computing the average value.

Most of the applications that include UIs, WSN, or the crowd as actors share a common characteristic: their logic is highly process-driven. These applications are of interest for people and companies that already use software instruments for the modeling, execution and management of processes. For example, distribute UIs can be used to coordinate the work of doctors, exchanging information on patients between different wards; a WSN can be used to track items and rise an alarm when a hazard item is placed nearby heat sources; a crowd can tag a large set of picture, or transcribe an audio speech in a very short time. For these reasons we focused the research on extensions of process modeling languages whose *goal* is to provide instruments for modelers, people that design processes, to model and execute processes that interact with the new actors, we called them *extended processes*.

Today, instruments for modeling and executing processes are tailored to different types of actors, namely a machine (e.g., a computer or a web service) or a person (e.g., an employees). With machines or employees as actors of a process there is a different type of modeling. Actors' capabilities are known and described a priori (e.g., what are the operations of a web service), they execute one task at a time in an isolated fashion (without interaction with others), thus one simply invokes them to execute a job. Distributed UIs, WSNs, and the crowd as actors are different from what usually is an actor of a process. Thus, to support applications that leverage these new actors we have to integrate and unify them within process modeling and execution instruments. *Integration* refers to the capacity of these new actors to execute part of the application logic (for example a network of sensors that senses the temperature of a room and reports the average value) whose result triggers the execution of other parts of a bigger process. There is a need to achieve a tight integration between the

actors of a same process. *Unification* refers to the availability of a single specification that allows one to create logics that are executed by a central process and by distributed UIs, or by a WSN, or by the crowd.

1.1 Challenges

To unify and integrate the new actors into business process modeling and execution languages we have several **challenges** to tackle. This dissertation addresses the challenges of each research focus one at a time.

First, we started with the orchestration of *distributed user interfaces*. The *goal* is to coordinate distributed mashups by using a process logic to specify the orchestration of different UIs that are deployed and executed in a distributed fashion. To do so, there is a need to *unify* the modeling language to specify the orchestration of UIs, and to *integrate* the execution of processes with the orchestration of UIs. In particular we have to:

- Understand how to componentize UIs, thus how to abstract UIs capabilities (e.g., a list that can display data, a map that can display a point) in a way that can be used to construct pages and to orchestrate the various pieces of UIs in different pages.

- Provide a method to define the logic of an application, so that a modeler can specify how the UIs interact, for example, how a form in a page updates a list in another page. However, since most of the applications are not only a composition of user interfaces this logic should also give the possibility to interact with other actors, such as web services, to support the creation of wider types of applications.

- Create a language and a tool to enable the modeling of distributed UI applications. We aim at having a tool that allows developers to design and execute their application.

- Develop a runtime system that is able to execute the orchestration of UIs, thus a system that communicates with the UIs, supports the exchange of data, and interacts with the actors of the process.

Next, we focused on the possibility to orchestrate and manage *WSNs* with a process modeling language. As of today, WSN logic is mostly created with programming languages [62] and the integration with business processes is via web-services that expose sensor operations as a set of APIs [7, 78]. In this focus of the research the *goal* is to support the creation and execution of applications whose logic spans across the process and the network of sensors. We do not want only to interact with the sensor, meaning to have the possibility to query the nodes to gather information, but also to program the logic of a WSN, thus create the operations that are later executed by the WSN. For example, we have to find a solution that allows a modeler to specify the logic to sense the CO_2 value of meeting rooms every 10 minutes without asking him to write code for the WSN. This requires to *integrate* the execution of the process that runs in the back-end and of the WSN and to *unify* the modeling language in a way that both logics can be specified within a single modeling solution. We have thus the following challenges:

- To provide an easy access to WSN capabilities. Each node of the network has its own characteristics (e.g., where it is deployed) that for a process modeler could be difficult to grasp but that are important to create an application, for example how to specify a specific sensor inside a specific room. We have to understand how to abstract the characteristics of the network, such as sensor types, node locations and the like, in an easy and understandable way that can be used by a modeler, which is not an expert of WSNs.

- To provide a set of modeling concepts to enable the specification of

sensor network functionality. While a WSN expert could be interested in having control of the low-level functionality of the network, a modeler would like to have a more high-level view. We have then to find a trade-off between the complexity of the sensor network and the high-level modeling language, hiding low-level network details without limiting the possibility to specify WSN functionality.

- To provide a unified modeling for WSN applications that seamlessly embraces the needs of the process applications and of the WSN logic using a single modeling paradigm.
- To create a tool and a language for the creation of process-driven WSN applications.
- To support the process execution. We have to enable the hybrid execution (on the sensor and on the back-end) and we have to support the exchange of information between the participants.

Then, we focus on the possibility of integrating the crowd as an actor into a process. With this focus our *goal* is to give modelers the possibility to create applications whose logic is partially executed by a crowd. To do so, we have to give the modeler the possibility to specify what are the tasks that are executed by the crowd and how the crowd has to execute them (e.g, competing for a task, as for the logo creation; or submitting results for an instance of a huge task, as in the photo tagging). Similarly, we have to provide support for the management of data that the crowd produces and consumes. This requires to *unify* the modeling of the crowd characteristics, the management of data produced and consumed by crowd tasks within what the process already supports, and to *integrate* the execution of crowd tasks and processes. In detail we have to:

- Understand how to specify tasks for the crowd and how these tasks

can be combined into a process logic.

- Define modeling constructs that allows a modeler to specify how each task for the crowd is executed, that is, how the results are collected, how workers are evaluated and rewarded.
- Define a modeling language that is able to orchestrate the crowd and other participants, such as web services. This language has also to support the management of the data that are produced and consumed by the crowd.
- Provide a tool that enables the creation of crowd processes.
- Create a runtime environment that is able to execute the logic of the application, thus that is able to crowdsource a task, collect the results and manage the data to be used by other tasks.

1.2 Methodology

By analyzing the challenges of each focus we can see that they are similar and can be summarized as the need for:

- Abstracting new actors, capabilities and characteristics;
- Defining suitable process modeling constructs to specify tasks for the new actors;
- Enabling the design of the internal execution logic of each actor's tasks;
- Creating a language and tools for the creation of extended processes;
- Supporting the execution of processes that span between all the actors involved in an extended process.

To address the challenges we adopt a similar **approach** in all the three cases. In Figure 1.1 we schematize the work methodology as a process model, which is divided in phases:

- The first phase is the *analysis* of the requirements for the new actor. We analyze its characteristics, its capabilities and functionalities that later have to be abstracted in the language. A reference scenario, taken from a real use-case, is analyzed to understand its composition needs, deriving requirements for the process language. The output of this first phase is a set of requirements, for both the components and the language, that we have to satisfy.
- The second phase is the *creation* of the components for the new actor and of the extensions for the language. The components abstract the actor in a high-level modeling convention, for example a task specific for the WSN that has dedicated parameters. These components are paired with an actor's descriptor, which is a reference document that describes, at a high-level, the capabilities of each actor. For example, an actor's descriptor for a node of a WSN contains information on the location where it is deployed and on the operations it can execute. In this phase we conceptualize the extensions of the language to support the creation of an extended process. The extensions of the language have to support the design of extended processes and the design of the actor's task logic (e.g., the design of a WSN logic that specifies the sensing of CO_2 in a specific room). At the end of this phase we have a conceptual set of extensions.
- The third phase is a first *evaluation* of the extended language. In this phase we implement, on paper, the scenarios and other possible applications that we have identified. At the same time, when possible, we test with the help of experts (e.g., our WSN partners) the effectiveness

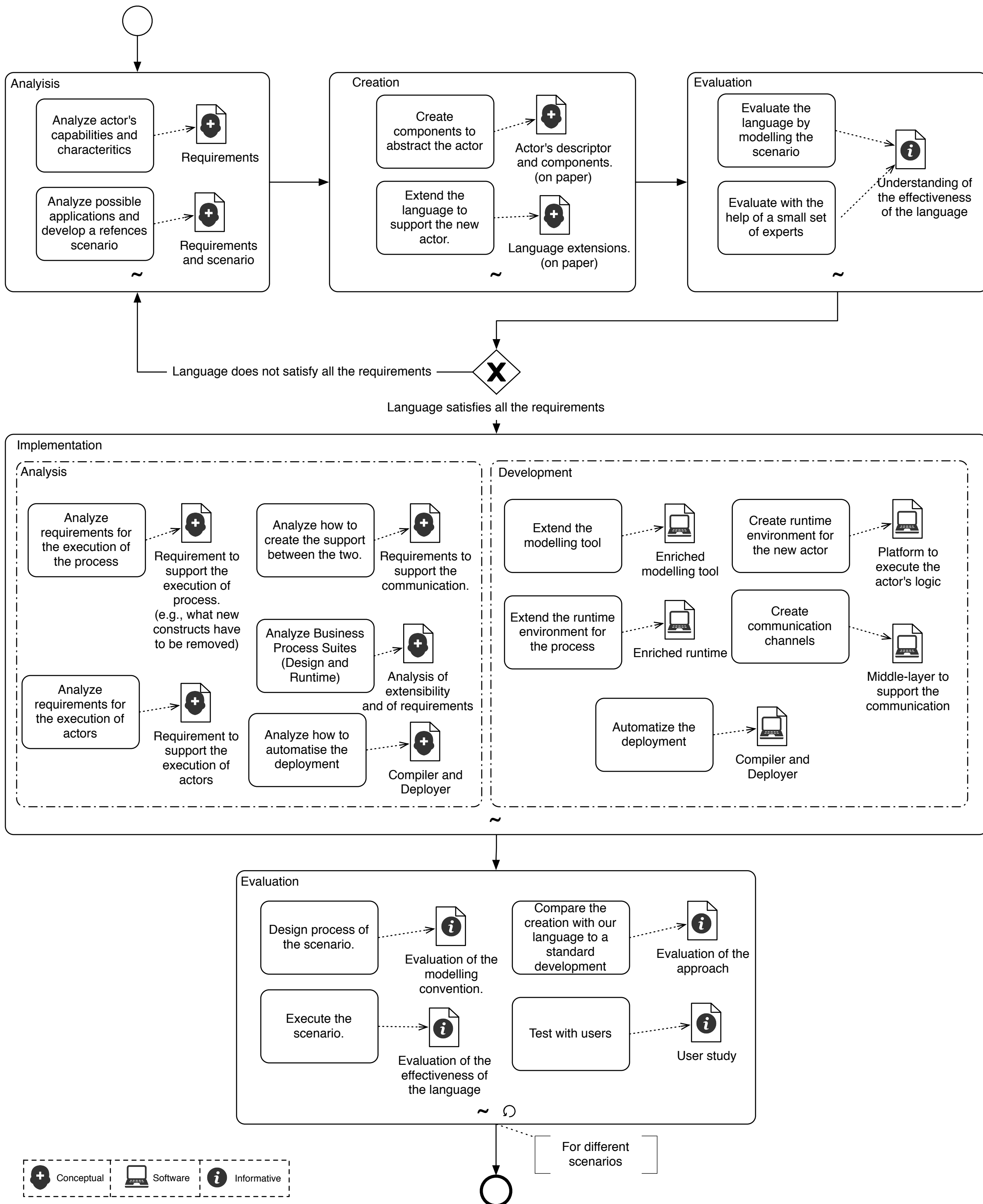


Figure 1.1: The process model that summarizes our work methodology.

of the language. The output of this phase is a first assessment for the extended modeling language. After this phase, if the language misses some characteristics, we re-iterate from the first phase, otherwise we move to the next phase.

- The fourth phase is the *implementation* of the extended language. This phase has several activities that can be grouped in two sub-groups: *analysis* and *development*. In the analysis group we analyze existing tools and software systems to understand what are the limitations we have to overcome and the requirements we have to satisfy to enable the creation and execution of extended processes. As output of the analysis phase we have a set of conceptual outputs that we use in the second group, the development. The *development* group contains the activities for the creation of the modeling tool, of the runtime for the process and for the actors, and of the communication channels. The output of this phase is a toolchain that enable the design and execution of extended processes.

- The last phase is the *evaluation* of the overall solution. In this phase we evaluate the modeling language, with the help of the tools, by modeling and executing the scenario and other application logic. We run user studies, when possible, to measure the effectiveness of the language, and we apply our approach to the development of the scenarios to understand if our language simplifies its development. The output of this phase is a set of informative documents from which we can understand the success of our language. After this phase, if problem arises or if we have new process logic to support, we iterate from the first phase, otherwise the process is completed.

The technical solution, which enables the design and execution of extended processes, is similar in the three focuses. We created a framework

that is divided in three layers (each of which is intended to support a step for the the creation and execution of the processes): design, deployment and execution. The *design* phase enables the creation of processes. In this phase the process modeler creates the extended process by using our modeling tool. In the *deployment* phase the compiler takes the modeled processes and extracts the information needed to create the logic for the new actors. For example, each WSN task is translated into code for sensors. The compiler also establishes the communication channels to enable the exchange of information among the involved actors. The deployer takes the generated code and the process and deploys the former into the runtime for the actors and the latter into the process engine. The *runtime* phase supports the execution of the extended process: the process and the actor run their logic autonomously, interacting when needed.

1.3 Contributions and Results

We applied this work methodology to the three research focuses. We started first with distributed UIs. Later, leveraging on the knowledge acquired from the work on distributed UIs, we focused on WSNs, and, taking advantage from the similarities that WSNs and the crowd have (e.g., they are networks of executors, both can execute tasks in an autonomous fashion), on crowdsourcing. The contributions of the three works can be summarized as follows.

Distributed UIs. For distributed UIs we developed a solution based on WS-BPEL (Web Services Business Process Execution Language)[64], which is the dominant solution to design and execute processes that interact with participants through web service operations. The approach allows a modeler to integrate UI components directly in a process model, to specify their

deployment, and to orchestrate their execution and interaction [27, 28, 29]. In particular, we contribute with:

- An abstraction of UIs that specifies the characteristics of each UI, such as operations and events, in a way that can be used in a process language (we called this WSDL4UI).
- An extension of a WS-BPEL to specify the orchestration of UIs (we called this BPEL4UI). With this extension we enable the modeling of distributed UI orchestration.
- A visual editor to model processes for distributed UIs. Thus we implemented a tool that allows a modeler to use the BPEL4UI language to create their processes.
- A runtime environment to support the execution of the process. This runtime environment also contains the compiler and deployer for the translation of the process and creation of the UI coordination logic.

These contributions found their application in the MarcoFlow project, funded by and jointly conducted with Huawei, China (which also deposited a patent in China [75]). The goal of the project was to solve the problem of UI and service integration in the context of the service-oriented architecture for applications whose user interfaces are distributed over multiple web browsers. In this project we contributed providing the languages and a prototype.

The *publications* for this work are:

- A demo at the 8th international conference on Business Process Management (2010) titled: "MarcoFlow: Modeling, Deploying, and Running Distributed User Interface Orchestrations" [27].
- A paper at the 8th international conference on Business Process Management (2010), which was nominated for the best paper award, titled:

"From People to Services to UI: Distributed Orchestration of User Interfaces" [29].

- An article published in the Information Systems journal, Elsevier (2012), titled "Distributed orchestration of user interfaces" [28].

WSNs. For WSNs, we extended the BPMN (Business Process Modeling and Notation)[67], which is a standard notation for business process modeling, adding specific constructs to support WSN characteristics and to model the execution logic of WSN-dedicated tasks. In details we provide:

- An abstraction of WSN capabilities that allows one to describe, in a high-level fashion, the characteristics of each node of a network. This abstraction also enables the description of pre-defined operations that the network can execute as a sort of library that a modeler can use out of the shelf.
- An extension of BPMN to specify *WSN tasks*, thus the tasks that are executed by the network, as part of a process model.
- A set of modeling constructs for WSNs to allow a modeler to specify the internal logic of WSN tasks. This modeling solution, which we called *WSN task specification*, provides high-level constructs for the modeling of sensing or actuating operations.
- A graphical editor to model processes for WSNs.

These contributions found application in the makeSense project[1] whose goal was to ease the programming of WSNs and the integration with business processes [15, 17, 18, 25, 83]. In this project we tightly collaborated with SAP AG for the creation of the modeling language that we called

[1] `http://www.project-makesense.eu/`

BPMN4WSN and the implementation of the graphical editor. The process translator and process runtime environment was developed by SAP AG. Other partners of the projects provided support for the WSN part, creating the runtime environment for the sensor networks. The modeling solution we proposed was tested with a user study run by professionals from SAP [33]; the overall approach and relative outcomes were tested in a real-world deployment [25].

The *publications* for this work are:

- A poster at the 8th European Conference on Wireless Sensor Networks (EWSN 2011), titled “makeSense: Easy Programming of Integrated Wireless Sensor Networks” [18].

- A demo at the 9th European Conference on Wireless Sensor Networks (EWSN 2012), titled “From Business Process Specifications to Sensor Network Deployments” [17].

- A paper at the 34th ACM/IEEE International Conference on Software Engineering (ICSE), NIER track 2012, titled: “Towards business processes orchestrating the physical enterprise with wireless sensor networks” [15].

- A paper at the 10th International Conference on Business Process Management (2012), titled: “Process-Based Design and Integration of Wireless Sensor Network Applications” [83].

- A paper at the 4th International Workshop on Networks of Cooperating Objects for Smart Cities 2013 (CONET/UBICITEC 2013), titled: “makeSense: Real-world Business Processes through Wireless Sensor Networks” [25].

The crowd. To support the crowdsourcing of tasks, we extended BPMN to enable the modeling and execution of crowd processes, we called the language BPMN4Crowd [50, 82]. In particular we contribute with:

- An extension of the language to support tasks for the crowd. This extension contains a task specific for the crowd and tasks for the management of data created and consumed by the crowd.
- A set of modeling constructs to specify the execution logic of crowd tasks. That is how the crowd executes a task, if the workers compete for the task (e.g., the creation of a logo), or if they contribute to a bigger task (e.g., the tagging of pictures).
- A set of patterns that describe common crowd task logics. These processes are part of the language and describe the execution logic of a crowd task. A modeler can use the patterns to specify the execution logic of each task.
- A visual editor to design crowdsourcing processes based on our language.
- A runtime environment to support the execution of these processes. It is composed of a crowdsourcing platform (crowd computer[2]), which enables the crowdsourcing of tasks, a process engine, which executes the process logic, and a communication mechanism to exchange information between the two.

The outcomes of this focus of the research are part of the BPM4People[3] project, which aims at providing software for the support of Social Business Process Management. In this project we contributed with the process

[2] `http://www.crowdcomputer.org/`
[3] `http://www.bpm4people.org/`

language for crowdsourcing and the implementation of the tool and runtime environment to support the execution.

The *publications* for this work are:

- A paper at the 5th International Workshop on Business Process Management and Social Software(BPMS2 2012), titled "Business Processes for the Crowd Computer" [50].
- An article to be submitted to the ACM Transactions on the Web (TWEB) journal, titled "Modeling and Enacting Flexible Crowdsourcing Processes" [82].

1.4 Structure of the thesis

The rest of this dissertation presents in details each of the three focuses. Chapter 2 presents the work to orchestrate distributed UIs; this chapter is based on the work presented in [28] and illustrates an initial step to orchestrate distribute actors. Chapter 3, based on the work presented in [83], focuses on distributed and autonomous actors and presents the research work to design process-based WSN applications. Chapter 4 focuses on distributed, autonomous, and intelligent actors, the crowd, presenting the research to support the creation of crowd processes; this chapter is based on the work presented in [82][4]. In Chapter 5 we conclude the dissertation with an analysis of the research and lessons learned, highlighting possible future work. All the material of this thesis, plus videos, and prototypes are available online at: `http://phd.stefanotranquillini.me`[5].

[4] not yet published

[5] last time checked February 2014

Chapter 2

Distributed Orchestration of User Interfaces

In this chapter we present our research on the orchestration of distributed UIs. This work is a first step toward the orchestration of actors that are distributed and that interact with processes in a different way compared to classical actors. With this work we enable the development of mashup-like applications that require process support by integrating a process language (BPEL) with the orchestration of distributed UIs. The content of the chapter is an extract of the journal paper [28], which is an extension of the paper presented at the Business Process Management conference in 2010 [29]. In the same conference we also presented a demo [27], which is the implementation of the scenario presented in this chapter. A screen cast of the demo is available online.

2.1 Introduction

Workflow management systems support office automation processes, including the automatic generation of form-based user interfaces (UIs) for executing human tasks in a process. *Service orchestrations* and related languages focus instead on integration at the application level. As such, this technology excels in the reuse of components and services but does not facilitate the development of UI front-ends for supporting human tasks and complex user interaction needs, which is one of the most time consuming tasks in software development [63].

Only recently, *web mashups* [90] have turned lessons learned from data and application integration into lightweight, simple composition approaches featuring a significant innovation: integration at the UI level. Besides web services or data feeds, mashups reuse pieces of UI (e.g., content extracted from web pages or JavaScript UI widgets) and integrate them into a new web page. Mashups, therefore, manifest the need for reuse in UI development and suitable UI component technologies. Interestingly, however, unlike what happened for services, this need has not yet resulted in accepted component-based development models and practices.

This chapter tackles the development of applications that require service composition/process automation logic but that also include human tasks, where humans interact with the system via possibly complex and sophisticated UIs that are tailored to help perform the specific job they want to carry out. In other words, this work targets the development of ***mashup-like applications that require process support***, including applications that require distributed mashups coordinated in real time, and provides design and tool support for professional developers, yielding an original composition paradigm based on web-based UI components and web services.

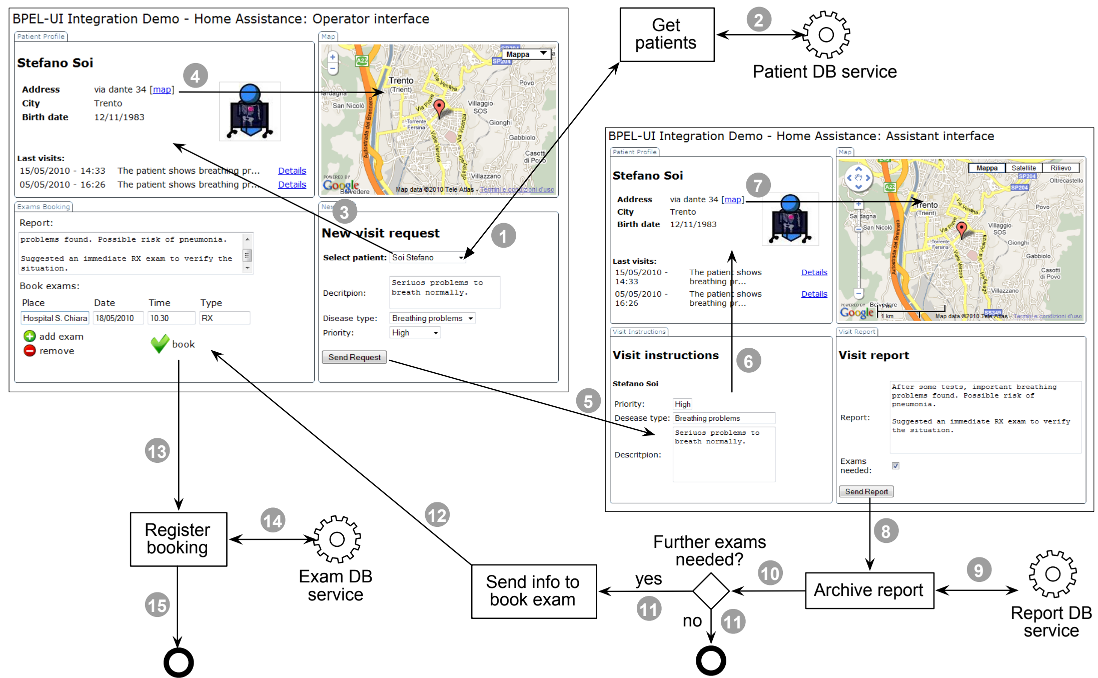


Figure 2.1: A home assistance application integrating both web services and UI components into a process-like orchestration logic.

This class of applications manifests a common need that today is typically fulfilled by developing UIs in ad hoc ways and using and manually configuring a process engine in the back-end for process automation. As an example, consider the ***scenario*** in Figure 2.1: The figure shows a home assistance application for the Province of Trento whose development we want to aid in one of our projects. A patient can ask for the visit of a home assistant (e.g., a paramedic) by calling (via phone) an *operator* of the assistance service. Upon request, the operator inputs the respective details and inspects the patient's data and personal health history in order to provide the assistant with the necessary instructions (steps 1-5). There is always one assistant on duty. The *home assistant* views the description, visits the patient, and files a report about the provided service (steps 6-7). The report is processed by the back-end system and archived (steps 8-9). If no further exams are needed, the process ends (steps 10-11). If exams are

instead needed, the operator books the exam in the local hospital asking confirmation to the patient via phone (steps 12-13). Upon confirmation of the exam booking, the system also archives the booking, which terminates the responsibility of the home assistance service (steps 14-15).

The application in the scenario includes, besides the process logic, two mashup-like, web-based control consoles for the operator and the assistant that are themselves part of the orchestration, need to interact with the process, and are affected by its progress. In addition, the UIs are themselves component-based and created by reusing and combining existing UI components that are instantiated in the users' web browsers (both web pages in Figure 2.1 are composed of four components). The two applications, once instantiated, allow the operator and assistant to manage an individual request for assistance; each new request requires starting a new instance of the application.

In summary, the scenario requires the coordination of the individual actors in the process and the development of the necessary distributed user interface and service orchestration logic. Doing so requires addressing a set of ***challenges*** (each leading to a specific contribution):

1. Understanding how to *componentize UIs* and *compose* them into web applications;

2. Defining a logic that is able to *orchestrate both UIs and web services*;

3. Providing a *language and tool* for implementing distributed UI compositions; and

4. Developing a *runtime environment* that is able to execute distributed UI and service compositions.

In Section 2.2 we introduce the state of the art of the related composition approaches and technologies. In Section 2.3, we derive requirements

from the above scenario and outline the approach we follow in this chapter, including the architecture of our *MarcoFlow* platform that will serve as a guide throughout the rest of the chapter. In Section 2.4, we then introduce the concept of HTML/JavaScript UI component and show how defining a new type of binding allows us to leverage the standard WSDL [20] language to abstractly describe them. We then build on existing composition languages (in particular WS-BPEL [64]) to introduce the notions of UI components, pages, and actors into service compositions (Section 2.5) and explain how such extension can be used to model UI orchestrations (Section 2.6). In Section 2.7 we discuss the different types of UI orchestrations that can be implemented. In Section 2.8, we show how we extended the Eclipse BPEL editor to support design, and we describe how to run UI orchestrations. Finally, in Section 2.9 we report on the lessons we learned with MarcoFlow and conclude the chapter in Section 2.10.

2.2 State of the Art in Orchestrating Services, People and UIs

Workflow or business process management systems are the traditional solution to coordinate people; web services have been integrated over the last decade, while support for UI development is still rather weak. For instance, the Oracle BPEL Proccess Manager (`http://www.oracle.com/technetwork/middleware/bpel`) uses Workflow Services to handle the work-lists of each user and to allow them to perform their tasks. The tool provides two solutions for creating user interfaces: automatic generation, where the tool generates the forms, and custom generation, which enables the modeler to select the template and the parameters to display. Both solutions produce a JSP-based form. Bonita Studio (`http://www.bonitasoft.com`) has an extension of the tool to create

forms. The software allows the developer to use existing form templates; alternatively, forms can be created using a WYSIWYG interface. Forms can be customized by hand and exported as portlets. Similarly, also the tool based on the popular workflow language YAWL [39] and its extension (YAWL4Film [21]) do not go beyond custom or automatically generated web forms (based on the Java Server Faces technology). WebRatio BPM [9] allows the developer to generate WebML [19] web application templates starting from BPMN process models. The templates can then be refined by the developer to equip each page (for task execution) with the necessary data and application functionality, which enables the tool to automatically generate the necessary application code.

All these solutions provide good means to render input and output parameters of tasks as HTML forms, which can either be based on pre-defined form templates or custom forms implemented by the developer. None of the approaches, however, supports the reuse of third-party UIs (e.g., a Google map) as first-class application components and, hence, they are not able to orchestrate them. The synchronization of the two pages in our reference scenario, requiring direct UI-to-UI communications, is thus out of the reach of these tools.

In ***service orchestration*** approaches, such as BPEL [64], there is no support for UI design. Many variations of BPEL have been developed, e.g., aiming at the invocation of REST services [68] or at exposing BPEL processes as REST services [53]. IBM's Sharable Code platform [58] follows a slightly different strategy in the composition of REST and SOAP services and also allows the integration of user interfaces for the Web; UIs are however not provided as components but as ad-hoc Ruby on Rails HTML templates filled at runtime with dynamically generated content.

BPEL4People [3] is an extension of BPEL that introduces the concept of people task as first-class citizen into the orchestration of web services.

The extension is tightly coupled with the WS-HumanTask [2] specification, which focuses on the definition of human tasks, including their properties, behavior and operations used to manipulate them. BPEL4People supports people activities in the form of inline tasks (defined in BPEL4People) or standalone human tasks accessible as web services. In order to control the life cycle of service-enabled human tasks in an interoperable manner, WS-HumanTask also comes with a suitable coordination protocol for human tasks, which is supported by BPEL4People. The two specifications focus on the coordination logic only and do not support the design of the UIs for task execution.

The systematic development of ***web interfaces and applications*** has typically been addressed by the web engineering community by means of model-driven web design approaches. Among the most notable and advanced model-driven web engineering tools we find, for instance, WebRatio [1] and VisualWade [37]. The former is based on a web-specific visual modeling language (WebML), the latter on an object-oriented modeling notation (OO-H). Similar, but less advanced, modeling tools are also available for web modeling languages/methods like Hera [84], OOHDM [74], and UWE [48]. These tools provide expert web programmers with modeling abstractions and automated code generation capabilities for complex web applications based on a hyperlink-based navigation paradigm. WebML has also been extended toward web services [57] and process-based web applications [10]; reuse is however limited to web services and UIs are generated out of dynamically filled HTML templates.

A first approach to ***component-based UI development*** is represented by portals and portlets [80], which explicitly distinguish between UI components (the portlets) and composite applications (the portals). Portlets are full-fledged, pluggable Web application components that generate document markup fragments (e.g., in (X)HTML) that can however

only be reached through the URL of the portal page. A portal server typically allows users to customize composite pages (e.g., to rearrange or show/hide portlets) and provides single sign-on and role-based personalization, but there is no possibility to specify process flows or web service interactions; also the WSRP [65] specification only provides support for accessing remote portlets as web services.

Finally, the ***web mashup*** [90] community has produced a set of so-called mashup tools, which aim at assisting mashup development by means of easy-to-use graphical user interfaces targeted also at non-professional programmers. For instance, Yahoo! Pipes (`http://pipes.yahoo.com`) focuses on data integration via RSS or Atom feeds via a data-flow composition language; UI integration is not supported. Microsoft Popfly (`http://www.popfly.ms`; discontinued since August 2009) provided a graphical user interface for the composition of both data access applications and UI components; service orchestration was not supported. JackBe Presto (`http://www.jackbe.com`) adopts a Pipes-like approach for data mashups and allows a portal-like aggregation of UI widgets (so-called mashlets) visualizing the output of such mashups; there is no synchronization of UI widgets or process logic. IBM QEDWiki (`http://services.alphaworks.ibm.com/qedwiki`) provides a wiki-based (collaborative) mechanism to glue together JavaScript or PHP-based widgets; service composition is not supported. Intel Mash Maker (`http://mashmaker.intel.com`) features a browser plug-in that interprets annotations inside web pages supporting the personalization of web pages with UI widgets; service composition is outside the scope of Mash Maker.

In the mashArt [24] project, we worked on a so-called universal integration approach for UI components and data and application logic services. MashArt comes with a simple editor and a lightweight runtime environment running in the client browser and targets skilled web users. MashArt

aims at simplicity: orchestration of distributed (i.e., multi-browser) applications and complex features like transactions or exception handling are outside its scope. The CRUISe project [69] has similarities with mashArt, especially regarding the componentization of UIs. Yet, is does not support the seamless integration of UI components with service orchestration, i.e., there is no support for complex process logic. CRUISe rather focuses on adaptivity and context-awareness. Finally, the ServFace project [35] aims to support even unskilled web users in composing web services that come with an annotated WSDL description. Annotations are used to automatically generate form-like interfaces for the services, which can be placed onto one or more web pages and used to graphically specify data flows among the form fields. The result is a simple, user-driven web service orchestration. None of these projects, however, supports the coordination of multiple different actors inside a same process.

As this analysis shows, existing development approaches for web-based applications lack an integrated support for service orchestration, component-based UI development, *and* coordination of users, three ingredients that instead are necessary to fully implement applications like the one described in our example scenario.

2.3 Distributed User Interface Orchestration: Definitions, Requirements, and Architecture

If we analyze the home assistance scenario, we see that the envisioned application (as a whole) is highly distributed over the Web: The UIs for the actors participating in the application are composed of UI components, which can be components developed in-house (like the `Patient Profile` component) or sourced from the Web (like the `Map` component); service orchestrations are based on web services. The UI exposes the state of the

application and allows users to interact with the application and to enact service calls. The two applications for the operator and the assistant are instantiated in different web browsers, contributing to the distribution of the overall UI and raising the need for synchronization.

The key idea to approach the coordination of (i) UI components inside web pages, (ii) web services providing data or application logic, and (iii) individual pages, as well as the people interacting with them, is to split the coordination problem into two layers: *intra-page UI synchronization* and *distributed UI synchronization and web service orchestration.* We call an application that is able to manage these two layers in an integrated fashion a ***distributed UI orchestration*** [29].

2.3.1 Requirements and approach

Supporting the development of distributed UI orchestrations is a complex and challenging task. Especially the aim of providing a development approach that is able to cover all development aspects in an *integrated* fashion poses requirements to the whole life cycle of UI orchestrations, in particular, in terms of design, deployment, and execution support.

Indeed, supporting the ***design*** of distributed UI orchestrations requires:

- Defining a new type of component, the *UI component*, which is able to modularize pieces of UI and to abstract their external interfaces. For the description of UI components, we slightly extend WSDL [20], obtaining what we call *WSDL4UI*, a language that is able to deal with the novel technological aspects that characterize UI components by reusing the standard syntax of WSDL.

- Bringing together the needs of *UI synchronization and service orchestration* in one single language. UIs are typically event-based (e.g., user clicks or key strokes), while service invocations are coordinated

via control flows. In this chapter, we show how to extend the standard BPEL [64] language in order to support UIs. We call this extended language *BPEL4UI*.

- Implementing a suitable, *graphical design environment* that allows developers to visually compose services and UI components and to define the grouping of UI components into pages. BPEL comes with graphical editors and ready, off-the-shelf runtime engines that we can reuse. For instance, we extend the Eclipse BPEL editor with UI-specific modeling constructs in order to design UI orchestrations and generate BPEL4UI in output.

Supporting the ***deployment*** of UI orchestrations requires:

- Splitting the BPEL4UI specification into the *two orchestration layers* for intra-page UI synchronization and distributed UI synchronization and web service orchestration. For the former we use a lightweight UI composition logic, which allows specifying how UI components are coordinated in the client browser. For the latter we rely on standard BPEL.
- Providing a set of *auxiliary web services* that are able to mediate communications between the client-side UI composition logic and the BPEL logic. We achieve this layer by automatically generating and deploying a set of web services that manage the UI-to-BPEL and BPEL-to-UI interactions.

Supporting the ***execution*** of UI orchestrations requires:

- Providing a *client-side runtime framework* for UI synchronization that is able to instantiate UI components inside web pages and to propagate events from one component to other components. Events of a UI

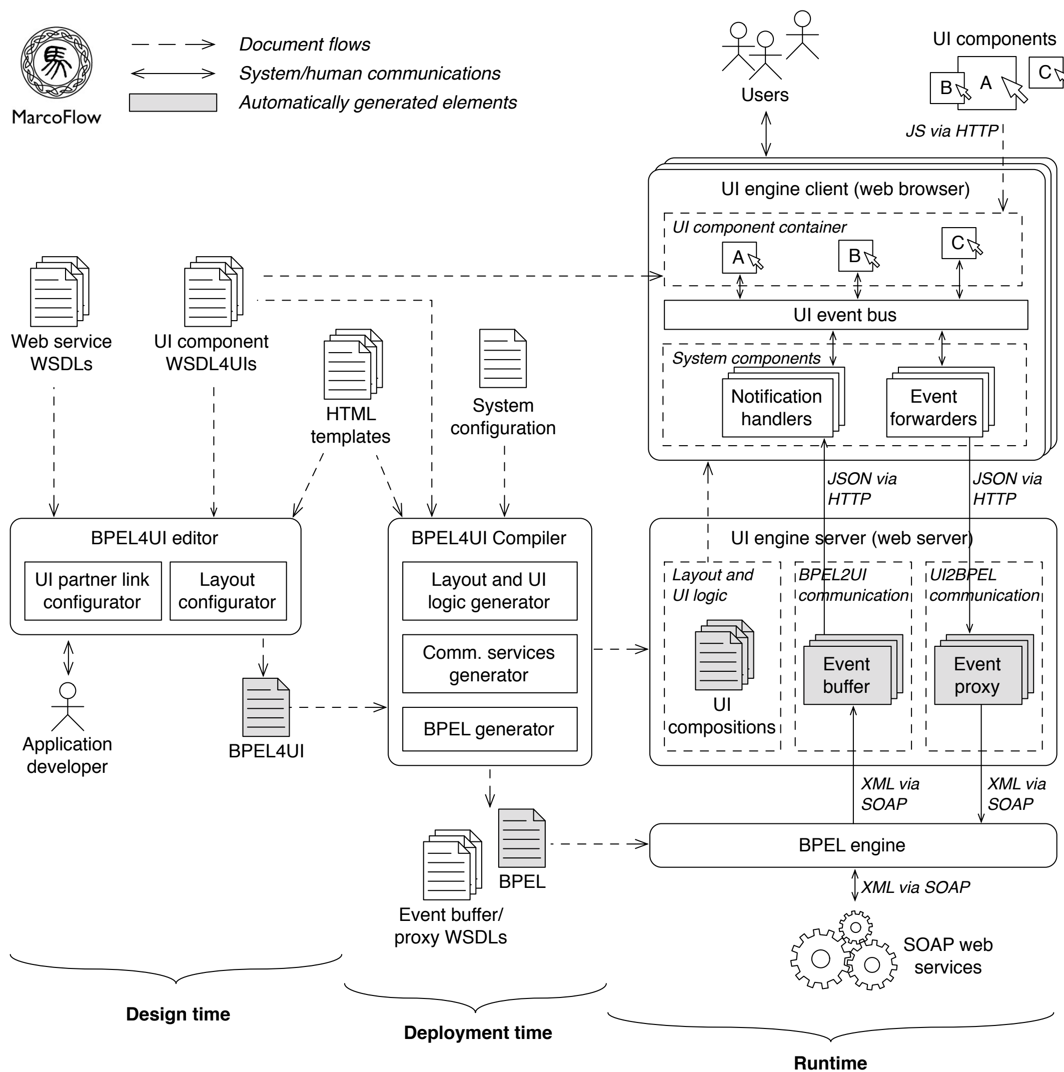


Figure 2.2: From design time to runtime: overall system architecture of MarcoFlow.

component may be propagated to components running in the same web page or in other pages of the application as well as to web services.

- Providing a *communication middleware* layer that is able to run the generated auxiliary web services for UI-to-BPEL and BPEL-to-UI communications. We implement this layer by reusing standard web server technology able to instantiate SOAP and RESTful web services.
- Setting up a *BPEL engine*, in charge of orchestrating web services and distributed UI-to-UI communications, and implementing a *management console* for both developers and participants in UI orchestrations, enabling them to deploy UI orchestrations, to instantiate them, and to participate in them as required.

These requirements and the respective hints to our solution show that the main methodological goals in achieving our UI orchestration approach are (i) relying as much as possible on existing standards (to start from a commonly accepted and known basis), (ii) providing the developer with only few and simple new concepts (to facilitate fast learning), and (iii) implementing a runtime architecture that associates each concern with the right level of abstraction and software tool (to maximize reuse), e.g., UI synchronization is handled in the browser, while service orchestration is delegated to the BPEL engine.

2.3.2 Architecture

A possible system architecture that meets the above requirements is shown in Figure 2.2. It is the architecture of our MarcoFlow platform, which has been developed jointly by Huawei Technologies and the University of Trento. For presentation purposes, we discuss a slightly simplified version and partition its software components into design time, deployment time, and runtime components.

The ***design*** part comprises a BPEL4UI editor, which comes with a UI partner link configurator, enabling the setup of UI components inside a UI orchestration, and a layout configurator, assisting the developer in placing UI components into pages. Starting from a set of web service WSDLs, UI component WSDL4UIs, and HTML templates the application developer graphically models the UI orchestration, and the editor generates a corresponding BPEL4UI specification in output, which contains in a single file the whole logic of the UI orchestration.

The ***deployment*** of a UI orchestration requires translating the BPEL4UI specification into executable formats. In fact, as we will see, BPEL4UI is not immediately executable neither by a standard BPEL engine nor by the UI rendering engine (the so-called UI engine in the right hand side of the figure). This task is achieved by the BPEL4UI compiler, which, starting from the BPEL4UI specification, the set of used HTML templates and UI component WSDL4UIs, and the system configuration of the runtime part of the architecture, generates three kinds of outputs:

1. A set of *communication channels* (to be deployed in the so-called UI engine server), which mediate communications between the UI engine client (the client browser) and the BPEL engine. These channels are crucial in that they resolve the technology conflict inherently present in BPEL4UI specifications: a BPEL engine is not able to talk to JavaScript UI components running inside a client browser, and UI components are not able to interact with the SOAP interface of a BPEL engine. For each UI component in a page, the compiler therefore generates (i) an event proxy that is able to forward events from the client browser to the BPEL engine and (ii) an event buffer that is able to accept events from the BPEL engine and store them on behalf of the UI engine client. The compiler also generates suitable WSDL files for proxies and buffers.

2. A *standard BPEL specification* containing the distributed UI synchronization and web service orchestration logic (see Section 2.6.1). Unlike the BPEL4UI specification, the generated BPEL specification does no longer contain any UI-specific constructs and can therefore be executed by any standards-compliant BPEL engine. This means that all references to UI components in input to the compilation process are rewritten into references to the respective communication channels of the UI components in the UI engine server, also setting the correct, new SOAP endpoints.

3. A set of *UI compositions*[1] (one for each page of the application) consisting of the layout of the page, the list of UI components of the page, the assignment of UI components to place holders, the specification of the intra-page UI synchronization logic (see Section 2.6.1), and a reference to the client-side runtime framework. Interactions with web services or UI components running in other pages are translated into interactions with local system components (the notification handlers and event forwarders), which manage the necessary interaction with the communication channels via suitable RESTful web service calls.

Finally, the BPEL4UI compiler also manages the deployment of the generated artifacts in the respective runtime environments. Specifically, the generated communication channels and the UI compositions are deployed in the UI engine server and the standard BPEL specification is deployed in the BPEL engine.

The ***execution*** of a UI orchestration requires the setting up and coordination of three independent runtime environments: First, the interaction with the users is managed in the client browser by an event-based JavaScript runtime framework that is able to parse the UI composition

[1] Details about the format and logic of these UI compositions can be found in [24].

stored in the UI engine server, to instantiate UI components in their respective place holders, to configure the notification handlers and event forwarders, and to set up the necessary logic ruling the interaction of the components running inside the client browser. While event forwarders are called each time an event is to be sent from the client to the BPEL engine, the notification handlers are active components that periodically poll the event buffers of their UI components on the UI engine server in order to fetch possible events coming from the BPEL engine.

Second, the UI engine server must run the web services implementing the communication channels. In practice we generate standard Java servlets and SOAP web services, which can easily be deployed in a common web server, such as Apache Tomcat. The use of web server technology is mandatory in that we need to be able to accept notifications from the BPEL engine and the UI engine client, which requires the ability of constantly listening. The event buffer is implemented via a simple relational database (in PostgreSQL, `http://www.postgresql.org`) that manages multiple UI components and distinguishes between instances of UI orchestrations by means of a session key that is shared among all UI components participating in a same UI orchestration instance.

Third, running the BPEL process requires a BPEL engine. Our choice to rely on standard BPEL allows us to reuse a common engine without the need for any UI-specific extensions. In our case, we use Apache ODE (`http://ode.apache.org`), which is characterized by a simple deployment procedure for BPEL processes.

We discuss each of the ingredients in the following.

2.4 The Building Blocks: Web Services and UI Components

Orchestrating remote application logic and pieces of UI requires, first of all, understanding the exact nature of the components to be integrated, i.e., web services and UI components.

For the *integration of application logic*, we rely on standard web service technologies, such as ***WSDL-SOAP web services***, i.e., remote web services whose external interface is described in WSDL, which supports interoperability via four message-based types of operations: request-response, notification, one-way, and solicit-response. Most of today's web services of this kind are stateless, meaning that the order of invocation of their operations does not influence the success of the interaction, while there are also stateful services whose interaction requires following a so-called business protocol that describes the interaction patterns supported by the service.

For the *integration of UI*, we rely instead on ***JavaScript/HTML UI components***, which are simple, stand-alone web applications that can be instantiated and run inside any common web browser [24]. Figure 2.3 illustrates an example of UI component (the `Patient Profile` UI component of our reference scenario), along with an excerpt of its JavaScript code. The figure shows that, unlike web services, UI components are characterized by:

- A ***user interface***. UI components can be instantiated inside a web browser and can be accessed and navigated by a user via standard HTML. The UI allows the user to interactively inspect and alter the content of the component, just like in regular web applications. UI components are therefore stateful, and the component's navigation features replace the business protocol needed for services.

- ***Events***. Interacting with the UI generates system events (e.g., mouse

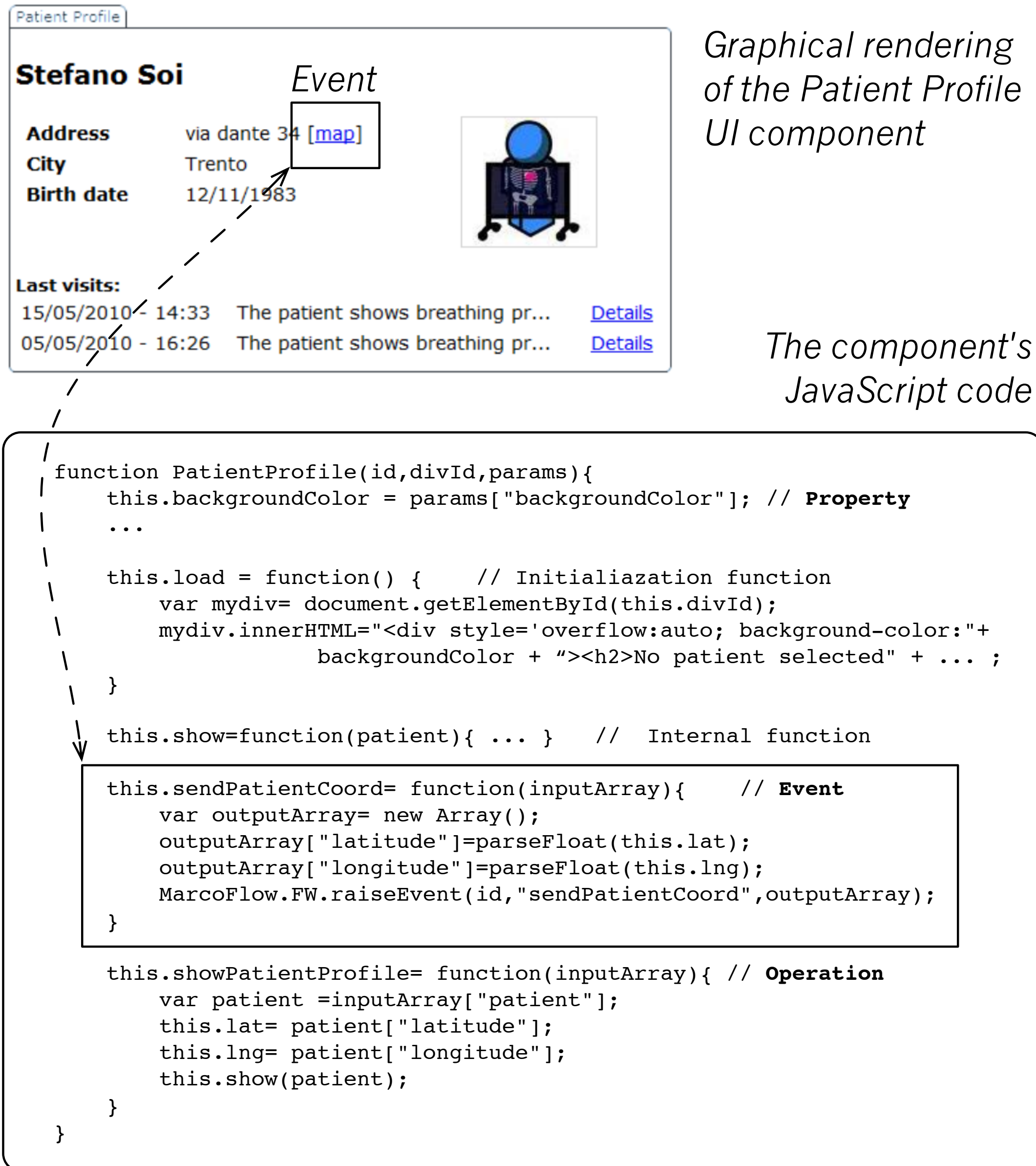
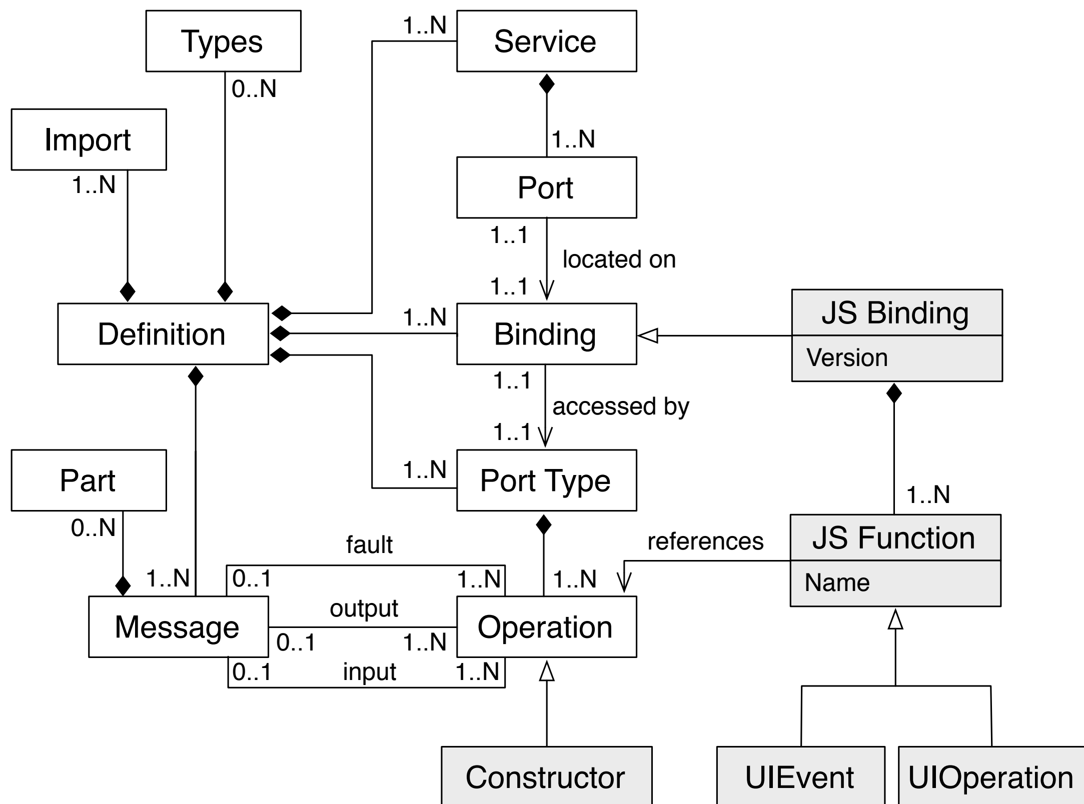


Figure 2.3: Graphical rendering and internal logic of a UI component



WSDL4UI conventions:
(1) All *Operations* are either *UIOperations*, *UIEvents*, or a *Constructor*.
(2) *UIOperations* only have inputs.
(3) *UIEvents* only have outputs.
(4) The *Constructor* is unique and has only inputs.
(5) The service's *port address* points to the JavaScript class of the UI component.

Figure 2.4: Simplified WSDL4UI meta-model (inspired by [23] and extended – via the gray boxes – toward UI components).

clicks) in the browser used to manage the update of contents. Some events may be exposed as component events, in order to communicate state changes. For instance, a click on the "map" link in Figure 2.3 launches a `sendPatientCoord` event.

- ***Operations***. Operations enact state changes from the outside. Typically, we can map the event of one component to the operation of another component in order to synchronize the components' state (so that they show related information).

- ***Properties***. The graphical setup of a component may require the setting of constructor parameters, e.g., to align background colors or set other style properties.

In order to make UI components accessible to BPEL, each component must be equipped with a descriptor that describes its events, operations, and properties in terms of WSDL operations. As already anticipated in the previous section, doing so requires extending the standard WSDL description logic, i.e., its meta-model, from web services to UI components. The result of this extension is called ***WSDL4UI***. Figure 2.4 illustrates its meta-model, from which we can see that the extension toward UI components occurs via two different techniques:

1. First, we introduce a set of *conventions* of how the abstract WSDL constructs can be used to describe UI components. The properties of the UI component are encapsulated by means of a dedicated *constructor* operation that can be used to set properties at instantiation time of the component. Next, all operations specified in the description are either UIOperations, UIEvents, or a constructor. *UIOperations* have only inputs; *UIEvents* have only outputs; the constructor is an operation. Finally, the *port address* of the described service corresponds

```
 1 <?xml version="1.0" encoding="utf-8"?>
 2 <wsdl:definitions name="PatientProfile" targetNamespace="http://www.unitn.it/
 3 JS/Patient" ... >
 4   <!-- types definition -->
 5   ...
 6   <!-- massages definition -->
 7   ...
 8   <wsdl:portType name="PatientPortType">
 9       <wsdl:operation name="constructor">
10            <wsdl:input message="tns:constructorMessage"/>
11        </wsdl:operation>
12        <wsdl:operation name="ShowPatientProfile">
13            <wsdl:input message="tns:ShowPatientProfileMessage"></wsdl:input>
14            </wsdl:operation>
15        <wsdl:operation name="SendPatientCoord">
16            <wsdl:output message="tns:SendPatientCoordMessage"></wsdl:output>
17        </wsdl:operation>
18    </wsdl:portType>
19
20    <wsdl:binding name="PatientJS" type="tns:PatientPortType">
21        <js:binding version="1.0" />
22        <wsdl:operation name="constructor">
23            <js:operation jsFunction="load" />
24        </wsdl:operation>
25        <wsdl:operation name="ShowPatientProfile">
26            <js:operation jsFunction="showPatientProfile" />
27        </wsdl:operation>
28        <wsdl:operation name="SendPatientCoord">
29            <js:event jsFunction="sendPatientCoord" />
30        </wsdl:operation>
31    </wsdl:binding>
32
33    <wsdl:service name="PatientProfile">
34        <wsdl:port name="PatientJS" binding="tns:PatientJS">
35            <soap:address location="http://www.unitn.it/JS/Patient.js" />
36        </wsdl:port>
37    </wsdl:service>
38 </wsdl:definitions>
```

Figure 2.5: Example of WSDL/UI description of a UI component.

to the URL at which the actual UI component can be downloaded for instantiation (in form of a JavaScript file).

2. Second, we introduce a new *JavaScript binding* that allows us to associate to each abstractly defined operation a JavaScript function of the UI component. Doing so enables the client-side runtime environment (the UI engine client) to parse the WSDL4UI description of a component, to invoke its constructor, and to correctly access events and operations in JavaScript.

Only WSDL files that conform to these rules are considered correct WSDL4UI descriptors of UI components. Figure 2.5, for instance, shows the descriptor of the `Patient Profile` UI component. Its interface is characterized by three WSDL operations: `ShowPatientProfile`, `SendPatientCoord`, and `constructor` (lines 9-17), corresponding, respectively, to a UIOperation, to a UIEvent and to the component's custructor, as stated in the JavaScript binding (lines 20-31). In the binding, there are also specified, through the related `jsFunction` attributes (e.g., line 23), the actual JavaScript functions implementing the operations, which are contained in the file located at the URL defined in the service's port address (line 35).

For the BPEL engine, in order to interact with a component, the BPEL4UI compiler introduced in Section 2.3.2 generates a respective event buffer and event proxy for the UI engine server and equips them with two standard WSDL descriptors. These descriptors contain the abstract service description as defined in the WSDL4UI file (the event buffer contains all operations of the UI components, the event proxy all events), yet their port addresses point to the newly generated services and their JavaScript binding is turned into a SOAP binding.

2.5 The UI Orchestration Meta-Model

Starting from web services and UI components, developing a UI orchestration requires modeling two fundamental aspects: (i) the *interaction logic* that rules the passing of data among UI components and web services and (ii) the *graphical layout* of the final application. Supporting these tasks in service orchestration languages (like BPEL) requires extending the expressive power of the languages with UI-specific constructs.

Figure 2.6 shows the simplified meta-model of BPEL4UI, addressing these two concerns. Specifically, the figure details all the new modeling constructs necessary to specify UI orchestrations (gray-shaded) and omits details of the standard BPEL language, which are reused as is by BPEL4UI (a detailed meta-model for BPEL can be found, for instance, in [89]). The code snippet in Figure 2.7 exemplifies the syntax that we use, in order to express the novel concepts in BPEL4UI.

In terms of standard BPEL [64], a UI orchestration is a *process* that is composed of a set of associated *activities* (e.g., sequence, flow, if, assign, validate, or similar), *variables* (to store intermediate processing results), *message exchanges*, *correlation sets* (to correlate messages in conversations), and *fault handlers*. The services or UI components integrated by a process are declared by means of so-called *partner links*, while *partner link types* define the roles played by each of the services or UI components in the conversation and the port types specifying the operations and messages supported by each service or component. There can be multiple partner links for each partner link type.

Modeling UI-specific aspects requires instead introducing a set of new constructs that are not yet supported by BPEL. The constructs, illustrated in Figure 2.6, are:

- ***UI type***: The introduction of UI components into service composi-

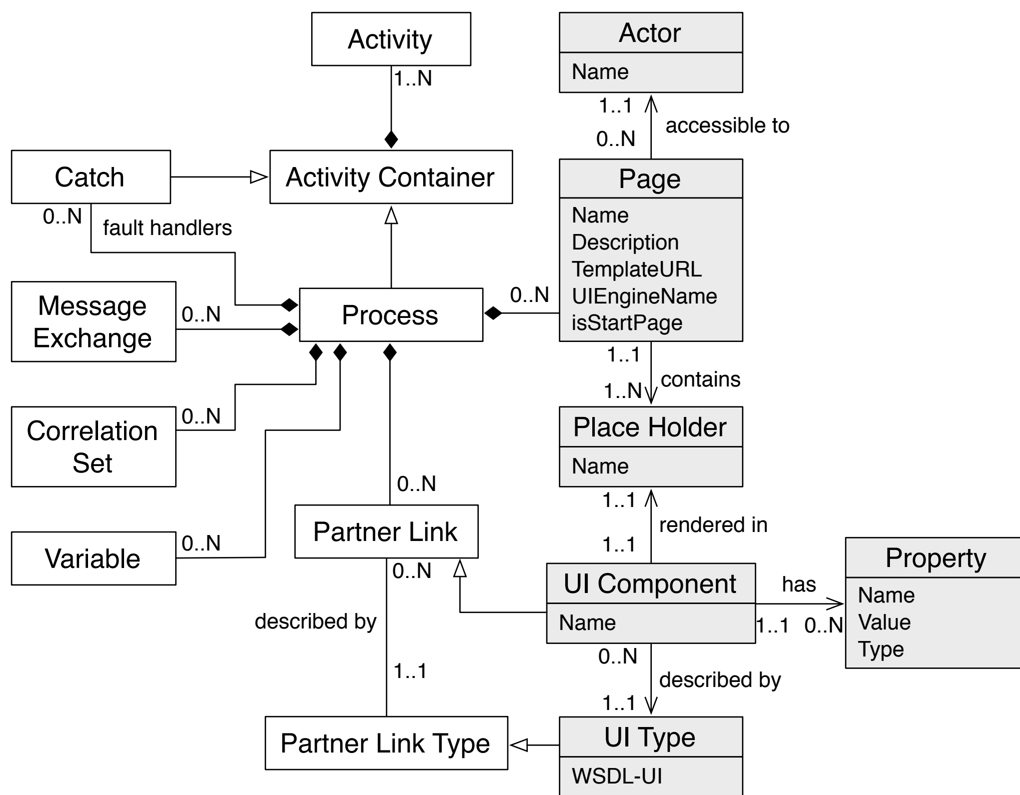


Figure 2.6: Simplified BPEL4UI meta-model in UML. White classes correspond to standard BPEL constructs [89]; gray classes correspond to constructs for UI and user management.

tions asks for a new kind of partner link type. Although syntactically there is no difference between web services and UI components (the JavaScript binding introduced into WSDL4UI comes into play only at runtime), it is important to distinguish between services and UI components as (i) their semantics and, hence, their usage in the model will be different from that of standard web services, and (ii) the UI orchestration editor must be aware of whether an object manipulated by the developers is a web service or a UI component, in order to support the setting of UI-specific properties.

As exemplified in Figure 2.7, we specify the new partner link type like a standard web service type (lines 7-10). In order to reflect the events and operations of the UI component, we distinguish the two roles. Lines 1-5 define the necessary name spaces and import the WSDL4UI descriptor of the UI component.

- ***Page***: The distributed UI of the overall application consists of one or more web pages, which can host instances of UI components. Pages have a *name*, a *description*, a reference to the pages' *layout template*, the name of the *UI engine* they will run on, and an indication of whether they are a *start page* of the application or not (as we will see in Section 2.7, inside a process model, not all pages allow the correct instantiation of the process).

  The code lines 13-20 in Figure 2.7 show the definition of a page called "operator", along with its layout template and the name of the UI engine on which the page will be deployed; the page is a start page for the process.

- ***Place holder***: Each page comes with a set of place holders, which are empty areas inside the layout template that can be used for the graphical rendering of UI components. Place holders are identified by

a unique *name*, which can be used to associate UI components.

Place holders are associated with page definitions and specified as sub-elements, as shown in lines 16-19 in Figure 2.7.

- ***UI component***: UI types can be instantiated as UI components. For instance, there may be one UI type but two different instances of the type running in two different web pages. Declaring a UI component in a BPEL4UI model leads to the creation of an instance of the UI component in one of the pages of the application. Each component has a unique *name*.

  We specify UI component partner links by extending the standard partner link definition of BPEL with three new attributes, i.e., *isUiComponent*, *pageName*, and *placeHolderName*. Lines 25-32 in Figure 2.7 show how to declare the `Patient Profile` component of our example scenario.

- ***Property***: As we have seen in the previous section, UI components may have a constructor that allows one to set configuration properties. Therefore, each UI component may have a set of associated properties than can be parsed at instantiation time of the component. We use simple *name-value* pairs to store constructor parameters.

  Properties extend the definition of UI component link types by adding property sub-elements to the partner link definition, one for each constructor parameter, as shown in lines 30-31 in Figure 2.7.

- ***Actor***: In order to coordinate the people in a process, pages of the application can be associated with individual actors, i.e., humans, which are then allowed to access the page and to interact with the UI orchestration via the UI components rendered in the page. As for now, we simply associate static actors to pages (using their *names*); yet,

```
1 <bpel:process name="HomeAssistance" targetNamespace="http://www.unitn.it/
2 example/HomeAssistance" xmlns:wsdl6="http://www.unitnt.it/JS/Patient" ...>
3  <bpel:import namespace="http://www.unitnt.it/JS/Patient"
4                location="Patient.wsdl" importType="http://
5                schemas.xmlsoap.org/wsdl/" />
6  ...
7  <bpel:partnerLinkType name="PatientPL">
8      <bpel:role name="receive" portType="wsdl6:PatientPortTypeReceive"/>
9      <bpel:role name="invoke" portType="wsdl6:PatientPortTypeInvoke"/>
10 </bpel:partnerLinkType>
11 ...
12 <bpel4ui:pages>
13     <bpel4ui:page name="operator" templateURL="operator.html"
14                uiEngineName="HAEngine" actorName="SteS"
15                description="the operator page" isStartPage="true" >
16         <bpel4ui:placeHolder name="marcoflow-top-left" />
17         <bpel4ui:placeHolder name="marcoflow-top-right" />
18         <bpel4ui:placeHolder name="marcoflow-bottom-left" />
19         <bpel4ui:placeHolder name="marcoflow-bottom-right" />
20     </bpel4ui:page>
21     ...
22 </bpel4ui:pages>
23
24 <bpel:partnerLinks>
25     <bpel:partnerLink name="PatientProfileUI_operator"
26                       partnerLinkType="tns:PatientPL"
27                       myRole="receive" partnerRole="invoke"
28                       isUiComponent="yes" pageName="operator"
29                       placeholderName="marcoflow-top-left">
30         <bpel4ui:property name="backgroundColor" type="xsd:string"
31                        value="white" />
32     </bpel:partnerLink>
33     ...
34 </bpel:partnerLinks>
35
36 <!-- orchestration logic definition -->
37 ...
38 </bpel:process>
```

Figure 2.7: Excerpt of the BPEL4UI home assistance process (new constructs in bold)

actors can easily be assigned also dynamically at deployment time or at runtime by associating roles instead of actors and using a suitable user management system.

Actors are simply added to page definitions by means of the *actor-Name* attribute, as highlighted in line 14 in Figure 2.7.

The addition of these new concepts to BPEL turns the service orchestration language into a language that, in addition to service invocation logic, is also able to specify the organization of an application's UI and its distribution over multiple servers and actors. Our goal in doing so was to

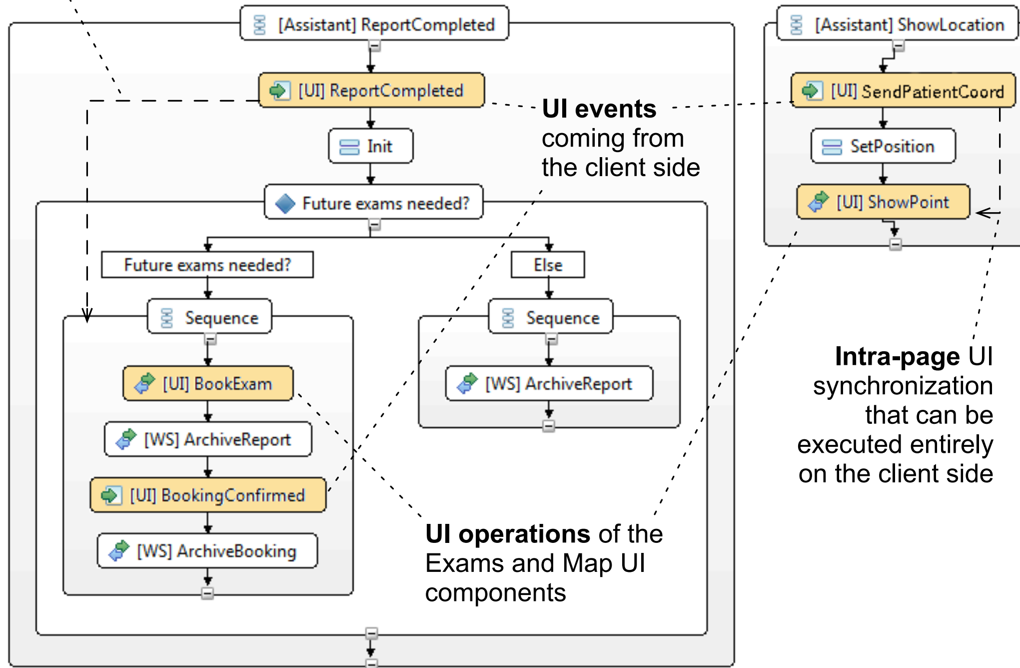


Figure 2.8: Part of the BPEL4UI model of the home assistance process as modeled in the extended Eclipse BPEL editor (the dashed and dotted lines/arrows have been overlaid as a means to explain the model).

keep the number of new concepts as small as possible, while providing a fully operational specification language for UI orchestrations.

2.6 Modeling Distributed UI Orchestrations

The code example in Figure 2.7 shows that the UI-specific modeling constructs have a very limited impact on the syntax of BPEL and are mostly concerned with the abstract specification of the layout and the declaration of UI partner links. The actual composition logic, instead, relies exclusively on standard BPEL constructs. Yet, since UI components are different from web services (e.g., it is important to know in which page they are

running), modeling UI orchestrations requires a profound understanding of the necessary modeling constructs and their semantics. In particular, it is important to understand the effect that individual modeling patterns have on the execution of the final application, i.e., the semantics of the patterns, and which other modeling tasks (data transformations, message correlations, and layout design) are necessary to fully specify a working UI orchestration.

2.6.1 Core UI orchestration design patterns

The first step toward this understanding is mastering the core design patterns that characterize UI orchestrations. As hinted at in Section 2.3 and illustrated in Figure 2.8, we distinguish three main design patterns:

- ***Intra-page UI synchronization***: The small model block (a BPEL *sequence* construct) in the right part of Figure 2.8 shows the internals of step 7 in Figure 2.1. When the assistant clicks on the "map" link, the patient's address is shown on the Google map. In BPEL terms, we receive a message from the `Patient Profile` UI component (the event) and forward it to the operation of the `Map` component, both running inside the web page of the assistant. The pattern, hence, implements a so-called *intra-page UI synchronization*, i.e., a synchronization of UI components that run inside a same page. From a runtime point of view, this kind of UI synchronization can be performed entirely on the client side without requiring support from the BPEL engine.
- ***Distributed UI synchronization***: The bigger model block (again a BPEL *sequence* construct) in the left part of the figure, instead, contains a *distributed UI synchronization* that cannot be executed on the client side only, as the two UI components involved in the com-

munication (`Visit Report` and `Exams Booking`) run in different web pages. The event generated upon submission of a new report is processed by the BPEL engine, which then decides whether an additional exam needs to be booked by the operator or not. As such, the BPEL engine manages two independent concerns, i.e., the forwarding of the event from one UI component to another and the evaluation of the condition, of which only the former is necessary to implement a distributed UI synchronization pattern. The execution of a distributed UI synchronization pattern always requires the cooperation of both the BPEL engine and the client-side runtime environment.

- ***Service orchestration***: The distributed UI synchronization also involves the orchestration of the `Report DB` and `Exam DB` web services, as well as some BPEL flow control constructs. In fact, the modeled logic checks whether the report expresses the need for further exams or not. In either case, the further processing of the report involves the invocation of either one or both the web services, in order to correctly terminate the handling of a visit request. The pure invocation of web services represents a service invocation pattern, whose execution can be entirely managed by the BPEL engine without requiring support from the client-side runtime environment.

The BPEL4UI excerpt in Figure 2.8 shows that, when modeling a UI orchestration, it is important to keep in mind who communicates with whom and which UI component will be rendered where. Depending on these two considerations, the modeled composition logic will either be executed on the client side, in the BPEL engine, or in both layers. For instance, it suffices to associate the `Map` component with a different page, in order to turn the intra-page UI synchronization in the right hand side of Figure 2.8 into a distributed UI synchronization and, hence, to require support from

the BPEL engine.

2.6.2 Data transformations

When composing services or UI components, it is not enough to model the communication flow only. An important and time-consuming aspect is that of transforming the data passed from one component to another. With BPEL4UI we support all data transformation options provided by BPEL by means of its `Assign` construct. This allows us to leverage on technologies, such as XPath, XQuery, XSLT, or Java, for the implementation of also very complex data transformations.

Yet, it is important to keep in mind that the *type* of data transformation may affect the logic of the UI orchestration: For instance, if the `SetPosition` activity in the top-right corner of Figure 2.8 does not transform data at all or only performs simple parameter mappings (with the BPEL `Copy` construct), we fully support the execution of the intra-page UI synchronization in the client browser. If instead a more complex transformation is needed, we rely on the BPEL engine to perform it.

The reason for this choice is that UI synchronization typically requires the exchange of only simple data (e.g., parameter-value pairs), which do not require complex transformation capabilities like the ones we need when interacting with web services. Supporting only simple parameter-parameter mappings on the client side allows us to keep the client-side runtime framework as lightweight as possible, without however giving up any of BPEL's data transformation capabilities.

2.6.3 Message correlation

Independently of the format of data, UI orchestrations may require a careful design of the messages used in the orchestration and of how these must

be *correlated*, in order to enable the runtime environment to dispatch each message to its correct UI orchestration instance. In fact, just like in conventional workflow or service orchestration engines, there may be multiple instances of UI orchestrations running concurrently in a same BPEL/UI engine. Message correlation is required in all those cases where the orchestration involves *multiple entry points* into the orchestration logic (e.g., callbacks from external web services or a condition that requires input from two different events).

If we look at our modeling example in Figure 2.8, we see that the intra-page UI synchronization in the top-right corner does not involve multiple entry points. It is therefore not necessary to implement any correlation logic in BPEL4UI, in order to propagate the `SendPatientCoord` event from the `Patient Profile` UI component to the `ShowPoint` operation of the `Map` UI component. Since both UI components involved in this synchronization run inside the same web page and, therefore, there is no ambiguity regarding which instance of the `Map` UI component is the target of the `SendPatientCoord` event. In Section 2.7, we will see that this is not always the case.

The distributed UI synchronization, instead, involves two UI events from two different actors and, hence, different pages: `ReportCompleted` and `BookingConfirmed`. In this case, it is necessary to configure a so-called *correlation set* (in BPEL terminology) that allows the BPEL engine to understand when two instances of those events belong to a same process instance. In the example in Figure 2.8, we use `UIOrchestrationID` (provided by the UI engine) and `VisitID` (part of the report) as correlation set.

2.6.4 Graphical layout

Finally, the complete definition of a UI orchestration also requires the design of suitable *HTML templates* and the assignment of UI components to their place holders inside the pages. As our goal is the development of an enabling middleware layer for UI orchestrations, for the layout templates we rely on standard web design instruments and technologies (e.g., Adobe Dreamweaver). The only requirement the templates must satisfy is that they provide place holders in the form of HTML DIV elements that can be indexed via standard HTML identifiers following a predefined naming convention: `<div id="marcoflow-..."></div>`.

Figure 2.9, for instance, depicts the empty HTML template of the assistant's web page, whose filled version we have already seen in Figure 2.1. The template is a simple HTML page with a page title and the four uniquely identified placeholders to be filled with UI components at runtime. Differently from dynamic HTML and most of the approaches discussed in Section 2.2, in which the template typically also contains the formatting logic for the data to be rendered inside the place holders, in our case the template only identifies the location of the UI components; the rendering of content is then managed autonomously by the UI components.

Once all HTML templates for all pages in the UI orchestration are defined, the definition of the pages and the association of UI partner links with place holders therein proceeds as exemplified in Section 2.6.

2.7 Types of UI orchestrations

So far we have seen how BPEL4UI supports the development of distributed UI orchestrations. Yet, developing correct UI orchestrations is still a non-trivial task, in that the distribution of UI synchronizations and service orchestrations over two different runtime engines (the UI engine and the

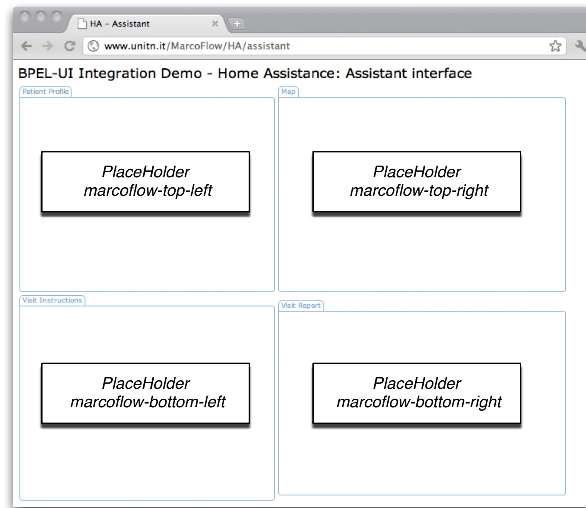


Figure 2.9: The HTML template of the assistant's web page highlighting the empty place holders for UI components.

BPEL engine) complicates the instantiation logic of distributed UI orchestrations, an aspect that developers should understand thoroughly. As illustrated in Figure 2.10, we identify four main types of UI orchestrations that can be implemented by means of the core patterns described in Section 2.6.1, i.e., *pure UI synchronizations*, *pure service orchestrations*, *UI-driven UI orchestrations*, and *process-driven UI orchestrations*. The developer needs to master these configurations if he doesn't want to encounter unexpected behaviors or errors at runtime. We discuss each of these configurations next.

2.7.1 Pure UI synchronizations

From a UI point of view, the basic type of UI orchestration is represented by applications that involve *UI components only* and, hence, exclusively focus on the synchronization of UIs via events. Typical examples of this type of UI orchestration are UI-based mashups, portlets/portals, applications that integrate widgets/gadgets, or similar component-based UI applications.

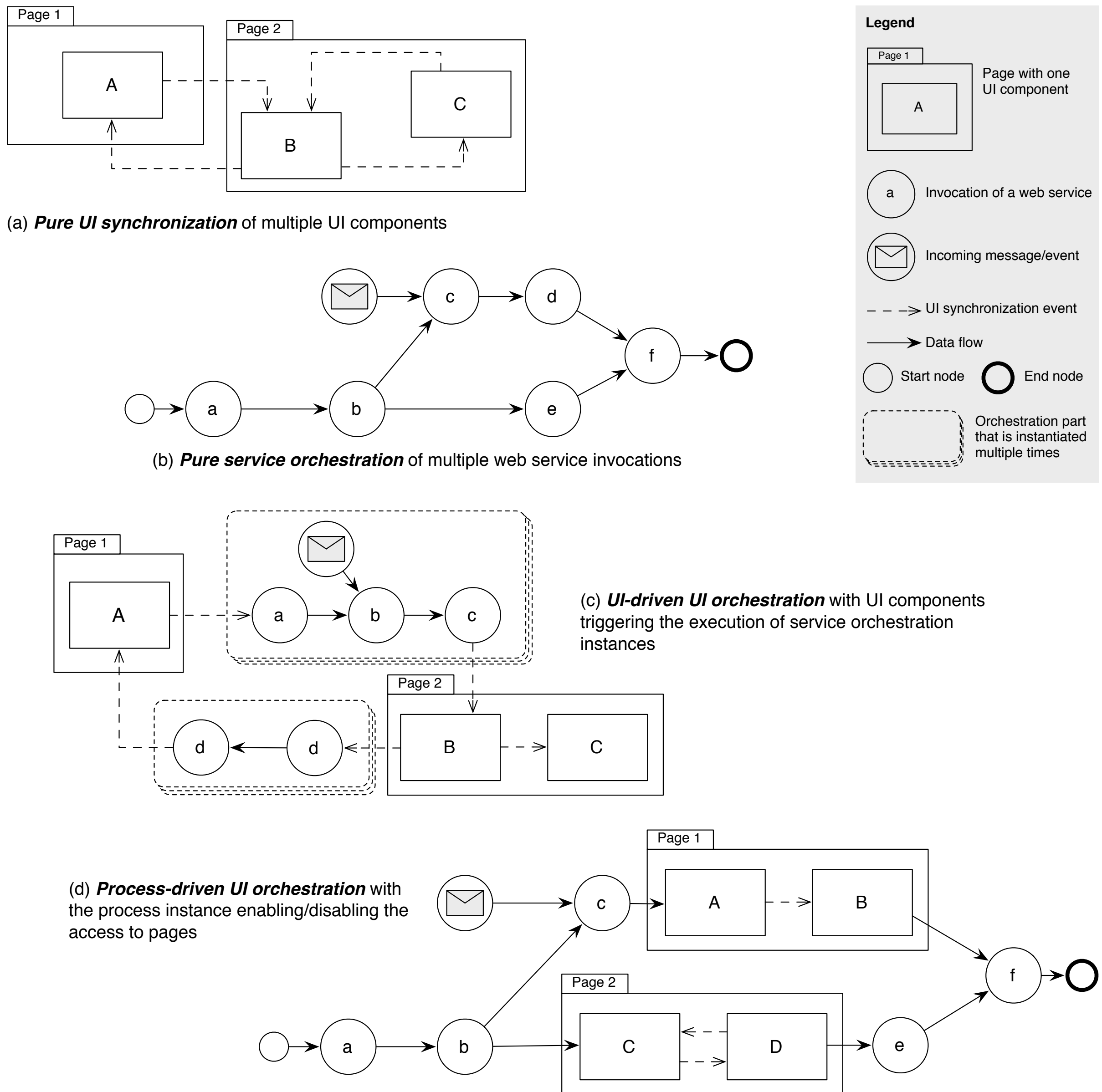


Figure 2.10: The four types of (UI) orchestration supported by BPEL4UI and the MarcoFlow system.

Figure 2.10(a) illustrates a simple example: There are two concurrent pages, possibly associated with two different users and with a total of three UI components, one in `Page 1` and two in `Page 2`. By interacting with the UI component `A`, the user can generate an event that synchronizes component `B` in the other page; likewise, another user can interact with `B` and synchronize both `A` and `C`, while `C` allows the user to synchronize again `B`. The three UI components are instantiated in their web pages and run until the users close their web browsers or navigate to another web page. As such, UI components are stateful: their UI constantly reflects the interaction state of the users with the component (e.g., in terms of selections or navigation actions performed). During their lifetime, each UI component may generate multiple events as output and accept multiple events as input. That is, while in one instance of the UI orchestration in Figure 2.10(a) each UI component is instantiated only once, there may be multiple instances of synchronization events (the dashed arrows).

Supporting the *execution* of this type of UI orchestration requires the presence of both a client-side runtime environment and a server-side environment. Specifically, the intra-page UI synchronization of `B` and `C` can be handled in the client, since both UI components run inside the same web page, i.e., web browser. The synchronization of `A` and `B`, instead, requires help from the server side, in that they implement a distributed UI synchronization. Therefore, the event proxy on the server side (cf. Figure 2.2) is needed, in order to forward communications among the two web pages.

Sending an event through the event proxy raises the need for *correlation*, in that there may be multiple instances of a same UI orchestration running concurrently and, therefore, it is necessary to identify which event belongs to which instance. The solution we adopt is to add to each generated UI event a so-called *UIOrchestrationID*, which uniquely identifies the UI orchestration instance. The identifier is generated by the UI engine

at application startup and shared with all the users participating in the orchestration. This feature is automated in our runtime framework and does not require any specific modeling at design time.

2.7.2 Pure service orchestrations

From a web service point of view, the basic type of UI orchestration is the one that completely comes *without UI*, i.e., a common web service orchestration. Although this configuration represents a "degenerated" UI orchestration (given that there is no UI), it is fully supported by BPEL4UI and deserves an explanation in that it represents the building block for the next UI orchestration types. Typical examples are order processing logics or payment processes.

Figure 2.10(b) provides an example: There are six web service invocations (specifically, synchronous request-response invocations) and one incoming event arranged in a typical service orchestration. For presentation purpose, we adopt a *data flow* logic to model the orchestration, as for the discussion in this section it is not important to explicitly distinguish between control and data flow. The important aspect of the model is that, upon instantiation of the service orchestration, each element in the model is instantiated exactly once – including the data flow connectors (differently from what happened with the UI synchronization events in Figure 2.10(a)). The data flow connectors rule both which service invocation can be performed and how data are passed from one invocation to another.

Executing such a service orchestration requires support from an orchestration engine/server, such as a BPEL engine, which is able to instantiate on orchestration model, to invoke the services as prescribed by the model, to transform data formats between service invocations, to accept incoming notifications or events, and to keep the state of the progress in the orchestration instance. The actual services run remotely, and are outside the

scope of the orchestration environment.

The important aspect of the model in Figure 2.10(b) is the incoming event (graphically represented by the letter in the circle), as the event raises the need for *correlation* in the service orchestration. In fact, without the incoming event, the model would consist only of synchronous service invocations, which could be processed easily step by step by the orchestration engine. The engine would simply invoke a service, wait for its response, pass the response to the next service, and so on till the whole orchestration logics ends. In the presence of the incoming event, instead, the engine must be able to correlate each incoming event it receives with the correct target orchestration instance of the event. Doing so requires sharing at least a simple key or identifier (the *correlation set*) among the running orchestration instance and the incoming event. For instance, the name of the person who starts the orchestration instance could be used as correlation identifier, as such could be known to both the engine and the external service sending the event – provided that there is always only one instance per person running in the engine.

2.7.3 UI-driven UI orchestrations

A "full" UI orchestration, however, is characterized by the joint use of both UI synchronizations and service orchestrations inside a same application. Depending on which of these two ingredients dominates the behavior of the application, we can have either UI-driven orchestrations (where service orchestrations are enacted by the UI) or process-driven orchestrations (where the UIs are enacted by the service orchestration). Here we focus on the former type, in the next section we discuss the latter. For instance, a web mashup that integrates RSS data from a Yahoo! Pipe may invoke the pipe processing logic multiple times while running.

Figure 2.10(c) abstracts this type of UI orchestration: There are two

pages with respective UI components and two service orchestration flows. While the intra-page UI synchronization of `B` and `C` does not involve any web service, the distributed UI synchronizations of `A` and `B` are based on intermediate service invocations in both directions. Just like we can have multiple UI synchronization events (the dashed arrows) for each instance of UI component, we now also have for each synchronization of `A` and `B` a new instance of the intermediate service orchestration logic (graphically represented by the dashed box around the service orchestrations).

In order to *execute* such a UI-driven UI orchestration, we need to join also the power of the runtime environments of the two previous configurations. Specifically, UI synchronizations involving service invocations can no longer be performed with a simple event proxy on the server side only (like in pure UI orchestrations); instead, the synchronization requires a tight integration of the client-side runtime environment for UIs with the server-side service orchestration engine. Specifically, a UI synchronization event from one page must be able to instantiate and provide input to a service orchestration logic on the server side, which, in turn, must be able to deliver its output in form of a UI synchronization event sent to another page. That is, we need to have a full two-way communication channel between the two runtime environments, a feature that is implemented by the UI components' event proxies and event buffers in the UI engine server.

In terms of *correlation*, all UI synchronization events carry the `UIOrchestrationI` as already introduced for pure UI orchestrations, while the service orchestration parts may require additional correlation information inside BPEL4UI, depending on their individual topology. For instance, the service orchestration enacted by propagating an event from `B` to `A` only involves synchronous service invocations and does therefore not require any additional correlation information. The other service orchestration in Figure 2.10(c), instead, also involves the reception of an external event, which re-

quires the setup of an additional correlation identifier, as already described for Figure 2.10(b).

2.7.4 Process-driven UI orchestrations

Finally, we have a process-driven UI orchestration each time we have an application that brings together UI synchronizations and service orchestrations in which the service orchestration dominates over the UI synchronization. For instance, workflow management or, more in general, business process management applications that integrate both web services and UI components and that orchestrate tasks (work items) to be performed by either users or automated resources, such as our reference scenario, can be considered of this type of UI orchestration.

Figure 2.10(d) schematically illustrates the situation: The application starts with a pure service orchestration that enacts a set of services and, only after the successful processing of services `a`, `b`, `c`, and `d`, allows the users to access their respective web pages. Inside the pages, there are UI components that allow the users to interact with the pages and to perform and conclude their tasks, which causes the UI orchestration to leave again and disable the pages and to proceed with the processing of the remaining part of the service orchestration. That is, in process-driven UI orchestrations pages are invoked like services, but they are targeted at users and, therefore, expose a UI the users can interact with. The overall UI orchestration keeps waiting until the user successfully completes his/her task, which is communicated via an outgoing UI synchronization event.

In terms of required *execution* support, process-driven UI orchestrations are similar to UI-driven UI orchestrations, with the difference that the main service orchestration is instantiated only ones, not multiple times.

Correlation requirements are similar, too. As shown in Figure 2.10(d), if there is an incoming event that needs to be injected into a running in-

stance of the UI orchestration, correlation is needed; otherwise, the whole UI orchestration can also be processed without correlation. UI synchronization events are again managed via the orchestration's unique identifier associated by the UI engine.

2.7.5 Complex UI orchestrations

The four types of UI orchestrations above represent those classes of UI orchestrations that characterize the most important application scenarios we encountered throughout the development of the MarcoFlow system. Yet, UI orchestrations may easily also get more complex. For instance, it is possible to use a process-driven UI orchestration (including again UIs and actors) in place of any of the simple service orchestrations in Figure 2.10(c), or it is possible to expand the simple pages in Figure 2.10(d) into complete UI-driven UI orchestrations (including new service orchestrations), or we could establish UI synchronizations among the two pages in Figure 2.10(d), and similar. While these kinds of UI orchestrations are theoretically possible and supported by BPEL4UI and MarcoFlow, luckily it is hard to find practical examples that indeed require such a level of complexity.

2.8 Implementing and Running UI Orchestrations

In order to ease the development, deployment, and execution of UI orchestrations, MarcoFlow comes with two tools that aid the different actors involved: a *graphical BPEL4UI editor* for developers and a *web-based management console* for both developers and users.

The ***graphical BPEL4UI editor*** for developers has been implemented as an extension of the Eclipse BPEL editor (`http://www.eclipse.org/bpel/`) and comes with (i) a panel for the specification of the pages in which UI components can be rendered and (ii) a property panel that al-

lows the developer to configure the web pages, to set the properties of UI partner links, and to associate them to place holders in the layout.

The screenshot in Figure 2.11 shows the editor at work. The layout structure of the editor is the same of the standard Eclipse editor, except for some differences in the right and bottom side. On the right side, now it is also possible to define the pages of the UI orchestration (as elements of the *Pages* group). Selecting a page in the list shows the respective details in the *Properties* panel in the lower part of the figure and allows the developer to assign the actor, i.e., the user that will be allowed to access the page, and the HTML template for the page. Still on the right side, where usually there are only partner links for web services, now it is also possible to define UI partner links for UI components. Selecting a partner link from the list again shows its details in the *Properties* panel. Ticking the *UI component* checkbox turns the partner link into a UI partner link and allows the developer to define in which page and place holder inside the page the UI component will be rendered. The actual composition logic is specified in the modeling canvas in the central part of the editor.

The ***web-based management console*** helps (i) developers deploy ready UI orchestrations and (ii) users in instantiating and participating in running UI orchestrations. Deploying a new UI orchestration requires the developer to pack all the project files (web service WSDLs, UI component WSDL4UIs, BPEL4UI specification, HTML templates, and the system configuration) into a single archive file and to upload it to the management console. Doing so allows the developer to deploy the application by means of a simple mouse click, which invokes the BPEL4UI compiler and generates the standard BPEL file, the event buffers and event proxies, their respective WSDL files, and the UI compositions and then deploys all generated artifacts in the respective runtime environments.

Figure 2.12, instead, shows the interface of the management console

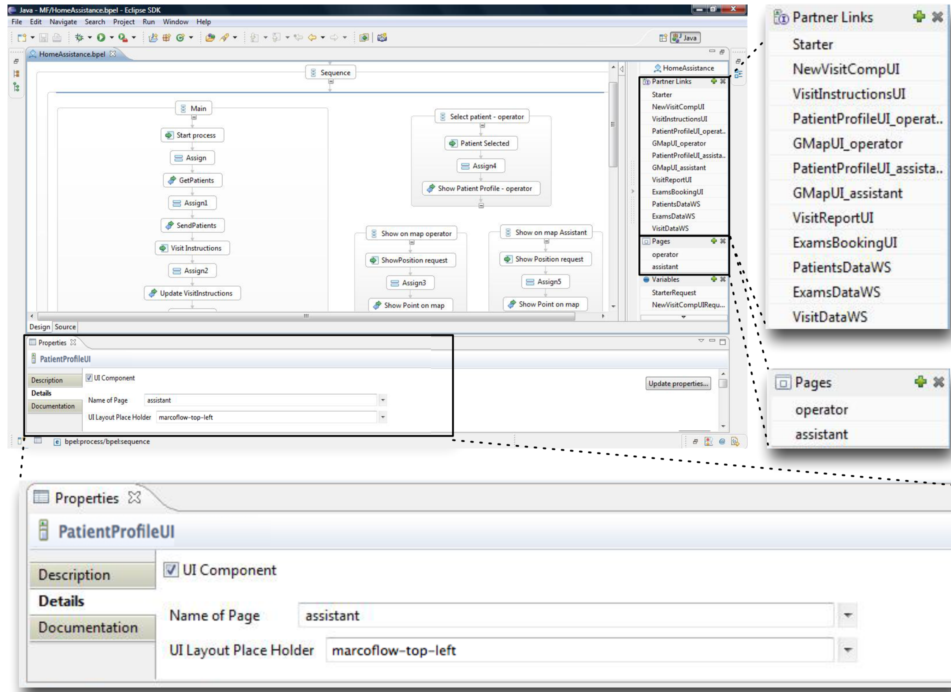


Figure 2.11: The extended Eclipse BPEL editor for developing UI orchestrations at work.

for regular users, where they can see which UI orchestrations have been deployed they have also access to. Specifically, a user can either start a new instance of UI orchestration (via the upper list in the figure) or participate in an already running instance of UI orchestration (via the lower list in the figure), which – in the case of the operator and assistant in our example scenario – leads him/her, for example, to one of the pages in Figure 2.1. The operator is allowed to instantiate the orchestration, and the assistant is enabled to participate.

The MarcoFlow system shown in Figure 2.2 is fully implemented and running (a demo of the tool is available at `http://goo.gl/XqdK79`). In our test setting, we run the UI engine server and the BPEL engine on the same machine, yet these components could also easily be distributed over different physical machines, a feature that is already supported by our code generator. Developing the MarcoFlow platform in a way that is

Figure 2.12: The management console for developers and users allowing them to deploy, instantiate, and participate in UI orchestrations.

fully functioning required taking some decisions on the *technologies* to be used. As shown in this chapter, we opted for BPEL as service orchestration engine, since BPEL natively supports communication with SOAP/WSDL web services, a requirement that stems from our scenario. We opted for JavaScript UI components, as this represents the current trend in mashups and web-based UI development. Yet, the contributions of this chapter are independent of these choices and more conceptual than technological (cf. Section 2.7). In fact, we can easily imagine substituting the BPEL editor with a BPMN editor, of course adding the necessary UI-specific extensions to it. Given the standardized mapping from BPMN 2.0 to BPEL, this would not affect the runtime part of the architecture. If we substitute the BPEL engine with another workflow or business process engine (provided that such already supports interaction with web services), this would require a change in the runtime architecture and the generated process model. But it would be straightforward and not change the philosophy

of the overall platform. Similarly, if we want to manage UI integration at the server-side (e.g., via server-side scripting languages like Perl or PHP, ASP.Net or JSP), this could be achieved, but for the cost of lower performance. User interaction occurs at the client side and, hence, UI events are generated inside the client browser. Using server-side technologies means going through the server each time we have a simple intra-page UI synchronization, which degrades the overall user experience. It could however be possible to use different client-side UI componentization technologies, such as W3C widgets (again based on JavaScript), for which we are already studying suitable mashup models [88].

2.9 Lessons Learned

We conclude the chapter with a few considerations on lessons learned while developing and applying MarcoFlow.

One observation is that developers seem to prefer a *web-based* environment rather than an Eclipse-based one. We had chosen Eclipse because it already comes with an open-source editor for BPEL, and we felt it was rather powerful and reasonably easy to extend as opposed to developing a new editor. In the end, working with the editor took a lot of time, so that we did not get the benefits of a web-based editor nor the time savings we hoped for.

A second issue relates to the number of *conversions* of messages from SOAP to REST and vice versa. In the current approach, even when two REST services are communicating we always need to SOAP-ify them. While we aim to minimize this kind of conversions as much as possible (by keeping intra-page UI synchronizations on the client), this limits the scalability if a single UI engine is used.

A limitation of the current implementation is that our notification han-

dlers inside the client browser continuously *poll* the server-side event buffers for updates, which further produces communication overhead and possibly delays the forwarding of events. With the growing support for HTML 5 web sockets, we will approach this limitation by pushing events from the server to the client.

Another limitation is the hard-coded assignment of *users* to pages. In our future work we will address this by investigating how resource managers known from workflow management systems can be adapted to our needs. Instead of assigning concrete users, we will therefore assign users roles to pages, which can then be instantiated either at deployment time or runtime.

An interesting finding we did not realize in the beginning is that, since UI orchestrations intermix *stateless* elements (web service invocations) with *stateful* elements (UI components) the need for correlation in UI orchestrations is higher than in pure web service orchestrations. Design-time and runtime constructs here may be needed to simplify specifications and make the engine more scalable.

However the main considerations that will drive our research are in terms of usability and applicability. While working with BPEL was a strong requirement initially, many companies are increasingly considering mashup languages for non mission-critical applications, targeting relatively simple ways to integrate and present web-accessible data. This would fit well with the MarcoFlow approach, which can be extended to deal with mashup languages.

Finally, working with MarcoFlow and experimenting its usage helped us strengthen our belief that BPEL, its variations, and actually even mashup languages are not suitable for end users, no matter how good development tools are. Our conclusion here is that if we want to bring development power to the end users or at least to knowledge workers we need to define domain-specific models and tools rather than general purpose ones. This

is what is presented in [16] by members of our research group. Yet, we also recognize that UI orchestrations are intrinsically complex, an observation that already inspired a critical survey paper on "process mashups" [26], in which we conclude that the kind of development scenarios supported by MarcoFlow hardly suits the capabilities of less-skilled developers or end users.

In summary, we are confident that the technological limitations of MarcoFlow (no web-based editor, message conversations, polling, user assignments) can easily be addressed in our future work. The conceptual limitations, that is, the intrinsic complexity of UI orchestrations, however, we cannot eliminate.

2.10 Conclusion

The spectrum of applications whose design intrinsically depends on a structured flow of activities, tasks or capabilities is large, but current workflow or business process management software is not able to cater for all of them. Especially lightweight, component-based applications or Web 2.0 based, mashup-like applications typically do not justify the investment in complex process support systems, either because their user basis is too small or because there is a need only for few, simple applications. Yet, these applications too demand for abstractions and tools that are able to speed up their development, especially in the context of the Web with its fast development cycles.

We introduced an approach to what we call *distributed UI orchestration*, a component-based development technique that introduces a new first-class concept into the workflow management and service composition world, i.e., UIs, and that fits the needs of many of today's web applications. We proposed a model for UI components and showed how dealing

with them requires extending the expressive power of a standard service composition language, such as BPEL. We equipped the language with a modeling environment and a code generator able to produce artifacts that can be executed straightaway by our runtime environment, which separates intra-page UI synchronization from distributed UI synchronization and service orchestration. The result is an approach to distributed UI orchestration that is comprehensive and free.

A strong point of the described approach is that it recognizes the need for abstraction and more expressive models and languages at design time, while – thanks to its strong separation of concerns and powerful code generator – it does not require any new language or system at runtime.

While the intrinsic complexity of UI orchestrations prevents the adoption of MarcoFlow by less skilled developers or end users (which was never the goal of the project), MarcoFlow does provide skilled developers with more expressive power compared to their current instruments: the experienced *BPEL developer* is able to integrate UIs and people into his service compositions; the *mashup developer* is able to design mashups that also involve long-running service orchestrations and user collaborations.

Chapter 3

Process-Based Design and Integration of Wireless Sensor Network Applications

In this chapter, leveraging on the knowledge gained from the research on distributed UIs, we focus on actors that are not only distributed but also autonomous. We present here the work to enable the modeling and execution of processes that integrate, coordinate, and control Wireless Sensor and Actuator Networks (WSNs). As in the previous chapter also in this we present an extension of a process language (BPMN) with domain-specific constructs and with modeling components to abstract the capabilities of a network at an higher level. This chapter is an extension of the work presented at the Business Process Management conference in 2012 [83]. In this chapter we added an explanation of the new constructs we introduced and a section on the evaluation of the approach made within the makeSense project.

In the same context we also published other papers that present the overall approach [15, 17, 18, 25] which are collected and available online.

3.1 Introduction

Today there is still lack of high-level, model-driven programming tools for Wireless Sensor and Actuator Network (WSN) applications and the integration with enterprise services requires significant effort and expertise in embedded programming of WSNs. Organizations are reluctant to install large-scale WSNs, as this still requires significant, costly, low-level programming of sensing and actuation logic for the WSN, in addition to the physical deployment of the WSN nodes (e.g., inside a building). Additionally, setting up the communication channel between a WSN and an enterprise's information system requires an even larger set of technologies and manually writing of custom code. Domain experts typically lack the necessary low-level programming skills.

To foster widespread adoption and more efficient use of sensor networks for enterprise information systems, a need for a specifically tailored integration technique that is able to bring together sensor networks and business applications [43] is perceived. The aim is to drastically improve the ease of programming of WSNs by enabling the graphical *modeling* of WSN applications, leaving low-level details to a model compiler and a run-time system. WSN programming should be accessible to domain experts, such as business process modelers. They should further be empowered to design the WSN's interaction with enterprise information systems using the methods of business process modeling they are familiar with. Our approach aims to:

- Provide a *conceptual model* that abstracts typical WSN programming knowledge into reusable tasks that can be integrated into modeling notations, such as the Business Process Modeling Notation (BPMN).
- Develop an *extension of BPMN* [67], BPMN4WSN, that enables the graphical modeling of WSN applications and their integration with

BPs based on an abstraction layer that hides low-level details of the sensor network.

- Introduce *tools* that enable the design, deployment, and execution of integrated WSN/BP applications. We do not reuse existing APIs toward the WSN; we *program* the WSN and automatically generate the necessary APIs.

- Evaluate our approach with a realistic *prototype* deployment, including a self-optimizing run-time system layer, and a report on the first experiences with its usage in the context of the EU project make*Sense*.

In order to create applications that span both a BPMN process and a WSN application, knowledge in both fields is required. We do not expect the *application developer* (the domain expert) to model an executable process. Rather, we suggest a two-phase approach, where a descriptive process model is created by the developer, which is then refined by a more technical *system developer* using the WSN extension integrated in the process diagram.

In the following, we outline an application scenario to better describe our approach. Then, in Section 3.3, the typical characteristics and components of WSNs are analyzed. In Section 3.4 it is outlined how the challenges identified in the scenario are approached conceptually, and in Section 3.5 the according extension of BPMN is described. Subsequently in Section 3.6 the implementation of the prototype, including the code generation logic for WSNs is described. Section 3.8 critically discusses the results achieved so far. Section 3.9 reviews related work before concluding the chapter.

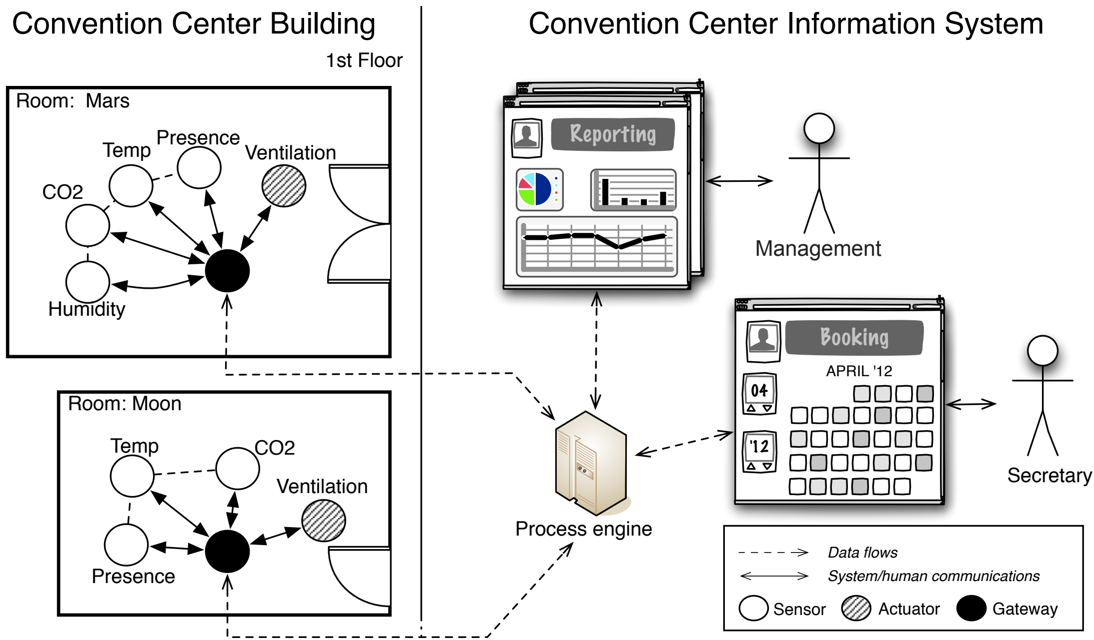


Figure 3.1: Integration of a convention center's BP engine with a WSN for HVAC.

3.2 Scenario: Convention Center HVAC Management

Our application scenario showcases the operation of a convention center (see Figure 3.1) that has a variety of meeting rooms, which can be booked for various events. Each room can be booked at a rate that partly depends on room characteristics (e.g., its size) and partly on the energy consumption of the event organized in the room. For this purpose, the convention center is equipped with a Heating, Ventilation, and Air Conditioning (HVAC) system including a WSN, which ensures comfortable levels of temperature, humidity, and CO_2 for each individual room for the booked duration of the respective event. In order to do so, the HVAC system must be instructed automatically by the convention center's information system about when to activate the ventilation and how long to control the room's temperature and CO_2 concentration for each room. Room conditions are only maintained during the booked times to save energy and only if presence of people is detected by presence sensing; air conditioning is shut off when

a meeting is not attended at all or ends prematurely. In turn, the HVAC system feeds back sensor data to the information system, which allow the information system to precisely compute the HVAC cost for each individual event. The information system is used for the booking of rooms, the reporting on energy consumption, and the billing of customers. This mode of operation is more energy efficient that today's common practice, where one would simply run the HVAC system at a fixed rate, independently of room occupation or environmental conditions — a practice that wastes much energy.

Technically, it is necessary to develop (i) the BP logic running inside the BP engine, (ii) the code running on the nodes inside the WSN, and (iii) a suitable set of communication endpoints supporting the interaction of the BPM with the WSN and vice versa. Note that it is *not* the goal of this work to optimize convention center operation or more generally building automation, but to provide a basic set of abstractions, tools, and methodologies that can be used in all scenarios where also WSNs are used. We use it merely as a device to depict a concrete application of our approach.

3.3 Relevant Properties of Wireless Sensor Networks

Before going into the details of the approach, the special properties of WSNs that are relevant at the application layer and that therefore underpin our model of the system are explained. A WSN is a distributed system, namely a network of wireless, battery-powered, autonomous, small-scale devices, so called nodes, each of which is equipped with one or more sensors or actuators or both. Nodes are battery-powered and replacing the battery is mostly not intended or not feasible from a Total Cost of Ownership (TCO) perspective. Therefore, they make use of ultra-low-power hardware, that is drastically limited in processing power, memory, and transmission

bandwidth and the application software running on the nodes, including wireless communication protocols, needs to be optimized for low power consumption to extend network lifespan. These limits typically prevent executing a regular BPMN engine on the devices that interprets BPMN models serialized as XML.

Sensors are used to sense information from the real world (e.g., temperature) while actuators perform actions that change the state of the environment (e.g., control a motor or a lamp). The typical number of nodes inside a network can vary from a few to hundreds or even thousands. Via radio links, a node can generally communicate with all other nodes in its transmission range and with nodes further away by multi-hop, routed communication. WSNs are able to self-organize, overcome network failure, and execute distributed computation logic, such as computing the average of sensor values while those are routed to a destination node. Often, WSNs are composed of heterogeneous nodes, each equipped with a custom set of sensors, so that, for example, one type of node can sense CO_2 and humidity while another type of node is able to control an automatic door, while a third has enough special hardware to compute complex arithmetics.

As a basis for modeling WSN application logic, a very simple model of the physical set-up that is sensed and acted upon is assumed: A given WSN monitors real world entities, each is referred to as an Entity of Interest (EoI) which can be a *location* or a *thing.* A thing is any physical object, while a location is a space that the sensor network is monitoring, e.g., a room or a building. A domain expert is usually only interested in the EoIs and the operations that can be applied to them, but not in the technical layer of sensors that sense or the actuators that influence them.

To overcome the limitations of WSN hardware and to maximize efficiency of operations, the research community has introduced a large number of programming abstractions to program wireless sensor networks [62].

By abstracting existing programming concept into high-level constructs [15] (described in Section 3.5.2) and assuming that all existing functionality can be expressed using them, one can use high-level constructs as basic building blocks for graphical modeling. Usually, a sensor network will perform some or all of these tasks:

Sensing: measuring one or more environmental parameters of an EoI, such as temperature or humidity, making use of the sensing equipment of the nodes.

Actuation: enacting operations physically affecting an EoI, e.g., controlling or moving it or flashing a LED. WSNs are often used to actuate or control the environment in reaction to sensed parameters, creating a control loop (as the actuation eventually triggers changes in the sensed values).

Task distribution: distributing operations that coordinate a subset of nodes, e.g., any in-network aggregation on the input values or the election of a controller node based on certain criteria. As WSNs consist of several nodes, several of which can monitor the same EoI, especially data aggregation operations are often required, e.g., to compute the average temperature of a room observed by many sensor nodes.

From the perspective of a domain expert, it is irrelevant which part of a WSN performs a task, e.g., whether an operation is carried out by a single node or the network as a whole as long as the operations are addressable by an EoI.

3.4 Requirements and Approach

In the convention center scenario, there is a need for collaboration between the reservations and billing systems in the back-end and the sensor network

that executes the sensing and actuating operations. Thus, the application runs on different types of systems which can be seen as two distinct participants in the process. This raises the need to model both the intra-WSN logic and its interactions with back-end systems as a collaboration of two process participants. While the back-end part is orchestrated using classical Business Process Management (BPM), modeling the process logic to be executed inside the WSN needs certain provisions (e.g., model extensions) to enable the specification of WSN logic in a high-level fashion and the creation of code that can be executed in the network.

Typically, the integration of WSNs into BPs is based on the invocation of services exposed by the network [6, 36, 78]. This results in a modeling approach that uses the network as set of available operations on which a process can be constructed, but that prohibits the programming of the WSN itself. This limits the possibility to define custom WSN logic to be carried out by the network as part of the process. Instead, the key idea of our approach is to develop a business process modeling notation that allows a domain expert to program both the BP *and* the actual network logic, without the need to know and specify all the low-level details. The created process model is later used to derive the code that will be executed by the WSN. In this way, the WSN logic is fully specified at the process level.

The specific requirements we identify can be divided into supporting *modeling*, *deployment* and *runtime*. Supporting modeling means defining a modeling paradigm that fits the needs of a domain expert and integrates back-end business processes and WSN logic using a single modeling language. This requires to:

- Provide an easy to understand and familiar way of expressing WSN logic; enable integrating WSN processes into back-end processes, coupling them and allowing for easy data sharing.

- Define a set of concepts to describe the logic and operations that can be combined for creating reusable, high-level WSN modeling constructs. We have to supply the modeler with the possibility to specify operations like sense, actuate and aggregate for measurements over EoIs.

- Model WSN capabilities and details. WSNs are usually heterogeneous regarding the type of sensors and actuators. Knowing the characteristics of the network is fundamental to have an overview of which things and locations can be controlled and monitored by the WSN as well as which operations the WSN is able to perform. Having such a model will give the domain expert the ability to express the desired processes in the familiar terms of EoIs and irrespective of technical systems.

- Supporting the modeler in designing only feasible processes by restricting the available modeling constructs to him to what the WSN is capable of executing.

Supporting the deployment of the process requires to:

- Split the process model into an intra-WSN part and a WSN-aware part (back-end). The process is divided between two actors that participate in the execution. These two parts of the process have to be separated and handled differently.

- Create WSN binary code. The intra-WSN part of the process has to be translated to binary code and injected into the nodes. This code is generated based on the flow of the process model and tasks that describe the operations.

- Create the endpoints and communication channels to handle the messages from and to the network. After having split the process in two

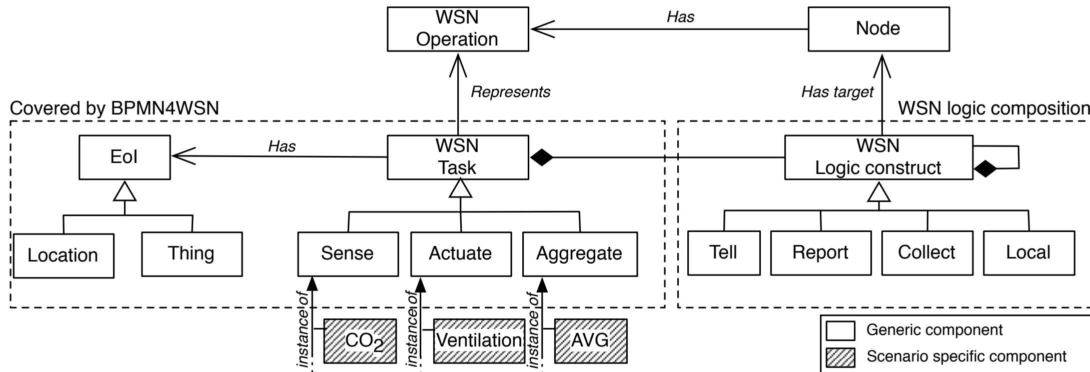


Figure 3.2: Conceptual model of WSN operations

parts and after having translated the WSN part into binary code the communication between these two participants has to be guaranteed. To do so, the endpoints and the communication channels through which the messages will be sent/received need to be available.

Supporting the execution requires to:

- Provide a process engine to execute the WSN-aware business process part. The process engine also handles the communication with the WSN.

- Run the code in the WSN. Part of the process actually runs inside the network without the need for external communication and control. The process is executed on the gateways and the actions are distributed on the nodes, guaranteeing the correctness of the process depicted by the modeler.

Figure 3.2 illustrates the conceptual model of how we approach WSN programming. The model is not meant to be an extension of the BPMN meta-model. Only part of it is related to BPMN4WSN, the other part is related to our own modeling formalism for the definition of low-level WSN logic. The two entities on the top represent the physical WSN, which we

abstract as composed of a set of *Nodes* (sensor or actuator nodes) supporting a set of native operations, the so-called *WSN Operations*, such as *sense* CO_2 for a sensor or *open* for a valve actuator. We allow the domain expert to use WSN operations by abstracting away from the network topology, i.e., nodes, and instead allowing him to reason in terms of EoIs via a dedicated task type, the *WSN Tasks*. A *WSN Task* is a generic action that can be used to express *sense*, *actuate*, and *aggregate* operations and that can be executed by the network. The *WSN Task* is logically connected to an EoI, which allows the modelers to scope the action. That is, the EoI specifies *where* the action will be executed; it could be a *thing* or a *location*. WSN Task and EoI represent the high-level constructs used to model WSN logic in BPMN4WSN. This level of abstraction is however not enough to describe all the needed details to generate binary code that runs on the nodes, which instead requires taking into account the topology of the network. The detailed specification is based on *WSN logic constructs*, which abstract operations that can be configured (e.g., by adding a concrete target node resolving a logical EoI) and translated into binary code. The composition of WSN logic constructs (the *WSN logic composition* box) allows the system developer to refine the process model designed by the domain expert and to fill WSN tasks with concrete logic.

Figure 3.3 shows the architecture of the tool chain for developing WSN/BP applications containing an extended BPMN editor in which the process is modeled, and a compiler for translating the high-level specifications into low-level executable binary code for the sensor network and for the process engine. Next, the *modeling* and *deployment* part are discussed in more detail; a first prototype of the tool is discussed afterwards.

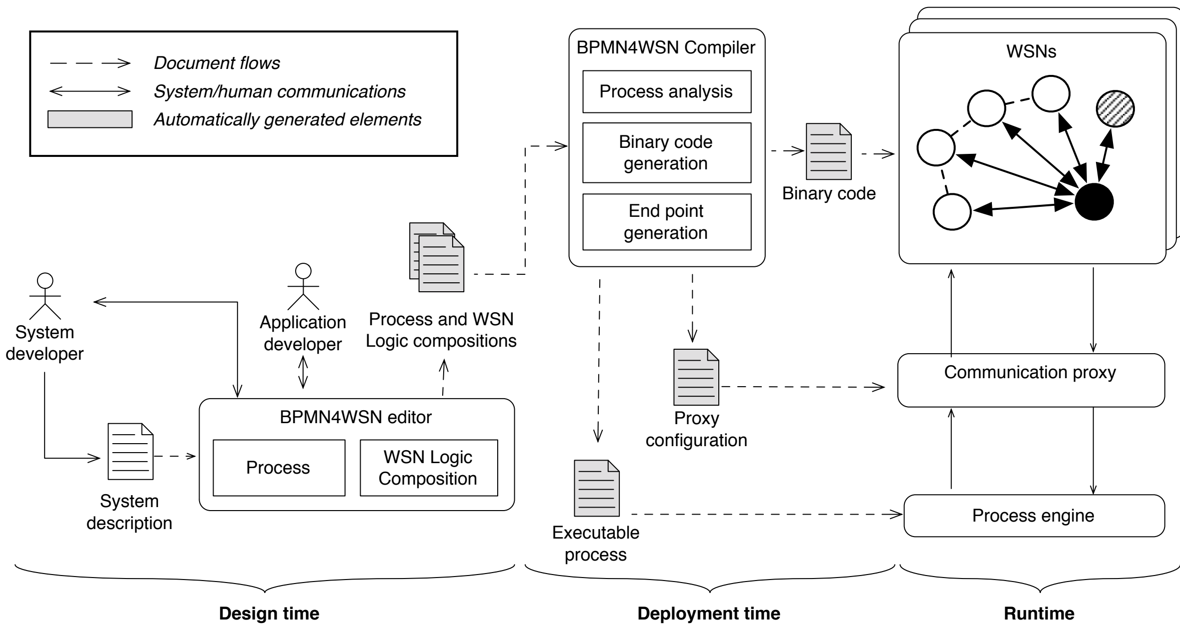


Figure 3.3: Architecture

3.5 BPMN4WSN

As illustrated in Figure 3.3, two types of developers jointly develop a process model: the application developer and the system developer. The application developer is the person who models the coarse process; he is an expert of the domain with experience in business process modeling and has some WSN background. The system developer is a WSN expert and has the task of creating the refined, XML-formated model of the system (see the bottom left corner of Figure 3.3). This model contains information of the network such as the EOIs, nodes and available sense and actuate operations. The two roles collaborate mainly in the design of WSN Tasks. The application developer creates a process that crosses the system boundary between standard IT and WSN including the specification of the behavior of the latter. He defines a descriptive, not yet executable version of the process. For instance, in the convention center use case, the application developer would specify a task for reading the latest sensor values or driv-

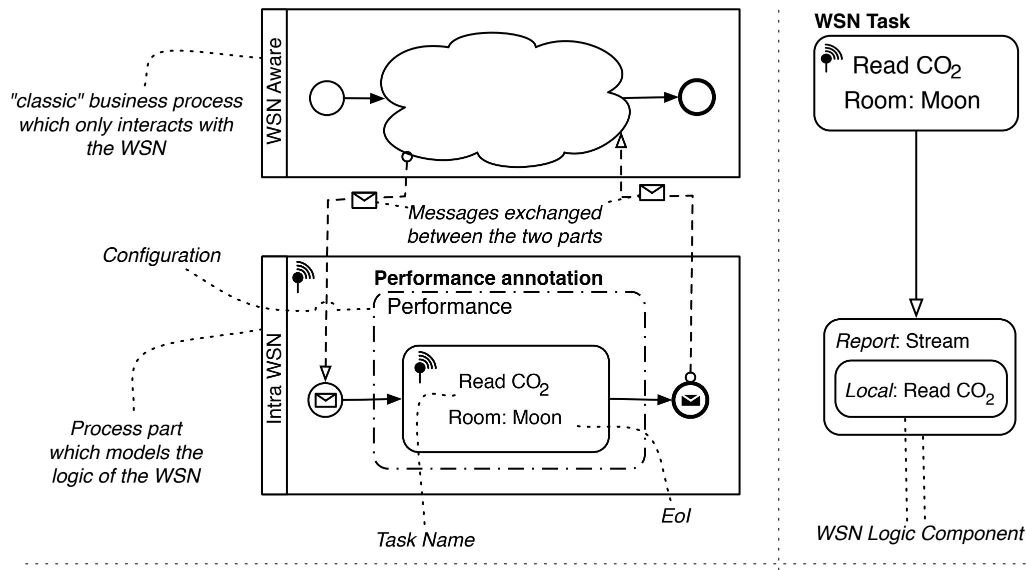


Figure 3.4: WSN-specific modeling constructs in BPMN4WSN.

ing an actuator based on the system descriptor model. Later, the system developer would refine this model by adding WSN logic components to make the tasks that involve the WSN executable.

3.5.1 Process Logic

In our solution the design of the business process is mainly carried out by the application developer, who uses BPMN [87] with some additions based on the extension points defined in the standard (without touching the BPMN meta model), designed to model the salient characteristics of the WSN. The extended language is referred to as *BPMN4WSN*. This extended version comprises both new components and modeling rules.

A BPMN4WSN process must be composed of at least two pools: an *intra-WSN* pool and *WSN-aware* pool; Figure 3.4 contains a minimal example. The intra-WSN pool is the part where the WSN logic is specified, while the WSN-aware pool is a classical BPMN process. The splitting into process logic executed inside and outside the WSN forces the modeler to

explicitly model interactions between the two parts as messages, directly mapping the run-time behavior (where messages are the only way of interaction between the parts) to the model. This separation also enables the clean generation of code.

In the intra-WSN pool, constructs that directly orchestrate WSN functionality (made available through high-level abstractions) are needed. This need is addressed by introducing a new activity type: the *WSN Task*, that can only be used in the intra-WSN part. It has two properties: a reference to a set of *WSN logic construct* definitions and the *EoI* to which the respective operations should be applied (see Figures 3.4 and 3.2). It has an antenna on the top-left corner to distinguish it from other tasks; if specified, the *EoI* value is written below the task name. For example, setting the EoI value to "room Moon" will execute the task on those nodes that belong to the "room Moon". In a nutshell it specifies *where* (i.e., by which subset of nodes) each WSN Task is executed.

The referenced *WSN logic construct* definition is the set of operations that have to be performed by the network. For simplifying the modeling of such low-level programming specification, a set of *WSN logic constructs* that describe the common operations and the way they can be combined is created.

To support the shift of the center of orchestration from the gateway node to one or more nodes in the WSN, we introduce the script tasks by extending sub-processes with annotations. With these sub-processes a modeler can define WSN logics that is executed and controlled by a subset of the network, rather than have a central gateway node that controls the whole network. To specify the nodes that execute the operation we introduce a *target* attribute for the sub-process. The target of the sub-process is specified with a *static* element and a *dynamic* element. The former element tells the make*Sense* tool chain on which kind of nodes the generated

subprocess code should be deployed, based on static node attributes that can be evaluated at deployment time (e.g., all the CO_2 sensors). The latter defines where to run the code, depending on runtime parameters, for example the roomNumber parameter.

In addition to the WSN Task and extended sub-processes, a *performance annotation* element, i.e., an extension of the BPMN *group* element which shows the chosen performance configuration on the top-left corner, is introduced. It is used for describing the network behavior from a performance point of view. This new component allows the application developer and system developer to decide when the network performance goal has to be changed (e.g., to optimize battery lifetime). For example, when a room is empty, the network will be set to *low energy consumption* mode in order to save battery and prolong node network lifetime at the cost of lower reactiveness and possibly less reliable message transfer. In cases where high performance is needed (at the cost of battery power), other performance annotations are used. At run-time the execution semantics of these annotations is that one performance mode is set for the whole WSN, depending on the number of the tasks in each performance group. The group that contains the most tasks to be executed sets the performance mode.

3.5.2 WSN Task Specification

WSN Tasks are modeled in two steps: (i) the *process design* and (ii) the *process refinement*. The process design is generally carried out by the application developer. He just specifies a *WSN Tasks* with a speaking name, which can be a *sense*, *actuate*, or *aggregate* operation and the *EoI* on which the operation has to be executed. This part of the modeling is represented in Figure 3.2 by the items inside the *BPMN4WSN* dashed rectangle.

The process refinement (an example is shown in Figure 3.4), instead, is

generally performed by the system developer. Its goal is to transform all high-level WSN Tasks into executable operations by combining *WSN logic constructs* which model the network behaviors. As shown in Figure 3.2, each WSN Task represents *WSN logic constructs* that are the basic functionality and instances of so called meta abstractions [15] that must be configured and instantiated:

Local actions are executed locally on each sensor node.

- The *tell/report actions* represent one-to-many/many-to-one communication.
- The *tell action* enables a node to delegate an embedded action to a set of other nodes.
- The *report action* enables the gathering of information from many nodes.
- *Collective actions* enable distributed, many-to-many collaborations.

Each of these distributed actions has a *target*, which is used to select the subset of nodes the action refers to (obtained by resolving logical EoIs into physical nodes, based on the system description). In addition there is also the possibility to specify *data operators* useful to perform mathematical operations during transmission of data (e.g., to compute the average). The composition of *WSN logic constructs* requires the system developer to nest *WSN logic constructs* one into the other, creating the logic he wants to specify as shown in Figure 3.2 and in Figure 3.8.

Each specific WSN deployment has its unique system-description, which is the starting point for modeling. It describes the details of the network and it is used as configuration for the model editor. The document provides a high-level description of application-specific details of the concrete WSN deployment to the business process editor and to the model compiler. It

is used by the editor to list only those attributes to the system developer that are actually available in a concrete deployment, such as the list of EoIs (simple or composed ones like "First Floor" comprising "room1" and "room2") and to restrict the selectable operations (e.g., CO_2 sensing can only be selected if EoI "room2" has been selected, because only that room is equipped with CO_2 sensors).

In the context of the project make*Sense* our partner SAP AG extended the modeling languge introducing a different modeling approach for both the process logic and WSN task specification [81]. For convention we call the modeling presented so far as *composed WSN logic constructs* while the newer as *parametrized WSN logic constructs*. The main difference between the two version is that the parametrized WSN logic constructs introduces additional WSN Task parameters to specify the WSN logic constructs. Thus, meta abstraction composition are now specified with parameters. For example the *tOperation* parameter is used to discriminate between a sense or an actuate operation. Distribute actions, which before required to nest a *local action* within a *collective action*, are created with the *command action* parameter set as *true*, which means that this operations *tells* to other sensors what to do, and by specifying the *target operation* (e.g., sense CO_2) and the *return operation* (e.g., average). With this approach each WSN Task can be configured to be executed on specific nodes of the network, with static and dynamic targeting. This removes the need of subprocesses, which are not present in the parametrized WSN logic construct version. The other components for modeling WSN process logic (i.e., WSN pool and performance annotations) are still present. Although this modeling does not require one to model the composition of meta abstraction it still requires the system developer to specify the same attributes of the meta abstraction composition that we presented here. In other word, it abstracts the *WSN logic constructs* composition with a set of configuration

Figure 3.5: The Startup page for the configuration of the scenario.

parameters for the WSN Tasks.

3.6 Prototype

The approach described in the previous sections has been implemented as a proof-of-concept prototype. Figure 3.3 depicts the architecture of the prototype, showing the document flow and the actors involved. The modeling process, defined by our tool chain, is divided into three phases: *modeling*, *translation*, and *execution*.

Modeling.

For the modeling part of the prototype, a well-known web-based BPMN editor called *Signavio Core Components* (`http://code.google.com/p/signavio-core-components`) has been extended. The editor has been modified by adding a start page for scenario selection and a model editor for the *WSN logic constructs*.

The start page (Figure 3.5) is used to select or create a separate workspace for each scenario to enable development for distinct WSN set-ups, each with its on system-description. In each workspace, only operations that can actually be executed inside the corresponding network are enabled, helping the modeler in creating correct executable processes. For instance, in our example scenario there would be the possibility to sense CO_2 and presence but no other environmental parameters as the WSN is only equipped with

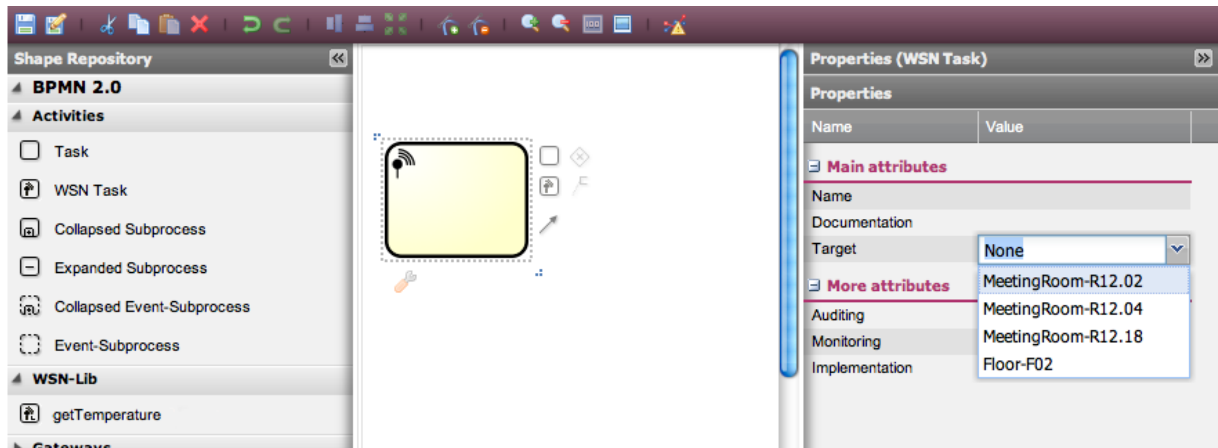


Figure 3.6: The editor for the creation of the BPMN process.

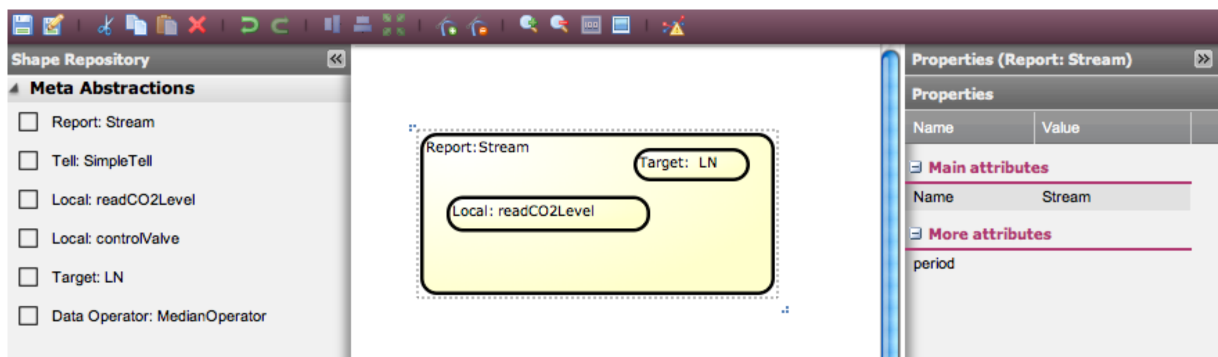


Figure 3.7: The editor for the meta abstraction composition.

these sensors.

BPMN extension points have been used to realize WSN Tasks, script tasks, and performance annotations as explained in Section 3.5 and in Figure 3.4. To support the design, the modeling tool has been extended with these three components (Figure 3.6). The editor also have library functions which are WSN Tasks that implement specific operations created ad-hoc for the chosen workspace. This feature has been introduced to provide to the application developer a set of operations that he can use without the need for composing *WSN logic constructs*

The *WSN logic construct* composition has been enabled by creating a new meta model inside the tool (Figure 3.7). By doing so, the modeler is

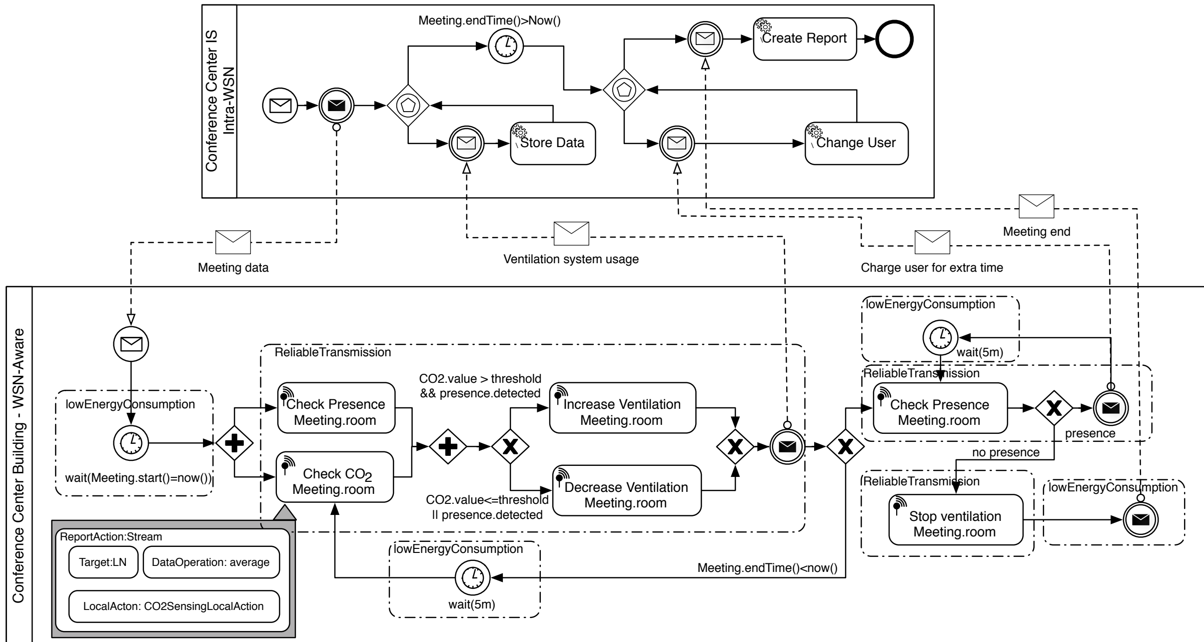


Figure 3.8: The HVAC process of the convention center.

given the possibility to compose *WSN logic construct* blocks by dragging and dropping and nesting them according to predefined composition rules that are checked by the tool. The composition is later translated into an internal format, and the files are used by the compiler to create the binary code for sensors. The editor shows only the *WSN logic construct* that are compatible with the chosen workspace.

Example.

In Figure 3.8 there is a screen shot of the process that models the scenario explained in Section 3.2. For the sake of clarity, in the intra-WSN process only CO_2 measurement and presence detection are modeled. For the CO_2 sensing operation we add the meta abstraction composition logic. This compositions is executed by the gateway and tells to the nodes that can sense the CO_2 to report the average value.

A new process instance is started when a new meeting is scheduled. The

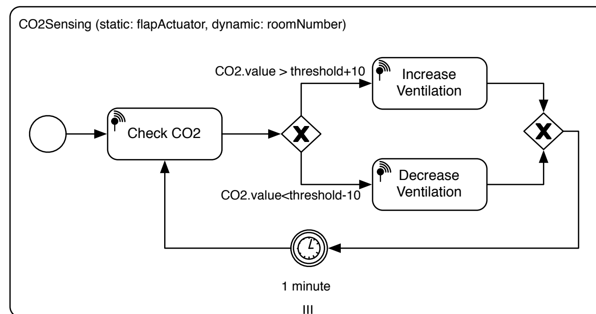


Figure 3.9: The CO_2 and ventilation operations modeled as a script task

WSN will be set to *low energy consumption mode* until the actual meeting starts. Throughout the duration of the meeting, the network checks the room conditions, increasing the ventilation when sensor values exceed a given threshold and a human presence is detected. After the scheduled meeting end time, the network checks if someone is still in the room, in which case the information system is informed, charging the user for *extra time*.

In Figure 3.9 there is modeled a script task, thus a sub process, for the sensing of the CO_2 that triggers the ventilation. This subprocess is deployed in all the rooms with *flapActuators* (static target) and run when the *roomNumber* correspond to the actual room. The CO_2 sensing operation is the same as before. In this case it is not executed by the gateway but by the static target, thus the actuator. The actuator instructs the CO_2 sensors to report the average CO_2 value. Thus, the sensing operation is performed remotely while the actuating operation locally by the actuators. The solution of Figure 3.8 and the one with sub-processes are both effective. The former executes the control on the gateway, while the latter directly on the sensor of each room.

In Figure 3.10 there is modeled the same logic as of Figure 3.9 with the WSN modeling convention presented in [81]. The "Calculate CO_2 Average" task specifies the action to sense the CO_2 value and report the

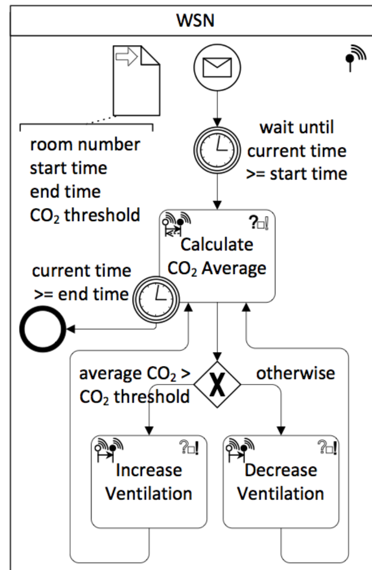


Figure 3.10: The CO_2 and ventilation operations modeled with the parametrized WSN logic constructs [81]

average value. The icon on top-right part shows how the task is a command tasks with a return operation, which *tells* the nodes of the room (*room number* is an input of this process) to execute *sense* CO_2 and to *report* the average value; the icon on the left-top of the task indicates that it is a sense operation (the question mark is in bold). Similarly, the tasks for the management of the ventilation are specified as command action, yet this time they execute an actuate operation (icon on top-right with the exclamation mark in bold) to trigger the ventilation. As for the sub-process, this modeling allows the deployment of the logic in multiple parts of the network. For example, this logic can be deployed in all the meeting rooms. Differently from the previous abstraction, the meta abstraction logic is specified with parameters instead of nesting operations.

Translation and Execution.

The WSN-aware part of the process is a standard BPMN model that can be executed by a process engine exterior to the WSN. The intra-WSN part, instead, is translated into executable code. A tool for translation called *model compiler* takes this part of the process and generates code implementing a custom execution engine. The executable program hence behaves similar to a regular BPMN engine interpreting the given BPMN model. The generated program implements a finite state machine, realizing the execution semantics of the translated process model including instance management and message correlation, and of course keeps track of all execution tokens in each process instance as specified in the BPMN 2.0 specification.

For example, an exclusive diverging gateway will be translated into a series of if statements (mapping the conditions on the outgoing flows) in the "main loop" of the program. Each WSN Task is translated using the *WSN logic construct* composition describing sensor logic. This is the most extensive generation step, as these sub-models need to be mapped to an API for instantiating, managing, and using those programming abstractions. The system-description describes the characteristic of each node of the network and it is used as input for the translator. The EoI of a WSN Task is mapped to attribute matching at run-time, e.g. if a WSN Task has been configured to operate on EoI "Floor 1" and the system developer contains information which room ids belong to that floor, this could be mapped to the expression `location='room1.1' or location='room1.2'`.

The two parts of the process can now be executed separately. To make them communicate, the model compiler maps the message flows between intra-WSN and WSN-aware process to communication endpoints that are created automatically on either side, enabling each part to receive and send

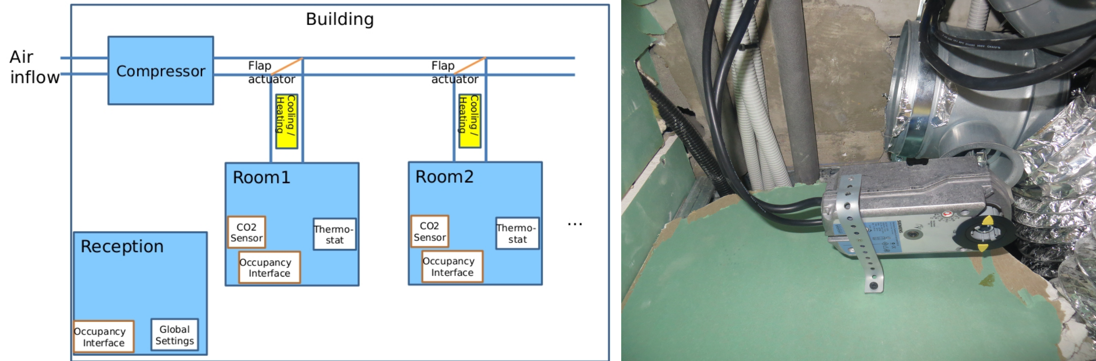


Figure 3.11: Deployment of the ventilation scenario in Cadiz, Spain. On the left part an overview of the setup and on the right part an actuator with the flap. [25]

messages. As the message format and transmission encoding are out of scope of the BPMN specification, a simple message format and an efficient transmission encoding are defined and implemented in both the generated intra-WSN executable and as an extension to a regular BPMN execution engine. In order to support the coordination of multiple instances, each message contains a field that is used for instance correlation and the execution of message start events creates instance IDs that need to be used by either side of a same process instance.

3.7 Evaluation of the approach

Within the `make`*Sense* project we run an evaluation to test the effectiveness of the modeling language and of the overall approach in a real-world deployment. Both are described in details in [33] and here we summarize the outcomes.

Evaluation of the `make`Sense approach A deployment of the ventilation scenario, similar to what we presented here as scenario, was deployed in Cadiz, Spain. A student dorm was equipped with sensors and actuators. A process was modeled to adjust the ventilation inside a student's room based

on the level of CO_2 and presence. The system ran correctly for a week, triggering the ventilation when the CO_2 level was above the threshold and the student was inside his room.

This real world deployment shows how the make*Sense* approach can be effectively used to model real-world applications. An estimation of the costs [33] shows how the presented approach can save more than 60% of the final cost compared to a conventional deployment.

Evaluation of BPMN4WSN To test if the BPMN4WSN approach simplifies the development of WSN process, our partner SAP AG ran a user study with 6 developers that did not have any previous knowledge of BPMN4WSN. The test was conducted on the composed WSN logic constructs, which is the version presented in this work, and that is the language integrated in the toolchain of the make*Sense* project. Among the participants one was familiar with WSN; three had an average knowledge of BPMN; and the others had little or no knowledge of both fields. The test case was based on the Cadiz deployment [25], and participants were asked to perform different modeling operations with various degree of difficulty, form opening the editor to specify static and dynamic targets (Table 3.1 shows the steps). Results of each modeling tasks were collected using a three values scale: *no success*, *success with help*, and *success*. After the modeling exercise a questionnaire was given to the users to collect feedbacks on a 7-point scale.

1	Open editor, select scenario
2	Open properties table
3	create Business Model Diagram
4	Create additional BPMN Pool and name "reservation system"
5	Create WSN Pool and name "WSN"
6	Create Start Event in WSN pool that reacts on a message

7	Create message flow from reservation system to newly created start event	
8	Create a message that is assigned to the message flow	
9	Use the message flow to trigger a room reservation. Search the XML file for the appropriate message name.	
10	Use a data object to store the in the previously defined message (use context menu of message object)	
11	Name the newly created data object "master data"	
12	Create event for delayed start of next process step	
13	Configure delayed time event to "15 min. before start of meeting"	
14	Create WSN Task and name it CO_2 monitoring	
15	Select the sensor nodes that should be addressed by this task by entering the right type in the StaticTargetExp field	
16	Use the DynamicTargetExp field for specifying the correct location as indicated in the received message.	
	Start abstraction editor and open property pane	
17	Instruct the previously identified sensor node to report values.	Part 2
18	Instruct the previously identified sensor node to sense CO_2 values. Make this local action a part of the reporting task. (save abstraction composition)	
19	Store the sensor sensing result in a data object call CO_2Data	
20	Use gateway to specify if ventilation is turned on or not. * model decision with two outcomes based on CO_2 values	
21	Configure out-going edges in such a way that ventilation is turned on if CO_2 is greater than 1000 ppm otherwise ventilation is not turned on. Manually label the edge with value ¿1000 "CO_2 too high", the other edge "CO_2 ok".	
22	Label the WSN task "start ventilation" and "stop ventilation"	
23	Chose the appropriate static and dynamic targets for turning on and off the ventilation. Program the actuators (vans). Hint: set the correct parameter for the local action in the abstraction editor.	
24	Ensure energy efficiency while process instance is sleeping until meeting starts	

Table 3.1: Exercise steps [33]

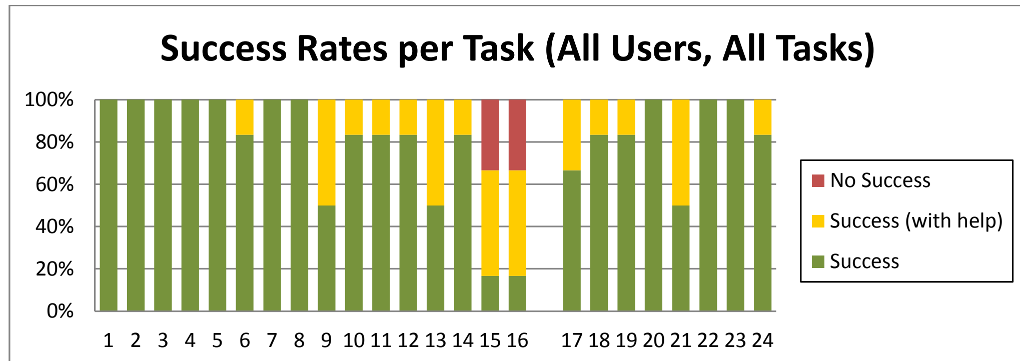


Figure 3.12: Task Completion [33]

The results (Figure 3.12) of the tests showed how, out of 24 modeling tasks, 22 were completed by all the users. Most challenging tasks were the ones that asked to specify static and dynamic targets with boolean expressions (only one user completed it without any help, and two were not able to complete the tasks at all). Yet, a similar tasks, which was presented later in the study (next to the last task), was successfully completed by all the users, showing how the required skill can be learned in a short time period. From the questionnaire emerged a positive feedbacks but also some difficulties in the use of the modeling tool, especially related to the meta-abstraction compositions and to the specification of the target for actions.

From the data collected with the user study emerges that the BPMN4WSN can be easily learned and used to create rather complex application in short time period. This study demonstrates how people with low or no prior knowledge of WSN were able to use the BPMN4WSN to model and program a WSN application. Part of the modeling relative to the WSN abstraction may be too complex for non technical people, yet their composition rules can be learned in a rather short time. Overall, from the study emerges that the BPMN4WSN achieves the goals we fixed: it can be used by people, even by non domain experts, to create and maintain

WSN applications, yet it still requires to learn the modeling language and how to use the new modeling components.

3.8 Discussion and Lessons Learned

Our approach was guided by the core requirement presented in Section 3.4, i.e., to integrate WSN programming into business process modeling. We address this requirement by offering unified modeling in one model editor, hiding model artifacts that are not relevant in a given modeling context, splitting work between application developer and system developer, and providing model compilation and execution as a custom engine in the WSN.

The work described in this chapter integrates WSNs with BPs, combining classical business process modeling with ad-hoc extensions for WSNs that hide low-level network details. This integration allows an application developer to design process logic both inside and outside the sensor network, without requiring intimate knowledge of how to program distributed computations inside a WSN; an intuitive understanding of EoIs and sensing and actuating actions is enough. The system developer instead only focuses on the refinement of WSN Tasks. The described tool-chain takes care of splitting the two logics (intra-WSN and WSN-aware) and of the binary code generation. Endpoints for communication between the business process and the network are created following the model of the process.

The main limitation, which also emerges from the user study, is that the modeling language may not suits application developers that do not have the competence of a system developer. In this work we tried to abstract WSN details to a high-level modeling language. However, the intrinsic complexity of WSN cannot be fully hidden. WSN logic constructs provide a high level modeling of WSN operations, yet this modeling convention may not be easy to use by people not familiar with WSNs. With the li-

braries, which are present in the editor, we tried to provide users with workspace-specific operations that can be used without additional configurations. This could foster the adoption and non experts can use WSN operations without the need to ask a refinement by a system developer. Similarly, the specification of static and dynamic targets is not always a simple task. We tried to simplify this operation by adding EoIs as the way to abstract targeting of WSN nodes. However, in some case the specification of EoIs may not be enough (e.g., for the sub-process which are executed on specific nodes of a room) that then requires to specify static and dynamic targets.

In summary, this work is of help for application developers that have to create applications for specific contexts, for which the operations are already defined and available in the editor, and for application developer that can be helped by system developer (or that have the knowledge of a system developer) for the refinement of the process. We do not see BPMN4Crowd as a solution that can be used by any user, due to the unavoidable complexity for the specification of WSN details.

3.9 Related work

Building commercially relevant applications on resource-constrained, networked embedded systems (the front-end) such as WSNs while integrating them into business processes of an enterprise (the back-end) is a complex, challenging task that has to be repeated for each combination of front-end and back-end. Numerous efforts have been made, aiming also at demonstrating the business benefit.

Approaching the problem bottom-up, i.e., from the WSNs, several solutions have been proposed to simplify programming. Although many programming abstractions have been introduced, most of them aim at simpli-

fying the activities of skilled WSN programmers [62]and cannot be used directly to specify high-level process constructs by domain experts without WSN expertise.

The COBIS project (`www.cobis-online.de`) aimed at integrating heterogeneous WSNs with back-end systems by providing a web service facade to the WSN's functionality. The proof of concept was trialled in an environment, health, and safety application scenario, more specifically by enforcing physical storage rules for hazardous goods managed in an enterprise system [78, 79].

The SOCRADES project (`www.socrades.eu`) targeted industrial automation with the goal to almost eliminate the need for any proprietary intermediate layers between embedded services and the business back-end by directly service-enabling devices themselves [43]. The approach was based on the WS-* family of web service standards and only for very resource-constrained and legacy devices a gateway/service-mediator concept was developed to enable those to participate in service orchestrations.

Other proposed solutions for modeling sensor network applications using a process-based design include the Graphic Workflow Execution Language for Sensor Network (GWELS) [36], which enables the design of data-flow as workflow, and an ad-hoc architecture for handling the communication. Similarly, [6] uses a process paradigm for defining WSN applications, easing the configuration for non-experts of the field. Mash-up composition is also promising; in [38], the authors wrap smart-objects with web services, introducing an architecture and a web-based mash-up tool for composition and execution. These solutions enable the modeling of WSN logic in a model-driven fashion but without deriving the executable logic of the network.

Recently, BPMN has gained interest as method to *program* WSNs. Caracas et al. [13, 14] presented studies on the expressiveness of the lan-

guage and its potential to be compiled into source code for WSN nodes. As results they produce a system that creates WSN applications by compiling BPMN processes. The outcomes highlight that, as it is, BPMN is powerful enough for specifying the high-level behavior (if modeled with correct patterns) more than low-level one. At the same time they prove how a process can be compiled into native source code for WSNs, without losing too much performance compared to hand-written code. These preliminary works show the possibility to compile the BPMN for creating binary code. However, the example shown in this work users a higher-level API, that does not allow one to fine-tune communication in the WSN as it is possible with our approach.

In the past months, extensions of BPMN for modeling smart objects have been proposed as outcome of the IoT-A (`www.iot-a.eu`) project [59, 77], an idea that shares some common ground with our approach. The idea is to extend the BPMN language to model Internet of Things (IoT) aspects. However, this approach differs as they propose modeling extensions that affect the language at a high level of abstraction; in fact their goal is to use this language to model IoT services instead of creating the logic from the process.

Approaches like SysML [86] are only remotely related to our approach. This modeling framework, derived from UML, allows the modeling of low-level details of a WSN system. Yet, SysML models are graphical models without a standard serialization, therefore they are not directly usable for process-based integration.

3.10 Conclusion

In the era of the IoT, collaboration and integration of non-conventional IT devices, such as entertainment and automotive equipment, RFID devices

and tags, or WSNs, with Enterprise services is of paramount importance [43]. In this chapter, we focused on one relevant representative of this need, i.e., WSNs, which typically still represent isolated and impenetrable realities from a business IT point of view. We proposed a layered approach for developing, deploying and managing WSN applications that natively interact with enterprise information systems, such as a business process engine and the processes running therein. We did not try to crack the whole problem at once, e.g., by aiming at a business-view-only approach to WSN application development, and rather foster current practice, equipping both the application developer (holding the process knowledge) and the system developer (holding the WSN knowledge) with effective languages and instruments to co-develop advanced, process-based WSN applications with non-trivial distributed sensing and actuation logics.

Chapter 4

Modeling and Enacting Flexible Crowdsourcing Processes

This chapter focuses on distributed, autonomous (and intelligent) actors, as the crowd of worker is, and presents our research to support the design and execution of processes whose logic is partially executed by a crowd of workers. We present the extensions of a process language (BPMN) that add domain-specific constructs to orchestrate tasks executed by the crowd and to design their internal execution logic (called tactics). As part of this work we also present the crowd computer, a flexible crowdsourcing platform that, by making available crowdsourcing operations, enable the execution of crowd tasks without imposing pre-defined choices or tactics. This chapter is an extract of [82]. On the same topic, we have also published a paper [50] that introduces the ideas and concepts that are developed in this chapter.

4.1 Introduction

Since their invention, computers were the main source of computational power able to generate solutions in a split second. However, even with the advancements in IT, computers are still not able to solve all the kinds of problems that one may face, such as to identify a good enough picture to advertise a new restaurant. These are tasks that human beings can solve easily, but that are less efficient with pure computations.

Crowdsourcing is a relatively new approach to execute tasks that also require human capabilities instead of only machine computations. It refers to the practice of outsourcing a work to an undefined and large network of people via an open call for contribution [40]. This approach is based on crowdsourcers (companies or individuals that crowdsource a work) that make available their tasks of work to the crowd instead of assigning them to employees. Workers, people who execute a task, can accept and perform the tasks, receiving a reward when the crowdsourcer decides that their result is accepted. The power of crowdsourcing lays in its workforce, the crowd, which is large, always available, and can be requested on demand. Crowdsourcing platforms, which are web applications such as Amazon Mechanical Turk (AMT), help crowdsourcers in managing the crowd (e.g., with solutions to select workers based on skills), tasks (e.g., with systems to post and dispatch tasks and collect results), and connect crowdsourcers and workers. Ideally, platforms should allow crowdsourcers to create *any* kind of crowdsourcing task, even as complex as writing an encyclopedia like Wikipedia. However, existing platforms implement fixed and pre-defined logics, such as how tasks can be executed by the crowd and how results are collected, logics that instead may differ from one crowd task to another.

Today, crowdsourcing is typically adopted to solve *atomic tasks* many of which are creative (e.g., logo creation) and micro (e.g. tagging of pic-

tures) tasks. *Crowdsourcing processes* (CP), which require more than one atomic task or a more sophisticated execution logic, are created with ad-hoc solutions [51]. This type of logic requires to specify the control flow, to describe in which order tasks are executed, and the data flow, to describe how data is produced or consumed by each task. This makes the applications highly-process driven [45, 50, 60, 72].

The goal of this work is to enable the creation and execution of crowdsourcing processes. We propose a new kind of crowdsourcing platform and a modeling language to program crowdsourcing processes. The platform is inspired by the idea of a computer that, instead of pure CPUs, has humans and machines as computational units; we call it *crowd computer*[1]. The crowd computer exposes a set of *API* each of which abstract the operation of the units, the blocks of the platform, each of which has a specific goal, such as managing the crowd. To specify this logic, i.e., to program the crowd computer we present a modeling language created to be used at different levels of granularity: one level to specify the logic of the process with human, crowd, an machine tasks; one level to specify the execution logic of crowd tasks, and a level to specify the configuration of crowd task internal based on reusable patterns. With this abstraction a modeler (the person who creates the process) can specify the logic of the whole application and also define and refine the internal execution logic of each task. The contributions of this work are:

- A crowdsourcing platform, the *crowd computer* (as extension of [50]), that supports the development of applications where the key work performer is the crowd.
- A BPMN-based modeling language that support the modeling of crowdsourcing processes and tasks and the management of data which is

[1] `www.crowdcomputer.org`

generated and consumed by the crowd. A visual modeling environment to allow crowdsourcer to program crowdsourcing processes.

- A compiler that transforms crowdsourcing processes into executable processes and that enable the deployment of processes on a process engine for their execution and their integration with the crowd computer.

The chapter is structured as follow, in Section 4.2 we introduce crowdsourcing. In Section 4.3 we introduce the requirements and explain our approach toward a solution. In Section 4.5 we present the crowd computer. Section 4.4 and Section 4.6 are dedicated to the explanation of the process language extensions, specifically to the crowdsourcing processes and crowd tasks, with relative tactic and configurations. In Section 4.7 we discuss our implementation of the solution, and in Section 4.8 evaluate it with an use case. In Section 4.10 we review related work.

4.2 Crowdsourcing: Concepts and State of the Art

Crowdsourcing is a young, yet already complex practice, especially as for what regards the different ways work can be outsourced and harvested. In the following, we conceptualize the necessary background and define the problem we approach in this article.

4.2.1 Core concepts

Howe [40] defines crowdsourcing generically as "the act of taking a job traditionally performed by a designated agent (usually an employee) and outsourcing it to an undefined, generally large group of people in the form of an open call." We specifically focus on crowdsourcing in the context of the Web and on work that is crowdsourced with the help of so-called

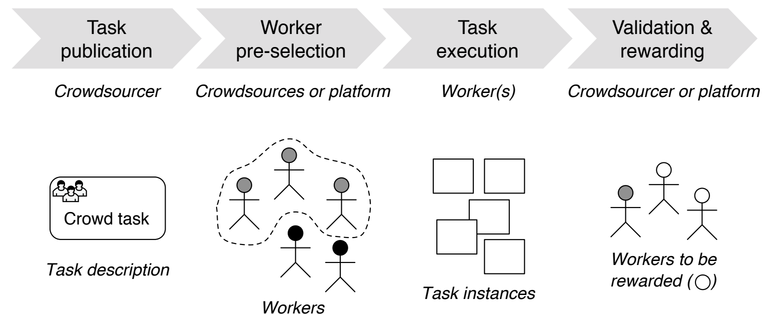


Figure 4.1: The high-level steps of crowdsourcing and the respective actors

crowdsourcing platforms, which are on-line brokers of work that mediate between the *crowdsourcer* (who offers work) and the *workers* (who perform the work). These latter form the *crowd*.

Crowdsourcing a task using a platform typically involves the steps illustrated in Figure 4.1 (not all steps are mandatory): The crowdsourcer *publishes* a description of the task (the work) to be performed, which the crowd can inspect and possibly express interest for. In this step, the crowdsourcer typically also defines the reward workers will get for performing the task and how many answers he would like to collect from the crowd. Not everybody of the crowd may, however, be eligible to perform a given task, either because the task requires specific capabilities (e.g., language skills) or because the workers should satisfy given properties (e.g., only female workers). Deciding which workers are allowed to perform a task is commonly called *pre-selection*, and it may be done either by the crowdsourcer manually or by the platform automatically (e.g., via questionnaires). Once workers are enabled to perform a task, the platform creates as many *task instances* as necessary to collect the expected number of answers. Upon completion of a task instance (or a set thereof), the crowdsourcer may inspect the collected answers and *validate* the respective correctness or quality. Work that is not of sufficient quality is not useful, and the crowdsourcer *rewards* only work that passes the possible check.

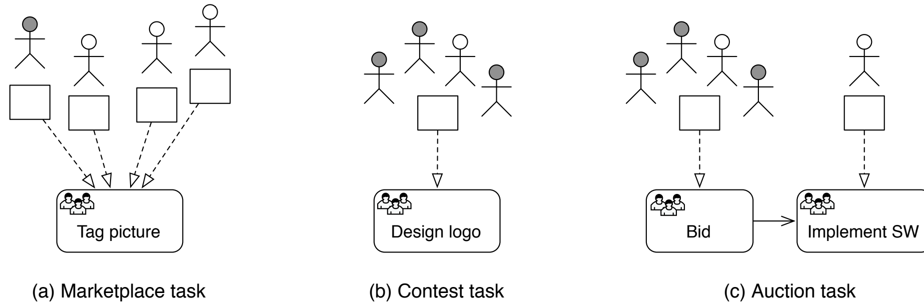


Figure 4.2: The most prominent tactics to crowdsource work

4.2.2 Crowdsourcing tactics

Depending on the *acceptance criteria* by both the crowdsourcer and the worker to enter a mutual *business relationship* (after all, this is what crowdsourcing is about), different *negotiation models* may be adopted to crowdsource a piece of work. For simple tasks (e.g., tagging a photo), it is typically not worth to start a complex negotiation process; more complex tasks (e.g., designing a logo or developing a piece of software), instead, may justify a process in which crowdsourcer and worker commonly agree on either the quality of the delivered work or its reward. Since it is the crowdsourcer who starts the crowdsourcing process and approaches the crowd, we call these negotiation models *crowdsourcing tactics* (often also called *crowdsourcing models*).

Three major tactics have emerged so far (see Figure 4.2):

(a) *Marketplace*: The marketplace tactic targets so-called micro-tasks of limited complexity, such as tagging a picture or translating a piece of text, for which the crowdsourcer typically (but not mandatorily) requires a large number of answers. Usually, the acceptance criteria by the crowdsourcer for this kind of tasks are simple and clear, e.g., all answers are valid or only answers that pass a given correctness check. Rewards for micro-tasks commonly range from nothing (workers perform tasks for fun or glory), to few cents or dollars, without

any margin for negotiation. If workers find the offer fair, they perform the task, otherwise they skip it. Prominent examples of crowdsourcing platforms that implement the marketplace tactic are Amazon Mechanical Turk (`https://www.mturk.com`), Microworkers (`http://microworkers.com`), and CrowdFlower (`http://crowdflower.com`).

(b) *Contest*: The contest tactic is particularly suitable to creative tasks for which the crowdsourcer knows the budget he is willing to spend, while he does not have clear criteria to decide which work to accept. Designing a logo or the layout of a web page are examples of tasks that fall into this category. In order to enable the crowdsourcer to clarify his criteria, this tactic invites workers to conceive a solution to a task and to participate with it in a contest. Once a given number of contributions or a deadline is reached, the crowdsourcer can inspect all contributions and choose the solution he likes most, thereby electing the winner of the contest (there could be multiple winners). Only the winner gets rewarded. Examples of crowdsourcing platforms that implement the contest tactic are 99designs (`http://99designs.com`), InnoCentive (`http://www.innocentive.com`), and IdeaScale (`http://ideascale.com`).

(c) *Auction*: The auction tactic targets tasks for which the crowdsourcer has relatively clear acceptance criteria, but for which he is not able to estimate a just reward. Coding a piece of software is an example of this kind of task. An auction allows the crowdsourcer to publish his requirements and workers to express the reward for which they are willing to perform the task. Typically, but not mandatorily (this depends on the adopted auction model), the worker with the lowest offer gets assigned the task and is payed accordingly upon delivery of the agreed on work. An auction can thus be seen as a combination of

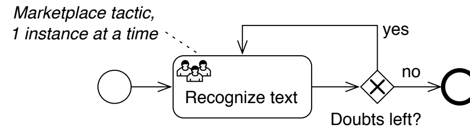


Figure 4.3: A simple crowdsourcing process in BPMN [67]: the text recognition task is iterated automatically until there are no doubts left about the correct wording

a contest (to win the auction) and a marketplace task with pre-defined worker assignment (to perform the task). An example of auction-based crowdsourcing platform is Freelancer (`http://www.freelancer.com`), which allows programmers to bid for the implementation of software projects.

The latter two tactics aim at producing one results that satisfies the crowdsourcer's need. The marketplace tactic, instead, most of the times aims at producing a large number of results that jointly satisfy the crowdsourcer's need. For instance, the quality of a translation is higher the more workers contribute to it. Aggregating results and coordinating workers, is however out of the scope of crowdsourcing platforms.

4.2.3 Crowdsourcing processes

We call the structuring of multiple crowd tasks and task instances, in order to achieve a common goal, a *crowdsourcing process*. The goal of crowdsourcing processes is to distribute work to workers, coordinate workers, check quality, and/or integrate individual results into an aggregated one – all aspects that otherwise the crowdsourcer would have to manage manually. For example, the model of the crowdsourcing process illustrated in Figure 4.3 shows how to iteratively crowdsource the recognition of a line of text until the last worker has no doubts left (inspired by [55]).

Since these kinds of crowdsourcing processes are not natively supported by crowdsourcing platforms, a set of programming frameworks and higher-

level platforms have emerged, which are built on top of existing crowdsourcing platforms (most notably, Mechanical Turk) and extend them with additional features for the management of processes. Turkit [55], for instance, proposes a JavaScript-based scripting language for the development of human computation algorithms, e.g., based on iterative, sequential task executions. Jabberwocky [4] is a parallel programming framework inspired by MapReduce [30], with an own scripting language and support for crowd and machine tasks. CrowdForge [47] is similar in spirit to Jabberwocky, but the map and reduce steps are both performed by the crowd. Turkomatic [51] proposes a collaborative, divide and conquer approach in which the crowdsourcer and the workers can split tasks arbitrarily and merge results without the need for programming or to follow the rather rigid structure of MapReduce. Finally, CrowdWeaver [45] proposes an own graphical notation to model crowd processes with dedicated operators for data flows and transformations.

4.2.4 Problem statement

All aspects from the publication of a single task and the selection of a suitable crowdsourcing tactic to the design of an integrated crowdsourcing process affect the *quality* of the final outcome of a crowdsourced work. Making each step right makes crowdsourcing *complex*.

In this chapter, we focus on the model-driven development and execution of (i) *custom crowdsourcing tactics* and (ii) *crowdsourcing processes*. We do not further elaborate on how to most effectively describe tasks or on how to fine-tune rewards, so as to maximize crowd participation or quality. These are aspects that very strongly depend on the specific task to be crowdsourced, and good studies of the topic already exist [5, 32, 41].

As for the *tactics*, the three tactics described above are just the most prominent ones emerged today. The problem is that these tactics are cur-

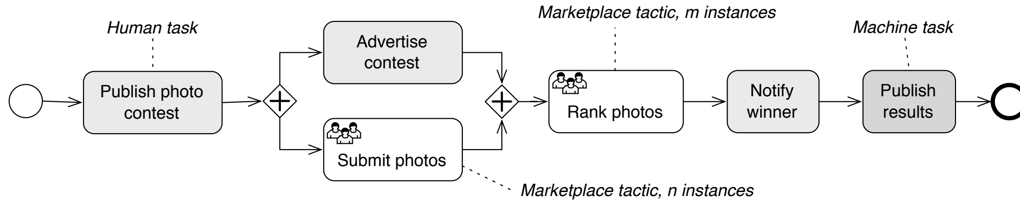


Figure 4.4: A crowdsourcing process involving different actors (humans, machines and the crowd) and possibly different crowdsourcing tactics

rently hard-coded inside crowdsourcing platforms; each platform has its own tactic with proprietary pre-selection, quality assessment and rewarding logics; and they all require a significant amount of manual labor by the crowdsourcer. It is not possible to freely choose and fine-tune how to negotiate a task with the crowd.

The process in Figure 4.4 illustrates how to implement a photo contest where the crowd both submits photos and ranks photos (crowd tasks), while the crowdsourcer takes care of initiating the contest, advertising it and notifying the winner (human tasks). The process is closed by a machine task that automatically publishes the results of the contest. The two crowd tasks internally adopt a marketplace tactic, which logic could be designed by the crowdsourcer just like he designed the process model in the figure.

There are multiple contexts that may benefit from this kind of advanced crowdsourcing processes, such as:

- *Product design*: Early feedback to new products is crucial to success. Crowdsourced feedback or even testing, if properly integrated into production processes, can be a significant competitive advantage.

- *Social marketing*: Marketing campaigns are increasingly conducted online. The integration of crowdsourcing into common marketing processes may allow organizations to boost and monitor their social presence.

- *Idea management*: Increasingly, organizations engage the crowd to ideate new products or services. Common social networks do not provide adequate support for this, and idea management systems may be too rigid. Custom crowdsourcing processes may make the difference.
- *e-Democracy*: In line with recent trends, crowdsourcing may enable the participation of the civil society to politics. How to involve society (e.g., via voting, promoting petitions or similar) is as crucial as election laws are. Each party may have its own preferences and goals, i.e., crowdsourcing processes.
- *Human computation*: Despite the increasing computing power of machines, there are still tasks that only humans can solve, e.g., telling whether a portrait photo is beautiful or not. Advanced crowdsourcing processes enable the flexible integration of both humans and machines, unleashing the computing power of both.

For the *crowdsourcing processes*, the state of the art is that they are still mostly executed manually. The frameworks introduced above do provide limited support for automation, alleviating the burden on the crowdsourcer. What is still missing is support for crowdsourcing processes that are natively integrated with common business process management practices, that bring together the crowd, individual actors and the machine, and that allow for the crowdsourcing of tasks using different tactics, depending on the specific needs of the crowdsourcer.

4.3 Modeling and enacting advanced crowdsourcing processes

Supporting the modeling and enacting of crowdsourcing processes is a complex task, especially if the aim is to provide a solution to create both crowd-

sourcing processes and crowdsourcing tactics with a single abstraction. A solution that can be used by crowdsourcers with various backgrounds and not only by technicians or expert of the field. We envision the typical crowdsourcer that uses our solution as a person that has: a background on business process, thus that knows the basics of process modeling; knowledge of crowdsourcing; a basic development knowledge of web pages; and that is able to understand how data are generated and consumed by the task he creates. We require the crowdsourcer to create web pages because we see crowd applications as complex and structured processes that may need different user interfaces (UIs) for different tasks. For this reason we envision a system where a crowdsourcer creates the UIs that are later presented to workers. Similarly, the crowdsourcer also needs to have a clear idea about what type of data are created and consumed by each task, this to manage and transform the data of tasks.

4.3.1 Requirments

We analyzed extensively crowdsourcing processes and we derived the requirements for the creation of crowdsourcing processes and for the execution. For the modeling we need to support:

R1 *Crowd tasks.* Crowd tasks are the essential part of crowdsourcing processes. They describe the work that is assigned to the crowd. This task has to allow a crowdsourcer to specify the task characteristics, such as the description of the task.

R2 *Crowdsourcing tactics.* The execution logic of a crowd task, the *tactic*, may differ from task to task even if they are in the same crowdsourcing process. Then, we need to be able to personalize the tactic, deciding its type, and to *configure* the logic in which rewarding and validation are executed. This to allow the crowdsourcer to decide and configure

the execution aspects of every crowd task.

R3 *Human tasks.* Human tasks are used when a task has to be executed by a designated human actor (not the crowd), such as the crowdsourcer. Human tasks are important, in that they allow a person to control the execution of the process, for example the crowdsourcer has the possibility to validate the task results.

R4 *Machine tasks.* Not all the logic of crowdsourcing processes can be executed with only human or crowd tasks. There may be the need for executing logics that can be performed by a machine, for example an operation to compute the average of a series of data extracted from the crowd results. For this reason we need the possibility to execute machine tasks.

R5 *Control flow.* Crowdsourcing processes are composed of various tasks. It is then necessary to be able to specify the order in which tasks are executed. The control flow does not only define the order, but also contains *control flow statements* (decision points) whose results may change the path to follow (e.g., an *if* condition).

R6 *Data flow.* Control flow specifies the execution order of the tasks. Yet, tasks produce and consume data, thus they need to have access to information. Data can be propagated in different fashion, e.g., sharing data with all the tasks or following a flow. Having a solution to specify the data flow, and how information are propagated, is then important. Data flow gives to a crowdsourcer the possibility to define clearly what data are produced and consumed by each task and where these data have to be sent or from where they have to be read.

R7 *Data transformation.* Data transformation is another fundamental aspect in crowdsourcing processes since crowd tasks consume and pro-

duce data. For example, input data of a crowd task has to be divided into partitions to be handled by each worker, while the task outputs have to be merged into a unique solution. Data have also to be loaded from external sources or filtered to remove results that are not satisfactory, operations that right now are implemented by crowdsourcers in external applications.

For the execution we need to support:

R8 *Engine for crowdsourcing processes.* To run a crowdsourcing process we need an engine able to execute all the tasks that are permitted in a crowdsourcing process. Thus, the engine has to support the execution of crowd, human, machine, and task for the transformation of data.

R9 *Deployment of tasks on crowdsourcing platforms.* Crowd tasks are executed on crowdsourcing platforms. This requires to support the automatic deployment of tasks and tactics on crowdsourcing platforms. The deployment requires to create a communication channel to receive crowd task results and to send task parameters and runtime information to the crowdsourcing platform from the process engine.

R10 *Management of data.* The platform has to offer support for data management, providing a solution to execute common data operations that are configured by the crowdsourcer in the model of the process. Since the crowdsourcing process specifies the data and control flow the platform has to support the execution of both accordingly to what is designed in the process model.

4.3.2 Approach

In Figure 4.5 we depicted a possible architecture of our approach. It is divided in three parts: one for the creation, thus the modeling of the

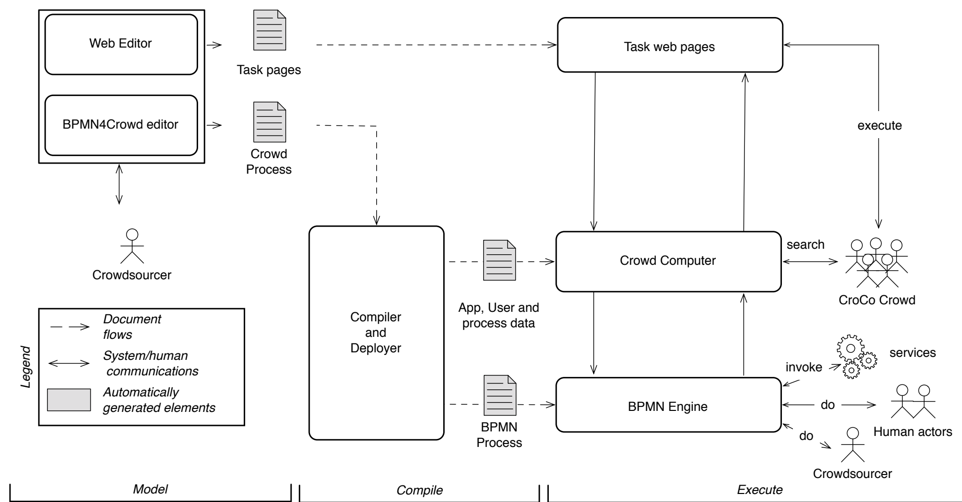


Figure 4.5: The architecture of our approach.

crowdsourcing process (model); one where the process is translated into executable code (compile); and a third part where the process is executed (execute).

For the *execution* of crowdsourcing processes (right part of the Figure 4.5) we use a business process (BP) engine, since it already supports part of the requirements (**R8**). A BP engine gives support for the execution of human task, machine task, control flow, and data flow. Data management operations are not part of a BP engine, thus we add to the engine the logic to support data operations (**R10**). For crowd tasks and crowdsourcing tactics we introduce the **crowd computer** (CroCo) that supports the development of applications where the key work performer is not the CPU but the people. The crowd computer provides primitives to, e.g., start and stop tasks, to assign them, to approve or reject the results, to reward, and it is able to keep track of work assigned, done, pending, and of the performances of each worker. With these primitives, programmers can specify programs and strategies on top. Therefore, *tactics* are encoded as reusable program templates or developed ad hoc by a crowdsourcer. Instead, we do not see the crowd computer as providing specific

support to the execution of the human task per se, the task execution data, and the computer-human interface. Each application has its own requirements needs for a specific UI. In our approach UIs are created by the crowdsourcers as external web pages.

The *compile* part of the architecture transforms crowdsourcing processes into process that can be executed by the engine, adding information that are needed for the execution. In this phase the system also deploys tasks on crowdsourcing platforms (**R9**), retrieving the information from the process model.

For the *modeling* of crowdsourcing processes we create an editor called BPMN4Crowd editor. This editor allows the creation of crowdsourcing process with the BPMN4Crowd language. This language is an extension of a business process modeling language, specifically BPMN (Business Process and Modeling notation) [67]. Already in [50] we proposed the use of BPMN as an intuitive way to express a crowdsourcing process that coordinates the work of a multitude of crowd members. The rationale behind the choice of a process language as BPMN is that crowdsourcing application logics can be expressed as processes. Moreover, by its nature, BPMN already satisfies some of the requirements we have, in particular: the possibility to model human tasks (**R3**) and machine tasks (**R4**), and to specify the control flow (**R5**) and the data flow (**R6**). Yet, the BPMN language lacks of components to model crowdsourcing processes. We approach this problem with a modeling notation to support crowdsourcing processes, called BPMN4Crowd. Based on the idea to have access to operations for basic crowdsourcing components (e.g., crowd, task, reward and quality), which are provided by the crowd computer, the BPMN4Crowd abstracts the operations and crowdsourcing functionalities with ad-hoc constructs, such as one for the crowd task or for data tasks. We introduce three conceptual layers of modeling (depicted in Figure 4.6). The layers abstract,

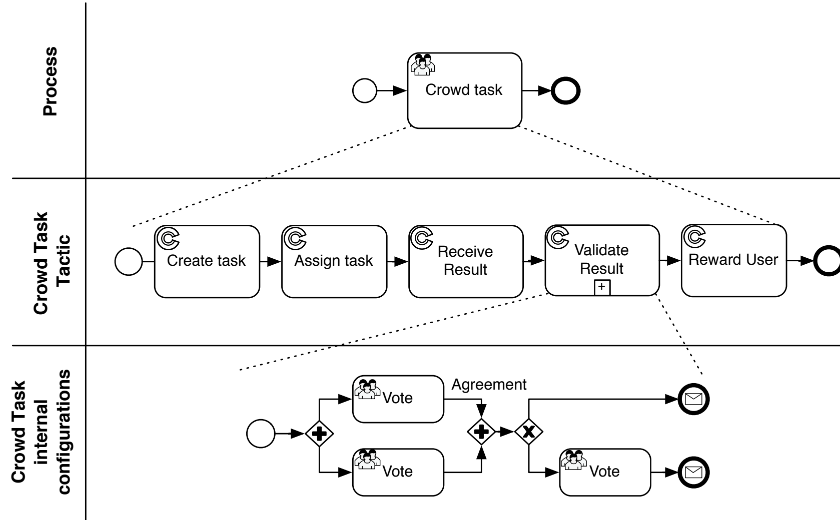


Figure 4.6: The visual representation of the layer approach.

with different granularities, the aspects that crowdsourcers have to specify to crowdsource a process: (i) the *crowdsourcing process* logic that contains the *crowd*, *machine*, *human* and, *data tasks*; (ii) the *tactic* of each task; (iii) and the *configuration* of each tactic. Each layer abstracts one of these points:

1. The first, high-level abstraction, is the *process layer*. This is the place where crowdsourcers model the *process logic*, which describes the *control flow* (**R5**) and *data flow* (**R6**) of *crowd* (**R1**), *human* (**R3**), *machine* (**R4**), and *data tasks* (**R7**). It is at this level that the logic of the crowdsourcing process is modeled.

2. The second level is the *tactic layer*. At this level crowdsourcers decide how to approach the crowd and how to manage the internal logic, the *tactic*, of each crowd task (**R2**).

3. The third, and lower, level is the *configuration layer*. This is the level where crowdsourcers decide the internal aspects of a tactic, the *configuration* (**R2**), such as the pre-selection of workers, how to reward

workers, and how to validate their work, aspects that typically can be implemented with various logics.

Tactics and configurations of tasks are very particular processes, which are not easy to model. We approach this problem by creating and providing various patterns that describe rigorously the most important tactics and configurations. These patterns are part of the language and can be used by crowdsourcers by choosing the desired one and set the parameters for the execution. With this approach we allow a crowdsourcer to choose and execute the most common tactics without modeling their processes.

4.4 Modeling Crowdsourcing Processes: BPMN4Crowd

In this section we introduce the higher level of modeling, which is where crowdsourcing processes are created. This level abstracts crowdsourcing concepts at an high level, hiding the detail to lower levels, allowing also non developer to create crowdsourcing processes. In the top part of Figure 4.6 there is depicted the most simple crowdsourcing process.

To enable the modeling of crowdsourcing processes we have to introduce new construct to the BPMN language to model crowdsourcing aspects and operations, for example a task to give the modeler the possibility to specify a task for the crowd. At this level of abstraction we introduce two new tasks: the *crowd task*, to create the tasks that have to be executed by the crowd; and *data tasks*, which implements the operations to manipulate the data that are produced or consumed by the tasks of a process.

4.4.1 Crowd task

First construct that we introduce is the *crowd task* to describe the work that the crowd has to execute. The crowd task (Figure 4.7) does not differ

from a standard BPMN task only by its icon, which represents a crowd, but also by the fact that this task is deployed and executed on the crowd computer. Within the task definition we introduced additional parameters to specify crowd related information, which are:

- *Description:* to specify the instructions that workers have to follow.
- *User interface:* to specify what UI has to be used as task interface.
- *Number of instances:* the number of task instances that have to be created, this represents how many instances must be created and executed.
- *Deadline:* a date and time after which the task expires.
- *Validation strategy:* to specify how the validation of worker results is conducted.
- *Reward:* to specify what is the reward for the job (e.g., 10 dollars).
- *Reward strategy:* how the reward will be give (e.g., to the best).

An important aspect of crowdsourcing is the pre-selection of the crowd. Standard BPMN grammar uses the concept of *lanes* and *pools* to define the actors. Yet, the task assignment to an actor is made at modeling time and the actor has to be *clearly* identified. In crowdsourcing, *clearly* identifying the person who executes a task at modeling time is not possible. The practice in crowdsourcing is that the requester defines a set of skills that a worker has to meet to be eligible to execute a task. For the crowd selection, instead of extend the *lane*, we decided that the pre-selection criteria is specified directly inside the properties of crowd tasks. This to limit the number of lanes that could be created if a process requires a different crowd for each task.

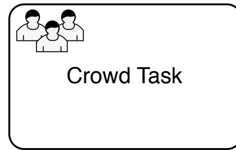


Figure 4.7: The crowd task.

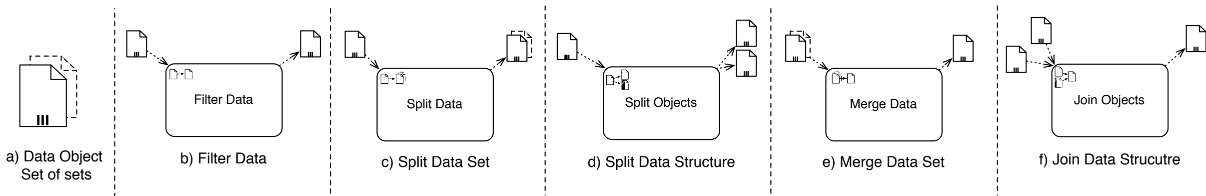


Figure 4.8: The data transformation notations.

4.4.2 Data transformation

BPMN has artifacts to specify the *data objects* and connections elements to specify the *association* of data. While the *association arrow* is enough to specify the data flow, the *data object* is not enough to specify the data that are produced or consumed by a crowd task. This artifact does not communicate any information on the structure of the data, making difficult for a modeler to understand the data structure that a task produces or consumes. In the standard BPMN data objects of any form (set of object or set of set of objects) are represented with the same element: the *collection*, which is a data object with three vertical line at the bottom. To discriminate the two structures of the data we introduce the *collection of collections* (*set of sets*) data object (the first element on the Figure 4.8).

Data management embraces also the possibility to execute operations on data. In crowdsourcing processes there is a need for *split*, *merge* and *filter* data. To help modelers we introduce a set of *data tasks* that represent common data operations. The benefit of this extension is to give to modelers a set of reusable operations that they can use and configure. To be more general, and to support different use cases, we created two sets

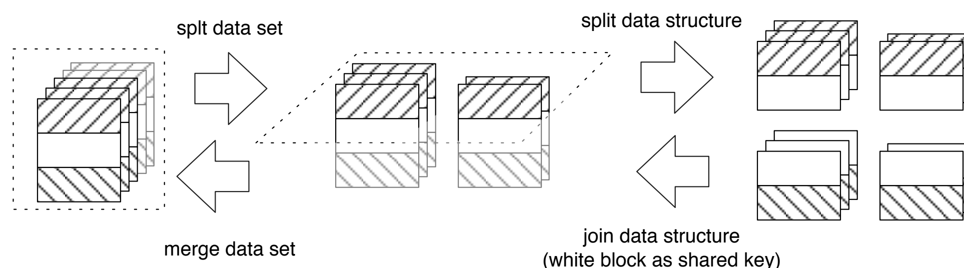


Figure 4.9: The data and object operations.

of operations: one to manage the *data set* and one to manage the *data structure*. The former work on a set level and are used to manage process data , such as inputs or outputs of a task (e.g., the set containing all the workers results) while the latter on the structure of the data(e.g., works on the data of a single worker's result). These operations are:

- *Filter data.* This function is created to filter a set of items, thus maintaining the data that match the condition and removing the others. It takes as input a set of elements and produces as output a sub-set of elements. For example, it can be used to filter the work of the crowd keeping the result sent by workers who did answer correctly at the question of the task. The filter data task is represented in Figure 4.8 item b). It takes in input a data item and returns a data items.

- *Split data set.* This function is used to split in partitions a dataset, it takes as input a set of items and produces as output a set of sets. The split data set task is represented in Figure 4.8 item c). The Figure4.9 (left part) shows how the split data works, it divides the list of objects in two lists (the dotted plan is where the cut is made). The split data operation does not have a singular execution logic, in fact there are different ways in which a dataset can be split into subsets. In Figure 4.10, we depict the visual representation of the split operations that we introduce here. The first element on the left is

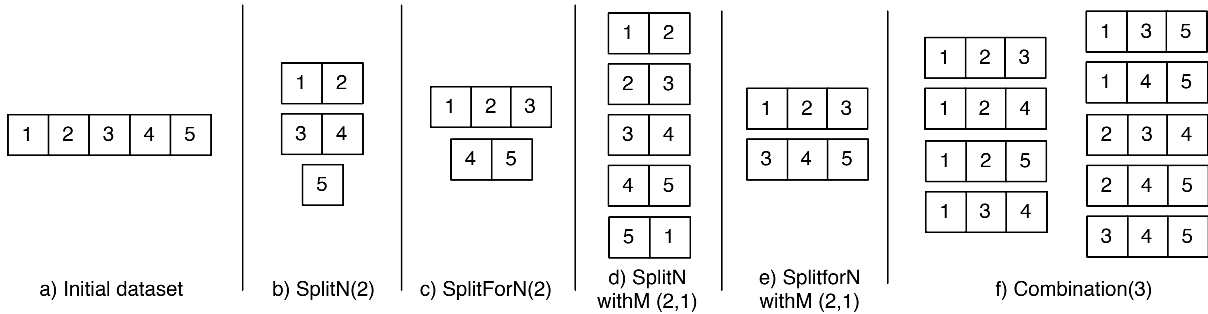


Figure 4.10: The different split data set operations.

the initial dataset while the others are the result of the operations. Operations that are:

- *SplitN:* this function is one of the most used, it splits the dataset in subset of N elements each. It is used when a requester wants to decide the size of the partition instead of the number of resulting subsets. In Figure 4.10 b) the result of the splitN (with N equal 2) applied on the initial dataset a). In this case the operations creates three sets, the first two of 2 elements while the last is of 1 element, the remaining one. This operation is generally used when the crowdsourcer wants to assign to each instance a precise number of item without worrying of how many instances will be created.

- *SplitForN:* in this function the number N is used to specify how many partitions have to be created. This operation is useful when there is the need for splitting the data in exact N subsets. In Figure 4.10 c) the result of the splitForN (with N equal 2). The operation creates two sets, one of 3 elements and one of 2. This operation is used when the crowdsourcer wants to have the control on how many instances will be created (each split set generally is assigned to an instance).

  - *withM:* this function can be combined with the previous two. It implies the fact that, among the partition, M elements are shared with another partition. In Figure 4.10 c) and in d) the result of the splitNwithM (with N equal 2 and M equal 1) and of the split-ForNwithM (with N equal 2, M equal 1). The former operation creates six sets each of which composed of 2 elements 1 of which shared with the following set. The latter operation crates 2 sets with 1 element shared among the two. This operation is used to create redundancy of elements and then have more data that can be used to validate the workers results, such as checking if the answer of two workers on the same data is the same or not.

  - *Combination:* this function generates all the possible combinations of K elements, thus some elements are shared among the combinations. The operation uses the value K as number of elements of each combination. Also this operation is used to have redundancy of results.

- *Split data structure* This operation splits the structure of each item instead of separate the items of a set as the data split. The split data structure task is represented in Figure 4.8 item d). The Figure 4.9 (right part) shows how the split data structure works, it divides the the items contaned by each object. In this case the white rectangle is in the middle of the cut, thus it is placed in both the results (last part of the Figure). The split object logic is unique and what changes is the configuration that specifies how to split the item structure (where the cut is made). The requester selects the attributes from the structure of the object and in which new object they will be placed. This configuration allow the requester to split the object into various other sub-objects. The operation takes as input an item and produces as

output two, or more, items that are sub-parts of the initial item. If in input is given a set of items, the split operation is repeated for every item in the list. For example, this operation can split an object that has id; name; description; tag; attributes into two objects one composed of id; name; description and one of id and tag.

- *Merge Data set* This operation is used to merge, thus recompose into a unique set, various set of items. It is the counter part of the split data set, and it results is shown in Figure 4.9 (left part) where the result of the split is merged back into a unique set. The merging logic is unique and works at the item level, thus no configuration is required. It takes as input a set of sets and produces a set of items. This operation is generally used to recompose the results of all various tasks or task instances into a unique set of results. This operation is the counter part of the split data operation. If the merge data is applied on the result of the split data operation what comes out is the initial dataset. The split merge data set is represented in Figure 4.8 item e).

- *Join data structure* This operation is similar to the merge data but executed on the structure of items. In this operation two or more objects are joint together. It is the counter part of the split data structure, and it results is shown in Figure 4.9 (right part) where the result of the split data structure is merged back into a the sets. The logic of the operation is unique but requires a configuration to specify a *shared key* (which in the Figure is the white block) that is used to identify what are the items to merge among the whole set (this is the same logic of a join in sql, where an ID is used to merge the data of two tables). For example it can be used to join the results of two different tasks (two list of items) into an unique set of items, using the id of the user as identification key. The join data structure is represented

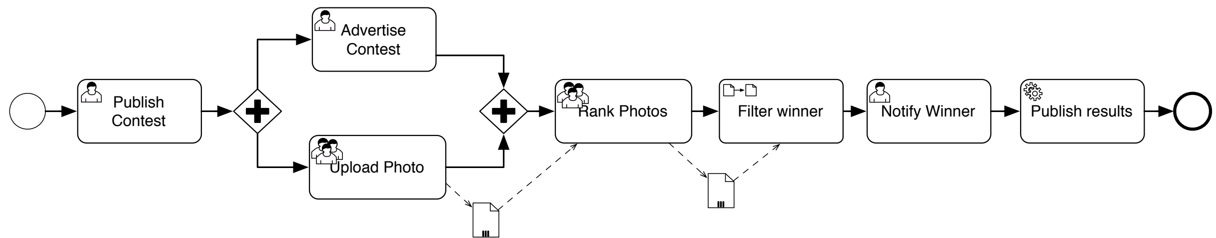


Figure 4.11: The process of the photo scenario modeled with BPMN4Crowd.

in Figure 4.8 item f).

4.4.3 Modeling a crowdsourcing process with BPMN4Crowd

In Figure 4.11 we modeled the process of the photo contest scenario with the BPMN4Crowd constructs. In the process we modeled two crowd tasks with relative data items and data flow. The first crowd task, the *upload photos*, asks the crowd to upload photos. For this task the crowdsourcer creates the task page, in which workers upload the picture, and set the parameters of the task accordingly to its need. The result of this task is a set of photos that the crowd has to rank. The entire set of photos is used as input for the the *rank photos* task, which has a similar configuration as the upload photos task and a dedicated page created by the crowdsourcer. In this task the crowd is asked to rank the photos. The result of this task is sent to a *filter data task* that is used to keep only the most voted picture. The other tasks of the process are modeled as human tasks, assigned to the crowdsourcer, and as a machine task that automatically uploads the contest results on the website. With BPMN4Crowd we are able to create a crowdsourcing process for the photo contest scenario. However, at this point we are not (yet) able to define the tactic of each task. Task and tactics heavily depend on the crowdsourcing platform one wants to use and existing platforms generally implement a single tactic. To have the possibility to model and decide tactics of tasks we need a crowdsourc-

ing platform that supports the creation and execution of various tactics without imposing pre-defined choices.

4.5 The crowd computer

For the execution of crowdsourcing processes we need a BPMN engine, which is in charge of executing the process, and a crowdsourcing platform to crowdsource the crowd tasks. Existing platforms only allow a limited personalization of crowd tasks (e.g., it is not possible to decide the tactic of a task), and often implement predefined choices that one cannot change (e.g., the type of payment). The crowd computer (CroCo) is created to provide a platform that gives a basic support to crowdsource various types of crowd tasks without imposing pre-defined tactics or configurations.

Figure 4.12 illustrates the essentials of the crowd computing environment. The crowd computer is composed of a central unit, the crowd engine, that receives the API calls and routes them to the different sub *units* (task, crowd, quality and reward managers). This give us the possibility to use the crowd computer to execute crowd tasks with various *tactics*, since tactics can be specified with different sequence of API calls. The crowd computer can also be used as proxy to publish a task into other crowdsourcing platforms. Yet, in this case, the execution of a tactic, different from what the chosen platform supports, is not possible.

To have a flexible execution of crowd tasks we designed the crowd computer with different units, each of which offers specific functions:

Task manager comprises a set of operations for task life cycle management and data propagation among tasks. Operations are activating a crowd task so that it can be instantiated (executed) by the crowd, assigning it to a crowd worker, canceling it while in execution, re-running it, deactivating it, enacting machine tasks, etc. These operations parse

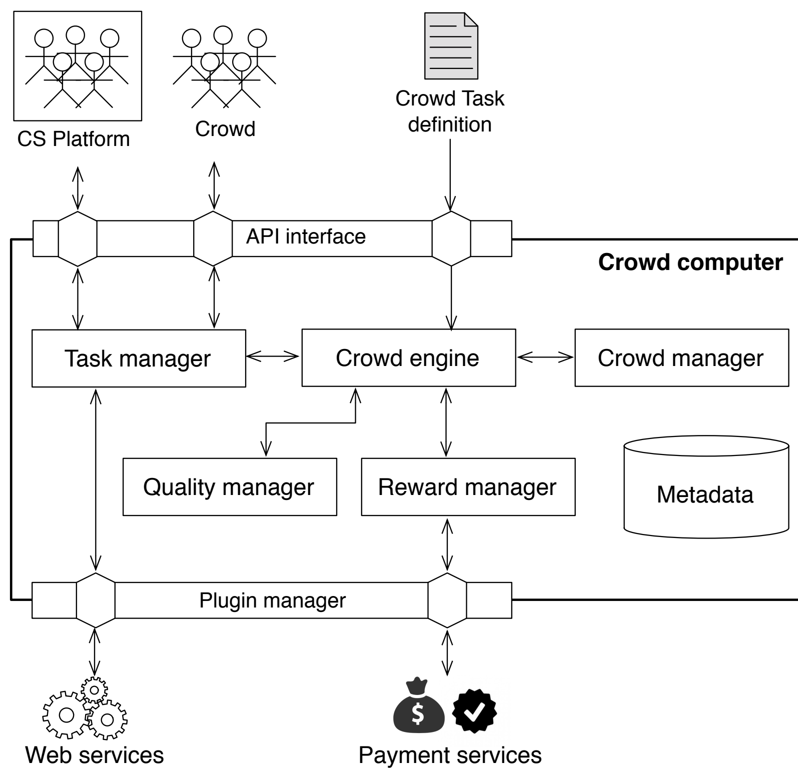


Figure 4.12: Functional architecture of the crowd computer.

the task definition with its configuration parameters (e.g., for the invocation of machine tasks), runtime parameters (e.g., for control flow decisions), and input and output data objects. Each task can be instantiated multiple times, so as to collect the amount of data the task is required to produce as output. Data objects store only references the application's task data repository; these are provided at runtime by the application to support data splitting and merging inside the CroCo. Data properties enable the association of descriptive information to data objects and enable, e.g., the definition of custom quality controls or the correlation of task instances referring to a same data object.

Crowd manager is in charge of human resource management and pre-selection. It comprises a set of operations for the management of users, such as resolving user roles, pre-selecting potential workers, keeping track of which user executed which task, sending direct invitations to people, ensuring separation of duties, etc.

Quality manager provides for the tracking of quality assessment tasks and respective evaluations. Specific operations are, for example, setting quality evaluations, checking threshold levels, filtering input data based on quality, etc. Quality is assessed via regular human/machine tasks; the CroCo only stores metadata. Quality control logics are implemented as reusable program templates or designed ad hoc. Data objects may have an associated quality and a quality threshold level. We propose that quality be simply expressed via numbers from 1-10, so as to equip the CroCo with a notion of quality that is can use to manage tasks. The threshold level separates "good" from "bad". The actual semantics of these quality levels and their computation is up to the application developer.

Reward manager is in charge of keeping track of which process instances have been rewarded, i.e., paid, and how. It provides for payment management. Each task may have an associated reward and payment service, yet actual payments may occur for individual task instances, bundles of task instances and similar. The crowd computer does also not impose any reward logic. Also reward logics can be implemented as reusable program templates or specified ad hoc. For example, processes that are modeled in the lower modeling layer (as presented in Figure 4.6) are process templates that the BPMN4Crowd offers for the reward of workers. These processes, which can be chosen from a repository, execute the logic of the reward and interact this unit only when the user has to be rewarded. The crowd computer does further not impose any concrete payment platform (e.g., PayPal, VISA); such can be plugged in dynamically by the developer.

The crowd computer exposes a set of API (explained in Table 4.1) that can be accessed by external applications. The API wraps the operations that each unit can perform. This approach allows developers to interact with the platform and create various crowdsourcing tactics and tasks. Together, these operations form an instruction set that supports a reasonably large class of programs, while keeping the instruction set focused to the core crowd management issues and, hence, simple, manageable and efficient.

Unit	Operation	Parameters	Description
Task	create	task information	Creates a task in the crowd computer. This operation is used to store the information given by the crowdsourcer.
	start	task id	Starts the specified task. When a task is started it is visible to the crowd.

	stop	task id	Stops the specified task. When a task is stopped it is invisible to the crowd. Workers' results for stopped tasks are not accepted.
	createInstance	task id	Creates an instance of the specified task. This operation also starts the instance, making it available for workers.
	stopInstance	task instance id	Stops the specified instance.
	assignInstance	task instance id, user id	Assigns the specified task instance to the specified user.
	storeResult	task instance id, data	Stores the data for the specified task instance.
	updateInstance	task instance id, data	Updates the status of the specified instance.
	updateInstances	task instance ids (array), data	Updates the status of all the specified instances.
Crowd	preselect	user id, task id	Executes the pre-selection, checking if the specified user meets the requirement for the specified task. This operation interacts with the Task unit from where receives the information of the task
Quality	validate	task instance id, data	Assigns the value (passed as parameter) for the quality of the selected instance.
Reward	give	user id, reward	Assigns the reward to corresponding worker

Table 4.1: List of the API. [2]

[2] Each unit implements CRUD operations and additional operations, which are not shown here. In this table we grouped only the operations that are used to model the processes that we present later in the chapter.

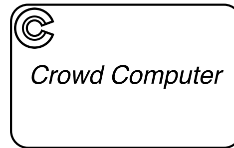


Figure 4.13: The crowd computer task to interact with the API.

4.6 Modeling crowdsourcing tactics

The crowd computer exposes operations that can be programmatically accessed to execute various type of tactics. A crowd task is not an atomic action executed only by an actor. It is a composition of various tasks executed by multiple actors, namely the worker, the requester and the platform. To create a tactic we have to program the crowd computer. To do so we introduced a secon level of modeling (as shown in Figure 4.6) in which a crowdsourcer can create the tactic of the task. In this section we introduce the components that allow one to create a tactic and also processes of most common tactics. The most basic process of a tactic is modeled in Figure 4.6 in the central part, it is the composition of the five main tasks.

4.6.1 Designing Tactics

In BPMN4Crowd we create a dedicated task that interacts with the crowd computer API, depicted in Figure 4.13. With this task it is possible to execute operations on the crowd computer and then enable the execution the desired tactic. To understand how the tactics are created we show in this section how the most important tactics, namely *marketplace*, *contest* and *bid* [52, 56, 70, 85], can be modeled with our notations. Since the tactics are complex process that requires a consistent modeling, for example each task instance created has to have a correspondent execution operation, we introduce in BPMN4Crowd the tactics as *reusable processes.* In addition

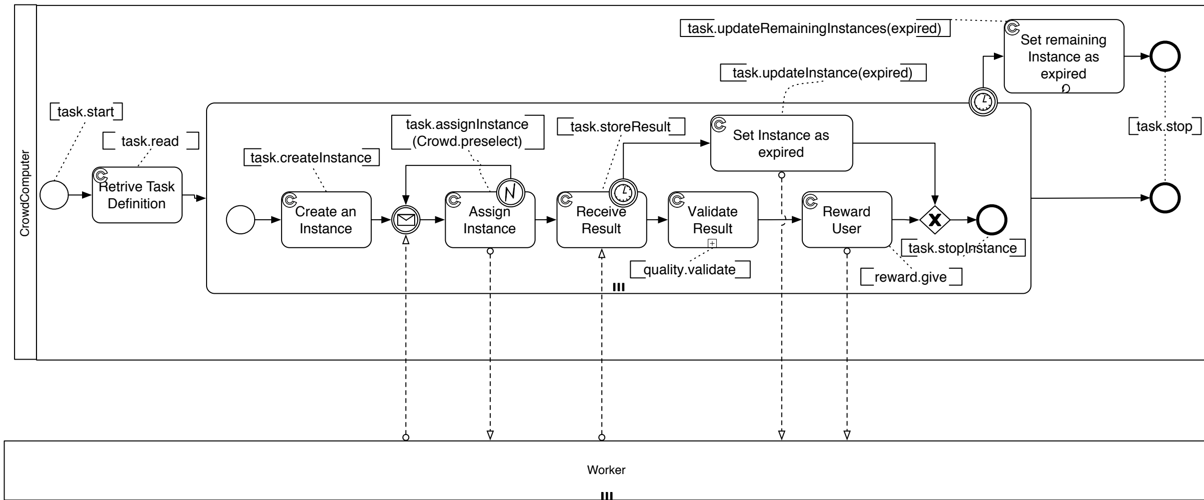


Figure 4.14: The process of the marketplace tactic.

to the three most used tactics we introduce here the *mailing list* tactic, which can be useful in many crowdsourcing scenario as it targets a precise group of people, for example for user studies. The list of tactics presented here is not complete but covers the most common ones. Yet, our modeling language allows modelers to create additional processes for the tactics, thus additional and new logic can be used to crowdsource work. The models of the tactic are designed with the use of the crowd computer task that calls the relative APIs. To make the models more understandable we gave meaningful names to the crowd computer tasks, annotating each task with the relative API call.

Marketplace

The marketplace is one of the most common tactics. The typical execution logic of this approach is that the requester posts in a shared space - the market - a task, which can have various instances that are decided by the requester. The crowd has the possibility to look through the list of available tasks and to accept to work on tasks. This tactic allows only one worker for each task instance and the instances are independent entities: there

is no exchange of information between workers or instances of the same task. Once executed the result is (generally) validated and, if accepted, the worker is rewarded, otherwise no reward is given. The reward is often a small amount of money (less than a dollar) but the reward is generally given when the answer is correct.

Figure 4.14 shows a process that represents one possible implementation of the marketplace. The model expresses the prospective of the crowd computer. The process starts with the platform loading the task definition that is stored in the crowd computer (created when the requester deployed the process). This operation interrogates the task management unit that replies with the task definition. The task definition is used for the creation of the task instances. In the model we created a *multiple parallel sub-process* that is used for the execution of the instances. In the subprocess the first operation creates a new task instance. The sub-process than waits for a worker to accept the task. When a worker sends the message that he has accepted the task, the platform execute the operation to assign the instance. The assign operation also performs the pre-selection: if the worker meets the requirements the instance is started and assigned to the worker, otherwise the instance is released and the process goes back waiting for another worker. When the instance is assigned the platform waits until the worker sends the result. The result sent from the worker is used to update the instance information inside the task unit of the crowd computer. In case of timeout (when a worker does not submit results on time) the platform updates the task instance information setting it as expired. After the process runs the validation task, which is a sub-process. The *validation sub-process* is modeled by the modeler and executed forwarding the task instance metadata. In its internal logic, the validation sub-process calls the API relative to the *quality* unit, updating the validation value with the result of the process execution. Later, the process executes the reward

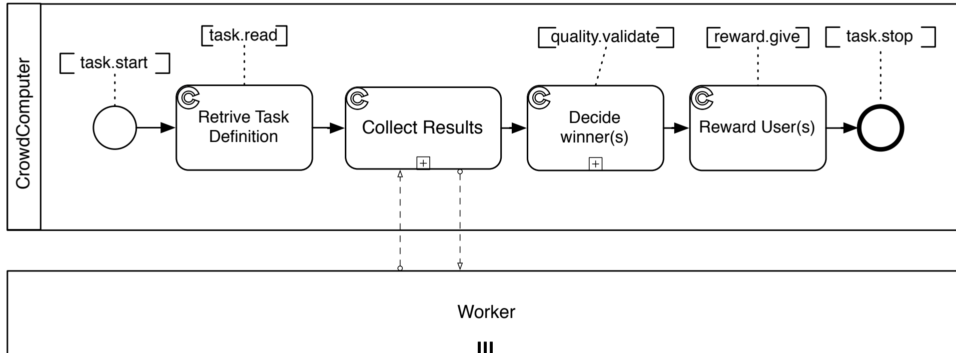


Figure 4.15: The process of the contest tactic.

task, which logic is *configured* by the requester. The execution logic of this task calls the *reward* unit API that is used to *give* the reward to users. Generally, in the marketplace tactic, the reward logic pays all the workers whose work has been validate as positive; yet requester can use other strategies such as *pay all* and give *bonuses* to certain workers. The sub-process that manages the task instances has a timer that is triggered if the instances are not completed, or accepted, within a time window. This timer triggers a task that sets as expired all the remaining instances, this ensures that all the instances are terminated. When all the instances are completed, or expired, the process stops the task.

This is only one possible representation of the marketplace and other exists, for example the minimal marketplace. The *minimal marketplace* is a marketplace with a single instance and without validation and reward. To model and execute it with BPMN4Crowd and crowd computer the process needs to have a task to *retrieve the task definition*. Then, without any sub-process, a task to *create an instance*, one to *assign the instance* and a task to *receive the results*. This is the minimal set of tasks that our solution needs to create a marketplace tactic.

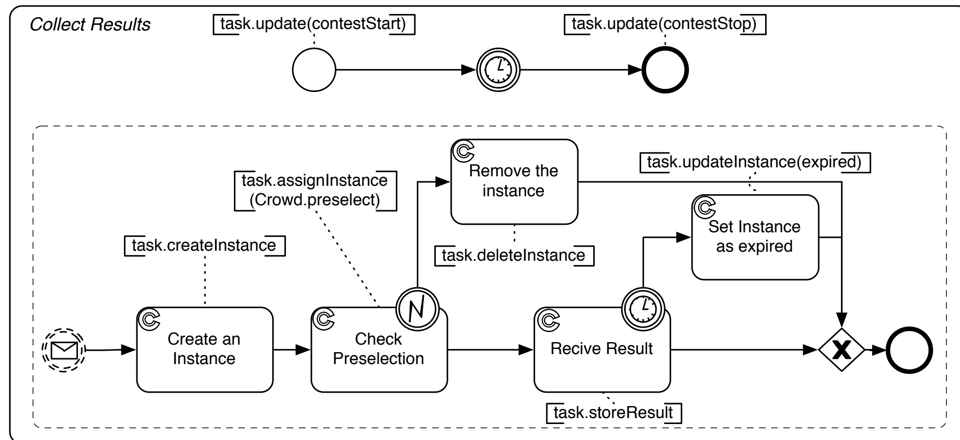


Figure 4.16: The process that models the collection of answers during a contest.

Contest

Another tactic is the *contest*, in which the negotiation is more articulated than in the marketplace tactic. When the task is published on the platform, all the workers can submit results. Usually there is no limit to the number of workers that can participate and neither to the number of solutions that each worker can submit. In this tactic the instances are dependent: the workers submit results for the same task, competing for the reward. Only after a specific time, when the contest closes, the requester (or someone else on his behalf) decides which work is the most valuable, thus the worker that gets the reward.

This tactic creates a competition among workers that are aware of this. The contest tactic is used to collect a widespread set of answers from various workers and then decide which are the best solutions. For this reason this tactic is used generally for creative task where the solution is subjective to the requester and more solutions can be evaluated as correct, for example the creation of a logo. To make the competition interesting for workers, reward is generally high (when monetary in the order of hundreds or thousands of dollars) but only the winner (best result), or few people

receive it. Thus, even with correct or good solution the workers may not receive the reward.

In Figure 4.15 and Figure 4.16 we model the process of the typical contest tactic. The process of the tactic is divided in two parts: the main process that is in charge of starting the task, validating the results, and rewarding the workers; and a sub-process that manages the contest execution. The main process (Figure 4.15) has similarities with the marketplace. The first task reads the task definition, that are used in the collect results task (a sub-process). Once the results are collected, the process executes the task to decide the winner, which is a sub-process (as in the marketplace) that interacts with the quality unit, and after executes the reward task that calls the reward unit API. However, the execution logic of the task instances is different from the marketplace. While in the marketplace each task instance has a task to collect the results, a validation task, and a reward task, in the contest the validation and reward are executed once for all the instances at the same time.

In the collect results sub-process (Figure 4.16) we modeled the collection of workers' answers. The sub-process starts and triggers a timer that after a predefined amount of time ends the sub-process closing the contest; this emulates the deadline of the contest. While the contest is open the sub-process collects answers from workers. For each worker that wants to participate the platform receives a message. This message triggers the event-based sub-process (the part surrounded by a dotted rectangle) that creates an instance, then executes the instance assignment, and if assigned waits to receive the answer from the worker (similar to the marketplace tactic). This event-based process can be trigged as many time as the workers submits data, and is automatically stopped when the timer reaches the deadline set by the requester. When the contest is closed all the answers are passed to the tasks that decide the winner and reward users.

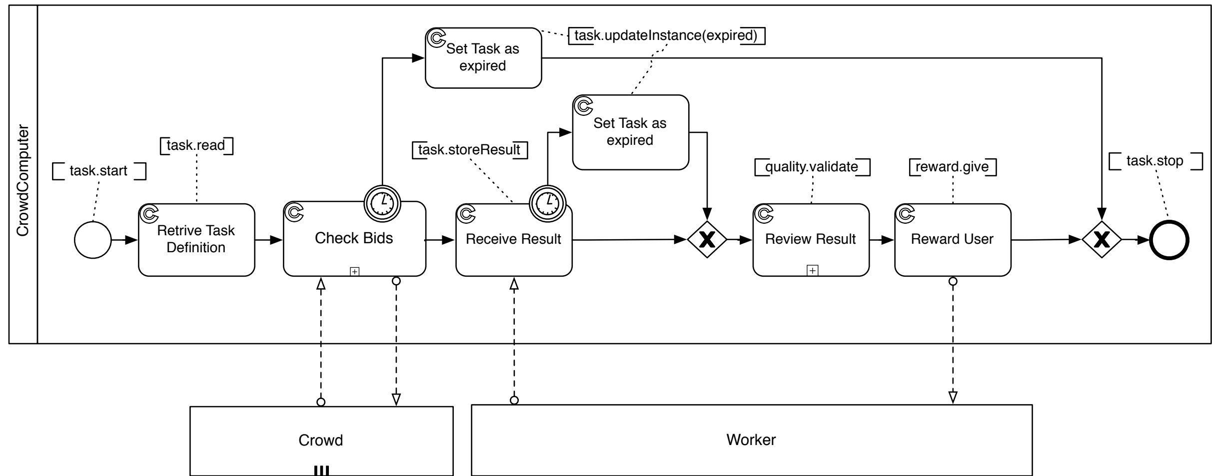


Figure 4.17: The process of the auction tactic.

Also for the contest various implementations exist, such as the minimal contest. For the *minimal contest*, in the the main process, a crowdsourcer can remove the rewarding. The other three tasks cannot be removed, one is needed to read the task definition, one for the execution, and the last to decide the winner. The logic of the collect result sub-process, instead, is standard and one cannot modify it much. What a crowdsourcer may want to change is the logic to decide the winner. In a minimal contest this logic could be of picking randomly one result or to select the first result sent. In more complex, and real, cases this logic can involve the crowdsourcer to decide the winner or can ask the crowd to rank the results and then, with a machine task, compute the average and find the winner.

Auction

The auction tactic is different from the previous two tactics we presented. While in the marketplace and contest reward is specified at the beginning, in the auction the reward is decided by the workers. There are various execution of the auctions, which for crowdsourcing are executed in a reverse fashion - where the winner is the worker who bids less than the others.

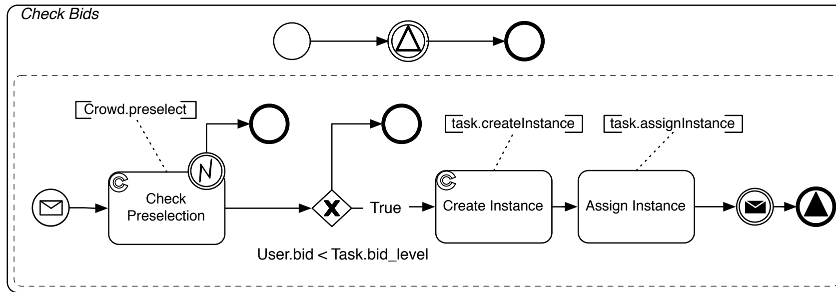


Figure 4.18: The process that models the selection of the winner, in this case it is the worker that first bids less than the bid level.

Among the existing mechanisms there is the (reverse) *English* auction, where workers bid against the others for a fixed amount of time. After this time the lowest bid wins. Another common auction mechanism is the (reverse) *Dutch* auction. The requester specifies an initial reward, if no workers accept the contract the reward is increased of a small amount. The first worker that accepts the reward wins. This type of auctions require a fair amount of time to participate in the bid, time that is not paid and that may discourage workers from participate. In crowdsourcing a common and used mechanism is the (reverse) *sealed first-price auction*. In this case the requester specifies the maximum amount of reward he is eager to pay. The first worker that bids less than this value wins. The auction tactic is generally used by platforms that have professional workers, such as programmers or freelancers. The idea is to offer rather complex tasks (e.g. development of a piece of code) to a crowd of competent people that will do the work for a reward that is lower than the one specified.

In Figure 4.17 and Figure 4.18 we modeled one possible process of the auction tactic, for this process we implemented the sealed first-price auction. The process shares common parts with the contest and the marketplace. From the contest tactic it borrows the idea of having multiple actors competing, in this case not for the work but for the bid (*check bids*). As soon as a worker bids less than the threshold the system creates an instance

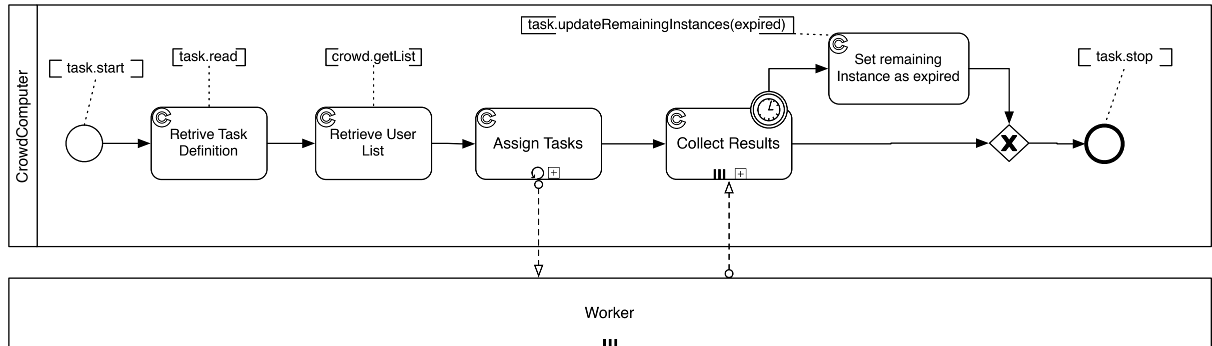


Figure 4.19: The process of the mailing list tactic.

and assigns it to the winning worker. Later, the worker performs the task sending back the solution that is validated, within the sub-process that models the logic of the evaluation; the worker is rewarded (generally if the work is evaluated as positive). The second part is similarly to the marketplace, where for each instance there is a worker that executes it, then the validation of the work, and after the reward based on the validation of results.

As said, other possible auction processes can be created. In this case, for the *minimal auction* what can be eliminated is the *review result* task, which assumes that the worker submits a satisfactory result. The reward task is mandatory, since the tactic is based on a negotiation of the reward. The bid logic (the sub-process) can be created with different auctions, what we modeled here is one of the simplest logics.

Mailing list

The idea of the *mailing list* tactic is to push the task to a group of people. This logic is the opposite of the other tactics, where are the workers that select the task they are willing to solve. Some platforms implement similar approach, for example AMT allows a requester to assign a task to a specific person. This tactic can be used to target a specific group, such as a mailing

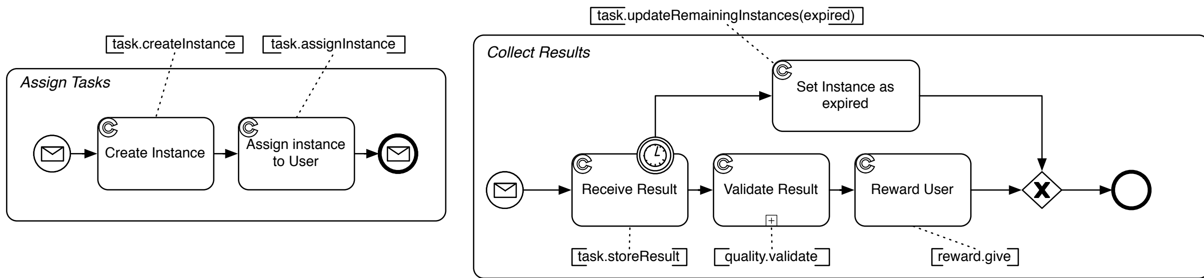


Figure 4.20: On the left the process that assign to each worker an instance. On the right the process that collects the worker's result and that validates and reward workers.

list of expert in some area, and ask them to participate in the task.

In Figure 4.19 we model the process of a possibler mailing list tactic. Differently from previous approaches, the instances are created and assigned to each selected worker in advance (repeating subprocess). This assignment is made to store the information of which user does which instance, information that are useful in the validation and reward. Later the platform waits for workers' solutions. Each time a worker sends a solution, the system stores the results. The solution is validated and reward given, generally all the workers are rewarded, even though for this tactic a non-monetary reward can be used. If the assigned worker are not willing to participate the system has a timer that after a predefined time set the remaining instances as expired. Different approaches exist. Here the system creates an instance for each worker. In other cases the system can create a number of instances accessible via a single link. The requester can then spread the link to a group of workers and the first people that access the link executes the task. For the *minimal mailing list* tactic one can remove the validation of the results and the rewarding, as in the marketplace.

4.6.2 Tactic configurations

With the tactic layer abstraction a modeler is able to decide and create a tactic for each task. Parts of the tactic can be executed without specifying

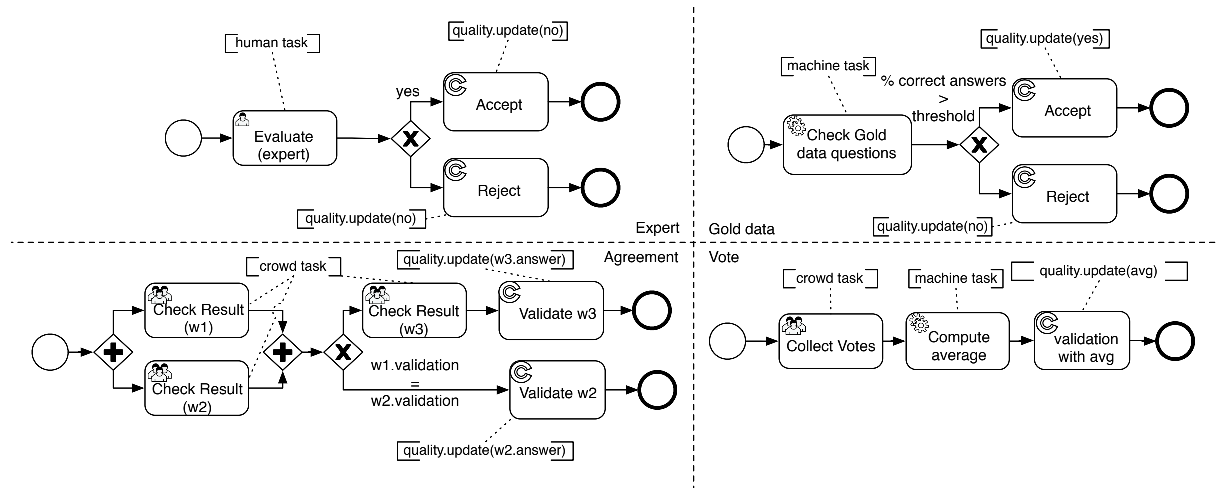


Figure 4.21: Processes of four validation configurations.

the execution logic, for example in our solution the pre-selection criteria are already specified in the task creation. Other parts need be configured to be executed, precisely what a requester has to specify is the *validation* process and the *reward* strategy. For these two tasks we created a list of patterns that one can reuse, patterns that describe some of the most common logics. This modeling is also present in our approach, it is the lower layer of Figure 4.6.

Validation

The quality unit allows the requester to control all the results sent by workers. In each tactic process there is a task that is used to validate, thus specify the quality value of the workers' results. To do so there are various logic that one can use and here we present a list of the most common. Validation logics are processes and with the BPM4Crowd we give the possibility to model a process to create a validation logic. Since some logics are well known and used, we add in the language these logics as reusable patterns. The patterns (process modeled in Figure 4.21) that we create are:

- *Requester/expert:* this validation is done by the requester himself/herself or by another person, an expert. The *validator* (the requester or an exeprt) executes a crowd task to validate the worker's result. The answer of the validator is used as value for decision, if positive the work is accepted otherwise rejected.

- *Gold data:* this validation relies on gold questions - their answers are known a priori - that are automatically checked by the platform [66]. This validation gives only an estimation of the task quality by checking control questions, and not a real evaluation of all the answers. In fact what is checked by the gold data method are only some of all the question of the task, and on this information an estimated quality is computed. On the other side, this validation is straightforward and does not require additional time or human activities for its execution. The validation process starts with in input the worker's answer and the gold data. A machine task compares the worker's answers with the gold data present in the system computing the percentage of correct answers. This value is compared with the threshold (decided by the modeler when creates the process), and if greater a positive validation is given, otherwise a negative one.

- *Agreement:* this validation assumes that two, or more, workers have to find an agreement on the validation. Two distinct workers are asked to validate the same result. If workers are in agreement, the agreed value (true or false) is chosen as validation. In the case the two workers are not in an agreement, the validation task is given to a third worker that does the validation. The third worker result is used as validation value, since he is in agreement with one of the previous two.

- *Vote:* asks the crowd to vote the results of workers, the decision is

based on the votes. This strategy is divided into two parts. In the first part people can vote on workers' results. Once the contest for voting ends, votes are collected and the system computes the average.

Reward

Reward is important in crowdsourcing since it drives the motivation of workers to perform the task. Here we present a list of logics that contains the most used one:

- *All / none.* This is the most trivial one. All the workers are paid (or none of them), even if the work is not satisfactory. This choice is not used often, yet, it is usually automatically executed by a platform if no instruction is given within a predefined amount of time.
- *On validation.* The payment on validation is one of the most used. Only the works that pass the validation step are rewarded. This type of strategy works well with tactics where workers are rewarded only if the result is satisfactory.
- *The best.* The reward is given only to the winner, or the best worker/s. This reward is used together with the voting strategy as validation. It is generally adopted in tactics where workers compete for the reward, thus send result for the same task where only one is accepted as correct.

In addition, these reward logics can be combined with other strategies, that are:

- *Bonus.* Reward is fixed and specified when a task is created. However, workers might provide astonishing results. For this reason, bonus rewards might be given to chosen workers. This strategy can be combined with the *on validation* and *winner.* It can be given also to people that were not awarded with the other strategy.

- *On milestone.* Some tasks can require a long collaboration between the worker and the requester. For example, when the task is on writing an application, which can be done with milestones. In this case the reward is also given on milestone. Before starting the task, the requester and workers agree on milestones and deadlines. At each milestone, if the work is done, the worker receives the corresponding reward. This reward strategy is generally used by platforms where workers have long-term tasks.

With this set of reward it is possible to cover most of the possible scenarios that a crowdsourcer may face. In addition the strategies are not mutual exclusive, thus they can be combined. For example a crowdsourcer can *pay all* and after give *bonus* to specific workers.

4.7 Prototype implementation

In this section we present the tools we implemented to ease the creation and execution of crowdsourcing processes:

- An enriched BPMN process editor to support the design of crowdsourcing processes (we called it BPMN4Crowd editor), based on the Activiti[3] modeling tool .

- A code generator (process compiler) to transform the crowdsourcing processes into executable BPMN processes and to extract data for the execution of crowd tasks and a process deployer to enable the deployment of crowdsourcing processes into the BPMN engine and crowd tasks into the crowd computer.

- An extended BPMN engine (based on the Activiti engine), which contains the libraries to enable the communication between the engine

[3] www.activiti.org

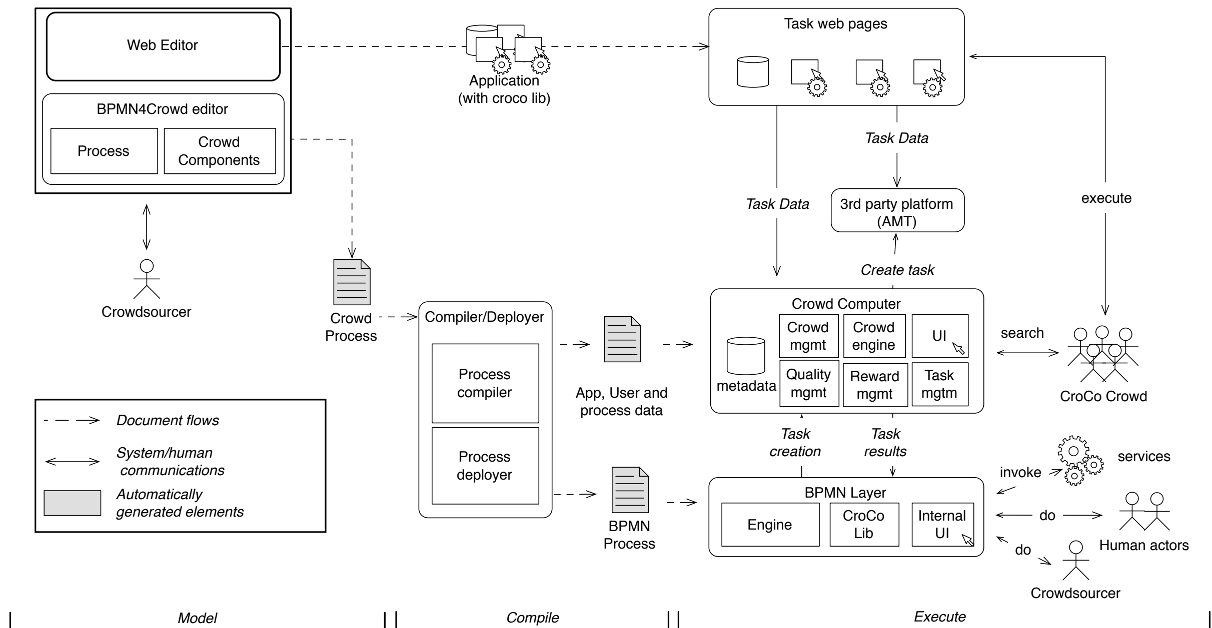


Figure 4.22: The implemented architecture of BPMN4Crowd solution.

and the crowd computer.

These three tools are components of our architecture, depicted in Figure 4.22. This figure, which is an extension of the one presented in the approach (Figure 4.5), shows in details the tools we created with their internal components. In this section we present how we implemented the three parts of the architecture, which are: *model*, *compile* and *execute*.

4.7.1 Model

To model crowdsourcing processes we extended the Activiti designer, which is a plugin for eclipse[4], adding the BPMN4Crowd components. The tool has dedicated extension points to create additional tasks, but it misses the possibility to create new artifacts for other elements, such as a new data item object. At the same time also the engine does not implement fully the BPMN specification. For this reason, in the BPMN4Crowd editor

[4]www.eclipse.org

we did not create all the extensions as we presented them before, but for few components we used a different abstraction. For example, the data items are not specified with data item artifacts connected to the task, but as parameters of tasks. Due to other limitations of the engine, tactics are implemented in the crowd computer, and made available as tasks in the editor. This limitation is also an advantage: crowdsourcers can use the tactic without implementing them as processes, but simply using the corresponding task. This change of paradigm does not limit the crowdsourcer (except the fact that tactics cannot be created) . The different layers of modeling (process, tactic, configuration of the tactic) are present, as well as, data transformation tasks and control and data flows.

With the BPMN4Crowd editor a crowdsourcer can create his own crowdsourcing process. In Figure 4.24 there is a screenshot of the editor. In the right part there is the palette that contains the crowd tasks (A), data tasks (B), and crowd computer API tasks (C). The crowd tasks (A) implement the crowd tactics (e.g., contest or marketplace) or give the possibility to post tasks on other platforms (e.g., the TurkTask to post task on AMT). We also added a machine task, differently from what BPMN already has, to ease the integration of external web services with the crowd computer. The machine task can be used to *get* data that are used in the process (load data) or to *post* data to execute external services. Each data task (B) implements a specific data operation (e.g., merge or split). For the crowd computer API tasks (C) we implemented only the task to call the quality unit, which is the only API that crowdsourcer can use in this implementation. Since the tactics are not implemented as processes, the crowdsourcer does not need to have access to all the APIs that are needed to create a tactic. What the crowdsourcer needs is the possibility to update the quality value from the validation process, for this reason we implemented only a single crowd computer task for the quality unit.

In the central part the BPMN4Crowd editor has the canvas where the process is modeled. The crowdsourcer can drag-and-drop the tasks from the palette and create his own crowdsourcing process. In Figure 4.24 we modeled a piece of the process of a scenario we developed for another project of our group (more information is given in Section 4.8). For each BPMN4Crowd task, which can be data transformation or crowd task, the crowdsourcer has to specify the parameters. By selecting a task in the process the editor opens the property tab where the parameters of the task are shown (lower part of Figure 4.24). The parameters are the same of what we introduced in the definition of the crowd tasks (Section 4.4.1), plus additional fields that are used as the new abstractions of components. In fact, there are additional fields to specify the data items used as inputs and outputs, and others that refer to the configuration of the tactic, such as the validation process and reward parameters. To create an executable crowd task, the crowdsourcer has to specify (letters correspond to the ones in Figure 4.24):

(a) *Description of the task.* This is the text that is presented to the worker when he starts the task. It should describe what the worker has to do to complete the task.

(b) *Task duration.* This parameter specifies for how long the task will be active. After this time the task will be automatically stopped by the system (deadline).

(c) *Number of instances.* With this parameter the crowdsourcer specifies how many instances of the task have to be created. In our implementation the instances are also correlated to the task and the data in input. Thus, if the dataset in input is a set of 5 pictures, and the number of instance parameter is set to 2, the system creates 2 instances for each set. In total the task will have 10 instance running, 2 instances for

each one of the 5 pictures.

(d) *Page URL.* This is the parameter that specifies the task interface of a crowd task. The crowdsourcer specifies here the URL of the pages that he created before.

(e) *Validation process.* Since each crowd task has its own validation process, the crowdsourcer can specify here what process has to be executed. In this implementation the crowdsourcer creates the process with our tool and writes in this field the process name. This parameters is used at runtime to execute the process when the tactic arrives to the validation task.

(f) *Reward.* this parameters is to specify the quantity of the reward that will be given to workers.

(g) *Reward platform.* The crowd computer has a plugin interface able to support various types of payments. With this parameter the crowdsourcer is able to decide what type of reward is given to workers.

(h) *Reward strategy.* The reward strategy is part of the configuration of the tactic. This parameter allows the crowdsourcer to select the strategy that has to be used for the reward.

(i) *Input data name.* This parameter is used to specify the name of the data item to read.

(j) *Output data name.* As above, this parameter specifies the name of the data item to write.

4.7.2 Compile

The second step after the modeling is the compilation and deployment of the process. For this we created a tool (code generator) that takes in input

a crowdsourcing process, modifies it, and creates a zip file that contains executable BPMN processes (the crowdsourcing process and all the validation processes). The code generator modifies the process in various ways. It changes process information to make it unique, this to ensure a correct execution. It adds a receive message after each crowd task. Crowd tasks are executed on the crowd computer that notifies the process when the task is completed. The receive message event is used to receive the notification from the crowd computer.

The compilation is repeated for all the validation processes linked to each crowd task. Once the compilation is finished, the crowdsourcer can take the zip file and upload it to the deployer. The deployer unzips the file, extracts crowd-related information, creating the necessary data structure to handle the execution of crowd tasks, and deploys the process on the BPMN engine. Then, the crowdsourcing process can be executed.

4.7.3 Execute

The start message for the process is sent by the deployer and received by the BPMN engine that executes the process. To handle the execution of a crowd task on the crowd computer we extended the engine with additional logic (Java classes). For every crowd task there is a Java class that sends to the platform (via an API call) the task parameters, which are specified in the process model, and runtime data such as the output of previous tasks. The crowd computer receives the data and creates a crowd task, which is a web page that embeds the UI created by the crowdsourcer. Once the task is created, the platform executes the crowd task following the tactic specified by the crowdsourcer. When the tactic reaches the validation task, thus when workers have sent responses for the task, the crowd computer invokes the validation process. This process receives as input the workers' information and their data, and updates the quality

metadata of each worker's result. Once all the workers' results have been validated, the process engine continues the execution of the tactic that (generally) invokes the reward strategy. The reward gives the reward to workers following the logic specified by the crowdsourcer. This terminates the execution of the crowd task. When a crowd task is terminated, all its metadata are stored in the crowd computer and sent to the BPMN engine within the message that notifies the completion of the crowd task. The engine receives the message and then continues the execution of the process.

Data tasks are executed in a similar fashion via API calls. For data tasks the execution is straightforward: the APIs accept in input data, execute the selected operations, and return the result to the engine.

4.8 Evaluation

In this section we evaluate our language introducing a real-case scenario and showing how it can be implemented with, and without, BPM4Crowd.

4.8.1 Scenario: crowd-based pattern mining

Harvesting knowledge from large datasets - *data mining* - is a domain where computers outperform humans when there is a large dataset and the algorithm is configured to understand what patterns have to be searched. When the dataset is small and algorithms cannot be trained to understand what a pattern is, humans may outperform machines. In [71] we use human capabilities to identify recurrent patterns in a set of mashup/workflow models called *pipes*. The goal is to test if and how different crowd algorithms - combinations and configurations of crowd tasks - enable the mining of patterns and with which quality. Moreover, we also want to test

if the crowd can be exploited for the quality assessment of crowdsourcing results. To test this, we created three experiments:

Naive: presents to a worker a single pipe and asks the worker to identify pieces of reusable knowledge (patterns);

Random3: presents three different pipes to the worker and asks to identify patterns that recur in at least two of the pipes;

ChooseN: allows the worker to select N pipes from a list of 10 pipes that are used as dataset for the second step where the worker identifies the recurring patterns

Each task for the crowd also includes a survey that is used to measure the knowledge of the topic by the workers.

Although the three algorithms may look different, their crowdsourcing logics can be abstracted into one configurable process:

- *Initialize and load pipes dataset:* the system loads the dataset of pipes.
- *Partition dataset and map to tasks:* the dataset is split in smaller subsets and for each study there is a different splitting algorithm (e.g., in sets of 1 element for Naive, in set of 3 elements for Random3). Each partition is mapped to an atomic task.
- *Deploy tasks on a crowdsourcing platform:* the tasks are deployed on a crowd platform and made available to the crowd.
- *Manage task execution and collect patterns*: tasks are executed. Once all the instances are performed, the patterns are collected.
- *Filter the results*: results from workers with poor performance (results of the survey) are eliminated.

As output of the three algorithms we have a series of patterns. To decide what algorithm performs best we:

- *Evaluate the output*: we evaluate the patterns, checking if the results meets our requirements (e.g., patterns need to have at least three connected elements).

- *Choose algorithm*: compare the results of the three algorithm and decide what is the algorithm that performs better.

As output of this process we have the algorithm that performs best. To understand if the crowd can also be used to asses the quality, we create another process that:

- *Loads pattern dataset*: the system loads the patterns of the winning algorithm.

- *Assesses quality (crowd)*: The results of the patter mining are given to other workers that perform the assessment of the quality.

- *Assesses quality (crowdsourcer)*: We asses the quality of the same set of patterns.

- *Compares quality assessments*: The quality assessment given by the crowd are compared to our assessment values.

- *Choose assessment strategy*: We chose the strategy based on the result of the comparison.

From this additional process we have the information on the quality assessment executed by the crowd.

In Figure 4.23 we modeled the logic of these processes in an unique crowdsourcing processes created with the BPMN4Crowd language.

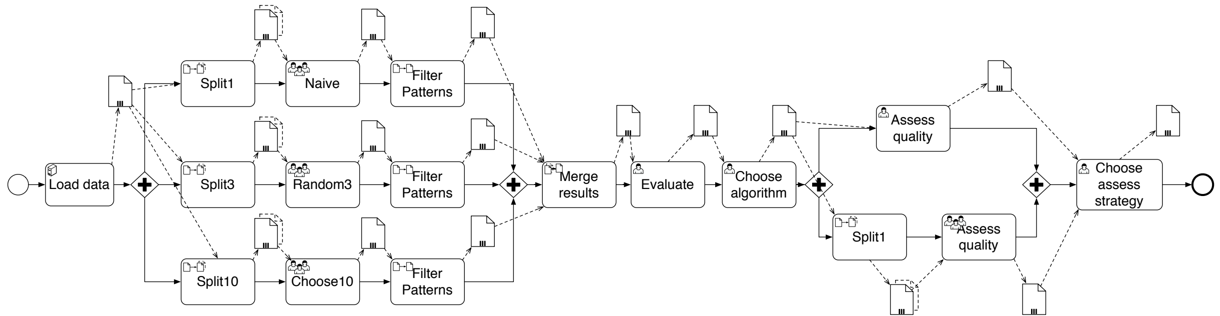


Figure 4.23: The crowdsourcing process of the crowd-based pattern mining pattern mining scenario.

4.8.2 Implementation

In this section we show how the scenario implementation changes *with* and *without* (which is what we did for the work in [71]) BPMN4Crowd.

Without BPMN4Crowd To execute the scenario we created *three* separated web applications, one for each study. In each application we developed the UI of the task, which was composed of an interactive form where workers could select the patterns. We manually configured the database and implemented the logic to load the dataset. We created the algorithms (one for each application) to divide the dataset and we manually mapped each partition to a task instance. We created in our application as many task instances as the partitions in order to have a task instance, thus a worker, for each partition. A crowdsourcing platform, CrowdFlower[5], was used to make the identification of patterns accessible to the crowd. We created a task on the CrowdFlower platform that contained a survey, to evaluate workers' expertises, plus a link to our web application where workers could access a task and identify patterns. The survey results were used as discriminant to accept or reject the identification results: workers who had a poor result in the survey had their results eliminated. The filter-

[5]www.crowdflower.com

ing operation, which for us was fundamental to have a high quality result, was done manually: data were downloaded from CrowdFlower and from our web application and matched. In the same filtering operation we also check if the patterns meet the requirements, eliminating the one who does not. When all the three applications were completed we compared the results and we decided which one was the best.

Consequently, we created an additional application to crowdsource the quality assessment. For this application we coded the logic that splits the set of pipes and that implements the interface for the assessment task. For each pipe we created an instance of the task on our application, creating a task also on CrowdFlower (as before). Results of these tasks were manually recombined to create statistic of the quality.

With BPMN4Crowd The modeling of the process in Figure 4.23 with BPMN4Crowd editor is almost straightforward. We replicated the logic within the editor, configuring each task. In Figure 4.24 we show the editor with the first part of the process and the parameters of the *naive* crowd task. The naive task is a *marketplace tactic* task, with an evaluation of the results based on gold data (created with a process that uses the logic presented in Figure 4.21) and a reward of 0.5$ based on validation. The description of the task tells what is the goal and the rules. The deadline is set to 1 month; the page url, the interface of the task, points to the page we created, which is deployed on a separate server. Data items for input and output are set to the "data_naive" variable. The other crowd tasks are configured in a similar fashion, yet with parameters specific for each task (such as the UI page).

The Figure 4.24 annotates data tasks with a description that shows how they are configured. For example, in the first branch, the *split* task is configured to divide the list of pipes into group of *one* element. Each

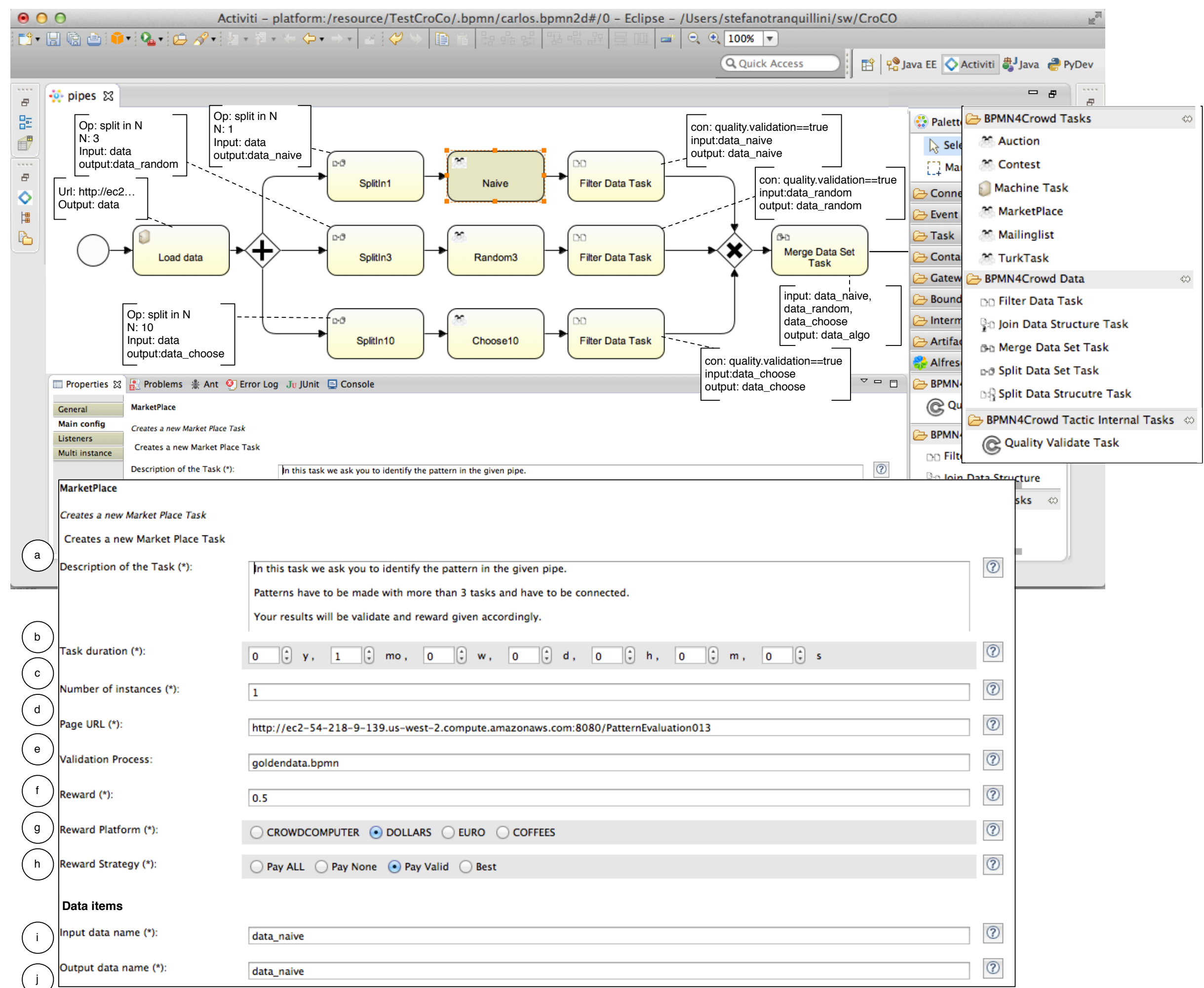


Figure 4.24: The visual modeling editor with the process of the crowd-based pattern mining scenario. The Figure has the BPMN4Crowd palette and the properties window zoomed.

branch of the process has a different data item (data_naive, data_random, data_choose) to keep the results of the three studies separate. Results are merged after the filtering, which is based on the validation of the work.

For the execution of this process we created the web pages that are used as task interfaces. The pages are similar to the ones created for the implementation without BPMN4Crowd: they contain an interactive form where workers can select the patterns. Yet, pages have less logic than in the previous implementation. The pages take in input the metadata from the process, which in this case are the *IDs* of the pipes, and load the corresponding data from the internal database displaying it to workers.

The assessment part does not require an additional process as in the manual development. This part is modeled (not visible in the figure) in the same process as in Figure 4.23. The human tasks are created with standard BPMN constructs and assigned to ourselves (the crowdsourcer), while for the crowd part we use a split task (split in 1) and a marketplace crowd task.

4.8.3 Analysis

The two approaches for the implementation of the scenario are both effective, yet they require a significantly different amount of time for their development. The manual development of the process is made with four different applications, which are developed almost from scratch every time and that have various tasks that require the intervention of a person to be executed. Each data management task is either executed inside the application (e.g., the split tactic) or executed manually (e.g., the filter of data). Split tasks require writing the logic of the splitting, which is different for each case. Filter tasks require going through the whole set of results from the crowd manually and selecting the ones that do not match the quality conditions, the same is true for the merge result tasks.

The differences are not only in the creation and execution of the process, but also in the creation of the task interfaces. Without BPMN4Crowd the interfaces are developed within the process application and implement the logic to load the data from database, split and display the data to workers, and to collect the answers that are also merged into a unique structure. With our approach each page receives the metadata (IDs of the pipes plus information that are necessary for the execution, such as the quality in the filter task) of the instance assigned to the worker that has opened the page. Each page uses the metadata to load the corresponding pipes data from the application database. This makes the creation of the pages easier, since the data logic does not have to be implemented. On the other hand, implementing pages for crowd tasks within the BPMN4Crowd language requires the knowledge of the library we implemented for sending and receiving data.

The execution of the scenario without BPMN4Crowd requires to: execute the three studies separately; to manually evaluate the results selecting the algorithm that produced better results; and to use the pattern data to execute the assessment application in a separated application. With BPMN4Crowd we created a single process that implements the three cases. The studies are executed in parallel, which has an impact on the overall execution time. Data management operations are automatically managed by the platform, thus there are no manual tasks to execute to filter results. The assessment part of the process is executed directly after the collection of patterns, and even in this case there is no need for any manual effort.

The advantage of BPMN4Crowd is in the automation of all the operations that before where done manually. The BPMN4Crowd approach automates 9 (all the data tasks) of the 17 total tasks of the process and makes easies the creation of crowd tasks and the connection among the two part of the scenario (algorithms and assessment). In addition, the vi-

sual modeling of the process, and the possibility to change parameters of each task, allows crowdsourcers to fine tune the process and re-execute it without spending time to code and create new applications.

4.9 Discussion and Lessons Learned

In this work we presented BPMN4Crowd, a process-based modeling language for enabling the design of crowdsourcing processes. Our targeted end-user is a modeler that wants to crowdsource complex and structured crowdsourcing processes. The possibility to intermix tasks executed by a crowd with tasks executed by other actors, such as the requester himself, makes the creation of structured logic feasible, compared to what crowdsourcing platforms support, and easier, compare to a programmatical approach. However, for simple tasks, which do not require particular logics or that do not need data management, our solution is still effective but may be too sophisticated. To crowdsource a single crowd task, a crowd platform is easier to use. Yet, the kinds of applications we support are crowdsourcing processes rather than single crowd tasks.

The choice of extending an existing process language, and relative technologies (e.g., a BPMN engine) has a twofold benefit: (i) it can foster the adoption of our extended language since BPMN is widely known and adopted by many people to model their business processes; (ii) people that already use BPMN can integrate our extensions and have tasks executed by the crowd in their existing processes. On the contrary, for people not familiar with the composition paradigm, our approach requires to learn the modeling language and the additional logic we introduced for the support of crowdsourcing; the respective complexity is only slightly bigger than that of BPMN.

In this work we proposed a different crowdsourcing platform, compared

to the existing ones, that provides a set of accessible crowd operations and that allow the implementation of various tactics. We have not yet tried to build a crowd of workers for our platform. Not having a crowd of workers limits the possibility to create applications and test their execution with real workers. To overcome this problem, which does not affect the language per se, we started the integration of existing platforms (i.e., Amazon Mechanical Turk), enabling the possibility to post tasks on these external platforms. This approach overcomes the problem of having a crowd but it introduces a limitation. Existing platforms implement a predefined tactic for the execution of crowd tasks and this limits the possibility to define, or use, different tactics on the same platform.

BPMN4Crowd supports the creation and specification of crowd tasks. What we require crowdsourcers to do is to create the UI of tasks as separated pages. On the one hand, this gives to crowdsourcers the freedom to create any UIs for the tasks, including any element and deciding what type of data have to be submitted to the platform and what data remain on their application. On the other hand, this requires knowing a programming language to create the task pages and to inject our library. Thus, creating a platform to help crowdsourcers in the creation of task UIs, or providing a set of ready-to-use pages, could foster the adoption of our solution also for crowdsourcers that do not have the knowledge to create a task interface.

The prototype we created contains the tools that are needed to design, deploy, and execute crowdsourcing processes. Building a platform like this is a complex and non-trivial development effort. It requires integrating different technologies, creating communication channels, and guaranteeing that the execution follows the modeled process. The result is that, besides being functional, the setup of the editor, the compilation, and deployment are not intuitive steps. We developed the platform with the goal of having a prototype to test our claims rather than having a product ready for the

market.

In addition to the language, we also proposed the crowd computer, an attempt to create a flexible and programmable crowdsourcing platform. While existing platforms are suitable mostly for the crowdsourcing of single tasks, we believe that flexible and programmable platforms are fundamental to support the crowdsourcing of more sophisticated and structured processes. These types of platforms require programming skills and knowledge of crowdsourcing to build applications on top of them and to specify the logic of the tactics. In our work we provided the tactics also as pre-implemented tasks, simplifying the work of less expert modelers. However, to exploit the full potential of crowdsourcing we cannot eliminate its intrinsic complexity.

4.10 Related work

Recently, *human computation*, the use of human abilities to solve problems, has been investigated and adopted as effective way to solve task of work. *Crowdsourcing*[40] is one aspect of human computation. Similarly, *social computing*, which studies social human behaviors facilitated by computers (example are blogs and wikis), is another aspect of human computation [70]. Social BPM is a recent trend of research that fuses social interactions, trough social software, and business process management to improve its life-cycle [34, 42]. Primary, in BPM, social aspects are used to improve the design of processes [49] or to enable coordination and collaboration during process execution [31]. Yet, there are social BPM approaches that extend business process languages, as we did, to support social interactions among online users. BPM4People [11, 12] proposes a set of extensions of BPMN to enable the modeling and deploying of social interactions over social networks, such as collection of votes or comments. With these ex-

tensions a modeler can define the action that actors have to do on a social network platform. BPM4People supports social computing on social network platforms, where work is mostly implicitly executed and actors may not be aware of taking part in a task of work. Instead, in our work, we focus on crowdsourcing, where the task of work is explicitly defined and where actors participate to offer or execute tasks of work.

Many researchers [22, 46, 52, 85] have highlighted how solutions that manage human and software based work with BPM or workflows are needed for crowdsourcing. The integration can be achieved in various ways. For example, [44] presents an extension of a BPMN engine capable to ensure the competition of crowd tasks before a deadline by adapting, at runtime, crowd task parameters, as the reward, and the required execution time. Among the different approaches there are some that involve business process modeling and that have people from a crowd as task executors. In [76] the authors use Human-Provided Services (HPS) [73], which abstract human capabilities as web services easing the interaction with people and integration with SOA systems, to create processes where task are executed by people taken from a social network. Schall et al. [72] in which some authors contributed in the previous discussed work, extended BPEL4People, an extension of the process language BPEL, with parameters to specify requirements specific to crowdsourcing, such as user skills and deadlines. Both approaches tackle the problem of crowdsourcing by abstracting worker capabilities within the definition of tasks or services. Approaches that are used to identify workers in a social network and that focus on the integration of the crowd within a service-oriented computation paradigm rather than providing a solution to model and enact crowdsourcing processes.

Our work provides a language and a platform to create and enact processes that have crowd, data, machine, and human tasks involved. In liter-

ature similar problems have been addressed from three main prospectives: *process composition*, *parallel computing*, and *procedural programming.*

Process composition Similarly to what we have presented in this work, other researchers have adopted a process composition solution to enact crowdsourcing processes. CrowdWeaver [45] is a process modeling tool built on top of CrowdFlower. The CrowdWeaver system offers a visual tool, with a graphic notation, to create and execute data-driven processes composed of machine and crowd tasks. Despite the fact that the work is oriented at providing a visual modeling solution for crowdsourcing process, it has similarity with the BPMN4Crowd. CrowdWeaver provides a visual modeling solution to create the process logic, supporting crowd and data tasks. On the other hand this solution is abstracting CrowdFlower operations at a higher level. This approach does not support crowd tactics and neither the possibility to execute a process into other crowd platforms.

CrowdLang [60, 61] is a model-driven language for programming generic human computations. Similarly to our solution, it expresses data and control flows, describes application logics with tasks executed by the crowd or machines, and supports operations to manage data. Yet, CrowdLang has an own notation, which may not be easy to understand for all the crowdsourcers. It is partially based on workflow modeling objects, such as rounded rectangle for tasks and diamond for conditions, but it also introduces additional concepts, such as the decision, which is modeled with a circle shape. Its modeling convention is not very effective, each task in CrowdLang is modeled as a task instance, thus crowdsourcers should create a task for each instance they need, which may require to insert a huge number of tasks.

Even thought both approaches are suitable to model crowdsourcing processes, they do not fully support crowdsourcing processes with the possi-

bility to define tactics and configuration. In our work we also support the integration with BPM, which makes the modeling more accessible to people that already know the language. Approach that also automates some of the steps of the process, such as human tasks that are executed by crowdsourcers.

Parallel computing Other researchers see a crowdsourcing process as a complex task that is made of smaller and simpler tasks executed in parallel. Crowdforge [47] and Turkomatic [51] adapt the Map-Reduce approach [30] to crowdsourcing to solve complex jobs. These frameworks model a complex job as a set of *split* and *recombine* tasks executed by workers. Workers have the possibility to solve a task or to split it in smaller tasks (sub-tasks), in this case the worker is in charge of recombining the solutions of sub-tasks into an unique solution. Similarly, Jabberwocky [4], implements a map-reduce approach. In addition, it offers a full-stack solution, providing not only a language but also an omni-comprehensive system. The Jabberwocky architecture is composed of three different pieces: Dormouse that provides operations to interact with machines and humans; ManReduce, similar to Crowdforge and Turkomatic, that implements the Map-Reduce idea having Dormouse as workers; Dog, a scripting language that can be used to specify the details of applications, e.g., defining users or task goals, which implements also the ManReduce.

The works of this type give one the possibility to use human computation and machine computation in a single application. Compared to our approach these systems support the creation of applications whose process is composed of a parallel executions of tasks, which can be replicated in BPMN4Crowd, but neglecting the possibility to create different process logics. In addition, this type of abstraction may be not suitable for crowdsourcers that are business analyst or are not experienced with programming

paradigms.

Procedural programming Researchers have applied programming to the creation of crowdsourcing process, inventing new languages able to cover crowdsourcing aspects. Turkit [54] is a programming language based on JavaScript that adds support for human computation. Turkit uses Amazon Mechanical Turk as platform where the human tasks are executed. Using Turkit, programmers can write software applications that use both human computation and machine computation. Automan [8] is a system for human computation, by integrating human computations into a programming language (Scala).

All these works abstract application logic at a programming language level. Programming languages allow crowdsourcers to create a variety of crowdsourcing processes, yet neglecting the support for tactic definitions and configuration. Moreover, implementing crowdsourcing process is not an easy task, as we also showed in our evaluation, even with frameworks that support it. Crowdsourcers that use these systems (or any programming language) are forced to code entirely their application logic. Coding operation that is not trivial and that makes difficult the maintenance and modification of its execution logic. As for the parallel computing, also with procedural programming its use is limited to people that are programmers, making it not accessible to others such as business analysts or people that have to conduct a user study as in our evaluation case.

4.11 Conclusion

In this work we introduced a language and the relative tool-chain to design and execute crowdsourcing processes. Our approach extends BPMN to provide support to crowdsourcing, with tasks dedicated for the crowd

and for the management of data, enriching the language with the definition of tactics and configurations that crowdsourcers can use or adapt. We equipped the language with a modeling tool and a set of software components able to transform the crowdsourcing processes into executable processes. To execute crowdsourcing tasks we created the crowd computer, a platform for crowdsourcing that gives the possibility to select, configure, and execute various crowd tactics, without imposing pre-defined logics. The result is an approach to the modeling of crowdsourcing processes that is comprehensive.

As future work, we plan to add the support for tactics modeled as processes in the prototype. We also plan to provide an online repository where common tactics, configurations, and processes can be shared by crowdsourcers, this to create a reusable knowledge base for crowdsourcing.

The advantage of our language lies in the automation of operations that otherwise require manual or non-trivial programming effort. The number of applications that can benefit for this work is large, both for individuals and companies. In fact, we enable the design of crowdsourcing processes, making it easier compared to the development with programming languages, and going beyond the single-task applications commonly supported by current crowdsourcing platforms.

Chapter 5

Conclusion

We conclude this dissertation with an analysis of the overall work. First we analyze our approach and results and we present the lesson we learned, then we conclude with possible future works and final remarks.

5.1 Lessons Learned and Limitations

The work of this dissertation is based on extensions of process languages. This choice was driven by the intrinsic process-driven nature that the applications of the three focuses have, and by the possibilities that the new actors could offer to people who already use process management systems in their activities. Commonly, processes are sketched by an analyst and designed by a skilled modeler who knows how to create an executable process. In our proposed languages we envision process design as a two phase process where first the high-level process is designed and later refined by an expert of the field. The visual aid given by the modeling tools eases the creation of these processes. Yet, from a usability point of view modeling a process requires knowledge of the modeling language and of the selected domain. Other modeling paradigms or ad-hoc languages could be investigated as an effective language to design extended processes. Nevertheless, due to the intrinsic complexity of these topics, and their adoption by do-

main experts, we do not foresee an immediate need for a modeling language that can be used by *any* kind of end-user. Similarly, we do not see our languages to be used by *any* end-user, but rather by domain experts that want to speed up the creation of applications, or by modelers that have interest for the integration of the new actors into their processes.

The languages presented in each chapter support and ease the development of the respective scenarios and of similar applications. Part of the work that before required manual and non-trivial programming effort (e.g., the programming of a WSN), with our languages is now automated and integrated in a high-level modeling convention; there is no need for further development effort. This is our evaluation to measure the success of our contributions: we made the work of people easier. We did this evaluation comparing the implementation of the scenarios with and without (that is a development with classical programming languages) our language. From this comparison we saw how the languages and software we proposed simplify part of the work by providing specifically targeted solutions that helps in the creation of the processes and software that automate the deployment and execution and that do not require the creation of code. The user study on the language for WSNs highlights how it can be effectively and efficiently used by modelers (after they are introduced to the new constructs) even if they are not experts of the domain. This claim can be extended to the other languages since the modeling conventions are similar. Yet, additional user studies could test in depth our claims and provide useful information regarding the usefulness of the language and on the use of different modeling languages to allow different types of users to design extended processes.

The languages that we created satisfy the initial set of requirements we found during the analysis of each new actor and relative scenarios. We do not solve *all* the problems and requirements of *any* possible application in

each field. For each language we proposed a set of domain-specific components that are the core to implement and cover the requirements of most of the possible processes in each field. The design of actor execution logic (i.e., UI interactions, WSN task composition logic, and crowdsourcing tactics) gives to modelers the possibility to create a wide range of applications. An important aspect of our languages is, in fact, the possibility to design the logic that is executed by the new actors. For WSNs and the crowd we do not only orchestrate tasks whose logic is already defined a priori, but we give to modelers the possibility to design the internal execution logic of these tasks.

The tools and systems we implemented support the design and execution of extended processes. At the current status, the setup, which is the installation of the editor, compiler, and runtime environment, is not an easy procedure. It requires some knowledge to install and configure the packages and additional software (e.g., databases and servers). Our primary goal was not to create a tool-chain that can be easily used by any person but rather implement prototypes to test and execute our scenarios and similar applications, that is, to prove that the concepts and solutions conceived indeed work.

An important lesson we learned is how complex it is to work on topics that require a transversal knowledge and that demand an important effort to build prototypes to test the outcomes. While working on mashups and distributed UIs was a natural evolution of the work conducted within our group, and for which we already had some knowledge, the research on WSNs was a new and challenging topic. Nodes of WSNs have a limited memory and processing capacity. Providing a high-level modeling language was thus a very ambitious goal, and a non trivial effort was the translation of process logic into executable WSN logic. We had several iterations with WSN experts to understand a WSN's capabilities and to find the right

abstractions between what a network of sensors can do and what can be modeled in a process. The fruitful collaboration with WSN experts made the creation of WSN code feasible and efficient, making it also possible to have a final prototype.

The difficulties in developing applications in these contexts can be summarized in the following points: (i) to develop the logic of the applications, that is, to write the code to coordinate the actors and the code that is executed by the actors; (ii) to create a platform that is able to support the execution of the application, which is composed of a runtime environment for the actors and a runtime environment for the coordination logic, plus a platform that manages the communication; (iii) to deploy and generate the code for the actors, the coordination logic, and the communication channels. In our work we provided: (i) high-level modeling languages to develop the logic with a process-based convention, which has domain-specific modeling components for the design of the actor logic; (ii) an architecture that supports the design, deployment, and execution of extended processes; (iii) and an automated system for the code generation and deployment of processes and of the actor logic. With our languages we simplify the work required to develop applications in these contexts, providing also an integration of business process languages with additional actors. Our simplification requires a modeler to learn the few new notions we introduced to abstract the new actor details. In fact, we simplified the creation of process-driven application logic but the intrinsic complexity of the new actors cannot be completely eliminated, only abstracted at a different level included in the modeling convention.

5.2 Future work

As discussed in the previous section, the research may benefit from some improvements; as future work we envision:

- To improve the languages with additional constructs to satisfy new requirements. With the proposed languages we support an initial set of requirements that comes from the analysis of the actors and of possible scenarios. There are new scenarios and possible applications that may arise in the future and that may need additional constructs to create processes, especially for WSNs and crowdsourcing that are becoming more and more adopted. In particular, we see as possible extensions:

  WSN

  – To enable the execution of multiple processes on the same network. Our toolchain supports the execution of a single process on a network since nodes do not have enough power and memory to execute multiple logics. However, when the nodes will permit it, an improvement will be enabling the deployment and the concurrent execution of various processes. The modeling language should not be changed, what needs a further refinement is the code generation and the runtime environment that will have to manage and coordinate the messages of different processes.

  – To create a runtime monitor. The set of tools we developed focus on the support of the modeling and execution of the process. To improve the usability and to have a better overview of the process execution a runtime monitor can be created. This monitor should give feedback on the status of the network, the status of

the execution, and possible problems that may arise at runtime (e.g., a node that stops its normal functionalities).

- To ease the composition of meta abstractions. Our language for WSN gives the possibility to create WSN task logic by composing meta abstractions. As emerged by the user study, this abstraction may be difficult to grasp for modelers that are not also domain experts. As started by our partner in the makeSense project (SAP), a newer modeling convention to specify WSN task logic could ease the creation and maintenance of WSN processes. Ideally, the newer composition language should not only abstract the WSN specific components to a different paradigm, but also provide the possibility to reuse existing implemented logic. For example, create a repository of implemented WSN logics accessible as libraries, or a repository of specific WSN patterns (e.g, a control loop for the sensing of CO_2) that a moder can easily access and use.

Crowd

- To support the creation of task UIs. Our language and toolchain assume that the task UIs are developed by crowdsourcers. This is an effective approach that also allows crowdsourcers to specify any type of UI for their tasks. However, a platform that helps a modeler to develop task pages, or that provides a repository of pages ready to use, can foster the adoption of our tools also by people that may not know how to create pages for the crowd tasks.
- To extend the support for additional existing platforms. We provided an implementation of our language and a platform that supports the creation and execution of crowdsourcing processes.

Our toolchain enables also the possibility to post crowd tasks on a third-party platform (Amazon Mechanical Turk). Adding support for other platforms, such as CrowdFlower, will have two benefits: (i) foster the adoption by additional crowdsourcers that are already familiar with these platforms, (ii) avoid the problem of finding a crowd of workers.

  - To extend the language to enable the execution of tasks instance-by-instance. Generally, in process languages all the instances of a task have to be completed before it is possible to execute the following task. For example, image a process that has a task to upload a picture (A) followed by a task to tag a picture (B); each task has ten instances. If we model this process with BPMN and execute it, we have to first execute all the ten instances of A (we collect all the images) and only then start the instances of B (we tag all the images). However, it would make more sense that as soon as a picture is uploaded (an instance of A is executed) an instance of the task to tag the picture starts (an instance of B). This new execution logic requires an extension of the language and an extension of the runtime environment.

- To improve the adoption of the languages and create communities of end-users. Having a large community of users gives the opportunity to test the languages and to have feedback and suggestions on additional requirements to satisfy. This requires: (i) to improve the usability of the modeling tools with additional libraries, to help modelers in the design, and with an online, easy to access, platform; (ii) to make the toolchain more stable and to create a portable and an online version of the runtime environment to allow people to easily access and use our toolchains.

- To conduct further user studies and analysis of the languages. Further user studies, with larger groups of people, ideally end-users that actively use our languages and tools, can give better feedback on how to fine-tune the language and on extensions that could be introduced. This requires to have an active community and to implement survives and software to analyze how end-users interact with the tools and how the language is used to model processes.
- To release all the code as open-source software to foster collaboration with other researchers and interested people and to improve the quality of the code.

5.3 Final Remarks

In this dissertation we presented modeling languages to design distributed UI, sensor, and crowd-oriented processes. We contributed to each area with a set of extensions for existing process languages, components to abstract the actors' capabilities, and with tools that support the design, deployment and execution of processes modeled with our languages. The proposed languages support the development of the scenarios, which reflect common application needs in the focused on domains. With our research we contributed to various projects, publicly and privately funded. Research outcomes were published in peer-reviewed conferences and journals specific to business process management domain and also specific to the domains we focused on. This demonstrates the viability of the work and its relevance not only to the specific domains but also to the business process management community.

Bibliography

[1] R. Acerbis, A. Bongio, M. Brambilla, S. Butti, S. Ceri, and P. Fraternali. "Web Applications Design and Development with WebML and WebRatio 5.0." In: *Objects, Components, Models and Patterns*. Vol. 11. LNBIP. Springer, 2008, pp. 392–411. ISBN: 978-3-540-69824-1.

[2] Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. *Web Services Human Task (WS-HumanTask) Version 1.0*. Tech. rep. 2007.

[3] Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. *WS-BPEL Extension for People (BPEL4People) Version 1.0*. Tech. rep. 2007.

[4] S. Ahmad, A. Battle, Z. Malkani, and S. Kamvar. "The jabberwocky programming environment for structured social computing." In: *UIST'11*. 2011, pp. 53–64. ISBN: 978-1-4503-0716-1.

[5] M. Allahbakhsh, B. Benatallah, A. Ignjatovic, H. R. Motahari-Nezhad, E. Bertino, and S. Dustdar. "Quality Control in Crowdsourcing Systems: Issues and Directions." In: *IEEE Internet Computing* 17.2 (2013), pp. 76–81. ISSN: 1089-7801.

[6] I. Amundson, M. Kushwaha, X. Koutsoukos, S. Neema, and J. Sztipanovits. "Efficient Integration of Web Services in Ambient-aware Sensor Network Applications." In: *BaseNets 2006*. 2006.

[7] J. Anke, J. Müller, P. Spieß, and L. Chaves. "A service-oriented middleware for integration and management of heterogeneous smart items environments." In: *DI FCUL TR 06 10* (2006).

[8] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor. "AutoMan: a platform for integrating human-based and digital computation." In: *SIGPLAN Not.* 47.10 (Oct. 2012), pp. 639–654. ISSN: 0362-1340.

[9] M. Brambilla, S. Butti, and P. Fraternali. "WebRatio BPM: A Tool for Designing and Deploying Business Processes on the Web." In: *ICWE*. Springer, 2010, pp. 415–429.

[10] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. "Process modeling in Web applications." In: *ACM Trans. Softw. Eng. Methodol.* 15 (4 2006), pp. 360–409. ISSN: 1049-331X.

[11] M. Brambilla, P. Fraternali, and C. Vaca. "A Notation for Supporting Social Business Process Modeling." In: *BPMN*. 2011, pp. 88–102.

[12] M. Brambilla, P. Fraternali, and C. K. Vaca Ruiz. "Combining social web and BPM for improving enterprise performances: the BPM4People approach to social BPM." In: *Proceedings of the 21st international conference companion on World Wide Web*. WWW '12 Companion. Lyon, France: ACM, 2012, pp. 223–226. ISBN: 978-1-4503-1230-1.

[13] A. Caracas and A. Bernauer. "Compiling business process models for sensor networks." In: *DCOSS*. IEEE, 2011, pp. 1–8.

[14] A. Caracas and T. Kramp. "On the Expressiveness of BPMN for Modeling Wireless Sensor Networks Applications." In: *3rd internation workshop on BPMN*. 2011.

[15] F. Casati, F. Daniel, G. Dantchev, J. Eriksson, N. Finne, S. Karnouskos, P. M. Montero, L. Mottola, F. Oppermann, G. Picco, A. Quartulli, K. Römer, P. Spiess, S. Tranquillini, and T. Voigt. "Towards Business Processes Orchestrating the Physical Enterprise with Wireless Sensor Networks." In: *ICSE 2012*. 2012.

[16] F. Casati, F. Daniel, A. D. Angeli, M. Imran, S. Soi, C. R. Wilkinson, and M. Marchese. "Developing Mashup Tools for End-Users: On the Importance of the Application Domain." In: *IJNGC* 3.2 (2012).

[17] F. Casati, F. Daniel, G. Dantchev, J. Eriksson, N. Finne, S. Karnouskos, P. M. Montero, L. Mottola, F. J. Oppermann, G. P. Picco, A. Quartulli, K. Römer, P. Spiess, S. Tranquillini, and T. Voigt. "From Business Process Specifications to Sensor Network Deployments." In: *9th European Conference on Wireless Sensor Networks, Trento, Italy*. 2012.

[18] F. Casati, F. Daniel, A. Dunkels, S. Karnouskos, P. M. Montero, L. Mottola, F. J. Oppermann, G. P. Picco, K. Römer, P. Spieß, S. Tranquillini, P. Valleri, and T. Voigt. "makeSense: Easy Programming of Integrated Wireless Sensor Networks." In: *8th European Conference on Wireless Sensor Networks (EWSN 2011)*. IEEE Press, 2011.

[19] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kauffmann, 2002.

[20] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1.* W3C Note. `http://www.w3.org/TR/wsdl`: W3C, 2001.

[21] O. Chun, M. La Rosa, A. ter Hofstede, M. Dumas, and K. Shortland. "Toward Web-Scale Workflows for Film Production." In: *IEEE Internet Computing* 12.5 (2008), pp. 53 –61. ISSN: 1089-7801.

[22] S. Curran, K. Feeney, R. Schaler, and D. Lewis. "The management of crowdsourcing in business processes." In: *Integrated Network Management-Workshops, 2009. IM '09. IFIP/IEEE International Symposium on.* 2009, pp. 77–78.

[23] A. D'Ambrogio. "A Model-driven WSDL Extension for Describing the QoS of Web Services." In: *ICWS'06.* 2006, pp. 789–796.

[24] F. Daniel, F. Casati, B. Benatallah, and M.-C. Shan. "Hosted Universal Composition: Models, Languages and Infrastructure in mashArt." In: *ER'09.* Gramado, Brazil: Springer, 2009, pp. 428–443. ISBN: 978-3-642-04839-5.

[25] F. Daniel, J. Eriksson, N. Finne, H. Fuchs, A. Gaglione, S. Karnouskos, P. M. Montero, L. Mottola, F. J. Oppermann, G. P. Picco, K. Römer, P. Spieß, S. Tranquillini, and T. Voigt. "makeSense: Real-world Business Processes through Wireless Sensor Networks." In: *CONET/UBICITEC.* 2013, pp. 58–72.

[26] F. Daniel, A. Koschmider, T. Nestler, M. Roy, and A. Namoun. "Toward Process Mashups: Key Ingredients and Open Research Challenges." In: *Mashups'10.* ACM, 2010.

[27] F. Daniel, S. Soi, S. Tranquillini, F. Casati, H Chang, and Y Li. "MarcoFlow: Modeling, Deploying, and Running Distributed User Interface Orchestrations." In: *Proceedings of the 8th International Conference on Business Process Management Demo Track.* 2010, pp. 23–27.

[28] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. "Distributed orchestration of user interfaces." In: *Inf. Syst.* 37.6 (2012), pp. 539–556.

[29] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. "From People to Services to UI: Distributed Orchestration of User Interfaces." In: *BPM'10.* 2010, pp. 310–326.

[30] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters." In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782.

[31] F. Dengler, A. Koschmider, A. Oberweis, and H. Zhang. “Social Software for Coordination of Collaborative Process Activities.” In: *Business Process Management Workshops*. Ed. by M. Muehlen and J. Su. Vol. 66. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2011, pp. 396–407. ISBN: 978-3-642-20510-1.

[32] S. Dow, A. Kulkarni, S. Klemmer, and B. Hartmann. “Shepherding the crowd yields better work.” In: *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. CSCW ’12. Seattle, Washington, USA: ACM, 2012, pp. 1013–1022. ISBN: 978-1-4503-1086-4.

[33] J. Eriksson, N. Finne, L. Mottola, T. Voigt, F. Casati, F. Daniel, A. Gaglione, D. Molteni, G. P. Picco, S. Tranquillini, B.-O. Holländer, F. J. Oppermann, K. Romer, S. Doeweling, N. Oertel, F. Probst, P. Spiess, and P. M. Montero. *Final application implementations and evaluation of system & Final evaluation of the programming model*. Tech. rep. EU FP7 Project makeSense Deliverable D3.6 & D5.4, 2013.

[34] S. Erol, M. Granitzer, S. Happ, S. Jantunen, B. Jennings, P. Johannesson, A. Koschmider, S. Nurcan, D. Rossi, and R. Schmidt. “Combining BPM and Social Software: Contradiction or Chance?” In: *J. Softw. Maint. Evol.* 22.67 (Oct. 2010), pp. 449–476. ISSN: 1532-060X.

[35] M. Feldmann, T. Nestler, K. Muthmann, U. Jugel, G. Hübsch, and A. Schill. “Overview of an end-user enabled model-driven development approach for interactive applications based on annotated services.” In: *WEWST’09*. Eindhoven, The Netherlands: ACM, 2009, pp. 19–28. ISBN: 978-1-60558-776-9.

[36] N. Glombitza, M. Lipphardt, C. Werner, and S. Fischer. “Using graphical process modeling for realizing SOA programming paradigms in sensor networks.” In: *WONS 2009*. 2009, pp. 61 –70.

[37] J. Gómez, A. Bia, and A. Parraga. “Tool Support for Model-Driven Development of Web Applications.” In: *WISE’05*. Vol. 3806. LNCS. Springer, 2005, pp. 721–730.

[38] D. Guinard, V. Trifa, and E. Wilde. *Architecting a Mashable Open World Wide Web of Things*. Technical Report 663. Institute for Pervasive Computing, ETH Zurich, 2010.

[39] A. Hofstede, W. van der Aalst, M. Adams, and N. Russell. *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2009. ISBN: 364203120X, 9783642031205.

[40] J. Howe. *Crowdsourcing: Why the Power of the Crowd Is Driving the Future of Business.* 1st ed. New York, NY, USA: Crown Publishing Group, 2008. ISBN: 0307396207, 9780307396204.

[41] P. G. Ipeirotis, F. Provost, and J. Wang. "Quality management on Amazon Mechanical Turk." In: *Proceedings of the ACM SIGKDD Workshop on Human Computation.* HCOMP '10. Washington DC: ACM, 2010, pp. 64–67. ISBN: 978-1-4503-0222-7.

[42] P. Johannesson, B. Andersson, and P. Wohed. "Business Process Management with Social Software Systems – A New Paradigm for Work Organisation." In: *Business Process Management Workshops.* Ed. by D. Ardagna, M. Mecella, and J. Yang. Vol. 17. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2009, pp. 659–665. ISBN: 978-3-642-00327-1.

[43] S. Karnouskos, D. Savio, P. Spiess, D. Guinard, V. Trifa, and O. Baecker. "Real World Service Interaction with Enterprise Systems in Dynamic Manufacturing Environments." In: *Artificial Intelligence Techniques for Networked Manufacturing Enterprises Management.* Springer, 2010.

[44] R. Khazankin, B. Satzger, and S. Dustdar. "Optimized execution of business processes on crowdsourcing platforms." In: *CollaborateCom.* 2012, pp. 443–451.

[45] A. Kittur, S. Khamkar, P. André, and R. Kraut. "CrowdWeaver: visually managing complex crowd work." In: *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work.* CSCW '12. Seattle, Washington, USA: ACM, 2012, pp. 1033–1036. ISBN: 978-1-4503-1086-4.

[46] A. Kittur, J. V. Nickerson, M. Bernstein, E. Gerber, A. Shaw, J. Zimmerman, M. Lease, and J. Horton. "The future of crowd work." In: *Proceedings of the 2013 conference on Computer supported cooperative work.* CSCW '13. San Antonio, Texas, USA: ACM, 2013, pp. 1301–1318. ISBN: 978-1-4503-1331-5.

[47] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut. "CrowdForge: crowdsourcing complex work." In: *UIST'11.* 2011, pp. 43–52. ISBN: 978-1-4503-0716-1.

[48] N. Koch, A. Kraus, and R. Hennicker. "The Authoring Process of the UML-based Web Engineering Approach." In: *IWWOST'01.* 2001.

[49] A. Koschmider, M. Song, and H. A. Reijers. "Social software for business process modeling." In: *JIT* 25.3 (2010), pp. 308–322.

[50] P. Kucherbaev, S. Tranquillini, F. Daniel, F. Casati, M. Marchese, M. Brambilla, and P. Fraternali. “Business Processes for the Crowd Computer.” In: *Business Process Management Workshops*. Ed. by M. Rosa and P. Soffer. Vol. 132. Lecture Notes in Business Information Processing. Berlin Heidelberg: Springer, 2013, pp. 256–267. ISBN: 978-3-642-36284-2.

[51] A. Kulkarni, M. Can, and B. Hartmann. “Collaboratively Crowdsourcing Workflows with Turkomatic.” In: *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. CSCW ’12. Seattle, Washington, USA: ACM, 2012, pp. 1003–1012. ISBN: 978-1-4503-1086-4.

[52] G. La Vecchia and A. Cisternino. “Collaborative workforce, business process crowdsourcing as an alternative of BPO.” In: *Proceedings of the 10th international conference on Current trends in web engineering*. ICWE’10. Vienna, Austria: Springer-Verlag, 2010, pp. 425–430. ISBN: 3-642-16984-8, 978-3-642-16984-7.

[53] T. v. Lessen, F. Leymann, R. Mietzner, J. Nitzsche, and D. Schleicher. “A Management Framework for WS-BPEL.” In: *ECOWS’08*. IEEE, 2008, pp. 187–196. ISBN: 978-0-7695-3399-5.

[54] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. “Exploring iterative and parallel human computation processes.” In: *Proceedings of the ACM SIGKDD Workshop on Human Computation*. HCOMP ’10. Washington DC: ACM, 2010, pp. 68–76. ISBN: 978-1-4503-0222-7.

[55] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. “TurKit: human computation algorithms on mechanical turk.” In: *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. UIST ’10. New York, New York, USA: ACM, 2010, pp. 57–66. ISBN: 978-1-4503-0271-5.

[56] T. W. Malone, R. Laubacher, and C. Dellarocas. *Harnessing Crowds: Mapping the Genome of Collective Intelligence*. Tech. rep. MIT Center for Collective Intelligence, 2009.

[57] I. Manolescu, M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. “Model-driven design and deployment of service-enabled web applications.” In: *ACM Trans. Internet Technol.* 5 (3 2005), pp. 439–479. ISSN: 1533-5399.

[58] E. M. Maximilien, A. Ranabahu, and K. Gomadam. “An Online Platform for Web APIs and Service Mashups.” In: *IEEE Internet Computing* 12 (2008), pp. 32–43. ISSN: 1089-7801.

[59] S. Meyer, K. Sperner, C. Magerkurth, and J. Pasquier. "Towards modeling real-world aware business processes." In: *Proceedings of the Second International Workshop on Web of Things*. WoT '11. San Francisco, California: ACM, 2011, 8:1–8:6. ISBN: 978-1-4503-0624-9.

[60] P. Minder and A. Bernstein. "CrowdLang - First Steps Towards Programmable Human Computers for General Computation." In: *Human Computation*. 2011.

[61] P. Minder and A. Bernstein. "CrowdLang: A Programming Language for the Systematic Exploration of Human Computation Systems." In: *Social Informatics*. Ed. by K. Aberer, A. Flache, W. Jager, L. Liu, J. Tang, and C. Guéret. Vol. 7710. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 124–137. ISBN: 978-3-642-35385-7.

[62] L. Mottola and G. Picco. "Programming wireless sensor networks: Fundamental concepts and state of the art." In: *ACM Computing Surveys (CSUR)* 43.3 (2011), p. 19.

[63] B. A. Myers and M. B. Rosson. "User interface programming survey." In: *SIGCHI Bull.* 23 (2 1991), pp. 27–30. ISSN: 0736-6906.

[64] OASIS. *Web Services Business Process Execution Language Version 2.0*. Tech. rep. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, 2007.

[65] OASIS. *Web Services for Remote Portlets*. Tech. rep. `www.oasis-open.org/committees/wsrp`, 2003.

[66] D. Oleson, A. Sorokin, G. P. Laughlin, V. Hester, J. Le, and L. Biewald. "Programmatic Gold: Targeted and Scalable Quality Assurance in Crowdsourcing." In: *Human Computation*. 2011.

[67] OMG. *Business Process Model and Notation (BPMN), Version 2.0*. `http://www.omg.org/spec/BPMN/2.0`. 2011.

[68] C. Pautasso. "BPEL for REST." In: *BPM'08*. 2008, pp. 278–293.

[69] S. Pietschmann, M. Voigt, A. Rümpel, and K. Meißner. "CRUISe: Composition of Rich User Interface Services." In: *ICWE'09*. San Sebastian, Spain: Springer, 2009, pp. 473–476. ISBN: 978-3-642-02817-5.

[70] A. J. Quinn and B. B. Bederson. "Human computation: a survey and taxonomy of a growing field." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: ACM, 2011, pp. 1403–1412. ISBN: 978-1-4503-0228-9.

[71] C. Rodriguez, E. Zaupa, F. Daniel, and F. Casati. *Crowd-Based Pattern Mining - On the Crowdsourcing of Reusable Knowledge Identification from Mashup Models.* Deliverable. UNITN, 2013.

[72] D. Schall, B. Satzger, and H. Psaier. “Crowdsourcing tasks to social networks in BPEL4People.” English. In: *World Wide Web* (2012), pp. 1–32. ISSN: 1386-145X.

[73] D. Schall, H.-L. Truong, and S. Dustdar. “Unifying Human and Software Services in Web-Scale Collaborations.” In: *IEEE Internet Computing* 12.3 (May 2008), pp. 62–68. ISSN: 1089-7801.

[74] D. Schwabe, G. Rossi, and S. D. J. Barbosa. “Systematic Hypermedia Application Design with OOHDM.” In: *HYPERTEXT'96*. Bethesda, Maryland, United States: ACM Press, 1996, pp. 116–128. ISBN: 0-89791-778-2.

[75] *Service composition realization method compiler.* CN Patent 102,158,516. 2013.

[76] F. Skopik, D. Schall, H. Psaier, M. Treiber, and S. Dustdar. “Towards Social Crowd Environments Using Service-Oriented Architectures.” In: *it - Information Technology* 53.3 (2011), pp. 108–116.

[77] K. Sperner, S. Meyer, and C. Magerkurth. “Introducing Entity-based Concepts to Business Process Modeling.” In: *3rd International Workshop and Practitioner Day on BPMN*. 2011.

[78] P. Spiess, H. Vogt, and H. Jutting. “Integrating sensor networks with business processes.” In: *Real-World Sensor Networks Workshop at ACM MobiSys*. 2006.

[79] P. Spiess and S. Karnouskos. “Maximizing the Business Value of Networked Embedded Systems through Process-Level Integration into Enterprise Software.” In: *ICPCA 2007*. 2007, pp. 536–541.

[80] Sun Microsystems. *JSR-000168 Portlet Specification.* Tech. rep. `http://jcp.org/aboutJava/communityprocess/final/jsr168/`, 2003.

[81] C. Sungur, P. Spiess, N. Oertel, and O. Kopp. “Extending BPMN for Wireless Sensor Networks.” In: *Business Informatics (CBI), 2013 IEEE 15th Conference on.* 2013, pp. 109–116.

[82] S. Tranquillini, P. Kucherbaev, F. Daniel, and F. Casati. “Modeling and Enacting Flexible Crowdsourcing Processes.” To be submitted to ACM TWEB. 2014.

[83] S. Tranquillini, P. Spiess, F. Daniel, S. Karnouskos, F. Casati, N. Oertel, L. Mottola, F. Oppermann, G. Picco, K. Römer, and T. Voigt. "Process-Based Design and Integration of Wireless Sensor Network Applications." In: *Business Process Management*. Ed. by A. Barros, A. Gal, and E. Kindler. Vol. 7481. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 134–149. ISBN: 978-3-642-32884-8.

[84] R. Vdovjak, F. Frasincar, G.-J. Houben, and P. Barna. "Engineering Semantic Web Information Systems in Hera." In: *Journal of Web Engineering* 2.1-2 (2003), pp. 3–26.

[85] M. Vukovic. "Crowdsourcing for Enterprises." In: *Services - I, 2009 World Conference on*. 2009, pp. 686 –692.

[86] T. Weilkiens. *Systems engineering with SysML/UML: modeling, analysis, design*. Morgan Kaufmann, 2007.

[87] S. White. "Introduction to BPMN." In: *IBM Cooperation* (2004).

[88] S. Wilson, F. Daniel, U. Jugel, and S. Soi. "Orchestrated User Interface Mashups Using W3C Widgets." In: *ComposableWeb'11 (ICWE 2011 Workshop Proceedings)*. Springer, 2011.

[89] WSPER.org. *WS-BPEL 2.0 Metamodel*. Tech. rep. `http://www.ebpml.org/wsper/wsper/ws-bpel20.html`, 2007.

[90] J. Yu, B. Benatallah, F. Casati, and F. Daniel. "Understanding Mashup Development." In: *IEEE Internet Computing* 12 (2008), pp. 44–52. ISSN: 1089-7801.