



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

**Artifact-centric Business Process Models in UML:
Specification and Reasoning**

— PhD Thesis —

MONTSERRAT ESTAÑOL
Advised by PROF. ERNEST TENIENTE

March 2016

A thesis presented by Montserrat Estañol
in partial fulfillment of the requirements for the degree of
Doctor per la Universitat Politècnica de Catalunya.

Author: Montserrat Estanyol

Address: Department of Service and Information Systems Engineering
Edifici Omega, Despatx S206
Jordi Girona, 1-3
08034 Barcelona, Spain

E-mail: estanyol@essi.upc.edu - montse.estanyol@gmail.com

To my grandmas

Acknowledgments

This thesis would not have been possible with the help and support of many people.

To begin with, I would like to thank my advisor, Ernest Teniente. I am very grateful for his guidance, rigor, support and encouragement during all these years as his PhD student. Not only has he taught me how to do research and given me insights on his teaching methods, but he has also shown me his own way of understanding life. It has been a truly enriching experience, both academically and personally, and I feel incredibly honored and privileged for having been his PhD Student.

Secondly, I would also like to thank Anna Queralt and Maria-Ribera Sancho, for their advice, help and support during the early stages of the PhD. And Maria also trusted me with her students, giving me the chance to teach in one of her courses. A special thanks also goes to Antoni Olivé, who actually gave me the first opportunity to teach a class at university.

A big thank you as well to all the people, mainly researchers, I've had the pleasure to work with during these years: Josep Carmona, Jorge Muñoz, Diego Calvanese, Marco Montali, Sylvia Díaz-Montenegro and Manuel Castro.

I would also like to thank my colleagues in both the research group and the office for their support and for their help whenever I asked. My gratitude also goes to the anonymous reviewers, whose insightful comments allowed me to see some aspects of this thesis in a different light and helped me to improve it in ways which would not have occurred to me.

I am also very grateful to my family and friends, for their support and encouragement during all this time. In particular, I would like to thank my parents, especially my mother, for encouraging me to study and work hard. Together with my aunt, they have been a constant source of support and have always been ready to lend an ear.

A special thanks go to Manel, who encouraged me from the very start to begin the PhD journey. If it had not been for him I would not be writing this today. He has always given me his unwavering support and has been there when I needed him. And his knowledge of programming and good practices has proved very useful whenever I had questions. Thanks for everything.

Finally, I would like to dedicate this thesis to my grandmas. Neither of them had easy access to education as children and had their own share of difficulties, including surviving a war. Unfortunately, one of them passed away before seeing this work completed, although I believe she would have loved to see it. It has been through their efforts, and those of many

other people, including my parents, that I've had a much easier life. Thank you.

The work presented here has been partly supported by Universitat Politècnica de Catalunya - Barcelona Tech, the Spanish Ministerio de Ciencia e Innovación under project TIN2011-24747, the Spanish Ministerio de Economía y Competitividad under project TIN2014-52938-C2-2-R and by the Catalan agency AGAUR under project 2014 SGR 1534.

Abstract

Business processes are directly involved in the achievement of an organization's goals, and for this reason they should be performed in the best possible way. Modeling business processes can help to achieve this as, for instance, models can facilitate the communication between the people involved in the process, they provide a basis for process improvement and they can help perform process management.

Processes can be modeled from many different perspectives. Traditional process modeling has followed the process-centric (or activity-centric) perspective, where the focus is on the sequencing of activities (i.e. the control flow), largely ignoring or underspecifying the data required by these tasks.

In contrast, the artifact-centric (or data-centric) approach to process modeling focuses on defining the data required by the tasks and the details of the tasks themselves in terms of the changes they make to the data. The BALS framework defines four dimensions which should be represented in any artifact-centric business process model: business artifacts, lifecycle, services (i.e. tasks) and associations. Using different types of models to represent these dimensions will result in distinct representations, whose differing characteristics (e.g. the degree of formality or understandability) will make them more appropriate for one purpose or another.

Considering this, in the first part of this thesis we propose a framework, BAUML, for modeling business processes following an artifact-centric perspective. This framework is based on using a combination of UML and OCL models, and its goal is to have a final representation of the process which is both understandable and formal, to avoid ambiguities and errors.

However, once a process model has been defined, it is important to ensure its quality. This will avoid the propagation of errors to the process's implementation. Although there are many different quality criteria, we focus on the semantic correctness of the model, answering questions such as *does it represent reality correctly?* or *are there any errors and contradictions in it?*.

Therefore, the second part of this thesis is concerned with finding a way to determine the semantic correctness of our BAUML models. We are interested in considering the BAUML model as a whole, including the meaning of the tasks. To do so, we first translate our models into a well-known framework, a DCDS (Data-centric Dynamic System) to which then model-checking techniques can be applied. However, DCDSs have been defined theoretically and there is no tool that implements them.

For this reason, we also created a prototype tool, AuRUS-BAUML, which is able to translate our BAUML models into logic and to reason on

their semantic correctness using an existing tool, SVTe. The integration between AuRUS-BAUML and SVTe is transparent to the user. Logically, the thesis also presents the logic translation which is performed by the tool.

Contents

I Preface	1
1 Introduction	3
1.1 Artifact-centric Business Process Modeling	7
1.2 Quality of Business Process Models	10
1.3 Goals and Contributions of this Thesis	13
1.3.1 The BAUML Modeling Framework	15
1.3.2 Reasoning on BAUML Models	16
1.4 Research Methodology	17
1.5 Structure of the Document	19
II Modeling Artifact-centric Business Process Models	21
2 Preliminaries of Modeling	23
2.1 The BALSAs Framework	23
2.2 State of the Art	25
2.2.1 Process-centric Approaches	25
2.2.2 Bridging the Gap between Process-centric and Artifact-centric Specifications	26
2.2.3 Artifact-centric Approaches	27
2.2.4 Summary & Conclusions	30
3 Artifact-centric Business Process Modeling in UML	33
3.1 The <i>BAUML</i> Framework	34
3.1.1 Business Artifacts as a Class Diagram	35
3.1.2 Lifecycles as State Machine Diagrams	36
3.1.3 Associations as Activity Diagrams	41
3.1.4 Tasks (Services) as Operation Contracts	44

3.1.5	A Note on the Models	46
3.2	Formalization of the <i>BAUML</i> Framework	46
3.2.1	Class Diagram and Integrity Constraints	47
3.2.2	State Machine Diagrams	48
3.2.3	Activity Diagrams	49
3.2.4	Tasks	50
3.3	An Example with Two Artifacts	50
3.3.1	Class Diagram	51
3.3.2	State Machine Diagrams	51
3.3.3	Activity Diagrams	55
3.3.4	Operation Contracts	56
3.4	On the Relationship with Soft. Eng. Methodologies	59
3.4.1	Object-oriented Analysis	59
3.4.2	Enterprise Architecture	61
3.5	Summary & Conclusions	62
 III Reasoning on Artifact-centric Business Process Models		65
4	Preliminaries of Reasoning	67
4.1	Basic Concepts	67
4.2	State of the Art	69
4.2.1	Simulation	69
4.2.2	Process Model Testing	70
4.2.3	Syntactical & Structural Reasoning	70
4.2.4	Semantic Reasoning	71
4.2.5	Summary	75
5	Reasoning Using Data-centric Dynamic Systems	77
5.1	Background	78
5.1.1	An overview of Data-centric Dynamic Systems	79
5.1.2	Mapping DCDSs to the BALSAs Framework	80
5.1.3	Assumptions	81
5.2	Translating a UML Artifact-centric BPM to a DCDS	84
5.2.1	Translating the Database Schema and Equality Constraints	85
5.2.2	Translating the State Machine Diagram	89
5.2.3	Translating the Activity Diagrams	92
5.2.4	Translating the Tasks	99
5.2.5	Summary & Overview	107

5.3	Reasoning with the Resulting DCDS	109
5.3.1	Verification Logic	109
5.3.2	Evolution of an Artifact	110
5.4	Summary & Conclusions	112
6	Reasoning in Practice: AuRUS-BAUML	115
6.1	Checking the Semantic Correctness of BAUML Models	116
6.1.1	Verification	116
6.1.2	Validation	120
6.2	AuRUS-BAUML: The Tool & Its Workflow	122
6.2.1	ArgoUML	124
6.2.2	AuRUS-BAUML	125
6.3	Translation of BAUML into Logic	128
6.3.1	Background on Logic Formalization	128
6.3.2	Overview of the Translation Process	128
6.3.3	Translation Algorithms	129
6.4	Formalization of Tests & Results	135
6.4.1	Verification Tests	136
6.4.2	Validation Tests	139
6.4.3	Some Test Results	140
6.5	Summary & Conclusions	143
7	Decidability	147
7.1	Background	147
7.1.1	2-Counter Machines	148
7.1.2	Running Example: An Online-Retailer	148
7.2	Results of Our Decidability Analysis	151
7.2.1	Unrestricted Models	152
7.2.2	Models with Non-Shared Instances	153
7.2.3	Models With Shared Instances	157
7.2.4	Applicability of the Results to the Bicing Example	158
7.3	Summary & Conclusions	159
IV	Closure	161
8	Conclusions	163
8.1	Contributions	163
8.1.1	Modeling Artifact-centric Business Process Models	164

8.1.2	Reasoning on Artifact-centric Business Process Models	165
8.2	Further Research	166
8.3	Impact of the Thesis	167
8.3.1	Artifact-centric Business Process Modeling	167
8.3.2	Reasoning on Artifact-centric Business Process Models	169
References		173
Appendix A Bicing: Full Example Specification		187
A.1	One Artifact	187
A.1.1	Class Diagram	187
A.1.2	State Machine Diagram	189
A.1.3	Activity Diagrams & Operation Contracts	189
A.2	Two Artifacts	191
A.2.1	Class Diagram	192
A.2.2	State Machine Diagram	193
A.2.3	Activity Diagrams & Operation Contracts	194
Appendix B Translation of Bicing into a DCDS		197
B.1	Data dimension	197
B.2	Condition-Action Rules	197
B.2.1	Actions	199
Appendix C Complexity: Proofs		203
C.1	Background on 2-Counter Machines	203
C.2	Theorems' Proofs	204
C.2.1	Unrestricted Models	204
C.2.2	Models with Non-Shared Instances	205
C.2.3	Models with Shared Instances	212

Part I

Preface

Chapter 1

Introduction

A process can be defined as “the combination of a set of activities within an enterprise with a structure describing their logical order and dependence whose objective is to produce a desired result” [5]. A similar definition is given by Weske [133], who states that a business process “consists of a set of activities that are performed in coordination in an organizational and technical environment [and which] jointly realize a business goal”. From both definitions, we can gather that business processes are a set of activities carried out in a particular manner in order to achieve a certain result or business goal.

Business process modeling, as the name implies, consists in representing business processes by means of a model. Some of the approaches to business process modeling represent both the process and the goals together [112, 113], thus highlighting the importance of goals. Bearing in mind, then, the close relationship between processes and goals, it is important to ensure that business processes are performed in the best possible way.

Business process models can help to achieve this, as they have several advantages, including the following [35]:

- They facilitate human understanding and communication. The use of a common language between the parties involved in the process (managers, analysts, modelers, etc.) facilitates this understanding.
- They provide a basis for process improvement.
- Having a process model helps support process management, as it provides a reference model to compare the actual behavior of the process to it.

In consequence, modeling processes correctly can play an important role in the success of a company, as it is through their processes that businesses add value to their products or services. However, one unique process model can hardly provide all the information that may be relevant for the business. Therefore, Curtis, Kellner and Over define different perspectives on process modeling [35], which share some similarities with the dimensions or spaces defined over information systems [107]. Each perspective emphasizes a particular kind of information that is relevant in the process. They are the following:

- **Functional perspective:** Shows which process elements are performed and the flow of informational entities (e.g. artifacts) that are relevant.
- **Behavioral perspective:** Shows when process elements are performed (for instance, by means of sequencing) and how (e.g. decision conditions, loops, entry and exit criteria, etc.).
- **Organizational perspective:** Shows where and by whom in the organization process elements are performed, the physical communication mechanisms used for the transfer of entities, and the physical media and locations used for storing entities.
- **Informational perspective:** Shows the informational entities produced or manipulated by a process. It should also represent the structure of informational entities and their relationships.

It is really difficult to unite all these perspectives in a single model. Therefore different languages or models will be more or less suited to representing a process from a particular perspective. For instance, BPMN (Business Process Modeling and Notation) is an ISO standard language for business process modeling, and its goal is to “provide a notation that is readily understandable by all business users, from the business analysts [...], to the technical developers [...], and finally, to the business people [...]” [69]. Despite being the standard language for process modeling, it has some drawbacks. For instance, it lacks expressiveness in terms of defining the resource assignment to tasks (as part of the organizational perspective) [24] or other authors find its procedural nature too restrictive [40].

Bearing this in mind, Aguilar-Savén [5], studies and analyses what she calls “process modeling techniques” (i.e. alternative ways of representing processes) in order to help researchers find the most appropriate technique depending on the desirable properties of the resulting model.

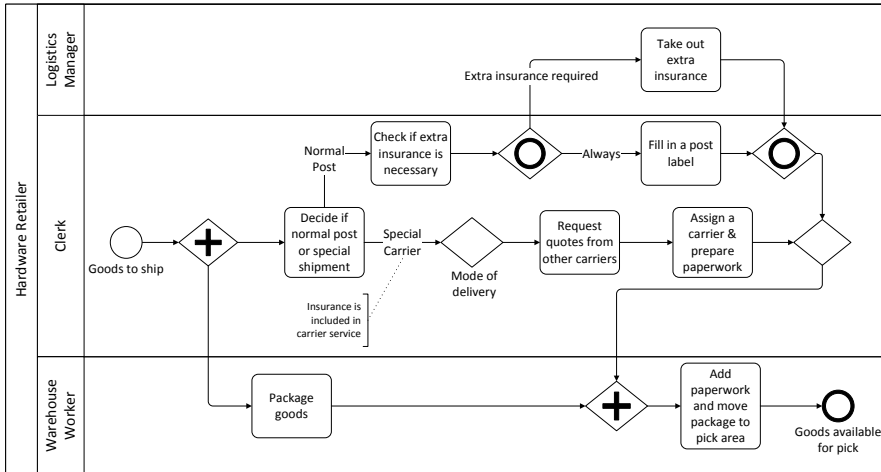


Figure 1.1: BPMN activity-centric diagram, obtained from [97].

As some researchers point out in [20, 19, 36, 37], traditional business process modeling has been centered on what they call a process-centric (or activity-centric) approach: all the attention has been on the sequencing of activities (i.e. the control flow), underspecifying or ignoring the data needed in each of these activities.

For example, Figure 1.1 shows a typical process-centric diagram in BPMN [98]. The diagram represents the shipment process of a hardware retailer. As it can be seen, it focuses mainly on the tasks required to achieve the final goal: having the goods available to be picked up.

The process begins once the goods are available to be shipped. At this point, there are two concurrent flows of activity: on the one hand, the goods are packaged by a warehouse worker. On the other hand, the clerk deals with the details of the shipment: she decides whether the post is normal or requires a special shipment. In the case of a normal post, she checks if it requires additional insurance, and if it does, the logistics manager gets it. In any case, she fills a label with the required details to post the package. If the package requires a special carrier, the clerk requests a quote, assigns it to a carrier and prepares the paperwork. Finally, when the goods have been packaged and the paperwork is ready, the latter is added to the package and it is moved to the pick-up area.

Notice that, given only the diagram in Figure 1.1, we can have an intuitive idea of what the process is doing, but it is lacking in detail. Although we clearly know the order of execution for the different tasks, we do not know exactly what each of the tasks is doing. For example: *how does the clerk decide if the package requires standard post or a special shipment?, what determines if additional insurance is necessary? or what changes are made to the system underlying the process?.* This would require information about the data which is manipulated by these tasks, and which is not specified in the diagram.

Relating this to the perspectives defined by [35], we can say that process modeling has focused on describing processes from a behavioral perspective or “control flow” perspective [25]. Even the two definitions of process that we have given at the beginning of this chapter are based on such perspective; especially the one from [5], as it emphasizes the fact that activities are carried out in a certain order and may have dependencies among them.

In recent times this lack of attention to the data required by the process has resulted in a new approach to process modeling: data-centric or artifact-centric business process modeling, first introduced in [77, 94]. This approach focuses on the data that is needed to carry out the different tasks in a process, and the dependencies between these data. It relies on the assumption that businesses need to record details of what they produce in terms of information, and business artifacts are the means to do it.

Nevertheless, despite the emphasis on data, artifact-centric process modeling goes further than just defining the data that is needed in the process. Artifact-centric approaches also model the services or tasks that are in charge of updating business artifacts, and they indicate when these tasks may be executed.

For example, Figures 1.2 to 1.5 on pages 7 and 8 show various fragments of an artifact-centric business process model extracted from [20]. Without delving into the details, the diagrams indicate the structure of the data, how it evolves, the conditions under which the tasks may take place and what each of the tasks is doing. For this reason, artifact-centric process modeling falls in between the functional, behavioral and informational perspectives described by [35].

More importantly, however, the artifact-centric approach has been successfully applied to real-life cases, as shown in [17, 18]. Important companies in the IT sector, such as IBM, have already incorporated some of the notions of artifact-centric process modeling into their services [64].

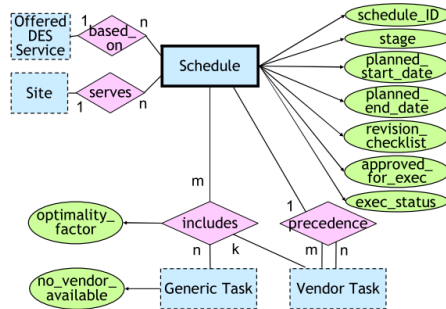


Figure 1.2: Fragment of an ER diagram representing the business artifacts, extracted from [20].

1.1 Artifact-centric Business Process Modeling

A brief look into the research literature shows the different approaches that have been proposed and are used to model artifact-centric business processes since Nigam and Caswell first defined them in 2003 [94]. Alternatives range from predominantly graphical notations such as [38, 78] to logic-based ones [11, 27, 37].

In order to facilitate the definition and the structuring of artifact-centric BPMs, the BALSAs framework, first described in [20, 64], establishes four different dimensions which should be present in any artifact-centric process model: business artifacts, their lifecycles, services and associations¹.

Intuitively, business artifacts represent the key data for the business and have a lifecycle which shows how they evolve during their life. These artifacts are manipulated by a set of services, and associations restrict the manner in how the services make changes to the artifacts.

For instance, Figures 1.2 to 1.5 show how these four dimensions are represented in [20]. In this particular case, the authors have used an ER diagram for the artifacts, a state machine diagram for the lifecycles, event-condition-action rules in natural language to represent the associations and an operation contract in natural language (with input, output, a precondition and a set of conditional effects) to specify the services.

In contrast, [27] uses an ontology to represent business artifacts, condition-action rules for associations, and actions - which have an input parameter and

¹See Section 2.1 for more details.

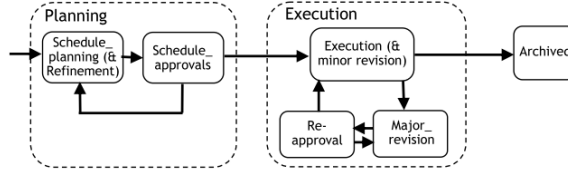


Figure 1.3: Lifecycle of artifacts, extracted from [20].

R1: Initiate schedule

- event** request by performer p to create a schedule instance for Offered DES Service artifact o , Customer artifact c , and Site artifact si
- condition** the appropriate non-disclosure agreements (NDAs) are in place for c
- action** invoke $create_schedule(o, c, si)$
- by** performer p where $offer_manager$ in $role(p)$ and $qualification(p, o, region: si.region) \geq 5$

Figure 1.4: Event-condition-action rule showing the conditions under which a certain service (i.e. action) can execute, extracted from [20].

The IOPE specification for $adjust_task_dates$ has the following properties:

- Inputs:
 - A *Vendor Task* artifact t , information about specific requirements for the customer and site associated with t 's schedule, and about the current status of various steps (government approvals, equipment availability, etc.).
 - A *Vendor* artifact v , and specifically information about v 's availability, about the cost for re-scheduling the task, etc., for the vendor assigned to perform t .
 - A *Schedule* artifact sch , and specifically information about immediate predecessors and successors of t in sch .
 - A list T of triples of form $(Task, date, date)$.
- Outputs:
 - Updates to start and/or end dates of t .
 - (Possibly) updates to the *revision_checklist* of sch .
 - (Possibly) updates to the *status* fields of each *Vendor Task* artifact t' that is a successor of t in sch , if the modification to t impacts t' , and invalidating the dates of each such artifact.
- Pre-condition
 - Vendor task t is assigned to supplier v .
 - Vendor task t occurs in Schedule sch .
 - T is the list of tuples (t', s', e') , where t' is a task that succeeds t in sch according to the *Precedence* relationship, and s', e' are the start- and end-times of t' , respectively.
- Conditional effects
 - If true, then the start and/or end dates of t may have been overwritten
 - If the start date of t is overwritten, then it is after the end date of each predecessor of t .
 - If the start or end date of t is overwritten and this impacts the timing of any successor t' of t (i.e., any task occurring in T), then the dates for t' are invalidated and the *revision_checklist* of sch is updated accordingly.

Figure 1.5: Service definition, extracted from [20].

Bachelor \sqsubseteq Student	$\delta(\text{MNum}) \sqsubseteq$ Student	(funct MNum)
Master \sqsubseteq Student	Student $\sqsubseteq \delta(\text{MNum})$	(id Student MNum)
Graduated \sqsubseteq Student		

Figure 1.6: Ontology extracted from [27].

- ENROLLED(id, -, -, -, NULL) \rightsquigarrow GRADUATE(id)
- TRANSF_M(name, surname) \rightsquigarrow COMPL-ENR(name, surname)

Figure 1.7: Condition-action rules extracted from [27].

```

GRADUATE(id) : { GRAD(id2, m, t)  $\rightsquigarrow$  GRAD(id2, m, t),
                 TRANSF_M(n, s)  $\rightsquigarrow$  TRANSF_M(n, s),
                 ENROLLED(id2, n, s, t, d)  $\wedge$  id2  $\neq$  id  $\rightsquigarrow$  ENROLLED(id2, n, s, t, d),
                 ENROLLED(id, n, s, t, NULL)  $\rightsquigarrow$  ENROLLED(id, n, s, t, today()),
                 ENROLLED(id, -, -, t, NULL)  $\rightsquigarrow$  GRAD(id, getMark(id, t), t) }
COMPL-ENR(n, s) : { GRAD(id, m, t)  $\rightsquigarrow$  GRAD(id, m, t),
                   ENROLLED(id, n2, s2, t, d)  $\rightsquigarrow$  ENROLLED(id, n2, s2, t, d),
                   TRANSF_M(n2, s2)  $\wedge$  (n2  $\neq$  n  $\vee$  s2  $\neq$  s)  $\rightsquigarrow$  TRANSF_M(n2, s2),
                   TRANSF_M(n, s)
                    $\rightsquigarrow$  ENROLLED(getID(n, s, "MSc"), n, s, "MSc", NULL) }

```

Figure 1.8: Actions extracted from [27].

contain a set of effects to determine the changes - for services (see Figures 1.6 to 1.8). Notice that in this case there is no specific representation of the lifecycle dimension, as the authors do not follow the BALSAs framework.

What is interesting to see is the differences and the appropriateness of each approach in different circumstances. If we look at the first model (Figures 1.2 to 1.5), it employs graphical representations for the business artifacts and the lifecycle. For the services and the associations, it turns to natural language. On the other hand, [27] uses a system of representation grounded on logic (Figures 1.6 to 1.8).

In the first case, the whole artifact-centric business process model will be easier to understand due to the use of well-known notations (ER models and state machines) and natural language. In contrast, the second example uses a system of representation based on logic which is more complex and unpractical from the point of view of the business.

At the same time, the use of natural language can easily lead to errors and misunderstandings, as it is ambiguous. However, the artifact-centric process

model of [27] does not share this problem, as the use of logic makes the model formal and unambiguous.

Therefore, as we have seen by looking at these two examples, different types of models have different characteristics which will be more appropriate for one circumstance or another. By using different models for each of the dimensions in the BALSAs framework, we will obtain different business process models with different characteristics bringing them closer to one perspective or another.

One of the challenges is to find the most appropriate model to represent these dimensions [64] depending on the model's purpose. Ideally, the final model or set of models should be able to represent artifact-centric business process models in a formal way, in order to avoid ambiguities and errors, but in a way that is easy to understand for all the parties involved in the business process, such as managers, analysts and modelers. This is one of the challenges in artifact-centric business process modeling.

Challenge

Find a way to represent artifact-centric business process models that is easy to understand for the people involved in the business process and at the same time is formal, to avoid ambiguities and errors.

1.2 Quality of Business Process Models

However, having a model that is intuitive and easy to understand for business people and designers is not enough. Ensuring the quality of the model is important, for many different reasons, such as avoiding errors in the final implementation of the process or having a process that is more efficient and compliant with the business requirements, just to mention a few. This will, in turn, save the company money and make it easier to achieve their goals.

Although we can distinguish between the quality of business processes and the quality of the modeling process [95], we will focus on the quality of a model. Hommes [63] summarizes the most relevant criteria used to evaluate it:

- **Completeness:** The model should contain all statements of the domain that are relevant.

- **Correctness:** There are two types of correctness:
 - **Syntactic correctness:** The model complies with the syntax of the modeling language that is used.
 - **Semantic correctness:** The model represents the domain correctly.
- **Consistency:** The statements in the model are not contradictory.
- **Minimality:** The model should be as simple and as small as possible while at the same time representing the domain correctly.
- **Comprehensibility:** The model should be easy to understand by its users and developers.
- **Predictive value:** It should be easy to infer statements from the model that comply with the represented domain.

Checking whether a model fulfills all these criteria can be daunting. Some of the criteria, such as comprehensibility, are subjective to a certain degree and therefore difficult to check in an objective way. Other criteria, such as correctness and consistency, are more objective and easier to verify.

According to the literature review in [95], not many of the works that deal with business process quality focus on their correctness or semantic quality. From those that do, process-centric approaches have centered on verifying the syntactical and structural correctness (e.g. detecting deadlocks, lack of sync) of the models. Examples of these works are [3, 43, 116, 91]. These approaches, being process-centric, lack knowledge from the domain they represent, as they do not model the data. For this reason it is not possible to perform any kind of semantic tests automatically.

This can be easily seen by returning to the process-centric model in Figure 1.1. An example of syntactic check that we can perform is ensuring that the nodes have the appropriate number of incoming and outgoing edges. For instance, an end event (Figure 1.9) cannot have any outgoing edges or flows, or a gateway (Figure 1.9) must either have multiple incoming or multiple outgoing flows. Notice that in Figure 1.1 they do have the appropriate number of flows.

Let's see another example, this time of a structural property. In this case we could check if parallel paths are closed by a parallel gateway and not an inclusive or exclusive gateway to avoid lack of sync errors. If we look at the diagram, we can see that the flow is split in parallel paths right after it begins



Figure 1.9: Different BPMN nodes.

execution, and the paths are joined together again right before the end using a parallel gateway. Therefore, the diagram has no synchronization errors.

Another property that could be interesting to check is the executability of the tasks or activities. Notice that this would fall in the category of semantic correctness. For instance, at a certain point in the flow the clerk has to decide the mode of delivery, and it is either normal post or a carrier. Can we guarantee in some way that the tasks in both paths can be executed? Or maybe, for some unknown reason, only one of the paths is taken? Unfortunately, since we have no knowledge about the data, we cannot answer these questions without implementing the process, and even then, it may be difficult to find an answer.

In contrast, these types of question could be answered in an artifact-centric process model, due to the definition of the data and the specification of the tasks or services. Research in artifact-centric business process modeling has precisely focused on checking the semantic correctness of the model (see for instance [9, 10, 37, 27]) in order to answer them.

Returning to the two different artifact-centric models we saw before (pages 7 to 9), both have the potential to be checked to ensure their semantic correctness. For instance, in the first artifact-centric example (Figures 1.2 to 1.5 on pages 7 and 8) we could check if tasks `create_schedule` and `adjust_task_dates` can be executed. To do so, we would need to consider all the different models (in this case, the ER and the state machine diagrams, and the ECA rules and the specification of the tasks in natural language).

Although the first model/example has the potential to be checked semantically because of the data and the definition of the tasks, it has an issue that prevents it: the tasks and ECA rules are defined in natural language, natural language can be ambiguous and therefore it would require first a transformation into a more formal language. As it is, we cannot answer those questions.

On the other hand, the second model (Figures 1.6 to 1.8 on page 9) uses a formal notation grounded on logic for all the elements in the model. In this case, questions like *Can task graduate execute?* or *Do all enrolled students graduate?* can be answered applying techniques of model checking to the

model. One of the goals of the paper [27] is, in fact, to show how to ascertain specific properties over the model.

Most of the approaches that deal with semantic reasoning on artifact-centric business process models such as [9, 10, 37, 27] also use languages based on logic. These models are clearly difficult to understand by all people involved in the business process, but more particularly so by managers and business analysts. Therefore, another challenge is to be able to check automatically their semantic correctness, but the models should be defined at a high level of abstraction and use a language that can be understood by them.

Challenge

Find a way to check the semantic correctness of an artifact-centric process model defined in a high-level language which can be understood by the people involved in the business process.

1.3 Goals and Contributions of this Thesis

The goal of this thesis is to address the two challenges presented in the previous sections. More specifically, we wish to find a way to represent business processes following an artifact-centric perspective that has the following characteristics:

- It provides us with a high-level representation of the business process.
- It can be easily understood by the people involved in the different stages of the business process development, from definition to implementation.
- It has precise semantics so that it can be used to check the models' correctness automatically.
- It is possible to check the correctness of the model automatically.

These goals are summarized in the table below:

Goals of the Thesis

Model artifact-centric business process models following the BALSAs framework, using a high-level language that is easy to understand by the people involved in the business process and at the same time formal enough to avoid ambiguities.

Define a method to check the correctness of the artifact-centric business process model defined above.

Considering this, the contributions of this thesis are the following:

1. A framework, BAUML, for modeling artifact-centric business process models using UML and OCL. This provides us with process models which have a high-level of abstraction and use a language which is known in the academia and the industry.
2. An approach to check the semantic correctness of the artifact-centric business process models as defined in the BAUML framework. To do so, the models need to be translated into intermediate languages which can then be used to obtain the results. We use two different approaches to do so:
 - a) The first approach is based on DCDSs (Data-centric Dynamic Systems) [11], which is a theoretical framework for artifact-centric business process modeling grounded on logic. Once the models are translated into a DCDS, model checking techniques can be applied to it to determine their correctness.
 - b) The second approach relies on an existing tool, SVTe [53], to perform the correctness checks. In this case, the intermediate language is based on first-order logic.
3. A prototype tool, AuRUS-BAUML, to automatically translate the BAUML models into first-order logic as required by SVTe. By connecting the two tools, the translation and reasoning process becomes transparent to user/modeler.
4. A study on the conditions which guarantee decidability of reasoning over our models.

Table 1.1 summarizes the contributions, the chapters where they are and the related publications.

Table 1.1: Summary of the contributions, their location in this thesis and the related publications

Topic	Contribution	Location	Publication
Modeling	Framework	Section 3.1	[45, 46, 47, 48, 49]
	Formalization	Section 3.2	[28, 29, 51]
Reasoning	Using DCDS	Chapter 5	[50]
	Using AuRUS-BAUML	Chapter 6	[51]
	Decidability	Chapter 7	[28, 29]

1.3.1 The BAUML Modeling Framework

The first contribution of this thesis is the BAUML framework, which we use to specify business processes from an artifact-centric perspective. To do so, we use the dimensions in the BALSAs framework [64] as a basis. For each dimension, we propose a combination of UML and OCL models and explain how they are related.

Despite the fact that BPMN is the *de facto* standard for business process modeling, it is very limited when it comes to representing the data, a key element in artifact-centric business process modeling.

On the other hand, using UML and OCL to represent all the dimensions in the BALSAs framework provides several advantages:

- We represent all the dimensions of the BALSAs framework using the same languages.
- Both languages integrate in a natural way.
- UML is an ISO standard [67] and is a language known both in the academia and in the industry.
- Using these languages provides us with a high-level representation of the process, which is what we strive for.

The details of this framework and its formalization can be found on Chapter 3.

Related Publications A first approach to the framework was presented in [46]. Simultaneously, we also created a technical report [45] with its application to the EU-Rent case study, a fictional car-rental company. The framework was further refined and described in [47], where it was selected for a revised version to be published as a book chapter. In the revised version [48], we included the application of the framework to a more complex example with two artifacts and we created another technical report with the full specification of the example [49].

The framework was then formalized in [28] and [51].

1.3.2 Reasoning on BAUML Models

The second main contribution of the thesis deals with the correctness of BAUML models. Having a model that can be understood by the various parties involved in the business process is just a first step towards achieving our goal. The next step, and what adds more value, is being able to check the correctness of these models automatically.

As we have seen, we distinguish between syntactic and semantic correctness. We focus on semantic correctness, because the powerfulness of artifact-centric process models lies on the domain knowledge embedded in them due to the use of data.

We consider two types of semantic correctness: *verification* ensures that there are no inherent errors or contradictions in the model, whereas *validation* checks that the models represent the domain accurately. Verification can potentially be performed without user intervention, whereas validation requires, in most cases, user intervention to define the desirable properties that the model should fulfill to be a valid representation of the domain.

As there was no tool that could perform the verification and validation tests on the BAUML models, we opted to translate them into an intermediate language which can then be used for this purpose. We adopt two different approaches.

In the first approach, we translate the models into a Data-centric Dynamic System. DCDSs are grounded on logic and use properties defined in μ -calculus to state the characteristics that the system should fulfill. By applying model checking techniques, we can then determine if the model fulfills them.

Despite the potential of DCDS, there is no tool that can actually perform the kind of tests described above. However, there is a tool in our department, SVTe [53], that given a database schema, a goal, and a set of restrictions is able to generate a set of base facts necessary to achieve the goal without violating

any restrictions. If no such solution exists, then the tool returns a list of the restrictions that do not allow reaching the goal.

Therefore, we take advantage of the tool by translating the BAUML models into the logic required by it. Finally, in order to prove the feasibility of our approach, we have implemented a prototype tool, AuRUS-BAUML, that translates automatically the BAUML models into a first-order logic suitable for SVTe. SVTe performs the reasoning, and the result obtained by it is then provided to the user. The interaction between the two tools is transparent to the user.

As a last contribution in this area, we also study the complexity of reasoning on our BAUML models and determine the restrictions which guarantee decidability over them.

Related Publications The work described above has resulted in several publications. The use of DCDS to perform reasoning on our models was first described in [50]. This work has been extended in the present thesis and can be found on Chapter 5. Similarly, the approach using SVTe has been explained in [51]. The complexity results have been published in [28, 29].

1.4 Research Methodology

The research methodology that we follow in this thesis is called design-science research. Design science, as Hevner and others explain, “creates and evaluates IT artifacts intended to solve identified organizational problems”[61]. These artifacts may be represented as software, logic, mathematics or even informal descriptions in natural language.

As [61] points out, design science is both a process and a product. There are two different processes or activities: build and evaluate. Building consists in creating the product; evaluating means checking that the performance of the product is appropriate for the purpose it was first built [90].

According to [90] there are four product types: constructs, models, methods and instantiations. Constructs are the common language for communication; models represent the design problem and its solution; methods indicate the way to perform certain activities; instantiations show that constructs, models and methods can be implemented [61, 90].

There are several guidelines that can be followed to perform design-science research [61]. They are the following:

1. **Design as an artifact:** The creation of an innovative artifact or product.

2. **Problem relevance:** The artifact should provide some utility for a specified problem.
3. **Design evaluation:** The artifact should be evaluated exhaustively.
4. **Research contributions:** The artifact should be innovative.
5. **Research rigor:** The artifact must be defined, represented formally, and it should be coherent and consistent.
6. **Design as a search process:** The search for an artifact should comply with laws in the problem's environment using available means.
7. **Communication of research:** The results of the research should be communicated in a way that is understandable by both technical and business people.

In this thesis we followed the guidelines above. To begin with, the result of our work is a framework to model artifact-centric business process models using UML and OCL, and two different approaches to automatically check the models' correctness (Guideline 1). This is useful (Guideline 2) in two different ways. First of all, we present a model that is meant to be easy to understand and contributes to existing research by introducing an alternative way of representing artifact-centric process models. Secondly, we show how to check the model in order to ensure its quality. In this way, we avoid errors reaching the implementation stage.

We have validated this proposal and its translation into logic by applying it to different examples and one case study. We also created a prototype tool to show the feasibility of our approach (Guideline 3). Our proposal is innovative because none of the existing proposals uses a combination of UML and OCL to represent all the dimensions of an artifact-centric process model and, to the best of our knowledge, there was no tool to automatically perform the validation with these models (Guideline 4).

Moreover, the research process and results are rigorous, as we have based it on existing research and proven methods of reasoning, on top of which we have added our contributions (Guidelines 5 and 6). Finally, our research results have been published in relevant conferences and journals, and therefore have been communicated effectively to the relevant communities (Guideline 7). For a list of these publications and their abstracts, check Section 8.3 on page 167.

Last but not least, we would like to point out that Léelo [82], a Spanish company, expressed an interest in our work. For this reason, our university signed

a collaboration agreement with them, under project name *Kopernik: cambiando el origen de las cosas (datos vs tareas)* (*Kopernik: changing the origin of things (data vs tasks)*), in order to work together to adapt the modeling methodology to their specific needs. This confirms the relevance of the problem in the industry and shows that the impact of our work goes further than the academia.

1.5 Structure of the Document

This thesis is structured into four different parts:

Part I The first part has one chapter (Ch. 1: Introduction) and comprehends the introduction to the topic of the thesis, its contributions in the context of design-science research and its structure.

Part II The second part of the thesis focuses on BAUML, our methodology for artifact-centric business process modeling. It includes two chapters:

Chapter 2: Preliminaries of Modeling This chapter introduces the BALS framework, which we base our work on, and a review of the state of the art in terms of modeling business processes.

Chapter 3: Artifact-centric Business Process Modeling in UML This chapter presents our approach for artifact-centric business process modeling using a combination of UML and OCL models. It illustrates the approach by using an example based on a city-bicycle rental system such as Bicing in Barcelona. It also formalizes the approach and afterwards extends the example to include two artifacts. The chapter ends by giving a brief overview of the relationship between our approach and software engineering methodologies.

Part III The third part of the thesis focuses on reasoning on these models. It is structured in four different chapters:

Chapter 4: Preliminaries of Reasoning This chapter introduces the topic of reasoning on artifact-centric business process modeling. It also analyzes the related work by specifically focusing on semantic reasoning (i.e. considering the data and the tasks), although it also gives an overview of related topics such as simulation, syntactic/structural reasoning and process model testing.

Chapter 5: Reasoning Using Data-centric Dynamic Systems This chapter introduces DCDSs (Data-centric Dynamic Systems), a theoretical framework for artifact-centric business process models which, as it is grounded on logic, makes it suitable for reasoning and checking the correctness of the artifact-centric BPM. Therefore, the chapter details the translation process required for going from BAUML to DCDSs and how we can perform the correctness checks.

Chapter 6: Reasoning in Practice: AuRUS-BAUML Unfortunately, the DCDS framework in the previous chapter has been defined theoretically and no tool can actually perform the required checks to ensure the model's correctness. Bearing this in mind, this chapter presents some validation and verification tests, together with a prototype tool, AuRUS-BAUML. AuRUS-BAUML is able to translate the BAUML models into first-order logic as required by another tool, SVTe, which is able to answer the kind of questions we are interested in once they are formulated into logic. SVTe is integrated into AuRUS-BAUML seamlessly.

Chapter 7: Decidability Finally, the last chapter in this part studies the decidability of performing reasoning on our BAUML models. It proves that, without any restrictions, it is undecidable to check if the model fulfills a given property. By incrementally establishing restrictions over the models, we find out a class of models for which decidability is guaranteed.

Part IV The final part of the thesis contains only one chapter, (Ch. 8: Conclusions). This chapter summarizes the contributions of the thesis and points out further research. It also includes a section on the various publications in relevant conferences or as book chapters that have resulted from the work presented here.

Part II

Modeling Artifact-centric Business Process Models

Chapter 2

Preliminaries of Modeling

As we have explained, one of the goals of our thesis is to find a way to define business process models from an artifact-centric perspective in a modeler-friendly way, but using a language which at the same time has precise semantics. Before doing so, this chapter presents the necessary background. First of all, it introduces the BALSAs framework, which we use as a basis for our work. After this, we examine the related work in two related areas: process-centric and artifact-centric business process modeling.

2.1 The BALSAs Framework

To facilitate the analysis of artifact-centric process models, [64, 20] proposed the use of the BALSAs (Business Artifacts, Lifecycles, Services and Associations) framework. This framework establishes the common ground for artifact-centric business process modeling by defining four different dimensions which, ideally, should be present in any artifact-centric process model. We summarize here the most relevant characteristics of each dimension:

- **Business artifacts** represent the data required by the business and whose evolution we wish to track. Each artifact has an identifier and may be related to other artifacts, as represented by the associations among them.
- The **lifecycle** of a business artifact states the relevant stages in the evolution of the artifact, from the moment it is created until it is destroyed. Each business artifact will have the corresponding lifecycle.

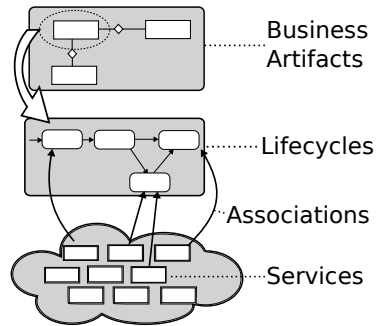


Figure 2.1: Dimensions in the BALSAs framework. Adapted from [64].

- **Services**, or tasks, represent atomic units of work and they are in charge of creating, updating and deleting the business artifacts.
- **Associations** represent constraints in the manner how services make changes to artifacts. That is, in general, services cannot freely manipulate the artifacts, and the associations correspond to the restrictions imposed over services in the manner how they make those changes. This implies that associations can restrict the order in which services are executed. They may be specified using a procedural specification (e.g. a workflow) or in a declarative way (e.g. condition-action rules).

Apart from business artifacts, businesses may also need to store data whose evolution does not result in relevant states from the point of view of the business. We will refer to this data as *objects*. Figure 2.1 shows the different BALSAs dimensions and how they relate to each other. As it can be seen, artifacts have a lifecycle, which is made up of several states, and the associations show how services can only be executed when the artifact is in a certain state.

If we think of a traditional, process-centric diagram, such as a flowchart, usually there is only one of the BALSAs dimensions represented there: the flow between the tasks in the process, called associations in the framework. In most cases, as the data has not been defined, neither are the tasks, and therefore the diagram does not provide information on the meaning or the semantics of the tasks. Consequently, we cannot consider that the *service* dimension is really represented.

We use the BALSAs framework as a basis in the remainder of this work for two main reasons: first of all, it defines what elements make up an artifact-

centric business process model. Therefore, by representing all these dimensions, we will ensure that we have an artifact-centric model. Secondly, it can be used to establish a basis to compare the different approaches for artifact-centric business process modeling, which is what we do in the following section.

2.2 State of the Art

After presenting the BALSAs framework, this section analyzes different alternatives to represent business process models. Although we are mainly interested in the artifact-centric approach, we also examine process-centric approaches, as they could be used to specify, if not all, at least one of the dimensions of artifact-centric process models. We also devote a small subsection to approaches which try to reconcile process-centric and artifact-centric approaches by establishing mappings and defining algorithms to change from one to the other.

2.2.1 Process-centric Approaches

There are several languages available to represent business process models following a traditional, or process-centric, approach. One of the most well-known is probably BPMN (Business Process Modeling Notation); however, there are others such as UML activity diagrams, Workflow nets or YAWL (Yet Another Workflow Language) [133].

Although some of these languages have the ability to represent the data needed in the flow, their focus is on the sequencing of the tasks that are carried out in the process. DFDs (data-flow diagrams) would be one example of this. While placing high importance on the data, the focus is on how these data move in the process, from one task to next, and little importance is given to their details or on the precise meaning of the tasks [134].

Another well-known language is BPEL (Business Process Execution Language). However, it is meant to be a web-service composition language following XML notation, and our focus is on defining processes at a high level of abstraction [133].

Apart from the languages themselves, there has been some process-centric research that takes data into consideration. For instance, [127] represents the associations between services in a WFD-net (WorkFlow nets annotated with Data). The tasks are annotated with the data that is created, read or written by the task.

Similarly, [89] uses WSM nets which represent both the control flow and the data flow, although the data flow is limited to read and write dependencies between activities and data. [85] represents associations in an operational model, which shows tasks (or services) as nodes connected using arrows or edges. The operational model also shows the transfer of artifacts between tasks by indicating them over the edges. However, details of artifacts are not shown.

A more complex representation can be found on [118]: they combine the use of ontologies with colored Petri nets (CPN). Then these Petri nets include annotations which refer to the ontology data in order to be able to “adapt” the Petri nets to the context.

To sum up, the majority of process-centric approaches focus on the sequencing of the tasks in the diagram; therefore, their effort is on representing the associations between the services, according to the BALS dimensions. There are some works that consider the data, however, this representation is limited to read and write dependencies and/or the data flow. In consequence, it lacks the richness of an artifact-centric representation.

2.2.2 Bridging the Gap between Process-centric and Artifact-centric Specifications

There are some works, such as [76, 79, 108], which attempt to bridge the gap between artifact-centric and process-centric specifications. Their final goal is to derive an artifact-centric specification from a process-centric one [79, 108] or vice versa [76]. We will focus on their representation of artifact-centric processes.

All these works represent lifecycles using variants of state machines, and associations are represented in different types of workflow diagrams. In particular, [79] uses UML state machine and UML activity diagrams, respectively. [108] uses the FlowConnect language, which merges concepts from UML state machine and sequence diagrams, to represent the lifecycle. [76] represents the lifecycle and associations in a diagram which resembles a combination of a state machine and activity diagram, as it shows both the potential states of the artifact and the tasks that may be applied to it.

In contrast, representation of artifacts (or the data) varies from one approach to the other. For instance, [76] represents artifacts in very simple diagrams showing only the dependencies between them. [108] uses a UML-like class diagram showing the relationships between the artifacts/objects. On

the other hand, [79] has no specific representation for them, other than the relationships between artifacts which may be derived from the lifecycle.

The main drawback of these approaches is that they do not specify the services/tasks in any way. Although in their context it is reasonable not to do so, as process-centric specifications do not model in detail with the data nor the meaning of the tasks, the specification of the tasks or services is a key element in artifact-centric specifications and in our approach, as we shall see.

2.2.3 Artifact-centric Approaches

After giving an overview of process-centric approaches, we will deal in more detail with works that specify business processes from an artifact-centric perspective. To facilitate the analysis, this subsection is structured according to the dimensions of the BALSAs framework for easier readability and comparison. Despite the fact that not all proposals use the BALSAs framework as a basis, in most cases it is easy to establish a correspondence between the different components of the models and a dimension in the framework.

A table summarizing our analysis can be found on page 31.

Business Artifacts Business artifacts can be represented in several ways. Many authors opt for a database schema [9, 11, 15, 31, 92, 37], while others consider artifacts as a set of attributes or variables [19, 54, 94]. Another alternative is to add an ontology represented by means of description logics on top of a relational database [27]. Although some of these alternatives describe the artifacts in a formal way, none of them represent the artifacts in a graphical way. This has some disadvantages: the models are more difficult to understand, e.g. it is more difficult to see how the artifacts relate to one another and to other objects.

There are also many works that represent artifacts in a graphical and formal or semi-formal way. For instance, [65, 39, 66] represent the business artifact and its lifecycle in one model, GSM, that includes the artifact's attributes. However, the relationships between artifacts are not made explicit. On the other hand, [87] represents artifacts as state machine diagrams defined by Petri nets, but does not give details on how the attributes of an artifact are represented. Closer to a UML class diagram is the Entity-Relationship model used in [20]. [52] uses a UML class diagram. Both the ER diagram and the UML class diagram are graphical and formal (or semi-formal) alternatives.

Finally, [78, 74] define the PHILharmonicFlows framework, which uses a diagram that falls in-between a UML diagram and a database schema representation. It is a semi-formal representation.

Lifecycles The lifecycle of a business artifact may be implicitly represented by using dynamic constraints in logic [9] or the tasks (or actions in the terminology of the papers) that make changes to the artifacts [15, 31, 11, 27].

In this context, however, we are interested in approaches that represent the lifecycles explicitly. In many cases, such as [20], they are based on state machine diagrams, as they show very clearly the states in the evolution of the artifact and how each state is reached and under which conditions. [92] derives the artifact's lifecycle in a state machine diagram from a BPMN model annotated with data.

The GSM approach represents the stages in the evolution of an artifact, the guard conditions, and the milestones in a graphical way. A milestone is a condition that, once it is fulfilled, it closes a state. In contrast to traditional state machine diagrams, the sequencing of stages is determined by the guard conditions and not by edges connecting the states, making it less straightforward. However, it is possible to use edges as a *macro*. Another difference is that in GSM various stages can be open, or active, simultaneously. GSM was first defined in [65] and further studied and formalized in [39, 66].

Another alternative to represent lifecycles is to use variants of Petri nets [52, 87, 75]. These representations are both graphical and formal. [78, 74], within the PHILharmonicsFlows framework, use a micro process to represent the evolution of an artifact and its states, which results in a graphical representation similar to GSM, without its strong formality.

Finally, some works opt for using a variable to store the artifact's state [37, 19]. Although it is an explicit representation, it only stores the current state of the artifact, instead of showing how it will evolve from one stage to the next. Therefore, it is a poorer form of representation in contrast to state machine diagrams, variants of Petri nets or GSM.

Services Services are also referred to as tasks or actions in the literature. In general, they are described by using pre and postconditions (also called effects). Different variants of logic are used in [9, 11, 37, 19, 54, 15, 27] for this purpose. [31, 39] omit the preconditions. The use of logic implies that the definition of services is precise, formal and unambiguous, but it is hardly understandable by the people involved in the business process.

Conversely, [20] uses natural language to specify pre and postconditions. In contrast to logic, natural language is easy to understand, but it is an informal description of services: this implies that the service definition may be ambiguous and error-prone.

Finally, [92] expresses the preconditions and postconditions of services by means of data objects associated to the services. These data objects are annotated with additional information such as what is read or written. [78, 74] define “micro steps” in the stages of their model which correspond to attributes that are modified. None of these approaches are as powerful as using logic nor the OCL language.

Associations In general, the different ways of representing associations can be classified on whether they represent them graphically or not. Many non-graphical alternatives are based on variants of condition-action rules. These alternatives have one main disadvantage over graphical ones: in order to know the order in which the tasks can execute, it will be necessary to carefully examine the rules. In contrast, graphical alternatives are easier to understand at a glance, although they are less flexible.

For instance, [9, 11, 31, 27] use a set of condition-action rules defined in logic. [19] calls them business rules. Similarly, [39, 65, 66] define guards in the GSM model following an event-condition-action style. In [37, 54, 15], preconditions determine the execution of the actions; as such, they act as associations. As they are defined in logic, they are formal and unambiguous.

Likewise, [20] uses event-condition-action rules, but they are defined in natural language. Using natural language makes them easier to understand than those defined in logic, but they have a severe drawback: they are not formal and because of this they may have ambiguities and errors.

Alternatively, [52] uses *channels* to define the connections between *proclets*. A procllet is a labeled Petri net with ports that describes the internal lifecycle of an artifact. On the other hand, DecSerFlow allows specifying restrictions on the sequencing of tasks, and it is used in [75]. It is a language grounded on temporal logic but also includes a graphical representation.

When it comes to graphical representations, [78, 74] use micro and macro processes to represent the associations between the services. [92] uses a BPMN diagram to represent the associations between the tasks. In this sense, it is very similar to our proposal to use UML activity diagrams. All these approaches are graphical and formal.

In contrast, [94, 17] opt for a graphical representation using flowcharts and, because of this, the resulting models can be easily understood. However, they do not use any particular language to define the flow and they do not define the semantics of the flowchart.

2.2.4 Summary & Conclusions

We have already explained the limitations of process-centric approaches, which focus almost exclusively on the association dimension of the BALSAs framework. In some instances they consider the data, but they never go into the details of how the data is modified.

In addition, the approaches that bridge the gap between artifact-centric and process-centric specifications lack, in some cases, a detailed definition of the data, and, in all of the analyzed works, there is no specification of the services or tasks. The service dimension is a key element in artifact-centric specifications and, considering the goals of this thesis (e.g. being able to reason with the artifact-centric business process models), it should be an indispensable component in our approach.

Table 2.1 shows a summary of the artifact-centric approaches. From the table, we can gather that many of the approaches do not represent all the dimensions in the framework. Those that do, will each have its own advantages and disadvantages. In particular, we will focus on the Philharmonic Flows [78, 74], the GSM modeling notation [65, 39, 66] and the proposals in [19, 20, 37, 92].

The approach in [20] uses several different models based on different languages to represent the dimensions in the BALSAs framework. Its main drawback is that it uses condition-action rules and actions defined in natural language, to represent the associations and the services, respectively. Therefore, the final model lacks formality and may be ambiguous.

As we have seen, [92] represents artifact-centric business process models using a BPMN model for the associations and the services, and a database schema for the artifacts. Lifecycles are inferred from the annotations in the BPMN model. Notice that the dimensions are not defined using the same language, and the specification of the services is limited to a graphical notation showing creation, updates and deletions over artifacts and relationships. This is not as powerful as using logic or a formal language to describe the tasks or services, although it is more intuitive due to the use of a graphical representation.

		Approach	Graphical?	Formal?
Artifacts		[9, 31, 15, 11]		
	DB Schemas	[92, 37, 19]		✓
	Attributes	[54, 94]		(✓)
	Ontology	[27]		✓
	ER Model	[20]	✓	✓
	UML Class Diag.	[52]	✓	✓
	Data diagr. (P.F.)	[78, 74]	✓	✓
	GSM's attributes	[39, 65, 66]		✓
	Petri-Nets	[87]	✓	✓
Lifecycle				
	State Machine	[20, 92]	✓	✓
	Variants of Petri-Nets	[75, 52, 87]	✓	✓
	GSM	[39, 65, 66]	✓	✓
	Micro proc. (P.F.)	[78, 74]	✓	✓
	Variable	[37, 19]		
Serv.		[9, 37, 19, 54, 15]		
	Pre / Post. in Logic	[27, 11, 31, 39]		✓
	Natural Language	[20]		
	Micro steps (P.F.)	[78, 74]	✓	✓
	Data Objects	[92]	✓	(✓)
Associations	Business/CA Rul. - Logic	[19, 9, 11, 31, 27]		✓
	Preconditions	[37, 54, 15]		✓
	Guards (GSM)	[39, 65, 66]		✓
	ECA Rul. - Nat. L.	[20]		
	DecSerFlow	[75]	✓	✓
	Channels (Proplets)	[52]	✓	✓
	Mic./mac. proc. (P.F.)	[78, 74]	✓	✓
	BPMN	[92]	✓	✓
	Flowcharts	[94, 17]	✓	

Table 2.1: Overview of alternative representations of artifact-centric process models. P.F stands for *PHILharmonicFlows*.

On the other hand, [37] defines artifacts in a database, and their lifecycles are stored in a status variable. The services have preconditions and postconditions defined in logic, and the services' preconditions actually act as the associations, determining when a service may take place.

Similarly, [19] represents artifacts using a tuple of attributes which includes identifiers and a status variable to represent the artifact's lifecycle. Services have a precondition and a set of effects represented in logic, but in this case business rules determine when a certain service may execute.

Both approaches in [37, 19] the advantage of being formal, but at the same time they are not practical from the point of view of the business, due their lack of a graphical, more understandable notation.

The Philharmonic Flows framework [78, 74] has been created with the purpose of defining artifact-centric business process models. The authors create their own models for each of the dimensions, and include other perspectives such as access control and automatic generation of forms. However, it is not based in a well-known language.

Similarly to Philharmonic flows, the GSM notation [65, 39, 66] does not rely on existing specific models. It is grounded on a graphical representation of the stages in an artifact's life, which includes the conditions which determine the opening and closing of a stage. However, its great potential for representing complex scenarios is also one of drawbacks: the presence of multiple guards and milestones which may open and close one or several stages in the diagram can lead to several difficulties to read and even create the diagrams. Moreover, the definition of the artifacts is based on a set of attributes, and therefore it is difficult to see the relationship between the various artifacts involved in the process.

In summary, all the examined approaches have their advantages and disadvantages. In particular, we find that none of the examined works have *all* of the following characteristics:

- It uses a well-known, formal language, to represent all the dimensions in the BALSAs framework.
- This language can be understood by business modelers and developers.
- It bridges the gap between artifact and process-centric approaches.
- There are available tools to model the business process following the approach.

Chapter 3

Artifact-centric Business Process Modeling in UML

According to [64], one of challenges in the artifact-centric business process modeling world is finding the most appropriate way to model business processes from an artifact-centric perspective. In the previous chapter we have seen different alternatives for artifact-centric business process modeling. Although they all have their purpose, they do not completely cover our needs.

For this reason, in this chapter we detail our proposal for modeling artifact-centric business process models using a combination of UML and OCL [48, 47], with the BALSAs framework as a basis.

Both UML and OCL are ISO standard languages [67, 68] and integrate with each other naturally. UML is traditionally used to represent the specification of systems and it provides models to represent both the static and the dynamic characteristics of the system.

In particular, we propose to use the UML class diagram, the UML state machine diagram, and the UML activity diagram to represent, respectively, the business artifacts, their lifecycles, and the associations between the services or tasks. To show the details of the services, we use OCL operation contracts.

To do so, we follow, in a sense, a top-down approach. That is, we define the process models from scratch in order to implement them later. However, an alternative way would be to follow a bottom-up approach: for already running processes, we could derive the artifact-centric process models from the log traces as done in [102, 101].

Moreover, we would like to emphasize the fact that our approach focuses on

specifying the process from a high-level/analysis perspective: we are interested in *what* the process does, not on *how* it does it [81]. This means that, unlike other approaches such as [78], we completely abstract away from the presentation of the data to the user or the process's implementation.

This chapter is structured in the following way. The first section introduces the different UML/OCL models and their use within the BALSAs framework and the second section formalizes the models. The third section further illustrates the proposal by showing its application to an example with more than one artifact involved in the business process, and the fourth section briefly compares our approach to software engineering methodologies. We end the chapter with some conclusions.

3.1 The *BAUML* Framework

As we have just mentioned, the modeling approach we propose is based on representing the BALSAs dimensions using UML and OCL: UML class diagrams for business artifacts; UML state machine diagrams for lifecycles; UML activity diagrams for associations, and OCL operation contracts for services. We call our approach *BAUML* (BALSAs UML, for short).

Figure 3.1 shows the dimensions in the BALSAs framework and their representation in our approach. Roughly, our methodology behaves as follows. Business artifacts correspond to some of the classes in the class diagram. For each artifact, a state machine diagram is defined stating its lifecycle. Then, each transition of the state machine diagram is further specified by means of an activity diagram determining the associations between the services or tasks of the artifact. Finally, the behavior of the atomic activities or tasks from each activity diagram is defined through an operation contract.

The remainder of this section presents in more detail our methodology for artifact-centric business process modeling using the *BAUML* approach. We will illustrate it by means of an example based on a city bicycle rental system, such as Bicing in Barcelona. Bicing users may take a bicycle, anchored in one of the many stations throughout the city, and return it to another station after the user reaches his destination. To keep it simple, we will assume that if a user returns a bicycle to a station immediately after it has been picked up, then the bicycle is not in good shape and it needs to be checked and repaired.

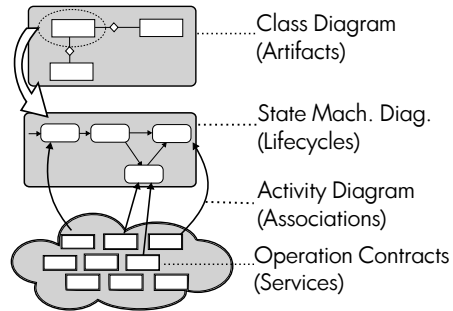


Figure 3.1: Representation of the BALS dimensions in our approach, adapted from [64].

3.1.1 Business Artifacts as a Class Diagram

Business artifacts represent the relevant data for the business. A business artifact has an identity, which makes it distinguishable from any other artifact, and can be tracked as it progresses through the workflow of the business process execution. It will also have a set of attributes to store the data needed for the execution. Business artifacts may be related to other business artifacts and objects.

We make the following distinction between *artifact* (or business artifact) and *object*. An *artifact* is an element whose evolution results in relevant states from the point of view of the business. These states are made explicit and appear in the lifecycle of the artifact. In contrast, an *object* may be created, deleted and updated by the process but these changes do not correspond to relevant, explicit states from the point of view of the business.

In order to represent the artifacts, the objects and the relationships between them we opt for the UML class diagram, which allows us to represent them graphically. Each artifact and object will correspond to a class. To distinguish them clearly from objects, artifacts will have stereotype `«artifact»`.

Each artifact and its states will be part of a hierarchy. The artifact will be the top class in the hierarchy, and the leaves will correspond to the states. They are dynamic subclasses, so that the artifact can change its type from one subclass to another as it evolves. They must fulfill the disjointness constraint (since we consider that an artifact cannot be in two states at the same time), but they can fulfill the completeness constraint (i.e. the artifact must have one of its subtypes) or not. If the artifact has a multi-level hierarchy, these rules

apply to all the levels. See for instance Figure 3.4.

Although the use of these dynamic classes may be controversial, because of the implementation problems when changing an object from one class to another, we are modeling the business process from an analysis perspective, focusing on *what* the process does and not on the *how* (i.e. the implementation details).

The advantage of using a hierarchy of subclasses to represent the potential states of an artifact is that it is possible to represent the attributes and relationships that are needed in each of the possible states while keeping the artifact's original identifier and the relationships that are common to all states (or several substates).

Apart from classes (which includes artifacts and objects) and relationships, the UML class diagram also allows to represent integrity constraints in a graphical way, e.g. the cardinalities in the relationships. However, those constraints that cannot be represented graphically should be written in OCL to ensure their formality. However, they could also be specified using natural language for easier readability¹.

The class diagram for our Bicing example can be seen in Figure 3.2. In this case we have only one artifact, *Bicycle*, marked with the corresponding stereotype. Apart from the business artifact, there are several objects, such as *AnchorPoint* (identified by *number*) and *User* (identified by *id*). We have kept things simple and the classes have few attributes.

The business artifact *Bicycle* has three subclasses: *Available*, *InUse* and *Unusable*², which keep track of the different stages of the *Bicycle* while containing information which is relevant only for that particular stage. For instance, when a bicycle is *InUse*, it is linked to a certain *User* and has an attribute that stores when it is expected to be returned. As shown in Figure 3.2, these subclasses also require restrictions to ensure that the dates are coherent.

3.1.2 Lifecycles as State Machine Diagrams

The lifecycle of a business artifact states the relevant stages in the possible evolution of the artifact, from inception to final disposal and archiving. As artifacts cannot evolve from one state to another randomly, the state machine diagram shows how the transitions from one state to the next are triggered.

¹Note that if natural language is used then the formality is lost and the reasoning explained in later chapters cannot be applied.

²These subclasses should be called *AvailableBicycle*, *InUseBicycle*, etc. We use instead shortened names for our convenience.

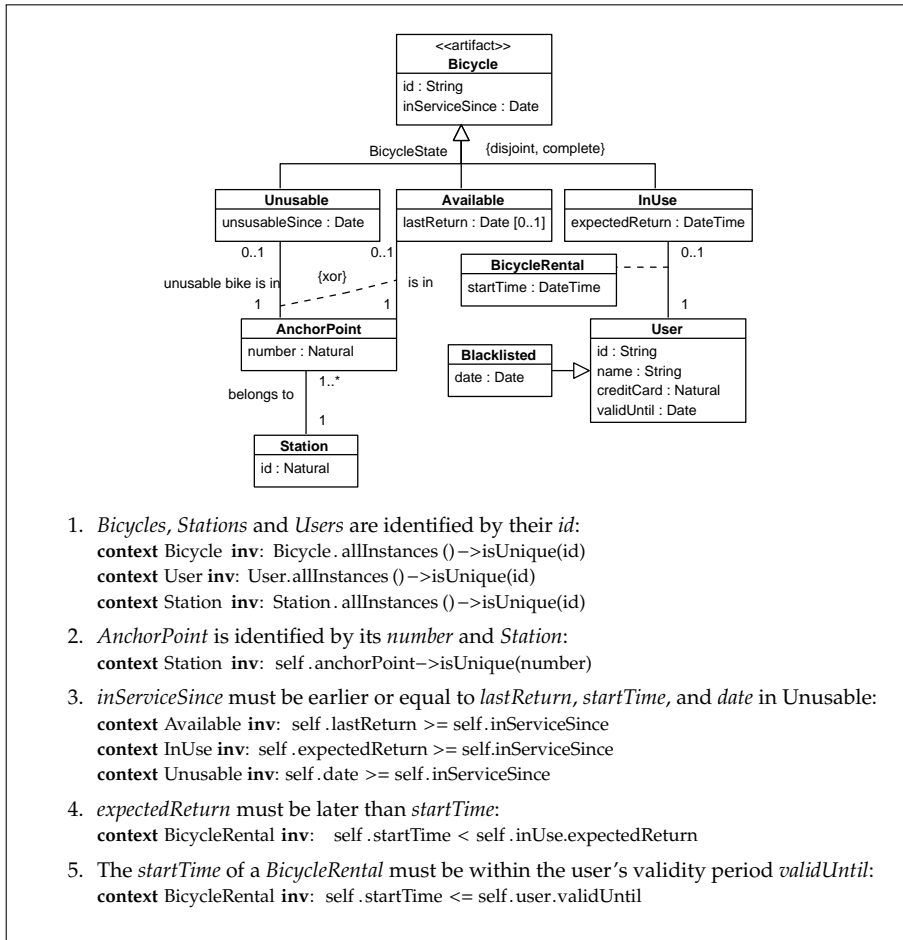
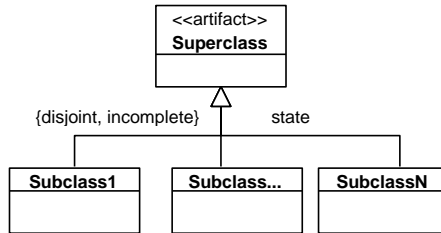
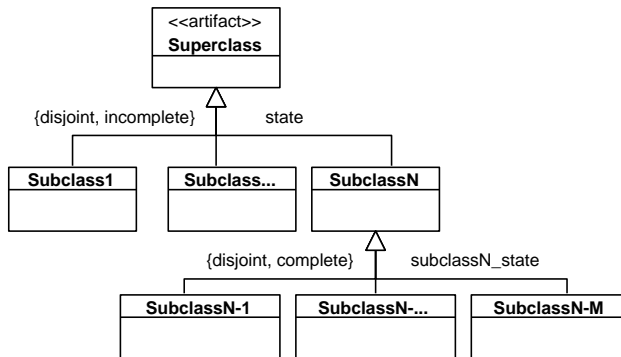


Figure 3.2: Class diagram of Bicing with the corresponding integrity constraints.

Figure 3.3: An example of *incomplete* hierarchy.Figure 3.4: An example of *incomplete*, multi-level hierarchy.

This state machine diagram will have a set of states, a set of events, a set of effects and a set of transitions between pairs of states.

Given an artifact, the states in the state machine diagram will correspond to its subclasses in the class diagram if the hierarchy is complete. If it is incomplete, then the state machine diagram will have another state for the superclass. In this context, the state that corresponds to the superclass will represent an instance of an artifact that does not have any of the subtypes. These rules apply to any multi-level hierarchy in the artifact.

For instance, if we have the hierarchy shown in Figure 3.3, there would be the following states in the corresponding state machine diagram: Superclass, Subclass1, Subclass..., SubclassN.

On the other hand, if we had a multilevel hierarchy such as the one shown in Figure 3.4, the states would be the following: Superclass, Subclass1,

Subclass..., SubclassN-1, SubclassN-..., SubclassN-M. Notice that now we have no corresponding state for subclass SubclassN: the reason is that SubclassN is the superclass of a *complete* hierarchy, and therefore a instance of type SubclassN will always have one of its subtypes: either SubclassN-1, SubclassN-... or SubclassN-M.

Note that we do not consider compound states nor orthogonality for the definition of the lifecycle dimension. We assume that an artifact can only be in one state at a certain point in time from the point of view of its evolution. This tallies with the way the class hierarchies for the artifacts have been defined.

Apart from the states that correspond to the subclasses (or superclass in some cases), we also have an initial and a final state.

The state machine diagram represents the transitions between states. Each transition will have a source state and a target state. Moreover, it may also have an OCL condition over the class diagram, an event and a tag representing the result from the execution of the event. The initial transitions are those which have the initial state as source and result in the creation of a new artifact instance. We differentiate between three types of transitions (the elements inside parenthesis are optional):

1. ([OCL]) ExternalEvent ([tag])
2. ([OCL]) TimeEvent (/ Effect)
3. [OCL] (/ Effect)

The first transition type occurs when ExternalEvent takes place and the OCL condition (if there is any) is true. If there is a tag, then the result of the execution of ExternalEvent must coincide with tag for the transition to take place. The second transition type will be triggered when there is a TimeEvent and the OCL condition is true. If there is an Effect, the changes specified by it will also be made. Finally, the last transition type is similar to the second excepting the occurrence of a time event. These transition types cover the types of transitions allowed in the UML specification that are significant at the specification level, as explained in [96].

An ExternalEvent will have as input parameters the artifacts in whose transitions it appears or the identifiers of those artifacts. The details of the execution of these events and their respective tags (if any) will be defined in an activity diagram.

A tag is a user-defined expression which provides information about the execution of the activity diagram. In most cases, there are only two tags

needed, *success* and *fail*, which will denote the successful or unsuccessful execution of the activity diagram representing the external event³. However, more tags could be added if necessary, e.g. *cancel*.

Effects correspond to atomic tasks that have as input parameters the artifacts involved in the transition. OCL is an OCL expression which starts from *self* or *Class.allInstances()->...* where *Class* is any of the classes in the class diagram. A *TimeEvent* represents an occurrence of time. We distinguish between relative and absolute time expressions. An absolute expression has the form *at(time_expression)*; a relative expression has the form *after(time_expression)*.

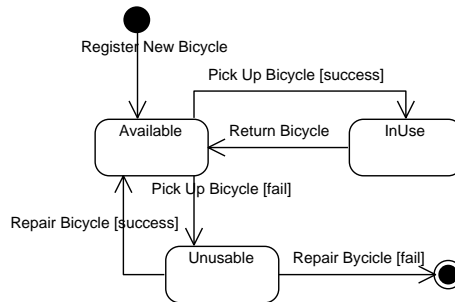
Notice that this state machine diagram does not follow exactly the UML standard described in [67]. This is due to the fact that it has tags, which we use to take into consideration the outcome of the event. In traditional UML state machine diagrams, events are atomic and there is no need for such conditions.

In addition, we also allow more than one outgoing transition from the initial node. This is useful when the artifact can be created in different ways. Alternatively, this situation could be represented using one outgoing transition from the initial node, leading to a state called *InitialState*. From this state, we could have the outgoing transitions that start from the initial node and leave the rest of the state machine diagram as it is. However, representing the lifecycles in this way does not contribute any relevant information and adds complexity to the final diagram.

Figure 3.5 shows the state machine diagram representing the evolution of the artifact *Bicycle*. When a bicycle is created (*Register New Bicycle*), it is in state *Available*, and ready to be picked up. If a user does pick up a bicycle (event *Pick Up Bicycle*), there are two possibilities: either everything goes smoothly and he takes it with him (tag *success*), or the bicycle is in bad shape and the pick-up fails (tag *fail*). In the first case, the bicycle is *InUse* while in the second case it is *Unusable*. While a bicycle is in state *InUse*, it will become *Available* again when the user returns it (*Return Bicycle*).

When a bicycle is *Unusable*, it needs to be repaired (*Repair Bicycle*) in order to become available again. There are two possible outcomes. If the bicycle is repaired successfully (tag *success*) it changes its state to *Available*. If, on the other hand, it is beyond repair (tag *fail*), the bicycle is destroyed and deleted from the system.

³We assume that in both cases the tasks in the activity diagram execute correctly (i.e. no constraints or preconditions are violated). The tag represents the outcome of this execution from the point of view of the business.

Figure 3.5: State diagram of *Bicycle*.

3.1.3 Associations as Activity Diagrams

As we have just mentioned, external events in a state machine diagram consist of a set of services or tasks, that is, they are not atomic. Consequently, for each external event we need to specify the services it consists of and the associations between them. Associations in the BALSAs framework establish the conditions under which services may execute.

For every `ExternalEvent` in a state machine diagram, there will exist exactly one activity diagram. An activity diagram will have a set of nodes and a set of transitions between those nodes. More specifically, the activity diagram will have exactly one initial node and one or several final nodes. Transitions will determine the change from one node to the next. Apart from a source node and a target node, transitions may also have a **guard condition** and a **tag**. The tag will determine the result of the execution of the activity diagram, and the triggering of transitions in the state machine diagram depends on this outcome.

We distinguish between the following node types:

- **Initial Node:** Point where the activity diagram begins
- **Final Node:** Point where the flow of the activity diagram ends.
- **Gateway Node:** Gateway nodes are used to control the execution flow. We distinguish between *decision nodes*, *merge nodes*, *fork nodes* and *join nodes*.

Decision and merge nodes deal with several paths which are mutually exclusive: the former splits one path into several, the latter joins them.

In contrast, fork and join nodes deal with parallel (i.e. simultaneously active) paths. The first splits the flow into several, the second joins them.

- **Activity:** An activity represents work that is carried out. We differentiate three types of activities:
 - A *task* corresponds to a unit of work with an associated operation contract. The operation contract will have a precondition, stating the conditions that must be true for the task to execute, and a postcondition, indicating the state of the system after the task's execution. Both are formalized using OCL queries over the class diagram.
 - *Material actions* correspond to physical work which is carried out in the process but that does not alter the system.
 - Finally, a *subprocess* represents a “call” to another activity diagram, and as such may include several tasks and material actions.

We assume the following: decision nodes and fork nodes have one incoming flow and more than one outgoing flow; merge nodes and join nodes have several incoming flows and exactly one outgoing flow; activities have one incoming flow and one outgoing flow; initial flows have no incoming and one outgoing flow; and final nodes may have several incoming flows but no outgoing flow.

Guard conditions are only allowed over transitions which have a decision node as their source. The guard condition may refer to either:

- The result of the previous task
- An OCL condition over the class diagram
- A user-made decision

On the other hand, tags are only allowed over those transitions that have as target a final node, as they represent the outcome of the activity diagram and connect it to the state machine diagram.

During the execution of the activity diagram we assume that the constraints established by the class diagram may be violated. However, at the end of the execution they must be fulfilled, otherwise the transition does not take place

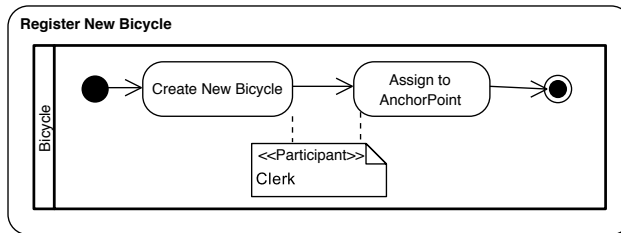


Figure 3.6: Activity diagram for *Register New Bicycle*

and the changes made during the execution of the activity diagram are “rolled back”⁴.

Finally, activity diagrams may also represent the main artifact or object involved in each of the tasks and its participants (i.e. the role of the person who carries out a particular activity) using swimlanes and notes, respectively. Although this information is not strictly necessary to represent artifact-centric business process models, in some instances it may be relevant to have it.

More specifically, swimlanes will provide additional information by indicating the main business artifact or object that is involved in each of the tasks or material actions. As tasks and material actions use the same notation in UML activity diagrams, material actions are represented using stereotype «material» in the swimlane or in the task itself. Tasks, on the other hand, deal with the representation of the real-life artifact or object. They are shown in the activity diagram as tasks without stereotypes in the swimlane. This distinction is important, because we will only be able to specify services that deal with informational resources (i.e. what we call *tasks*) and not material actions. Moreover, we will use notes with stereotype «Participant» to show the role/s of the people who carry out a particular task.

As we have mentioned, each external event in the state machine diagram has its corresponding activity diagram. Figure 3.6 shows the activity diagram of *Register New Bicycle*. First of all, the clerk provides the information of the new bicycle and, after this, he assigns it to an anchor point. The main artifact

⁴This should not be confused with the fact that the activity diagram may end in tag *fail*. The tag merely means that, following that path, the event has not fulfilled its goal, but not that the changes made are not valid. However, if the activity diagram ends in tag *fail* and the integrity constraints are violated, then every change made by it is “rolled back”, just like it would if there were no tags or the tag was success.

involved in both services is the *Bicycle*. In this particular case, both tasks are atomic and deal with information and not material resources.

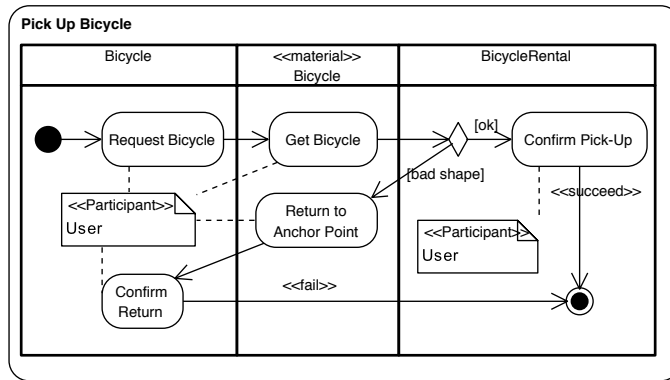


Figure 3.7: Activity diagram for *Pick Up Bicycle*

Figure 3.7 represents the activity diagram of *Pick Up Bicycle*. The user first requests a bicycle to the system and then he physically picks it up from its anchor point. If the bicycle is not in good shape, he returns it to an anchor point and then he confirms the return. Notice that in this case the activity diagram ends in failure. On the other hand, if the bicycle is usable, he takes it with him and confirms the return. Then the activity diagram ends successfully. It is important to make this distinction between success and failure as depending on the result of the activity diagram, the bicycle will change to state *InUse* or *Unusable*, as shown in Figure 3.5.

In this particular diagram, there are two tasks or services that deal with material resources: *Get Bicycle* and *Return to Anchor Point*, which correspond to physically getting the bicycle from its anchor point and placing it on the anchor point, respectively.

3.1.4 Tasks (Services) as Operation Contracts

In the context of the BALSAs framework, a service is a unit of work meaningful to the whole business process. Services create, update and delete business artifacts. In turn, this may make artifacts evolve to a new stage meaningful from the business perspective. In our approach, services correspond to the tasks in the activity diagrams and the effects in the state machine diagram.

As we have mentioned, each of the tasks in the activity diagrams will have an associated operation contract. The same applies to effects in the state machine diagrams. The contract will have a set of input parameters, a precondition, a postcondition and may have an output parameter. The input and output parameters may be classes or simple types (e.g. strings, integers). If several tasks belong to the same activity diagram and their input parameters have the same names, we assume that their value does not change from one task to the next.

The task can only be executed when the precondition is met, and the postcondition specifies the state of the system after the execution of the operation. We also assume a strict interpretation of operation contracts to avoid redundancies [104]. We deal with the frame problem by assuming that those classes that do not appear in the postcondition keep their state from before its execution. Below we show the operation contracts for the tasks in Figure 3.7.

Listing 3.1: Code for task *RequestBicycle*

```
operation requestBicycle(b: Bicycle)
pre: -
post: b.ocIsTypeOf(InUse) and not b.ocIsTypeOf(Available) and
      b.ocAsType(InUse).expectedReturn = now() + hour(3)
```

Task *Request Bicycle* (Listing 3.1) only changes the state of the given *Bicycle* from *Available* to *InUse*. Notice that the precondition does not check if the bicycle is indeed *Available*, as this is already guaranteed by the state machine diagram (see Figure 3.5).

Listing 3.2: Code for task *ConfirmPickUp*

```
operation confirmPickUp(b: Bicycle , u: User)
pre: -
post: BicycleRental.allInstances()->exists(x | x.ocIsNew() and x.user=u and
      x.inUse = b.ocAsType(InUse) and x.startTime = now())
```

Confirm Pick Up, in Listing 3.2, creates the *BicycleRental* and assigns the given *User* and *Bicycle* to it.

Listing 3.3: Code for task *ConfirmReturn*

```
operation confirmReturn(b: Bicycle , ap: AnchorPoint)
pre: -
post: not b.ocIsTypeOf(InUse) and b.ocIsTypeOf(Unusable) and
      b.ocAsType(Unusable).anchorPoint=ap and
      b.ocAsType(Unusable).unusableSince = today()
```

In contrast to *Confirm Pick Up*, *Confirm Return* (Listing 3.3) changes the bicycle state to *Unusable* and assigns it to an *AnchorPoint*. We do not check

if the anchor point is empty, as the cardinality and xor constraints control this. This task is executed when the user decides to return the bicycle before confirming the pick up, probably because the bicycle is in bad shape and cannot be used.

Notice that the bicycle given as input has the same name in all the operation contracts. We assume that it refers to the same bicycle.

3.1.5 A Note on the Models

This section has presented our framework for modeling artifact-centric business processes. We have used a set of models based on a combination of the UML and OCL languages. However, as long as the semantics of the diagrams are preserved, the dimensions in the BALSAs framework could be represented using other alternatives.

For instance, in the case of the business artifacts, the class diagram could be substituted by an Entity-Relationship (ER) [32] or an Object Role Modeling (ORM) [59, 2] diagram. Both diagrams also allow defining the artifacts, the objects and their relationships in a graphical way.

Although we use a variant of UML state machines, any other notation based on state machines would be useful to represent the lifecycles of the artifacts.

For activity diagrams, there are many different available notations (as long as they follow the semantics of our activity diagrams) such as BPMN [98] or DFDs [134]. BPMN is probably the language that is most used to represent business process models, and as such it offers a great variety of syntactic sugar for the basic node types described above. Data-Flow diagrams (DFD) are also another alternative, as they show the task and the inputs and outputs of data required and generated by them. Figure 3.8 shows the type of nodes we handle, with their corresponding representation in BPMN and UML.

Finally, for the tasks, alternatives could be using natural language, which is not formal and may have errors, or using logic, which is complicated to understand and use from the point of view of the business.

3.2 Formalization of the BAUML Framework

This section formalizes the BAUML framework presented in Section 3.1 and is structured according to the diagrams that we use for each dimension in the BALSAs framework. We will use it in the remainder of this thesis.

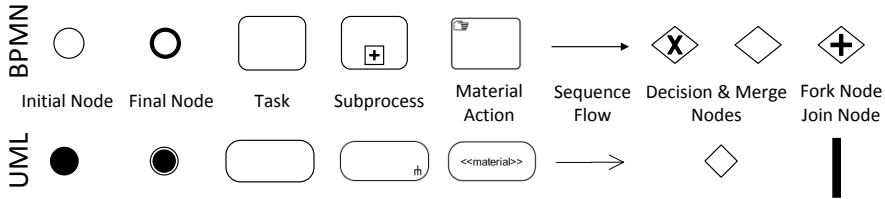


Figure 3.8: Nodes that we use in the activity diagram

3.2.1 Class Diagram and Integrity Constraints

\mathcal{M} is a UML class diagram, in which some classes represent (business) artifacts. Given two classes A and B , we say that A is a B , written $A \sqsubseteq_{\mathcal{M}} B$, if $A = B$ or A is a direct or indirect subclass of B in \mathcal{M} . Furthermore, given a class A and a (binary) association R in \mathcal{M} , we write $A =_{\mathcal{M}} \exists R$ ($A =_{\mathcal{M}} \exists R^-$ resp.) if A is the domain of R (image of R resp.) according to \mathcal{M} . We also denote by $R|_1$ and $R|_2$ the role names attached to the domain and image classes of R . We denote the set of artifacts in \mathcal{M} as $\text{ARTIFACTS}(\mathcal{M})$ and, when convenient, we use $\text{ARTIFACTS}(\mathcal{B})$ interchangeably. Each artifact is the top class of a hierarchy whose leaves are subclasses with a dynamic behavior (their instances change from one subclass to another). Each subclass represents a specific state in which an artifact instance can be at a certain moment in time. We denote by $\text{A-CLASSES}(\mathcal{M})$ ($\text{A-CLASSES}(\mathcal{B})$ resp.) the set of such subclasses, including the artifacts themselves. These subclasses must fulfill the disjointness constraints (i.e. they must have at most one of the subclasses type at a certain point in time.) Given a class $S \in \text{A-CLASSES}(\mathcal{M})$, we denote by ART_S the class S itself if S is an artifact, or the class A if A is an artifact and S is a possibly indirect subclass of A . Given an artifact $A \in \text{ARTIFACTS}(\mathcal{M})$, we denote by $\text{A-STATES}(A)$ the set of leaves in the hierarchy with top class A if the hierarchy is complete (i.e. every superclass must have one of the subtypes). If the hierarchy is incomplete, $\text{A-STATES}(A)$ will include the set of leaves and the superclass. We denote the classes in \mathcal{M} as $\text{CLASSES}(\mathcal{M})$, and the associations in \mathcal{M} as $\text{ASSOCIATIONS}(\mathcal{M})$. When convenient, we may refer to them as $\text{CLASSES}(\mathcal{B})$ and $\text{ASSOCIATIONS}(\mathcal{B})$.

Apart from the classes and associations, a class diagram will also have a set of graphical and textual integrity constraints. The latter will be represented in OCL. We denote both graphical and textual constraints as \mathcal{O} .

3.2.2 State Machine Diagrams

\mathcal{S} is a set of UML state machine diagrams, one per artifact in $\text{ARTIFACTS}(\mathcal{M})$. More formally, for each artifact $A \in \text{ARTIFACTS}(\mathcal{M})$, \mathcal{S} contains a state transition diagram $S_A = \langle V, v_o, v_f, E, X, T \rangle$, where V is a set of states, $v_o \in V$ is the initial state, $v_f \in V$ is the final state, E is a set of events (either *external* or *time* events), X is a set of effects, and $T \subseteq V \times \text{OCL}_{\mathcal{M}} \times E \times C \times X \times V$ is a set of transitions between pairs of states, where $\text{OCL}_{\mathcal{M}}$ is an OCL condition over \mathcal{M} that must be true in order for the transition to take place and C is a tag on the result of the execution of the event in E . Note that v_o cannot have any incoming transition, and v_f cannot have any outgoing transition.

The states $V' \subset V$ of S_A , such that $V' = V - \{v_o, v_f\}$, exactly mirror the classes in $\text{A-STATES}(A)$, so that S_A encodes the allowed event-driven transitions of an artifact instance of type A from the current state to a new subclass (i.e. a new artifact state). Moreover, the initial transitions starting from v_o always result in the creation of an instance of the artifact being specified by S_A .

We distinguish three different kinds of transitions, labeled as follows (elements inside parenthesis are optional):

- $([\text{OCL}_{\mathcal{M}}])$ ExternalEvent(a_1, \dots, a_n) ($[C]$), where a_1, \dots, a_n are the artifacts manipulated by ExternalEvent
- $([\text{OCL}_{\mathcal{M}}])$ TimeEvent ($/X$)
- $[\text{OCL}_{\mathcal{M}}]$ ($/X$)

In the first case, the transition will take place if $\text{OCL}_{\mathcal{M}}$ is true when the external event is received and the execution of the event results in tag C , if any (its possible values are `success` and `fail`). In the second case, it will take place if $\text{OCL}_{\mathcal{M}}$ is true when the time event occurs. This transition will also modify the contents of \mathcal{M} as stated by the effect X . The third case is similar, but the transition does not require the occurrence of any time event.

$\text{OCL}_{\mathcal{M}}$ is an OCL boolean expression over \mathcal{M} , which must begin with `self` or `Class.allInstances()->...`, where `Class` may be any $c \in \text{CLASSES}(\mathcal{M})$. A `TimeEvent` represents an instant of time defined by an expression. This expression may be relative with respect to another point in time or absolute. If it is relative it uses expression `after(time_expression)`; otherwise it uses `at(time_expression)`, as defined in [67]. `ExternalEvent(a1, ..., an)` must appear at least in a transition of the state machine diagram of each artifact a_i . Given a state machine diagram $S \in \mathcal{S}$, we denote the set of external events in S as $\text{EXTEVENTS}(S)$.

The execution of external events and the tags C resulting from this execution are driven by activity diagrams. Each effect X corresponds to an atomic task to be performed when making the transition, and whose parameters are exactly the artifacts involved in the transition.

Given an artifact $A \in \text{ARTIFACTS}(\mathcal{M})$, we denote by $\text{CONDITIONS}(A)$ the set of conditions appearing in the state transition diagram S_A , also considering all activity diagrams related to S_A . We then define $\text{CONDITIONS}(\mathcal{B}) = \bigcup_{A \in \text{ARTIFACTS}(\mathcal{M})} \text{CONDITIONS}(A)$.

3.2.3 Activity Diagrams

\mathcal{P} is a set of UML activity diagrams, such that for every state machine diagram $S = \langle V, v_o, v_f, E, X, T \rangle \in \mathcal{S}$, and for every event $\varepsilon \in \text{EXTEVENTS}(S)$ there exists exactly one activity diagram $P_\varepsilon \in \mathcal{P}$.

P_ε is a tuple $\langle N, n_o, n_f, F \rangle$, where N is a set of nodes, $n_o \in N$ is the initial node, $n_f \subset N$ is the set of final nodes and $F \subseteq N \times G \times C \times N$ is a set of transitions between pairs of nodes where C is a tag (success or fail) denoting the correct or incorrect execution of the transition, and G a guard condition.

There are four different types of nodes $n \in N$ in an activity diagram P_ε : initial nodes (denoted as $\text{INI}(P_\varepsilon)$), final nodes ($\text{FINAL}(P_\varepsilon)$), gateways ($\text{GATEWAYS}(P_\varepsilon)$) and activities ($\text{ACTIVITIES}(P_\varepsilon)$).

As we have seen, *initial* and *final* nodes indicate the points where the activity diagram flow begins and ends, respectively. *Gateways* are used to control the sequence flow. They may be either a *decision node*, a *merge node*, a *fork node* or a *join node*.

An *activity* may be a *subprocess*, an (atomic) *task* or a *material action*. We will denote the set of tasks of a certain activity diagram p as $\text{TASKS}(p)$. Each subprocess sp is defined by means of an additional activity diagram P_{sp} ; while each task is associated to an *operation contract*, which expresses a precondition on the executability of the task, and a postcondition describing its effect, both formalized in terms of OCL queries over \mathcal{M} . *Material actions* represent physical work that is done in the process but that does not change the system.

We make the following assumptions over each activity diagram P_ε : decision nodes and fork nodes have one incoming flow and more than one outgoing flow; merge nodes and join nodes have more than one incoming flow and exactly one outgoing flow; tasks have exactly one incoming and one outgoing flow; initial nodes have no incoming flow and exactly one outgoing flow; and final nodes have one or several incoming flows but no outgoing flow.

We only allow guard conditions over a transition $f = \langle n_s, g, c, n_t \rangle \in F$ if n_s is a decision node. Then, g may correspond to the result of task t_k , where $f' = \langle t_k, \emptyset, \emptyset, n_s \rangle \in F$, to an OCL condition over \mathcal{M} or to a label representing a user-made decision. Similarly, we only allow c over $f \in F$ such that $f = \langle n_s, g, c, n_t \rangle$ and $n_t \in \text{FINAL}(P_\varepsilon)$.

With a slight abuse of notation, given a state machine diagram $S \in \mathcal{S}$, we denote by $\mathcal{P}_S \subseteq \mathcal{P}$ the set of activity diagrams referring to all external events appearing in S .

As we have explained previously, during the execution of an activity diagram the constraints in \mathcal{O} may be violated, as we follow a strict interpretation of operation contracts [104]. However, these must be fulfilled at the end of the execution, otherwise the transition in the state machine diagram does not take place and all the changes made are “rolled back”.

3.2.4 Tasks

\mathcal{T} is a set of atomic tasks, each of which has an OCL operation contract. Its semantics is that the task can only be executed when the current information base satisfies its precondition, and that, once executed, the task brings the information base to a new state that satisfies its postcondition. If, during the execution of an activity diagram the precondition of one of the tasks is not met, then we assume that the corresponding transition does not take place and that no changes are made.

Given an artifact $A \in \mathcal{M}$, we denote by $\text{TASKS}(A)$ the set of tasks appearing in the state machine diagram S_A , also considering all activity diagrams related to S_A . We then define $\text{TASKS}(\mathcal{B}) = \bigcup_{A \in \text{ARTIFACTS}(\mathcal{M})} \text{TASKS}(A)$. Moreover, we assume that every task in $\text{TASKS}(A)$ that does not belong to the activity diagram of an initial transition has as input an instance of the artifact in S_A .

3.3 An Example with Two Artifacts

In the previous sections we have presented our framework (both intuitively and formally) and illustrated it by means of a simple example, based on a city bicycle rental system (Bicing) and which includes only one artifact. Our framework can also logically handle business processes with more than one artifact. In consequence, in this last section we expand the example to include another artifact: *User*, in order to further illustrate our approach. We now make the following assumptions:

- If a user takes a bicycle with him and does not return it within three days, we consider that the user has stolen or lost the bicycle. For this reason, the user is blacklisted and the bicycle is considered to be lost.
- A user can take more than one bicycle with him (three at most), so that families can use more bicycles without having to register their children as users.

The rest of this section presents how we would represent the example using the BAUML framework. Although the main focus of the section is on the example itself, we clarify some points regarding the interaction between artifacts for some transition types.

3.3.1 Class Diagram

Figure 3.9 shows the class diagram with the changes and additions mentioned above. In contrast with the previous example (see Figure 3.2) we now consider that bicycles may be lost by a user when he does not return them by a certain date. This is reflected in the diagram by the fact that *Bicycle* has a new subclass: *Lost*.

Apart from artifact *Bicycle*, we have another artifact: *User*. *User* has three different subclasses: *Active*, *Idle* and *Blacklisted*, and the hierarchy is *disjoint* and *complete*, stating that a *User* must have exactly one of its subtypes (or subclasses), just like in the case of *Bicycle*.

Because a *User* may have more than one bicycle rental, it may be the case that a *Blacklisted* user may still have rented bicycles, an *Active* user has at least one rental, and an *Idle* user has none.

3.3.2 State Machine Diagrams

As we have seen, there are two artifacts in the class diagram in Figure 3.9: *Bicycle* and *User*. Each of them will have its own state machine diagram, shown in Figures 3.10 and 3.11. Notice that each subclass of *Bicycle* and of *User* in the class diagram in Figure 3.9 has the corresponding state in their respective state machine diagram, as both hierarchies are *complete*.

The state machine diagram of *Bicycle*, shown in Figure 3.10 is very similar to the one we saw previously. The main difference is the fact that now it includes a new state: *Lost*. A *Bicycle* will change its state to *lost* if the bicycle is not returned within three days of the rental, as stated in condition *notReturned*. If, somehow, a lost bicycle is found, it then changes its state to *Unsuable*, as

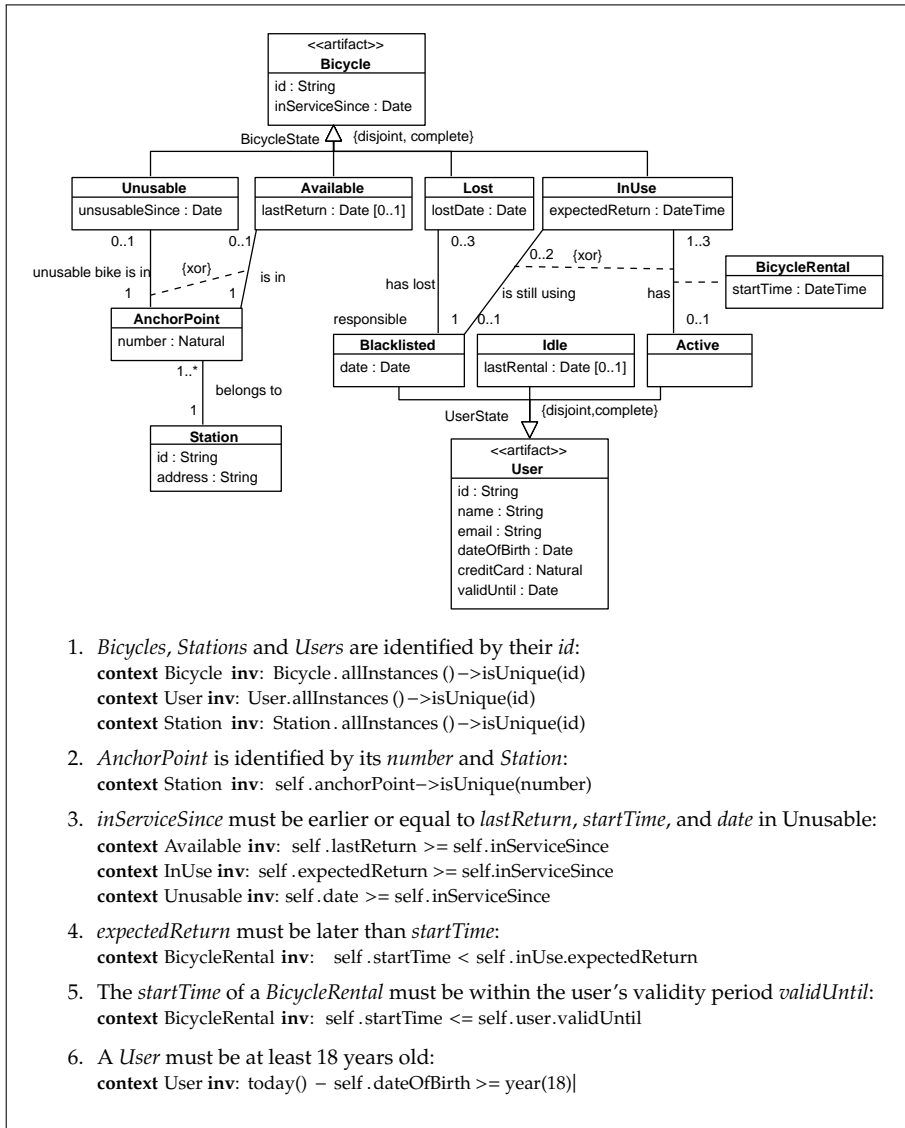


Figure 3.9: Class diagram and integrity constraints for the Bicing example with two artifacts.

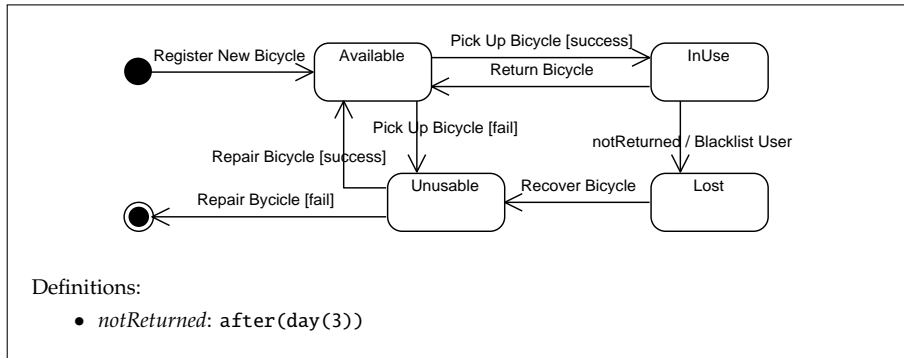


Figure 3.10: State machine diagram showing the evolution of the artifact *Bicycle*.

we assume that, after being lost, it will require a checkup and probably some repairs.

The evolution of a user (shown in Figure 3.11) is similar to that of a bicycle, and it shares many of its events. However, notice that in some cases there are conditions over the transitions. For instance, if an *Active* user returns a bicycle (event *Return Bicycle*), the final state will depend on whether there is only one bicycle left to return ($\#bicycles = 1$) or more ($\#bicycles > 1$).

The Interaction Between Artifacts

As we have seen in the class diagram in Figure 3.9, *User* and *Bicycle* are two artifacts that are directly related to one another. Their state machine diagrams have several events in common. We will now look at how they interact by means of these events.

Previously we distinguished between three types of transitions:

1. ([OCL]) ExternalEvent ([tag])
2. ([OCL]) TimeEvent (/ Effect)
3. [OCL] (/ Effect)

We will first focus on the transitions of the first type. In our example, there are three external events which appear in the state machine diagrams of both

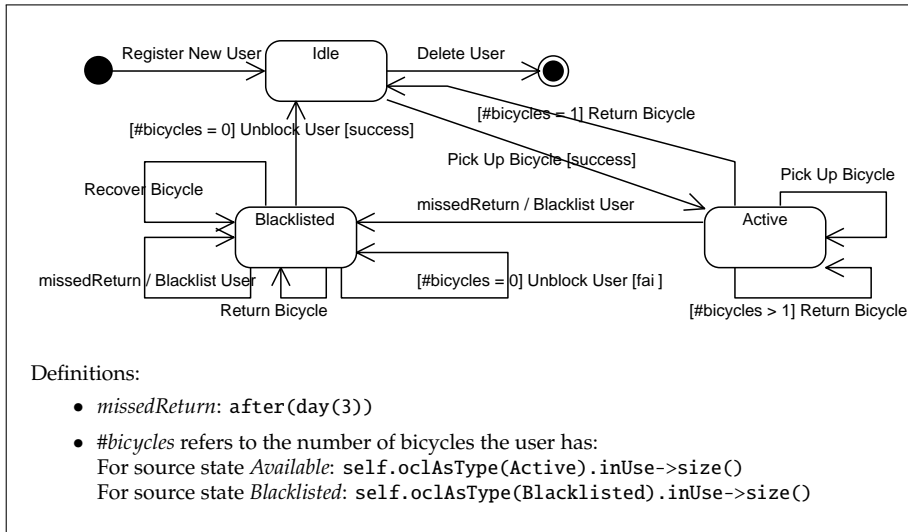


Figure 3.11: State machine diagram showing the evolution of the artifact *User*.

artifacts: *Pick Up Bicycle*, *Return Bicycle* and *Recover Bicycle*. External events are triggered by a user who wishes to carry out a task (or several tasks), and although we do not represent it explicitly in the state machine diagram, they have as input the artifacts involved in the external event. In the case of *Pick Up Bicycle*, it will require an *Available* bicycle and either an *Idle* or *Active* user, as shown in both diagrams in Figures 3.10 and 3.11. For this particular external event, there is no prior relationship between the user and the bicycle, as the goal of the event is to assign a bicycle to a user.

Return Bicycle also involves a user and a bicycle, so its implicit parameters are an *Active* or *Blacklisted* user and a bicycle that is *InUse* (see Figures 3.10 and 3.11). Unlike the previous external event, in this case the artifacts are related to each other; that is, it is not any user that returns the bicycle, it is the user who had rented it. For this reason, we have to ensure that the *User* and the *Bicycle* involved in the transition are related. As we shall see in the following sections, in this case we can obtain the user given the bicycle, and for this reason the only input parameter in the tasks will be the bicycle.

Recover Bicycle would work similarly to *Return Bicycle*, as the *User* and *Bicycle* involved in the transition must be related. Notice that one execution of

the external event triggers the appropriate transitions in both diagrams.

For transitions of types two and three, things are a bit more complex. They can only take place when certain conditions are met or when time goes by and, when this happens, an effect executes automatically and makes the appropriate changes to the system. Therefore, following strictly the semantics of transitions in state machine diagrams, these types of transition would be triggered independently for each artifact. However, there should be only one execution of the effect. For this reason, we will assume that given two artifacts with their respective state machine diagrams, if there are two transitions with exactly the same time event, OCL condition and effect, the execution of the effect should only be triggered once.

Bearing this in mind, there is one transition in Figures 3.10 and 3.11 that fulfills these criteria: `after(day(3)) / Blacklist User`. The goal of this transition is to blacklist a user if he does not return a bicycle within 3 days after taking it. In addition, it will also mark the bicycle as lost. Notice that it does not make sense to execute the effects of this transition twice for the same bicycle and user.

3.3.3 Activity Diagrams

This subsection will show some of the activity diagrams for the external events in Figures 3.10 and 3.11. The remaining activity diagrams can be found in Appendix A.2 on page 191.

Return Bicycle (see Figure 3.12) is very simple. First of all, the user places the bicycle in an anchor point, and afterwards the bicycle return is confirmed. The only task in the diagram is *Confirm Bicycle Return*.

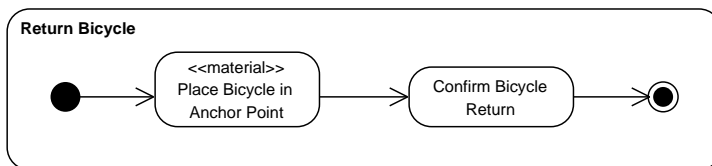


Figure 3.12: Activity diagram for *Return Bicycle*

Figure 3.13 shows the activity diagram for external event *Unblock User*. It has three different activities: the first activity is a material action and consists in revising the user's history. If it is successfully revised and the user is "forgiven", the blacklisted user must then pay a fine (it is also a material

action) and only, after this, the user's state is changed to *Idle*. On the other hand, if the user's history is deemed to be unforgivable, no changes are made and the user remains in state *Blacklisted*.

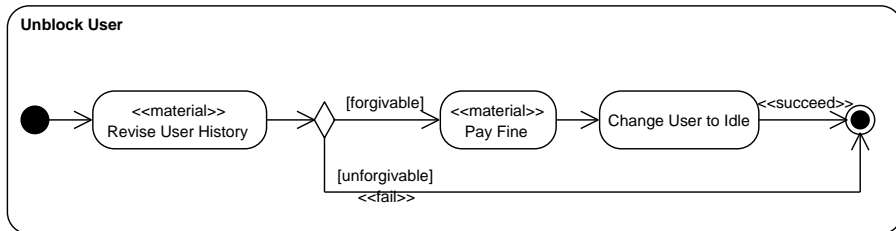


Figure 3.13: Activity diagram for *Unblock User*

3.3.4 Operation Contracts

Just like in the case of activity diagrams, many of the operation contracts do not change from the ones that we had in our first example. In this section we will show four different operation contracts which are part of various activity diagrams, except for the last one, which corresponds to the effect *Blacklist User*. The remaining operation contracts can be found in Appendix A.2.

The first contract specifies a task which is part of an external event exclusive to artifact *User*. The second one corresponds to an operation contract that changes from the one specified in the simpler version of the example in Subsection 3.1.4. The third contract corresponds to the only task in *Return Bicycle*, which is interesting because it involves both artifacts in our example, and one depends on the other in this context. Finally, the last contract shows the specification of effect *Blacklist User*, which also involves two interrelated artifacts of different types (*Bicycle* and *User*).

Unblock User

The operation contract for task *Change User to Idle* is very simple. It basically changes the user's state to *Idle* without making any other changes. It does not check if the user is blacklisted at precondition time because this is already guaranteed by the source state in the state machine diagram.

Listing 3.4: Code for task *Change User to Idle*

```
operation changeUserToIdle(u: User)
pre: -
post: not u.oclIsTypeOf(Blacklisted) and u.oclIsTypeOf(Idle)
```

Pick Up Bicycle

The activity diagram for external event *Pick Up Bicycle* (see Figure 3.7) does not change in the two-artifact version of the example. However, the operation contract of the task *Confirm Pick Up* does require some changes. As artifact *User* has now several states, when he rents a bicycle we need to ensure that his state is *Active*. This is what is done in Listing 3.5.

Listing 3.5: Code for task *Confirm Pick Up*

```
operation ConfirmPickUp(b: Bicycle , u: User)
pre: -
post: BicycleRental.allInstances()->exists(br | br.oclIsNew() and
      u.oclIsTypeOf(Active) and br.active=u.oclAsType(Active) and not
      u.oclIsTypeOf(Idle) and br.startTime=now() and br.inUse =
      b.oclAsType(InUse))
```

Return Bicycle

Listing 3.6 shows the operation contract for the only task in *Return Bicycle* (Figure 3.12). As we have mentioned, *Return Bicycle* involves two different artifacts - Bicycle and User - which are interrelated. The operation contract below shows that, as we can obtain the user given the bicycle, the only input is the bicycle itself.

The postcondition of the contract changes the bicycle state to *Available* and assigns it to an *AnchorPoint*. There is no need to check if the *AnchorPoint* is empty, because if it is not, no changes will be made to the system because the integrity constraints in the class diagram will be violated, according to the strict interpretation of operation contracts that we use [104].

Then, if the user related to the bicycle had exactly one rental at precondition time, this implies that at postcondition time he must change his state to *Idle*, as he will no longer have bicycle rentals.

Listing 3.6: Code for task *Confirm Bicycle Return*

```
operation ConfirmBicycleReturn(b: Bicycle , ap:AnchorPoint)
pre: -
post:
let u: User = b.oclAsType(InUse)@pre.active.oclAsType(User) in
```

```

not b.oclIsTypeOf(InUse) and b.oclIsTypeOf(Available) and
  b.oclAsType(Available).lastReturn = today() and
  b.oclAsType(Available).anchorPoint = ap and
  (u.oclAsType(Active).inUse@pre->size()=1 implies (u.oclIsTypeOf(Idle) and
not(u.oclIsTypeOf(Active) and u.oclAsType(Idle).lastRental=today()))

```

Blacklist User

Blacklist User corresponds to the only effect in the state machine diagrams in our example. As effects are executed automatically, the only input parameters that they have are the artifacts which they make changes to. In this case, there are two parameters: the *User* and the *Bicycle*. As one is not independent of the other (i.e. the *user who is blacklisted* is the one who has not returned a *certain bicycle* in a specified time frame), the precondition of the task ensures that they are related to each other. Notice also that, as the user may already be blacklisted, the precondition considers both cases: *Active* user or *Blacklisted* user.

The postcondition changes the state of bicycle to *Lost* and links the *Lost* bicycle to user. In addition, it changes the state of user to *Blacklisted*, if he was not already, and in this case it also ensures that the blacklisted user is related to other bicycles he may still have rented and which cannot be considered yet as lost.

Listing 3.7: Code for task *Blacklist User*

```

action BlacklistUser(b: Bicycle , u: User)
pre: (u.oclIsTypeOf(Active) implies b.oclAsType(InUse).active =
  u.oclAsType(Active)) or (u.oclIsTypeOf(Blacklisted) implies
  b.oclAsType(InUse).blacklisted = u.oclAsType(Blacklisted))
post:
b.oclIsTypeOf(Lost) and b.oclAsType(Lost).lostDate = today() and not
  b.oclIsTypeOf(InUse) and
(u@pre.oclIsTypeOf(Blacklisted) implies b.oclIsTypeOf(Lost) and
  u.oclAsType(Blacklisted).lost -> includes(b.oclAsType(Lost)))
and
(u@pre.oclIsTypeOf(Active) implies u.oclIsTypeOf(Blacklisted) and not
  u.oclIsTypeOf(Active) and u.oclAsType(Blacklisted).date = today() and
  u.oclAsType(Blacklisted).lost ->includes(b.oclAsType(Lost)) and
  u@pre.oclAsType(Active).inUse -> forAll(bi | bi <> b implies
  u.oclAsType(Blacklisted).inUse -> includes(bi))

```

3.4 On the Relationship with Software Engineering Methodologies

After presenting the BAUML framework and using it to model an example with two business artifacts, we believe it is interesting to give an overview of our work in relation to some software engineering approaches, focusing on some of the most well-known object-oriented analysis methodologies and enterprise architecture frameworks. A detailed comparison, however, is out of the scope of this thesis.

We will begin by looking at specific object-oriented analysis approaches and then we will examine enterprise architectures.

3.4.1 Object-oriented Analysis

This subsection deals with some of the most well-known object-oriented analysis approaches. In particular, we will look at the work of Craig Larman [81], OO-Method [100] and MERODE [120].

To begin with, we would like to acknowledge Craig Larman's [81] influence on our work. His work is based on the Unified Process (UP), which is a software development methodology. The UP establishes different stages (e.g. business modeling, requirements, analysis & design, etc.) in the development of software and each of these stages has a different level of abstraction.

In particular, the analysis stage describes the domain of interest, answering the question *what should the system do?*. This contrasts with the design phase, which answers the *how?* question, that is, how the system is going to be implemented [81]. As we have seen, we focus on the analysis stage, since in BAUML we abstract away from the implementation.

For this stage, [81] uses the following diagrams to represent the static and dynamic elements in UML: a class diagram shows the relevant concepts or data and the system sequence diagrams specify the behavior. These sequence diagrams consider the system as a black box, and as such the operations which make up the sequence diagram are not assigned to objects. These operations have pre and postconditions, which can be specified in OCL.

Note the similarities and differences of this approach to ours. To start with, we use a class diagram in UML to represent the business artifacts, which is very similar to the class diagram for the domain model. Secondly, although we do not use system sequence diagrams, we also assume the system to be a blackbox and we specify the meaning of the tasks using pre and postconditions in OCL. In a sense, our activity diagrams have a similar role to the sequence diagrams,

as they also establish an order for task execution. Finally, state machine diagrams have a clearly defined purpose in our work (defining the lifecycle for artifacts and acting as the starting point for the definition of the activity diagrams representing the associations), whereas Larman [81] recommends using them to model use cases or the behavior of dynamic classes, but does not establish a clear pattern for their use.

The next approach, the OO-Method [100], uses an extended UML class diagram to represent the data, state machines to show how the classes may evolve, and a functional model which defines the meaning of the services or tasks. These models are grounded on the OASIS framework to give them formality.

Despite these similarities to our work (i.e. the use of a class diagram for artifacts and the state machine to represent the lifecycles), there are several differences, other than the notational ones. First, OO-Method requires the services to be assigned to the classes and hence, it has a lower level of abstraction than our approach. Secondly, attributes are restricted to three different types which in turn impose limitations on the way that a certain attribute may be modified. Thirdly, it does not use textual integrity constraints to complement the class diagram. In our approach, the use of textual integrity constraints helps us to avoid redundancy in the operation contracts, as we do not check in the preconditions the conditions guaranteed by the class diagram and its constraints.

On the other hand, the goal of the MERODE [120] approach to systems' modeling is to be able to generate models without contradictions and errors, to ensure their completeness and their validity. To do so, the method constructs different views over a single model, by applying automatic or semi-automatic construction techniques to obtain them. These views have been formally defined in process algebra, and they are the following: an existing-dependency graph, an object-event table and a finite state machine to show the evolution of classes/data/objects.

The existing-dependency graph shows the structure of the system, similar to a class diagram but representing the evolution over time of the classes/object-s/data and their relationships. The object-event table indicates which events have an impact on which objects, and the state machine shows the evolution of the objects or data. In addition to this, the object-event table gives a very generic definition of the impact (create, modify or delete) of what the event, or task, does. Thus, it does not have the level of detail that we wish to have in our model. Plus, events are atomic and there is no specific way to define preconditions, or conditions to determine the execution of a task or event

other than the state represented in the finite-state machine. Consequently, this approach uses states as a way to limit the potential tasks or operations that may execute over an artifact, which may lead to unnecessary states from the point of view of our approach. Moreover, like OO-Method, MERODE does not consider textual integrity constraints to complement the class diagram.

3.4.2 Enterprise Architecture

Another area related to our work is that of Enterprise Architecture. Enterprise Architecture approaches tend to adopt a holistic view of the enterprise, covering the "what", "how", "who", etc. and ranging from a generic high-level-of-abstraction view to more concrete, lower-level details.

ARIS (Architecture of Information Systems) started as a general business process architecture and evolved into ARIS-HOBE (ARIS House of Business Engineering) and later into the ARIS-House of Business Process Management methodology, which covers from business process design to information technology deployment [117]. There are also several tools [121] available. As a methodology, it covers more aspects than ours, but it does not have a specific notation attached to it.

In contrast to ARIS, Archimate [80] is both a framework and a language for enterprise modeling. One of its goals is to provide a uniform representation for diagrams that describe enterprise architectures. According to this work, in enterprise architecture models is better to have coherence and overview than specificity and details, and for this reason, UML and BPMN are too fine-grained.

The resulting Archimate model provides an overview of the enterprise's structure, by modeling three different layers: business, application and technology, and how they are connected to each other. In spite of the claims to the formality of the language, according to [125] the behavioral constructs have no precise defined semantics. In addition to this, there is no precise definition for the services or tasks involved in the processes.

Our work also has some similarities to RM-ODP using UML (UML4ODP) [70, 84]. RM-ODP is a framework for open distributed processing, and as such it offers different viewpoints over a model. These are useful to provide a variety of views (enterprise, information, computational, engineering and technology) for different purposes, and they go from a higher level of abstraction, useful at the business level, to a lower level of abstraction, useful for the implementation.

As we have explained, our approach does not delve into the details of the implementation and stays at a higher level of abstraction. However, the diagrams that the BAUML framework and UML4ODP at the higher level of abstraction use are similar. UML4ODP also uses class diagrams to represent the concepts or object types (i.e. the artifacts), state machine diagrams to show the evolution of those objects, and activity diagrams to illustrate the steps in the different processes. However, unlike in our approach, the transitions in the state machine diagrams do not correspond to processes. Moreover, the details of the operations or tasks are shown by means of design sequence diagrams which are at a lower level of abstraction than our approach. Consequently, it lacks the process viewpoint of our approach and it does not provide a high-level abstraction of operations as we would wish.

3.5 Summary & Conclusions

In this chapter we have presented a framework, BAUML, for modeling business processes from an artifact-centric perspective. This proposal is based on using a combination of UML and OCL models to represent all the dimensions in BALSAs and is summarized in Table 3.1.

BAUML	
Business Artifacts	UML Class Diagram & OCL Integr. Constr.
Lifecycles	UML State Mach. Diagram
Associations	UML Activity Diagrams
Services	OCL Op. Contracts

Table 3.1: Representation of the BALSAs dimensions in BAUML.

This framework has several advantages: it uses two languages, UML and OCL, both of which are ISO standards. They allow us to define the business process at a high level of abstraction and thus they are independent of the final technological implementation. These languages, but more specially UML, can be understood by the business modelers and developers. In addition, as we have shown, our proposal has precise semantics and can deal with business processes that contain more than one artifact.

Moreover, although our proposal is artifact-centric, it shares some characteristics of process or activity-centric proposals, as the associations are represented using an activity diagram, which clearly shows the order for the

execution of the tasks, as in process-centric approaches. In addition to this, there are already many CASE tools to aid in the definition of the diagrams using UML and even OCL.

Finally, we have seen that BAUML shares some characteristics with approaches that deal with object-oriented analysis and enterprise architectures. However, the main goal of our work was to define a way to model artifact-centric business process models following the BALSAs framework, and obtaining a model or set of models whose correctness can be checked automatically. These methodologies and languages add dimensions and elements which we do not consider in our BAUML framework, so they could enrich it; however, our final goal is to check the correctness of the structural and behavioral components of the process, abstracting away from other elements such as resources, organizational units or other details which are only relevant for the implementation.

Part III

Reasoning on Artifact-centric Business Process Models

Chapter 4

Preliminaries of Reasoning

In the previous chapters we have presented a modeling framework, BAUML, for artifact-centric business process models using UML and OCL. Our proposal relies on the use of four different models - UML class diagram, UML state machine diagrams and UML activity diagrams, plus OCL operation contracts - to represent the four dimensions in the BALSAs framework, which should be present in any artifact-centric business process model.

The current and following chapters go further and deal with checking the semantic correctness (i.e. they consider both the static and the dynamic dimension) of these models. The different approaches that we use are meant to check the correctness of the models before they are put into practice, thus avoiding the propagation of errors to the final deployment of the process.

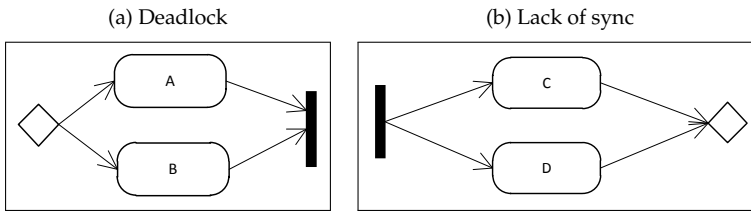
The present chapter is meant to be an introduction to reasoning on our models. It first presents the basic concepts (e.g. syntactic vs semantic correctness, validation vs verification) that we will use in the remaining chapters. Afterwards, it reviews the state of the art in terms of reasoning on artifact-centric business process models and on UML diagrams, as both topics are closely related to our work.

4.1 Basic Concepts

The goal of this short section is to establish the meaning, in the context of this thesis, of several terms that we will use throughout it. To begin with, when we talk about **reasoning** on our models, we refer to the ability of checking that they fulfill certain desirable properties.

We will particularly focus on the correctness of the model. We say that a model is **syntactically correct** if it conforms to the syntax of the language that is used. The **structural correctness** relates to whether the model avoids structural errors such as lack of synchronization or deadlocks (see Figure 4.1).

Figure 4.1: Examples of structural errors.



On the other hand, we say that a model is **semantically correct** if it represents the domain correctly. This includes, for instance, checking the model's consistency to avoid contradictions; and checking its minimality, in order to avoid redundancies in the model. Moreover, to a certain degree, we are also checking for the model's completeness. Completeness is defined as ensuring that all relevant information is included in the model.

Within the context of semantic reasoning, we distinguish between verification and validation. These terms are used with varying meanings [21, 1], and although they have similar definitions, it is important to differentiate them clearly for the purpose of this thesis.

Boehm [21] summarizes the difference between verification and validation of a product using the following questions. We can understand **verification** as asking "Are we building the product right?" whereas **validation** would ask "Are we building the right product?". In the context of business process modeling, verification would ask whether we are building the *model right*, and validation if we are building the *right model*.

As [106] explains, verification checks the internal correctness of the model, that is, it looks for contradictions and redundancies, whereas validation corresponds to checking the external correctness of the model, namely, it makes sure that the model fulfills the requirements and that it corresponds to the domain that it represents.

Therefore, semantic verification can be done automatically, because it checks the internal correctness of the model and does not require any additional information from the "outside world". Semantic validation, on the

other hand, requires user intervention in order to make sure that the domain or real world is represented correctly.

4.2 State of the Art

After having presented some concepts in the previous section, this section analyzes the state of art related to checking the correctness of business process models. In relation to this, Moreno et. al. [95] perform a literature review of the works dealing with business quality. The work gives an overview of the different topics related to business quality that have been researched and points out some areas of improvement and research gaps. However, it does not distinguish between artifact and process-centric approaches, nor it gives the details of the techniques employed by the different works.

The remainder of the section is structured according to four different topics: simulation, process model testing, syntactical reasoning and semantic reasoning on business process models. The last subsection summarizes the state of the art related to reasoning.

4.2.1 Simulation

One way of checking the correctness of the business process models considering their data dimension is to use simulation techniques. However, most of the existing works focus on optimizing and performing quantitative analysis of the business processes rather than on validating or verifying them.

For instance, [41] uses simulations of runs of the business process based on Petri nets to perform analysis of certain key indicators. Similarly, [13] runs simulations, based on Petri nets, to identify interdependencies and the impact between services when it comes to their availability. Additionally, [6] studies the use of business process simulation to optimize the business process.

On the other hand, [111] uses process mining techniques to derive simulation models from event logs (which reflect the real execution of the process), integrating various perspectives (i.e. control-flow, data and performance) into a single model. However, the uses of simulation are out of the scope of [111].

As we have seen, simulation is mainly used to optimize and quantitatively analyze business processes. In contrast, we are interested in checking their correctness. Therefore, simulation techniques are complementary to our goal, and they could be applied after checking the semantic correctness of the models.

4.2.2 Process Model Testing

Model testing techniques can be used to test different properties of business process models, including their correctness. In order to perform model testing, we require input data which is provided to the model, and then the result of the execution is compared to the expected result.

For instance, the authors of [130] use these techniques to check model transformations for certain errors. Although there are techniques to validate the whole transformation, they can easily lead to state explosion. For this reason, the authors opt for a more manageable approach which can also provide valuable information.

On the other hand, [22] performs a literature review of the existing approaches to process model testing. Although they can help to detect errors in the models, they are not complete, in the sense that they only ensure the correct result for certain input data. However, when the result is not what was expected, they can detect errors and they have the advantage of avoiding state explosion.

4.2.3 Syntactical & Structural Reasoning

When it comes to syntactical and structural reasoning on business process models, the greatest part of research follows a process-centric perspective. Most of this research has centered on detecting errors in the flow of activities and, for this reason, many authors translate different workflow models into Petri nets, which are formal.

For instance, [33] translates generic workflows into graphs in order to check certain properties such as soundness, deadlock, etc., similar to what is done in [83]. The most important difference between the two approaches is that [33] deals with cyclic workflows, whereas [83] does not. A more informal approach is used in [43], where EPCs (Event-driven Process Chains) are first reduced and then translated into a Petri net. However, this approach may require user intervention in order to determine whether a potentially conflictive situation is erroneous or not.

[123] gives a formal definition of UML activity diagrams. To do so, it maps the elements of activity diagrams, including the data flow, to colored Petri-nets. Due to this, standard properties checked in Petri nets such as state reachability can be checked in the activity diagrams. However, data is only considered in terms of the flow of the diagram and no formal definition of the tasks and their impact on the data is given.

A key paper dealing with structural reasoning is [3]. The authors study different variations of Workflow nets (similar to Petri nets) and how the different notions of soundness apply to them. Although not focused on actually verifying the workflows but rather on whether it is decidable or not to do so, the article offers an overview of existing techniques to verify them (coverability graph, invariants and graph reduction methods).

All these works make an important contribution by providing methods for the verification of the business process models used, which are usually based on Petri Nets, and their results can be applied to the models in our approach. However, being process-centric, they do not deal with artifacts nor the meaning of actions, both of which are key elements in our proposal.

4.2.4 Semantic Reasoning

As we have explained in the previous section, we are interested in performing semantic reasoning, which, for us, consists in validating and verifying the model. Although [62] does not deal with business processes, but rather on service composition, it is interesting because it focuses on both verification and validation. In this case, the services are specified using pre and postconditions defined in Fluent Calculus as supported by a tool, Flux. Fluent Calculus is a formal language but is not easy to understand. Flux has the ability of checking whether a certain service composition can achieve a certain goal, and can also, given a goal, return a service composition that fulfills it.

As the BAUML framework models artifact-centric business processes using a combination of UML and OCL models, the remainder of this section examines semantic reasoning in two different areas directly related to our work: artifact-centric business process models and UML diagrams. However, we also analyze what we call *data-aware* approaches: works that, while not explicitly artifact-centric, take the data into consideration when reasoning.

Data-Aware Approaches

There are some approaches that, although not specifically artifact-centric, do consider the data, such as [8, 119, 87]. [119] deals with soundness in WFD-nets (based on Petri nets) considering the read/write/delete operations in the process. [8] detects errors in the flow by considering the evolution of data from one state to the next. Given various Petri nets representing the evolution of artifacts, [87] creates a valid choreography (i.e. an interaction sequence), taking into consideration the policies that restrict the valid interaction and the

goal states. However, none of these works deals with the detailed meaning of the tasks (or operations) and they do not have an underlying conceptual schema representing the data and its relationships.

Similarly, [110] studies the conditions which guarantee the correctness of the data flow when making changes to the control flow or the data flow of a business process. However, both the data and the tasks themselves are represented in a very simple way.

The goal of [73] is to check if the business process model fulfills certain properties (i.e. compliance) that may take the data into consideration. To do so, the authors create an abstraction of the model to avoid the state explosion caused by the data. However, the structure of the data is not fully represented.

To perform the reasoning, the authors use model checking techniques applied to a state representation of the abstract system and a compliance rule defined in logic. In case of a compliance violation, feedback is provided to the user. There is a prototype tool that can perform these tests.

Semantic Reasoning on Artifact-centric BPM

Several approaches to reasoning on artifact-centric BPM use data-centric dynamic systems (DCDSs), grounded on logic, as the basis for reasoning [11, 27, 12]. [11] uses a relational database to represent the data, together with a set of condition-action rules and actions defined in logic. In contrast, [12] uses a Knowledge and Action Base defined in a variant of Description Logics to represent this data. Similarly, [27] maps an ontology to a DCDS in order to verify certain temporal properties expressed in a variant of μ -calculus.

[31] and [9] provide the basis for [11]. [31] represents artifacts in a database, lifecycles are shown by means of condition-action rules, and actions' (or services') pre and postconditions are represented as conjunctive queries. Properties of the data-centric model are expressed in logic using $\mu\mathcal{L}$, a variant of μ -calculus. [9] expands on [31] by adding inter-artifact and intra-artifact constraints in $\mu\mathcal{L}$, and allowing the use of negation, arbitrary quantification and Skolem functions in the actions' specification.

In [36], artifacts are represented using a set of variables, which are updated by services defined by pre and postconditions in first-order logic. This work is actually a summary of the results described in [37] and [42]. The authors check whether the model fulfills a set of properties defined in LTL-FO (a first-order extension of linear-time temporal logic), which is not as powerful as $\mu\mathcal{L}$. Despite this, it is possible to represent integrity constraints such as primary

keys and foreign keys, in contrast to [31, 9]. Moreover, arithmetic constraints are allowed in the services' definition, something not permitted by [31, 9, 11].

Similarly, [15] also checks whether a deployed artifact system, defined in logic, fulfills a property defined in FO-CTL (a first-order extension of CTL). The fact that the artifact system has been deployed implies that it does not deal with infinite data, but there is rather an upper bound on the number of elements in each state. Unlike [37, 36, 42], it is not possible to represent arithmetic constraints.

On the other hand, [55] goes as far as to define a specification language, ABSL, based on CTL, to specify the artifacts' lifecycle behavior and is able to check if the model satisfies a certain property defined in ABSL. However, like in the case of LTL-FO, CTL is not as powerful as μ -calculus.

Another alternative is [19]. It establishes a basis for reasoning on reachability, dead-ends and redundancy. Artifacts are represented using a set of attributes, an identifier and a state. Services (or tasks), consisting of preconditions and conditional effects, are defined by means of predicates *new*, *defined* and assignments between variables. Business rules are defined by means of *if* rules. However, this proposal does not deal with actual data, but rather an abstraction of it (hence the use of predicates *new* and *defined*).

All these works represent artifact-centric business process models in languages derived from logic. Consequently, the models under consideration are formal, but they are not practical for business people. Moreover, they have been proposed at a theoretical level and do not have a tool that implements them.

In contrast, the Guard-Stage-Milestone (GSM) approach provides a more business-friendly representation of artifact-centric business processes, and several works [122, 60, 58] study reasoning on these models.

[122] studies the decidability of verification over GSM models by translating them into a DCDSs. However, the presented results are theoretical, as there is no tool that can actually perform the reasoning. [60] presents a system to model and execute artifact systems. However, to our knowledge, the system is limited to simulating the behavior of the model given certain data, which differs from the work in this thesis.

All of the works in [16, 57, 58] use a tool, GSMC, to reason on GSM models. Although the tool is able to translate the GSM model into a symbol transition system for model checking, several restrictions are imposed on the data types and it only allows one instance per artifact [57]. [16] expands this work by allowing agent-based semantics on the GSM model in order to be able to verify the information about participants. In a similar way, [58] performs

model checking over GSM models from a multi-agent perspective; however the bound placed on the number of objects may sometimes lead to unreliable results when this bound is exceeded.

Similarly to the work presented in this thesis, [132] performs verification over process models considering the meaning of the tasks. It uses BPMN diagrams whose tasks are optionally annotated with preconditions and effects defined in logic, and use an optional ontology to define the underlying data. Time is not considered explicitly, but they have implemented a prototype tool that can perform some verification tests. However, as neither the ontology nor the details tasks are compulsory, the final results can only be partial or provide an intuitive idea of potential issues.

A closer work to ours is detailed in [23]. Starting from models defined in what the authors call *artifact union graphs* (a Petri-net like notation), they verify state reachability and weak termination. The artifact union graphs represent the states and the services that trigger the transitions between those states. Details of the services are defined in pre and postconditions which use a certain grammar. Constraints also use this grammar. They also have a tool which is able to perform the verification. Although their approach deals with several artifacts, the domain of the artifacts' attributes is constrained and the type of tests that can be performed is limited, as they only allow state reachability and weak termination.

Finally, a completely different approach is presented in [86]. Instead of checking the business process model's conformance to certain rules following a *compliance by detection* approach, the author uses a *compliance by design* approach: it generates business process models that already comply with the rules. However, it requires an initial business process model on top of which a new model is built which fulfills the rules. This means that there may be errors in the model which may not be detected. In addition, the notation used to represent the artifact-centric models is based on Petri nets to represent the lifecycle of the artifact, and no details of the tasks are given.

Semantic Reasoning on UML models

On the other hand, most of the proposals for reasoning on UML models deal with only one diagram. For instance, [106, 103] focus on the class diagram, [34] handles state-machine diagrams, and [44] focuses on activity diagrams. As far as approaches examining various UML diagrams, [88] offers a systematic literature review but only four of the analyzed papers perform reasoning on

more than one of the diagrams in our approach: they can handle class and state machine diagrams.

Other approaches [105, 26, 56] consider not only the static structure of the UML class diagram but also the OCL operations that modify it. On the other hand, [34] transforms state machine diagrams into colored Petri nets and verifies whether they fulfill certain properties defined in LTL or CTL.

[44] checks the consistency between UML activity diagrams and class diagrams. However, instead of dealing with the activity diagram's actions specification, it considers that the object flow acts as a precondition and postcondition of the actions. These constraints are derived automatically from the diagram. Therefore, it only focuses on create, read, write and update dependencies among tasks.

Other approaches such as [124] check the consistency between different UML diagrams using Description Logic, but targets very basic properties and does not include the definition of the operations or the additional constraints in the process.

Although not explicitly an artifact-centric approach, [109] studies the quality of business processes represented using UML activity diagrams. Despite the similarity of their models with our framework, the goal of the paper is different. We start from several UML models, whereas [109] starts from an "informal" (i.e. described in natural language) UML activity diagram from which a UML class diagram and the operation contracts for the tasks are derived. The resulting models are analyzed to detect mainly syntactic and structural errors.

4.2.5 Summary

As we have seen, existing research on reasoning on process-centric models focuses on checking their syntactic and structural correctness. On the other hand, reasoning on artifact-centric process models is normally oriented towards validation, by checking whether the model fulfills certain desirable properties. Although simulation may also be a way of checking the correctness of artifact-centric business process models, most of the works actually focus on optimization or quantitative analysis of the process. Process model testing may be useful to detect some errors, but cannot guarantee that certain properties are fulfilled by the model.

Moreover, most of the proposals that perform some kind of semantic reasoning represent artifact-centric process models in some variant of logic. This results in a formal definition of the system which makes it possible to reason on

its specification. However, specifications in logic are low-level and complex, and therefore difficult to understand and unpalatable for business people.

On the other hand, the GSM approach [122, 60, 58], is more modeler-friendly and has a formal semantics. However, the existing tools that can do some reasoning are limited [16, 57, 58].

Similarly, although [132] uses BPMN notation and an (optional) ontology for the data and there is a tool for the reasoning, the final results are only partial as the tasks are only partially annotated. [23] uses artifact union graphs to represent business processes and uses a tool to perform verification tests. However, the tests are limited to state reachability and weak termination.

Finally, after examining the literature that deals with semantic reasoning on UML models, we have found that none of the existing works deal, globally, with all the models in the BAUML framework.

Chapter 5

Reasoning Using Data-centric Dynamic Systems

Given a BAUML model, our goal is to be able to ensure that it fulfills a certain set of desirable properties. In particular, we wish to check that there are no errors in the model and that it represents reality correctly. Unfortunately, despite the advantages of the BAUML framework, there are no available methods to reason with the models that are used.

Data-centric Dynamic Systems (DCDSs) are an alternative way of representing data-centric business process models. They provide a formal representation at a lower level of abstraction than UML and OCL, but as shown in [11] it is possible to apply model checking techniques to these models in order to check if they fulfill a certain property.

For this reason, this chapter will present a way to translate a UML artifact-centric BPM into a DCDS, with the final goal of checking the correctness of the BAUML models. Figure 5.1 gives a general overview of this process. Although, as of yet, there is no tool that can perform this type of reasoning over DCDSs, this chapter shows the feasibility of our approach in terms of a firmly established proposal external to our own work.

To do so, it first introduces some basic concepts of DCDSs and some assumptions we make over our BAUML models. Afterwards it describes the translation process required to obtain a DCDS from a BAUML model. The last part of the chapter focuses on the types of reasoning that can be performed with the DCDS and the results that can be obtained.

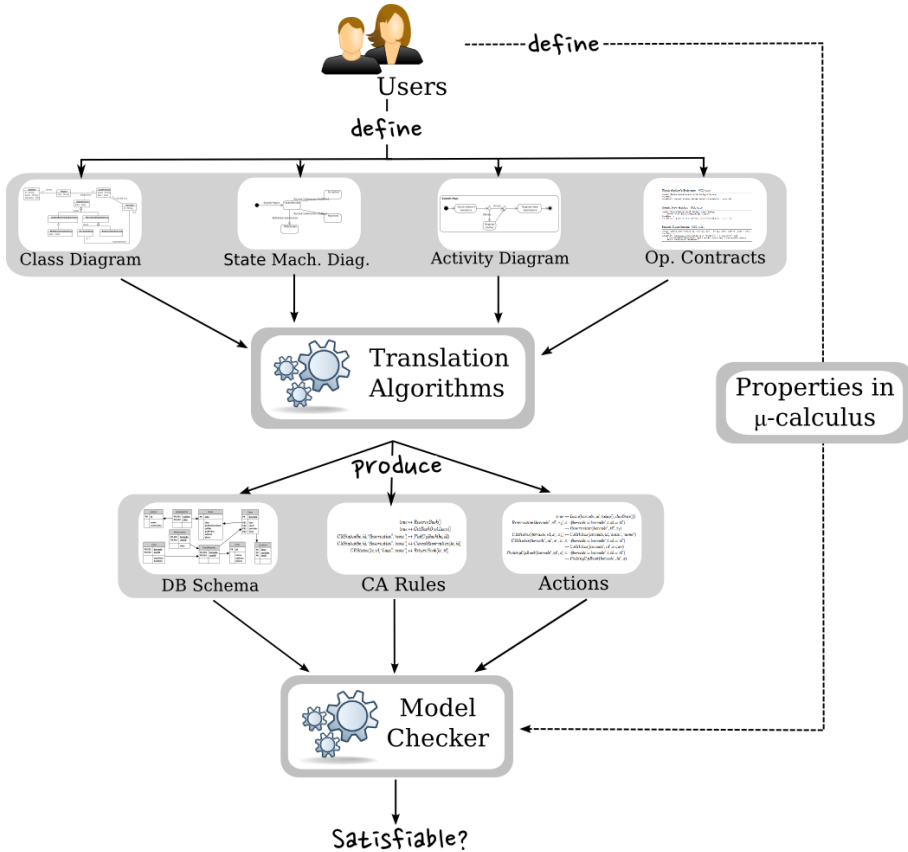


Figure 5.1: General overview of the reasoning process using DCDSs

5.1 Background

This section presents the required background for this chapter. It begins by introducing DCDSs and an intuitive mapping between the Balsa framework and DCDSs. After this, it details the assumptions that we make over our models before the translation process begins.

5.1.1 An overview of Data-centric Dynamic Systems

A relational Data-centric Dynamic System (DCDS) [11] is a tuple $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$, where \mathcal{D} corresponds to the *data layer* and \mathcal{P} to the *process layer*. More specifically, the data layer \mathcal{D} is defined as a tuple $\mathcal{D} = \langle C, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$, such that:

- C is a set of *values*.
- \mathcal{R} is a *database schema* containing a finite set of tables $\{\mathcal{R}_1, \dots, \mathcal{R}_n\}$.
- \mathcal{E} is a finite set of *equality constraints* of the form $Q_i \rightarrow \bigwedge_{j=1, \dots, k} z_{ij} = y_{ij}$, where:
 - Q_i is a domain-independent first-order query over \mathcal{R} using constants from the active domain $\text{ADOM}(\mathcal{I}_0)$ ¹ of \mathcal{I}_0 . The $\text{ADOM}(\mathcal{I})$ of \mathcal{I} is a set of constants or values c such that $c \in \text{ADOM}(\mathcal{I})$ if and only if c appears in \mathcal{I} .
 - z_{ij} and y_{ij} are free variables or constants in $\text{ADOM}(\mathcal{I}_0)$.
- \mathcal{I}_0 represents the *initial instance* of the database schema. Therefore, it conforms to \mathcal{R} and satisfies \mathcal{E} .

On the other hand, the process layer is a tuple $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$ such that:

- \mathcal{F} is a finite set *functions*. They represent the interface to external services.
- \mathcal{A} is a finite set of *actions*. They are in charge of evolving the data layer. They are executed sequentially and are atomic. An action $\alpha \in \mathcal{A}$ has the form $\alpha(p_1, \dots, p_n) : \{e_1, \dots, e_m\}$, where:
 - $\alpha(p_1, \dots, p_n)$ is the signature of the action, where α is the *action's name* and p_1, \dots, p_n represent *action parameters*.
 - $\{e_1, \dots, e_m\}$ is a set of *effects*. They take place simultaneously. Each effect e_i is defined as $q_i^+ \wedge Q_i^- \rightsquigarrow E_i$, where:
 - * $q_i^+ \wedge Q_i^-$ is a query over \mathcal{R} . Its terms are variables, action parameters, and constants from the active domain (ADOM) of \mathcal{I}_0 . q_i^+ is a union of conjunctive queries and Q_i^- is a first-order formula. Its free variables appear in q_i^+ .

¹Without loss of generality, we assume that all constants appear in \mathcal{I}_0 .

- * E_i is a set of facts for \mathcal{R} . It may include terms in $\text{ADOM}(\mathcal{I}_0)$, input parameters, free variables of q_i^+ and function f calls, where $f \in \mathcal{F}$.
- ϱ is a finite set of *condition-action rules*, defined as $Q \mapsto \alpha$:
 - Q is a *first-order query* over \mathcal{R} . Its free variables are the parameters of α . Its other terms can be quantified variables or constants in $\text{ADOM}(\mathcal{I}_0)$. Notice that Q refers to the content in the database schema \mathcal{R} .
 - α is an *action* in \mathcal{A} .

5.1.2 Mapping DCDSs to the BALSAs Framework

To facilitate the understanding of DCDSs, we map the dimensions in the BALSAs framework to their representation in a relational DCDS. Table 5.1 summarizes this mapping.

	DCDS
Business Artifacts	DB Schema
Lifecycles	CA Rules
Associations	CA Rules
Services	Actions

Table 5.1: Intuitive representation of the BALSAs dimensions in DCDSs

Because **business artifacts** contain the static information for the business, they will be part of the data layer. In DCDSs they are represented as a set of tables in a relational database schema \mathcal{R} . The restrictions over the database schema will be specified as a set \mathcal{E} of equality constraints of the form $Q_i \rightarrow \bigwedge_{j=1,\dots,k} z_{ij} = y_{ij}$.

Lifecycles show the evolution of a business artifact. As such, they do not have a direct representation in DCDSs. However, condition-action (CA) rules of the form $Q \mapsto \alpha$ establish that, when the condition on the left-hand side of the rule is true, the action on the right-hand side *may* be executed. Therefore, they take part in defining the evolution of business artifacts, and could be considered as a representation for the lifecycle.

Associations indicate the conditions under which a service may be executed. Therefore, they should be represented using CA rules, like in the previous case.

As **services** are in charge of making changes to the artifacts, the appropriate way to represent them using DCDSs is by means of actions α . As we have seen, actions can include calls to external services, represented as functions \mathcal{F} .

Important

The semantics of DCDSs state that content which is not explicitly copied is lost. Therefore, if we wish to keep all the information from one state to the next, we will have to add a set of effects in charge of doing so.

Intuitively, we could assume that one way of performing the translation process from BAUML to DCDSs is by mapping the models according to the BALS dimension which they represent. For instance, as business artifacts are represented by means of a class diagram in UML and by means of a database schema in DCDSs, it looks like we would be able to obtain the database schema of a DCDS by translating the class diagram.

However, the translation process is not as straightforward as it seems and requires additional considerations. Table 5.2 shows the elements of DCDS which result from the translation of each of the UML and OCL models in BAUML. Before detailing the translation, the next subsection states the assumptions that we make over our BAUML model.

5.1.3 Assumptions

This section presents some assumptions that we make over our initial BAUML models. To begin with, we only consider artifact-centric business process models with one artifact type. Having two or more artifact types would require tracking simultaneously the evolution of two artifact types adding much more complexity. So we begin our approach to the problem by considering only one artifact type and leave dealing with two artifact types for future work.

Secondly, of all the gateway nodes in activity diagrams, we only consider decision and merge nodes. If we had parallel execution paths this would result in state explosion of the possible combinations. Again, we leave those for future work.

Source Model		Target Model
<i>BAUML</i>		<i>DCDS</i>
Business Art.	Class Diagram. OCL Constr.	DB Schema Eq. Constr
Lifecycles	State Mach. Diagram	CA Rules DB Schema Actions
Associations	Activity Diagrams	CA Rules DB Schema Actions
Services	Op. Contracts	Actions DB Schema CA rules

Table 5.2: Translation of BAUML to the elements of a DCDS

In addition, we only allow two types of guard conditions: those stating conditions over the class diagram, and those referring to user-made decisions. Referring to the result of the previous task complicates the translation and in many instances it could be rewritten as a class diagram condition.

Finally, we also assume that our models are syntactically and structurally correct, as we wish to focus on checking their semantic correctness.

The remaining assumptions either correspond to limitations imposed by the target model (the DCDSs) or simplifications that we make to facilitate the translation process, but which could be straightforwardly implemented as part of the translation process. Therefore, the latter do not limit the potential of the reasoning in any way.

The list of limitations imposed by the DCDS are the following:

- As DCDSs cannot deal with time, we will assume that there are no time events in the state machine diagram.
- The tasks that create artifacts should be the first ones in the activity diagram. On the other hand, the tasks that delete artifacts should be the last ones in the activity diagram. This is necessary to ensure the proper tracking of the evolution through the tasks in the activity diagram in the DCDS.

- Furthermore, we do only consider data coming from a countably infinite unordered domain, and that can only be compared for (in)equality. We thus avoid any assumption on the structure of data domains, and consider only string and boolean attributes².

We also make the following simplifications over our initial models to facilitate the translation process:

- As material actions have no impact by themselves on the system, we assume that the initial BAUML model does not have any material actions in its activity diagrams. It should be straightforward to transform an activity diagram with material actions into one without.
- We do not deal with subprocesses in the activity diagrams. A subprocess is a node in the diagram which is decomposed in another activity diagram. It would be equivalent to embedding the activity diagram that corresponds to the subprocess in the original activity diagram³.
- There should not be two different tasks with the same names in a BAUML model. This could be solved by appending the activity diagram's name to the task name.
- Tasks can only appear once in an activity diagram. That is, there cannot be two task nodes with the same name, even if they are actually the same task (i.e. they have the same precondition and postcondition). To solve this, we can create two tasks with the same associated pre and postcondition and a different name. Another option is to model the diagram in a way in which the task only appears once.
- We will assume that effects in a state machine diagram are part of an activity diagram with only one task, which will correspond to the effect itself. This activity diagram will have the same name as the effect plus "AD".
- Although the state machine diagram shows how an artifact changes from one state to another, we assume that the change of state is performed

²A boolean attribute can be considered as a special string attribute that can only be assigned to the special strings `true` or `false`.

³Note that, as we are specifying elements from a high-level perspective, we do not assume any kind of isolation property between the different diagrams. The tasks in each diagram will manipulate the data given as input (which in this case will be provided by a user) and/or any other data.

by the tasks in the activity diagram or the effects. In most cases this redundancy is necessary in any case because the change to the new state requires adding specific relationships or attributes.

- We also assume that the previous node of a final node is always a task, to ensure that the algorithms provide the appropriate results.

In the remainder of this chapter we will use the Bicing example with one artifact that we presented in Section 3.1.

5.2 Translating a UML Artifact-centric BPM to a DCDS

After presenting the preliminaries, this section describes the translation process required to obtain a DCDS that is equivalent to a BAUML model, with the final goal of checking the correctness of a BAUML model. For each of the components in our BAUML representation, we show how its elements can be translated into a DCDS.

Algorithm 1 *translateBAUMLintoDCDS*($\mathcal{B} = \langle \mathcal{M}, \mathcal{O}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$)

```

CASet :=  $\emptyset$ 
actionSet :=  $\emptyset$ 
 $\langle \text{DBSchema}, \text{EC} \rangle = \text{translateCDandICs}(\mathcal{M}, \mathcal{O})$             $\triangleright$  Obtain DB and eq. constr. schema from class diag.
 $\langle \text{DB}, \text{CARules} \rangle := \text{translateSMD}(\mathcal{S})$                     $\triangleright$  CA rules from state mach. diagram
DBSchema := DBSchema  $\cup$  DB
CASet := CASet  $\cup$  CARules
 $\langle \text{DB}, \text{CA}, \text{actions} \rangle := \text{translateADs}(\mathcal{P})$               $\triangleright$  Definition of state mach. diag. actions
DBSchema := DBSchema  $\cup$  DB
CASet := CASet  $\cup$  CA
actionSet := actionSet  $\cup$  actions
 $\langle \text{DB}, \text{actions} \rangle := \text{translateTasks}(\mathcal{T})$ 
DBSchema := DBSchema  $\cup$  DB
actionSet := actionSet  $\cup$  actions
for all  $a \in \text{actionSet}$  do
    addRulesForFrameProblem( $a$ )
end for
return  $\langle \text{DBSchema}, \text{EC}, \text{CASet}, \text{actionSet} \rangle$ 

```

Algorithm 1 shows the main steps involved in the translation process. It is divided in steps according to the different components of our BAUML model. The algorithm begins by translating the class diagram and the integrity constraints into a database schema and a set of equality constraints. Afterwards it translates the state machine diagram, which will result in new tables which are added to the database schema and a set of condition-action rules.

Once this is done, the next step deals with the translation of the activity diagrams. This will return a set of tables, condition-action rules and actions that will be added to the already existing ones. Then we obtain the translation of the tasks into actions, which may also generate additional tables to be added to the schema. Finally, the last step in the diagram adds the necessary rules to deal with the frame problem: that is, it copies the content of all the tables that have not been modified by the action a , excepting table aux , which, as we shall see, it is used to ensure that integrity constraints are only checked at the end of a transition, that is, when an activity diagram finishes executing.

We will use the identifiers of the business artifact to track its evolution through its lifecycle and the tasks that are part of the corresponding activity diagrams.

The remainder of this section is structured according to the general steps described above. Table 5.3 shows the correspondence between the sections, their starting page, and the algorithms which may be part of the section showing the details of the translation.

Section	Page	Algorithm
Translating the Database Schema and Equality Constraints	85	N/A
Translating the State Machine Diagram	89	Algorithm 2
Translating the Activity Diagrams	92	Algorithm 3
Translating the Tasks	99	Algorithm 4

Table 5.3: Summary of the sections, their pages and the contained algorithms

5.2.1 Translating the Database Schema and Equality Constraints

As we have seen, a class diagram \mathcal{M} has a set of classes ($\text{CLASSES}(\mathcal{M})$) and associations ($\text{ASSOCIATIONS}(\mathcal{M})$). Apart from this, it also has a set of integrity constraints \mathcal{O} , represented either graphically or textually.

We need to translate all these elements into a DCDS. Considering that they represent the static information in the system, they will correspond to the data dimension \mathcal{D} in the DCDS. We explain below the main steps in this translation process:

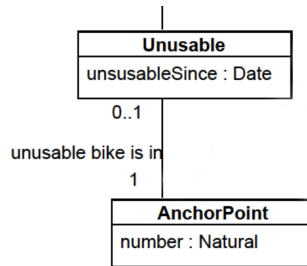


Figure 5.2: Fragment of the class diagram in Figure 3.2.

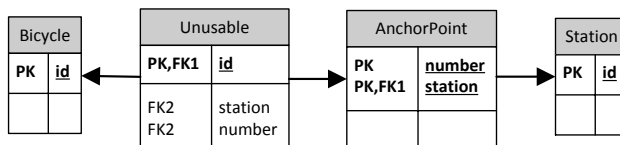


Figure 5.3: Possible translation of the relationship between *Unusable* and *AnchorPoint* into a DB schema. We want to avoid it.

Step 1: Translate classes and associations UML class diagrams can be translated into relational database schemas following well-known techniques of database design [126]. Notice that we do not specifically require the tables to be in any kind of normal form. We will make the following assumptions to keep the translation coherent and, in turn, easier:

- **Associations are always represented in one table**, even if the multiplicity in the relationship would allow them to be incorporated into the table of one of its participants. For instance, Figure 5.2 shows a fragment of the class diagram with classes *Unusable* (which is a subclass of *Bicycle*) and *AnchorPoint*. The cardinalities state that an unusable bicycle will always be assigned to exactly one anchor point.

Figure 5.3 shows a possible translation of these two classes into a database schema. Notice that the relationship between *Unusable* and *AnchorPoint* does not have its own table, but is represented by including a couple of attributes (the identifiers of *AnchorPoint*) which are foreign keys to *AnchorPoint*. We want to avoid this, and have a table for the relationship. See Figure 5.4 for the final translation of the class diagram.

- **Each subclass and superclass is always represented into its own table.**
The table corresponding to the subclass will have the identifiers of its superclass (which, of course, are also its own identifiers) and a foreign key to the table that corresponds to the superclass. In our example, we would have a table for *Bicycle*, *Unusable*, *Available* and *InUse*.

Figure 5.4 shows the result of applying this first step to our Bicing example presented in Section 3.1. As shown, each class and association is represented in its own table and they are related by means of foreign keys. To keep the translation shorter, we have only included the identifiers of each class and relationship as attributes.

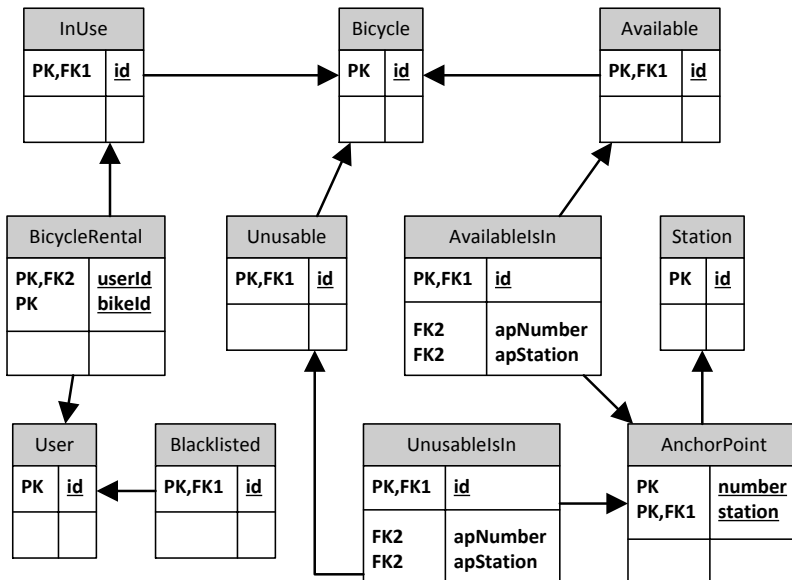


Figure 5.4: Database schema containing the tables obtained in the first step¹ to store the information.

Step 2: Translate integrity constraints The second step translates the integrity constraints into equality constraints, if possible. This includes both

textual constraints defined in OCL and graphical restrictions in the class diagram. A way to do so is by using [106] as a basis. This work normalizes OCL expressions into expressions of the form:

$$\text{path-exp} \rightarrow \text{select}(e \mid \text{body}) \rightarrow \text{size}() \text{ opComp } k$$

before they are translated. Once a constraint ic has been normalized, it is translated into logic. Notice that after the normalized expression expr is obtained, [106] translates it to the *denial form* $\neg \text{expr}$, which is exactly what we will need later on.

However, there are some differences between this work and ours which must be considered. In [106], attributes are part of binary predicates which also include the artifact's identifier. In the present work, the logic representation of tables is a n -ary predicate that includes all the attributes. The expressions have to be translated bearing this in mind.

Once the restrictions have been translated into logic, we need to transform them in the following way (adapted from [10]):

1. Add an auxiliary table to the database schema, of the form $\text{aux}(x, y)$.
2. Initialize $\text{aux}(x, y)$ with $\langle a, b \rangle$ at the end of the execution of every activity diagram. We only want to check their fulfillment at the end of the execution, not while it is taking place. We will have to bear this in mind when translating the tasks that make up the diagram.
3. For each expression IC obtained in the *denial form*, add the following equality constraint: $IC \wedge \text{aux}(x, y) \rightarrow x = y$

For instance, the denial form of the covering (i.e. restriction *complete* in the class hierarchy) constraint in our class diagram would be translated into logic like this:

$$\text{Bicycle}(id) \wedge \neg(\text{Unusable}(id) \vee \text{Available}(id) \vee \text{InUse}(id)),$$

and afterwards, we would apply the rules defined above.

Notice, however, that equality constraints have a limitation: they cannot be used to represent constraints that require greater than or less than symbols.

The primary and foreign keys that result from the translation of the UML class diagram into a database schema should appear in the DCDS as equality constraints. For instance, the equality constraints necessary to represent the constraints related to association *UnusableIsIn*, would have the following form:

$$\begin{aligned} & \text{UnusableIsIn}(id, n_1, s_1) \wedge \text{UnusableIsIn}(id, n_2, s_2) \wedge n_1 \neq n_2 \wedge \text{aux}(x, y) \mapsto x = y \\ & \text{UnusableIsIn}(id, n_1, s_1) \wedge \text{UnusableIsIn}(id, n_2, s_2) \wedge s_1 \neq s_2 \wedge \text{aux}(x, y) \mapsto x = y \\ & \text{UnusableIsIn}(id_1, n, s) \wedge \text{UnusableIsIn}(id_2, n, s) \wedge id_1 \neq id_2 \wedge \text{aux}(x, y) \mapsto x = y \end{aligned}$$

These constraints ensure that an unusable bicycle is assigned at most to one anchor point, and that an anchor point has exactly one bicycle.

5.2.2 Translating the State Machine Diagram

As we have mentioned previously, given a state machine diagram $s = \langle V, v_o, v_f, E, X, T \rangle \in \mathcal{S}$, our goal is to incorporate its information in condition-action rules of the following form: $Q \mapsto \alpha$, where Q corresponds to a condition, and α represents an action. When Q is true, then α may take place. Intuitively, given a transition $t = \langle v_s, o, e, c, x, v_t \rangle \in T$, the source state v_s and the OCL condition o will be part of Q . The event e and its tag c (if any) and the effect x will correspond to the action α .

We will follow this intuition for the types of transitions we can see below:

- ([OCL]) ExternalEvent ([tag])
- [OCL] (/ Effect)

Notice that, for the second type of transitions, the meaning in the DCDS differs from the original one. Originally, when the OCL condition is true, the transition *always* fires. According to the semantics of DCDSs, however, the resulting rule merely indicates the action *may* execute.

Therefore, in order to ensure an accurate result of the reasoning procedure, the modeler should do the following. Given two transitions $t_i = \langle v_{s_i}, o_i, e, c, \emptyset, v_{t_i} \rangle$ and $t_j = \langle v_{s_j}, o_j, \emptyset, \emptyset, x, v_{t_j} \rangle$, where $v_{s_i} = v_{s_j}$, then the condition o_i in t_i should be mutually exclusive with condition o_j in t_j . This applies to every pair of transitions of these types (i.e. $t_i = \langle v_{s_i}, o_i, e, c, \emptyset, v_{t_i} \rangle$ and $t_j = \langle v_{s_j}, o_j, \emptyset, \emptyset, x, v_{t_j} \rangle$). This will ensure that the only transition that can trigger when o_j is true is t_j ⁴.

⁴Note that a model as described here with transitions $t_i = \langle v_{s_i}, o_i, e, c, \emptyset, v_{t_i} \rangle$ and $t_j = \langle v_{s_j}, o_j, \emptyset, \emptyset, x, v_{t_j} \rangle$, where $o_i = o_j$ and $v_{s_i} = v_{s_j}$ would not make sense. $t_i = \langle v_{s_i}, o_i, e, c, \emptyset, v_{t_i} \rangle$ could never be triggered, as when o_i became true, $t_j = \langle v_{s_j}, o_j, \emptyset, \emptyset, x, v_{t_j} \rangle$ would be automatically triggered. However, if we do not establish the restriction described above, then the results of reasoning with the DCDS would be unreliable.

Algorithm 2 shows the translation process to obtain the condition-action rules corresponding to the state machine diagram. Its main steps are described below.

Algorithm 2 *translateSMD(S)*

```

CA := ∅
s = ⟨V, vo, vf, E, X, T⟩ ∈ S
PKi := getPrimaryKeyOfArtifact(s)
DB := {BAStatus(PKi, state, transition)}           ▷ Step 1: Create tables for lifecycle
DB := DB ∪ {Busy(PKi)}
for all t = ⟨vs, o, e, c, x, vt⟩ ∈ T do           ▷ Step 2: Create the corresponding CA rules for each transition
  if e is ∅ then
    action := x
  else
    action := e
  end if
  if vs is vo then                               ▷ vs is initial state
    if o is ∅ then
      CA := CA ∪ {¬Busy(PK'i) ↦ action()}
    else
      guard := translateOCL(o)
      CA := CA ∪ {guard ∧ ¬Busy(PK'i) ↦ action()}
    end if
  else                                           ▷ vs is not initial state
    if o is ∅ then
      CA := CA ∪ {BAStatus(PKi, vs, 'none') ∧ ¬Busy(PK'i) ↦ action(PKi)}
    else
      guard := translateOCL(o)
      CA := CA ∪ {BAStatus(PKi, vs, 'none') ∧ guard ∧ ¬Busy(PK'i) ↦ action(PKi)}
    end if
  end if
end for
return ⟨DB, CA⟩

```

Step 1: Encoding the states The first step in the algorithm creates a table which will encode the states corresponding to the lifecycle. It will be used to track the evolution of the artifact more easily. Each artifact *BA* will have a table of the following form: *BAStatus(PK_i, state, transition)*, where *PK_i* corresponds to the artifact's primary key and is a foreign key to that artifact, *state* will be any of the states $v \in V$ (except the initial and final states), which correspond to the artifact's subclasses, and *transition* corresponds to *none* or any event $e \in E$ or effect $x \in X$. *transition* is used to indicate the transition that is taking place for an artifact⁵. Its value will correspond to the name of the external event or

⁵Remember that a transition may have an external event. External events are made up of several atomic tasks represented in a state machine diagram, and therefore cannot be considered as atomic.

effect in the transition. Otherwise, when the artifact is not under the effect of any transition, attribute *transition* will have value *none*.

We will also create table $Busy(PK_i)$ which will be used to ensure that, once a transition for an artifact begins executing, no other transitions execute until it finishes. PK_i corresponds to the identifier of the artifact.

Step 2: Encoding the transitions After this, the algorithm iterates over the transitions $t = \langle v_s, o, e, c, x, v_t \rangle \in T$ in the state machine diagram to obtain the condition-action rules that will represent these transitions.

The general form of these rules will be the following:

$$BAStatus(PK_i, v_s, 'none') \wedge guard \wedge \neg Busy(PK'_i) \mapsto action(PK_i), \quad (5.1)$$

where $BAStatus$ refers to the transition's source state and $guard$ to the guard condition o in the transition, if any. The *none* in $BAStatus$ ensures that the artifact is not under the effect of any other transition.

$action$ will correspond to the external event e or effect x in the transition. Notice that $action$ includes as parameters the primary key PK_i of the artifact, as these transitions take place over artifacts that have already been created. We use PK_i to keep track of the artifact's evolution.

$\neg Busy(PK'_i)$ will ensure that, if a transition is taking place, a new transition does not begin execution.

There is one particular case. When the source state of the transition v_s is an initial node, the rule will have the following form:

$$\neg Busy(PK'_i) \mapsto action() \quad (5.2)$$

Notice that in this case $action$ has no input parameters, as the artifact has not been created yet. For the same reason, the condition part of rule 5.2 does not refer to table $BAStatus$.

Below we can see two CA rules that result from the translation of the state machine diagram in Figure 3.5 on page 41:

$$\neg Busy(id') \mapsto RegisterNewBicycle() \quad (5.3)$$

$$BicycleStatus(id, 'Available', 'none') \wedge \neg Busy(id') \mapsto PickUpBicycle(id) \quad (5.4)$$

The first rule corresponds to transition *Register New Bicycle*. It basically states that the transition can be executed at any time as long as no other transition is taking place. The second rule, on the other hand, states that if table $BicycleStatus$ contains an element with values *Available* and *none* (i.e. the

artifact is in state *Available* and not under the effect of any transition, hence the *none*, and no other transition is taking place ($\neg \text{Busy}(id')$), then action *PickUpBicycle* may execute over this artifact, identified by *id*. The remaining rules would be similar to the second and can be found in Appendix B.2 on page 197.

Notice that we should determine the effects of the actions that appear in the CA rules generated by the algorithm. These actions are used to make explicit the implicit connection between the state machine and activity diagrams. That is, they will prepare the DCDSs to execute the actions corresponding to the tasks in the activity diagram. The next section will specify the details of these actions as they will require some tables corresponding to the activity diagrams which have not yet been created.

5.2.3 Translating the Activity Diagrams

After translating the state machine diagram, we will do the same for the activity diagrams. The translation process mirrors that of the state machine diagram.

Algorithm 3 is in charge of this. It has three main steps: creating the database tables to track the evolution of an artifact through the tasks of an activity diagram (i.e. encoding the “states”), translating the associations in the activity diagram to a set of condition-action rules (i.e. encoding the “transitions”), and generating the effects for the initial actions that trigger the execution of the activity diagram.

Step 1: Encoding the “states” Like in the case of state machine diagrams, we will also need tables to keep track of the evolution of an artifact through the tasks in the activity diagram. For each $p \in \mathcal{P}$ we will have a table of the form $\text{tableAD}(PK_i, \text{lastTask})$, where tableAD is the name of p plus the suffix “ing”, PK_i corresponds to the artifact’s identifier and $\text{lastTask} \in \{\text{Tasks}(p) \cup \text{‘none’}\}$. This table will be used to keep track of the last task that has executed in the activity diagram. *none* is used when the activity diagram has not yet begun executing.

Step 2: Encoding the “transitions” Tasks execute in the context of an activity diagram. Given an activity diagram $p = \langle N, n_o, n_f, F \rangle$, a task $t \in \text{Tasks}(p)$ will only execute when the previous task in the diagram has taken place. In addition, if there is a guard condition on the edge leading to the task, this guard condition must also be true. Finally, tasks have a precondition stating the conditions that must be true for the task to execute.

Algorithm 3 *translateADs*(\mathcal{P})

```

DB :=  $\emptyset$ 
actionSet :=  $\emptyset$ 
CASet :=  $\emptyset$ 
for all  $p \in \mathcal{P}$  do
  AD := getName(p)
  tableAD := AD + "ing"
  PKi := getPrimaryKey(getArtifact(p))
  DB := DB  $\cup$  {tableAD(PKi, lastTask)}
  CARules :=  $\emptyset$ 
  actions :=  $\emptyset$ 
  for all  $t \in \text{Tasks}(p)$  do
    pre := translateOCL(t.pre)
    prevTasks := getPreviousTasks(t)
    ADName := getActivityDiagramName(t)
    tableAD := ADName + "ing"
    if  $t$  does not require split then
      for all  $prevT \in prevTasks$  do
        preart := true
        if  $prevT$  is InitialNode then
          Vs := getSourceState(p)
          if Vs contains InitialState then
            preart :=  $\neg BA(PK_i, \dots)$ 
          end if
          prevTName := 'none'
        else
          prevTName := getName(prevT)
        end if
        guard := translateOCL(getGuard(prevT, t))
        CA := {tableAD(PKi, prevTName)  $\wedge$  guard  $\wedge$  preart  $\mapsto$  t(PKi)}
        CARules := CARules  $\cup$  CA
      end for
    else
      for all  $prevT \in prevTasks$  do
        preart := true
        if  $prevT$  is InitialNode then
          Vs := getSourceState(p)
          if Vs contains InitialState then
            preart :=  $\neg BA(PK_i, \dots)$ 
          end if
          prevTName := 'none'
        else
          prevTName := getName(prevT)
        end if
        guard := translateOCL(getGuard(prevT, t))
        CA := {tableAD(PKi, prevTName)  $\wedge$  guard  $\wedge$  preart  $\mapsto$  t1(PKi)}
        CARules := CARules  $\cup$  CA
      end for
    if  $t$  creates a class then
      preimp := generateImpPre(t)
    else
      preimp := true
    end if
    CA := {tableAD(PKi, 't1')  $\wedge$  pre  $\wedge$  preimp  $\mapsto$  t2(PKi)}
    CARules := CARules  $\cup$  CA
  end if
end for

```

► The table will have the name of the AD + "ing"

► Step 1: Create tables for activity diagrams

► Obtains the precondition of the task

► Simple case: Task does not require split

► Will return true if it is a user-made decision

► Complex case: Task requires split

► Iterates over the previous tasks of t

► Obtains the name of the previous task

► Will return true if it is a user-made decision

► CA rule to execute t1

► Generates CA rule to execute t2

```

BA := getArtifact(p)
sourceStates := getSourceStates(p)
action := AD ▷ The action will have the same name as the AD
header := action()
for all  $v_s \in \text{sourceStates}$  do ▷ Step 3: Create SMD actions
  if  $v_s$  is InitialState then
     $fx := \{true \rightsquigarrow \text{tableAD}(\text{getPK}_i(), 'none')\}$ 
     $fx_2 := \{true \rightsquigarrow \text{Busy}(\text{getPK}_i())\}$ 
     $fxSet := fx \cup fx_2$ 
     $\text{firstTask} := \text{getFirstTask}(p)$ 
     $CA := \{\text{tableAD}(\text{PK}_i, 'none') \wedge \neg \text{BA}(\text{PK}_i, \dots) \mapsto \text{firstTask}(\text{PK}_i)\}$ 
     $CASet := CASet \cup CA$ 
  else
     $fx := \{true \rightsquigarrow \text{tableAD}(\text{PK}_i, 'none')\}$ 
     $fx_2 := \{true \rightsquigarrow \text{Busy}(\text{PK}_i)\}$ 
     $fxSet := fx \cup fx_2 \cup \{\text{BAStatus}(\text{PK}_i, v_s, 'none') \rightsquigarrow \text{BAStatus}(\text{PK}_i, v_s, AD)\}$ 
     $fxSet := fxSet \cup \{\text{BAStatus}(\text{PK}'_i, v_s, \text{trans}) \wedge \neg(\text{PK}_i = \text{PK}'_i) \rightsquigarrow \text{BAStatus}(\text{PK}'_i, v_s, \text{trans})\}$ 
  end if
   $\text{action} := \langle \text{header}, fxSet \rangle$ 
   $\text{actionSet} := \text{actionSet} \cup \{\text{action}\}$ 
end for
end for
return  $\langle DB, CASet, \text{actionSet} \rangle$ 

```

Therefore, there are three factors that need to be considered for task execution:

- The task's precondition
- The previous task(s)' execution
- Guard conditions (if there is a decision node)

Therefore, the generic form of the condition-action rules will be the following:

$$\text{tableAD}(\text{PK}_i, \text{prevTName}) \wedge \text{pre} \wedge \text{guard} \wedge \text{pre}_{art} \mapsto t(\text{PK}_i), \quad (5.5)$$

where *tableAD* will keep information about the last task/action that has been executed for the artifact identified by PK_i , *guard* represents the translation of any guard conditions that may result from a decision node, *pre* corresponds to the precondition of task *t*, and pre_{art} represents the condition ensuring that the artifact does not exist, if the task creates an artifact.

Overview on Task Splitting The tasks in our initial model rely heavily on the use of user-provided input parameters. In DCDSs, the way to incorporate fresh values into the system such as user input, is by calling services in \mathcal{F} . This is an important limitation in terms of the translation of our tasks \mathcal{T} into DCDS

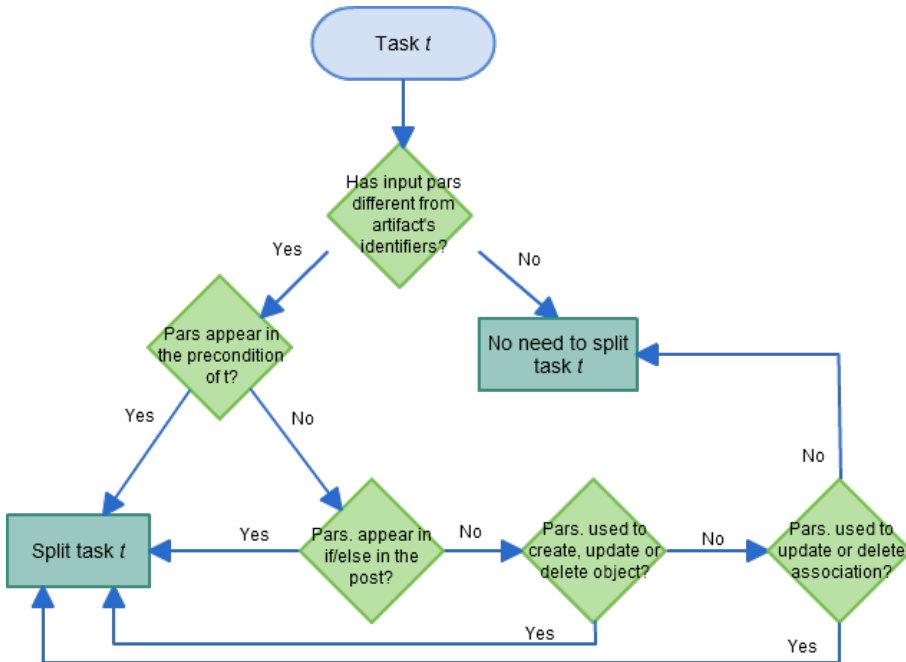


Figure 5.5: Diagram representing the conditions which require a task to be split.

actions α . As effects will be of the form $q_i^+ \wedge Q_i^- \rightsquigarrow Table_r(\dots, f(), \dots)$, where $f()$ is the call to an external service to obtain the user-provided parameters, notice that these values cannot be processed in any way by the task itself, as they are obtained, in a sense, at postcondition time, while in our models they are already available at precondition time.

Therefore, in order to solve this mismatch between BAUML tasks and DCDS actions, what we can do is to define two actions for a single task. Roughly, the first action will be in charge of obtaining the user input, and the second action will perform the changes required by the postcondition. However, we will only do this split for those tasks that need it. Intuitively, this will be when the input parameters (excepting the artifact's identifiers) are required to perform checks at precondition time or make changes to the instances of objects (i.e. creating, updating or deleting them).

Figure 5.5 shows the conditions that determine if a task has to be split into two actions. They are the following:

- The input parameters (different from an artifact’s identifiers) appear in the precondition of task t .
- The input parameters (different from an artifact’s identifiers) appear in an if/else block inside the postcondition of t .
- The input parameters are used to create⁶, update or delete an *object* (i.e. not an artifact).
- The input parameters (different from the artifact’s identifiers) are used to update or delete an association.

For instance, deleting an element requires an effect which performs a selection on all the objects that we want to keep (i.e. those elements that are different from the one we wish to delete). In order to carry out this task, if the required parameters to select the object are provided by the user (and do not correspond to the main artifact’s identifier), we will then have to split the original operation contract into two actions.

Notice that these rules do not apply to artifacts. As we use artifacts to track the evolution through the activity diagrams, the artifact’s identifiers are used as input parameters of every action in the DCDS. Therefore, they are already available at “precondition” time. The only exception is during the creation of an artifact, but in this case the parameters are obtained by the action that triggers the execution of the activity diagram, and used by a CA rule generated by Algorithm 3.

A last consideration that should be made has to do with implicit preconditions. In our BAUML models, we do not require any preconditions in the creation of an instance of a class (be it an artifact or an object), because we assume that there cannot be two elements with different object id (OID) and the same identifier, as stated in the integrity constraints.

However, in the translation process from UML classes to database tables we do not consider OIDs. Therefore, in order to ensure a proper translation under any circumstances, we will have to add an implicit precondition that ensures that there is not already an object with the given identifiers when creating classes.

⁶If the created class is a subclass of an existing superclass, then there is no need to split the task in two actions.

Details of the Algorithm for Step 2 If the task t does not require a split, the resulting rules are simpler. For every previous task of t , the algorithm generates a rule of the form:

$$tableAD(PK_i, prevTName) \wedge pre \wedge guard \wedge pre_{art} \mapsto t(PK_i), \quad (5.6)$$

which we have seen before.

In contrast, if the task needs to be split into two actions, the algorithm generates the following CA rules:

$$tableAD(PK_i, prevTaskName) \wedge guard \wedge pre_{art} \mapsto t1(PK_i) \quad (5.7)$$

$$tableAD(PK_i, 't1') \wedge pre \wedge pre_{imp} \mapsto t2(PK_i) \quad (5.8)$$

There will be as many CA rules of the first type (5.7) as previous tasks are there. On the other hand, the execution of the first action $t1$ can only lead to $t2$, as shown by rule 5.8.

Notice that rule 5.7 includes (if there is one) the translation of the guard condition that may lead to task t . It also includes the precondition required to ensure that there is not artifact with the given identifiers (pre_{art}), if applicable.

On the other hand, the condition part of rule 5.8 includes the translation of the precondition of t and, if there is one, an implicit precondition. The implicit precondition is required when creating a new class, to ensure that there is not an already existing class with the same identifier.

The precondition and implicit precondition are included in the second rule because they may require parameters obtained by action $t1$. Therefore, they would not be available when defining the first rule.

Some condition-action rules that would result from this process in our Bicing example can be seen below. The rules below correspond to the evolution of *Pick Up Bicycle* in Figure 3.7:

$$PickingUpBicycle(id, 'none') \mapsto RequestBicycle(id) \quad (5.9)$$

$$PickingUpBicycle(id, 'RequestBicycle') \mapsto ConfirmPickUp1(id) \quad (5.10)$$

$$PickingUpBicycle(id, 'ConfirmPickUp1') \mapsto ConfirmPickUp2(id) \quad (5.11)$$

$$PickingUpBicycle(id, 'RequestBicycle') \mapsto ConfirmReturn(id) \quad (5.12)$$

The actions on the right-hand side of the rules correspond to the tasks in each of the activity diagrams. The condition on the left-hand side establishes the conditions required for each task to take place.

Including the CA rules for *Pick Up Bicycle* is interesting because the activity diagram has a decision node (see Figure 3.7 on page 44) and task *ConfirmPickUp* is translated into two different actions.

In this particular case, the decision node refers to a user-made decision, and depending on this either *Confirm Pick Up* or *Confirm Return* will execute. As it depends on the user, the condition is not translated and we allow executing either of the corresponding actions once *Request Bicycle* has taken place. Notice that *Get Bicycle*, a material action, does not appear in the translation because it does not make changes to the system.

We also show below the condition-action rule that triggers the execution of *Register New Bicycle*. It is interesting because this external event is in charge of creating the artifact, and therefore the CA rule ensures that no other bicycle exists with the given *id*.

$$\text{RegisteringNewBicycle}(id, 'none') \wedge \neg \text{Bicycle}(id) \mapsto \text{CreateNewBicycle}(id) \quad (5.13)$$

Step 3: Generate the effects for the “initial” actions As transitions in state machine diagrams trigger the execution of activity diagrams, we will need to create an action in the resulting DCDS that represents this implicit connection. That is, we need to state what each of the actions that appear in the CA rules generated by Algorithm 2 does.

Intuitively, the actions do the following:

- Update table *BAStatus* to indicate that a transition is taking place, and therefore that no other transitions can take place simultaneously over the given artifact. The exception to this are the initial transitions (i.e. the ones whose source state is the initial state), because we do not have the identifiers when they begin execution and the artifact does not exist yet.

$$\text{BAStatus}(PK_i, status, 'none') \rightsquigarrow \text{BAStatus}(PK_i, status, AD) \quad (5.14)$$

PK_i corresponds to the primary key of the artifact, $status$ is a variable representing the state of the artifact, and AD refers to the name of the activity diagram p .

- Insert information in table *tableAD* to indicate that the activity diagram can begin its execution (effect 5.15), where *tableAD* corresponds to the name of the activity diagram (which in turn corresponds to an external event or an effect in the state machine diagram) that is being executed. If

it is an initial transition, it will obtain the identifiers of the main business artifact (effect 5.16).

$$true \rightsquigarrow tableAD(PK_i, 'none') \quad (5.15)$$

$$true \rightsquigarrow tableAD(getPK_i(), 'none') \quad (5.16)$$

- Insert information in table $Busy(PK_i)$ to indicate that no other transitions can execute while the current one is taking place.

$$true \rightsquigarrow Busy(PK_i) \quad (5.17)$$

$$true \rightsquigarrow Busy(getPK_i()) \quad (5.18)$$

For instance, in our example, the action corresponding to transition *Pick Up Bicycle* is represented below. The first effect is used to indicate that the bicycle is undergoing a transition. The second effect will be used to indicate that the execution of the tasks in the activity diagram can begin. The third effect will state that there is a transition taking place, so that no other transitions execute in the meantime. The last effect copies the content of $BAStatus$ that have not been modified.

Action *PickUpBicycle(id)*:

$$BicycleStatus(id, 'Av.', 'none') \rightsquigarrow BicycleStatus(id, 'Av.', 'PickingUp') \quad (5.19)$$

$$true \rightsquigarrow PickingUpBicycle(id, 'none') \quad (5.20)$$

$$true \rightsquigarrow Busy(id) \quad (5.21)$$

$$BicycleStatus(id', x, y) \wedge id' \neq id \rightsquigarrow BicycleStatus(id', x, y) \quad (5.22)$$

5.2.4 Translating the Tasks

Each task $t \in \mathcal{T}$ in our model has a header, a precondition and a postcondition. As we have already seen, the header of the action will only contain as input the identifiers of the artifact, and the remaining input parameters will be obtained by means of the effects of an action. We have also dealt with the precondition, which is incorporated into the CA rules. In this section we will see how the postcondition is transformed into a set of effects of an action. The steps are the following:

1. Determine what the task does: what classes and associations are created, deleted or modified by the task.

2. Translate the postcondition of the task accordingly.
3. Add the necessary effects to ensure proper evolution of the DCDS. This will include the rules that copy all the content that is not modified to avoid its loss.

Step 1: Identify what the tasks do The first step identifies the following OCL expressions in the postconditions of the tasks in order to determine what they do. We do so by following [105]. In this work, expressions are classified according to whether they create or delete a class or an association. Updates are considered as deletions followed by creations.

Once we have identified the expressions that create, delete and update the artifacts and their relationships in UML, we can determine how they can be translated into a DCDS.

Notice that in this step we will also find out whether the task requires to be split into two different actions or if it is not necessary, as shown on Figure 5.5. If it is the case that it requires two different actions, the next step in the translation process will have to deal with this.

For example, let's look at the operation contract for *RequestBicycle*, first presented on page 45. For easier readability, we include it again here (simplified with no attributes):

Listing 5.1: Code for task *RequestBicycle*

```
operation requestBicycle(b: Bicycle)
pre: -
post: b.oc1IsTypeOf(InUse) and not b.oc1IsTypeOf(Available)
```

The first part of the postcondition, `b.oc1IsTypeOf(InUse)`, creates an element of type *InUse*. The second part of the postcondition, `not b.oc1IsTypeOf(Available)`, does the opposite: it deletes `b` as an *Available* bicycle.

Step 2: Translate the task Looking at the operation contract above, one can notice that the input parameter does not have a simple type (i.e. such as `String`, `int`, etc.), but has instead the type of a class: artifact *Bicycle*, in this particular case. We use this method of specification for every operation contract that does not create an instance of a class. This simplifies the operation contracts' specification and makes them more readable. Note that it is equivalent to using the identifiers for the specification.

As DCDSs do not deal with objects, we will eventually substitute the object by its identifier (or primary key) when translating these tasks.

The translation of the task has two steps. The first step will obtain the effects for the first action if the task requires a split. The second step translates into effects the meaning itself of the task's postcondition. Algorithm 4 shows the main steps in this translation process, although the details for the translation of the postconditions have been abstracted away.

Algorithm 4 *translateTasks*(\mathcal{T})

```

actions :=  $\emptyset$ 
DBSchema :=  $\emptyset$ 
header :=  $t(PK_i)$  ▷ Creates header for action t
for all  $t \in \mathcal{T}$  do
  if t requires split then
    DBSchema := DBSchema  $\cup$  OutT1( $PK_i, att_j$ )
    header1 :=  $t1(PK_i)$  ▷ Creates action t1
    body1 := {true  $\rightsquigarrow$  OutT1( $PK_i, getAtt_j()$ )}
    body1 := body1  $\cup$  {true  $\rightsquigarrow$  tableAD( $PK_i, 't1'$ )}
    action1 := (header1, body1)
    actions := actions  $\cup$  action1
    header :=  $t2(PK_i)$ 
  else
    header :=  $t(PK_i)$  ▷ Creates header for action t2
  end if
  body := translatePost( $t.postcondition$ ) ▷ Returns a set of effects
  action := (header, body) ▷ Creates t or t2 with the translation of the postcondition
  actions := actions  $\cup$  action
end for
return (DBSchema, actions)

```

Step 2.1. Create additional action (if needed) In the case that the task is split into two actions, Algorithm 4 generates the effects for action $t1$:

$t1(PK_i)$:

$$true \rightsquigarrow OutT1(PK_i, getPars_j()) \quad (5.23)$$

$$true \rightsquigarrow tableAD(PK_i, 't1') \quad (5.24)$$

The first effect obtains the user input ($getPars_j()$) and stores it in table *OutT1*. Notice how we use PK_i to track the evolution through the various tasks/actions. The second effect updates *tableAD* to indicate that $t1$ has executed successfully.

Below we show the effects for action *ConfirmPickUp1(id)*. Note that *ConfirmPickUp* requires two actions when translated because it creates an association class, *BicycleRental*.

ConfirmPickUp1(id) :

$$true \rightsquigarrow OutConfirmPickUp1(id, getUserID()) \quad (5.25)$$

$$true \rightsquigarrow PickingUpBicycle(id, 'ConfirmPickUp1') \quad (5.26)$$

The first effect obtains the parameter corresponding to the user ID and the second effect advances the execution through the activity diagram.

Step 2.2: Translate the postconditions of the task After determining if the tasks require to be split into two actions and identifying what they do, the corresponding effects for the actions can be created.

Creation of a Class When a class is created in a class diagram (this includes association classes), we need to obtain values for its attributes and we will also create the required associations with other classes. There are two possibilities: the creation of the class is specified using a single action, or the creation of the class is specified using two actions.

The first case will be when we deal with a change of subclass in a hierarchy or the creation of an artifact. Unless there are other reasons for splitting the task (such as a precondition that uses the input parameters), we will have only one action for the task.

Creation of a subclass (not an artifact hierarchy, when the superclass already exists):

$t(PK_i)$:

$$q_i^+ \wedge Q_i^- \rightsquigarrow C(getPK_c(), getAtt_k()) \quad (5.27)$$

$$q_i^+ \wedge Q_i^- \rightsquigarrow Rel(getPK_c(), getAtt_l()) \quad (5.28)$$

Creation of an artifact or one of its subclasses:

$t(PK_i)$:

$$q_i^+ \wedge Q_i^- \rightsquigarrow C(PK_i, getAtt_k()) \quad (5.29)$$

$$q_i^+ \wedge Q_i^- \rightsquigarrow Rel(PK_i, getAtt_l()) \quad (5.30)$$

$q_i^+ \wedge Q_i^-$ will be *true*, if there is no if/else block, or the translation of the conditions in the if/else block following [106]. C corresponds to the table of the artifact or subclass that is created. In the case of a subclass which is not part of an artifact hierarchy, $getPK_c()$ will obtain the identifiers of the class that is created. Otherwise, the identifiers of the class will correspond to PK_i , which

is the input parameter of the action t . $getAtt_k()$ represents additional attributes of C , and $getAtt_l()$ corresponds the identifiers of the other class(es) taking part in the relationship Rel .

The first effects (5.27,5.29) correspond to the creation of the class, the second effects (5.28,5.30) correspond to the creation of the relationships between the created class and other classes.

For instance, as we have seen, the operation contract in our previous example (task *RequestBicycle*) creates a subclass, in this case it is part of an artifact hierarchy. Therefore, it would be translated in the effect below:

$$true \rightsquigarrow InUse(id) \quad (5.31)$$

The second case will deal with the creation of objects (i.e. classes that are not artifacts). In the case of artifacts, the identifiers and relevant attributes will have been obtained previously by the action corresponding to the state machine diagram.

$t2(PK_i)$:

$$OutT1(PK_i, PK_c, att_j) \wedge q_i^+ \wedge Q_i^- \rightsquigarrow C(PK_c, att_k) \quad (5.32)$$

$$OutT1(PK_i, PK_c, att_j) \wedge q_i^+ \wedge Q_i^- \rightsquigarrow Rel(PK_c, att_l) \quad (5.33)$$

$OutT1$ will be filled by the previous task, $t1$, which will be in charge of obtaining the parameters, as we have seen in the previous subsection. $q_i^+ \wedge Q_i^-$ will be *true*, if there is no if/else block, or the translation of the conditions in the if/else block following [106], like previously. PK_i corresponds to the primary key of the artifact, PK_c represents the primary key of the class we are creating, C , and $att_k \subseteq att_j$ will obtain the relevant attributes. Rel represents a table holding the information that corresponds to an association between C and another class, which is accessed by means of its role name in the OCL code. att_l represents the primary key of the class that is related to C , and $att_l \subseteq att_j$.

Below we show the remaining translation of task *ConfirmPickUp* into an action. In this case we show how *BicycleRental* is created using the parameters stored in table *OutConfirmPickUp*.

$ConfirmPickUp2(id)$:

$$OutConfirmPickUp1(id, uid) \rightsquigarrow BicycleRental(id, uid)$$

Creation of an Association If the task t only creates an association, then the parameters can be obtained directly in the action:

$$q_i^+ \wedge Q_i^- \rightsquigarrow Rel_{role_k}(getAtt_k(), getAtt_l()) \quad (5.34)$$

Like in the previous case, $q_i^+ \wedge Q_i^-$ will correspond to the translation of the conditions in the if/else block, if there is any. $getAtt_k(), getAtt_l()$ correspond to the functions that will obtain the identifiers of the elements that take part in the relationship.

Deletion Due to the frame problem, when we wish to delete information in a DCDS we need to copy everything except what we wish to delete. For this reason, the tasks that delete information have to be split into two different actions. The first task will be in charge of obtaining the identifiers of the elements that are deleted, as we have already seen, and the second task will perform the actual changes. This applies to both classes and associations. The exception to this are artifacts: the split will not be necessary to delete an artifact as their identifiers are part of the action's parameters.

Deletion of a Class We will first begin by looking at the deletion of an artifact or any of its subclasses. In any case, the deletion of a superclass implies the deletion of all of this subclasses. The translation of the OCL expressions would be the following:

$t(PK_i)$:

$$Artifact(PK'_i, \dots) \wedge PK'_i \neq PK_i \wedge q_i^+ \wedge Q_i^- \rightsquigarrow Artifact(PK'_i, \dots) \quad (5.35)$$

$$Rel(\dots, PK'_i, \dots) \wedge PK'_i \neq PK_i \wedge q_i^+ \wedge Q_i^- \rightsquigarrow Rel(\dots, PK'_i, \dots) \quad (5.36)$$

Artifact is the table that represents the artifact or the artifact's subclass, PK_i corresponds to the artifact identifier given as input, and $q_i^+ \wedge Q_i^-$ to an if/else condition. If there is not any, then it has a value of *true*. The first effect (5.35) will delete the artifact with the given identifier. The second effect (5.36) will delete any instance of a relationship *Rel* in which the artifact participates. Therefore, we will need as many instances of this effect as relationships in which the artifact participates.

This is, of course, assuming that the task does not require to be translated into two actions for another reason.

Returning to our example, apart from creating an instance of *InUse*, *Request-Bicycle* also deletes an instance of *Available*. Consequently, we would need to add the following effect to the action:

$$Available(id') \wedge id \neq id' \rightsquigarrow Available(id') \quad (5.37)$$

In addition, the deletion of an instance of *Available* also implies the deletion of the relationships in which it took part. In this particular case, we need to

delete an instance of association *AvailableIsIn*:

$$\begin{aligned} & AvailableIsIn(id', apN, station) \wedge id \neq id' \\ & \rightsquigarrow AvailableIsIn(id', apN, station) \end{aligned} \quad (5.38)$$

Then, the resulting action *RequestBicycle(id)* would have the following effects:

$$true \rightsquigarrow InUse(id) \quad (5.39)$$

$$Available(id') \wedge id \neq id' \rightsquigarrow Available(id') \quad (5.40)$$

$$AvailableIsIn(id', apN, apS) \wedge id \neq id' \rightsquigarrow AvailableIsIn(id', apN, apS) \quad (5.41)$$

On the other hand, the translation of the OCL expressions for the deletion of an object (i.e. not an artifact) would be the following:

$$\begin{aligned} & C(PK_c, \dots) \wedge OutT1(PK_i, att_c, att_k, \dots) \wedge PK_c \neq att_c \\ & \wedge q_i^+ \wedge Q_i^- \rightsquigarrow C(PK_c, \dots) \end{aligned} \quad (5.42)$$

$$\begin{aligned} & Rel(\dots, PK_c, \dots) \wedge OutT1(PK_i, att_c, att_k, \dots) \wedge \\ & PK_c \neq att_c \wedge q_i^+ \wedge Q_i^- \rightsquigarrow Rel(\dots, PK_c, \dots) \end{aligned} \quad (5.43)$$

PK_c corresponds to the identifier(s) of C , att_c contains the identifiers of the particular instance of C which needs to be deleted and att_k may contain additional parameters to perform other changes to the system. $q_i^+ \wedge Q_i^-$ corresponds to the translation of an if/else condition, if there is any.

The first effect (5.42) corresponds to deleting the instance of the class. The second effect (5.43) deletes the associations or relationships in which C participates.

Deletion of an Association Below we show the form of the effect that deal with the deletion of an association.

$$\begin{aligned} & Rel(\dots, att'_j, att'_k, \dots) \wedge OutT1(PK_i, att_j, att_k) \wedge att'_j \neq att_j \\ & \wedge att'_k \neq att_k \wedge q_i^+ \wedge Q_i^- \rightsquigarrow Rel(\dots, att'_j, att'_k, \dots) \end{aligned} \quad (5.44)$$

att_j and att_k correspond to the identifiers of the association and Rel corresponds to the table that represents the association. In the case of associations in which the primary key of the artifact adds as an identifier, we would omit att_j . Like in the case of the creation, if the deletion is inside an if/else block, then the translation of the condition of the if/else should be added to the $q_i^+ \wedge Q_i^-$ part of the effects.

Step 3: Ensure proper evolution So far we have only translated the content of a task t into a DCDS action. In every action we also need to make additional changes to other tables so that the DCDS can evolve properly. We distinguish two cases:

- If t is the last task in the activity diagram AD :
 - t will delete the row in the table that corresponds to the activity diagram AD with the primary key of the artifact that has been manipulated.
 - t will change table $BAStatus$:
 $BAStatus(pk, oldState, x) \rightsquigarrow BAStatus(pk, newState, 'none')$, unless t deletes the artifact.
 - t will delete the contents of table $Busy$, to enable the execution of another transition.
 - It will add $\langle a, b \rangle$ to $aux(x, y)$, to ensure that equality constraints are checked:
 $true \rightsquigarrow aux(a, b)$
- Otherwise:
 - t will change the row representing the execution of the activity diagram AD , to indicate that the task has already been executed:
 $true \rightsquigarrow tableAD(PK_i, tName)$,
 where $tableAD$ corresponds to the name of the table that tracks the evolution through the activity diagram, and $tName$ corresponds to the name of task t .

In our example *RequestBicycle* is the first task in the activity diagram. Therefore, we will only need to add the effect to ensure the proper evolution through the tasks in the diagram:

$$PickingUpBicycle(id, 'none') \rightsquigarrow PickingUpBicycle(id, 'RequestBicycle')$$

The final action would look like this:

RequestBicycle(id) :

$$true \rightsquigarrow InUse(id) \quad (5.45)$$

$$Available(id') \wedge id \neq id' \rightsquigarrow Available(id') \quad (5.46)$$

$$AvailableIsIn(id', apN, apS) \wedge id \neq id' \rightsquigarrow AvailableIsIn(id', apN, apS) \quad (5.47)$$

$$PickingUpBicycle(id, 'none') \rightsquigarrow PickingUpBicycle(id, 'RequestBicycle') \quad (5.48)$$

Logically, *RequestBicycle* would have to copy the contents of all the other tables that are not modified by the action, including table *Busy*.

5.2.5 Summary & Overview

In this section we have explained the translation process from the point of view of the source models in the BAUML framework: the class, state machine, activity diagrams and the tasks. As a summary, this subsection looks at the elements involved in the translation process from the point of view of the target model, the DCDSs. That is, for each element (database schema, condition-action rules and actions) we show the elements and the algorithms involved in the process of obtaining it.

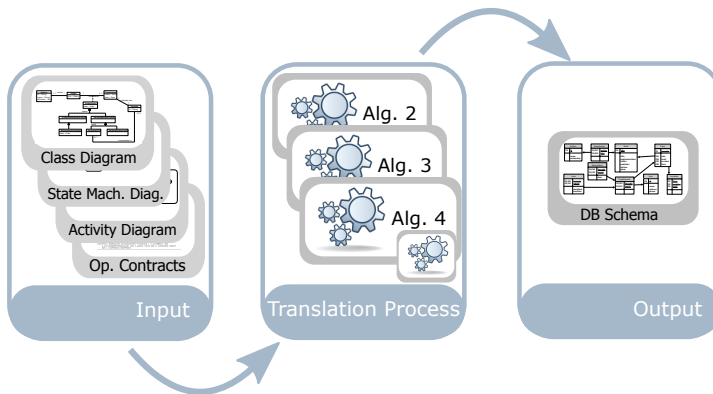


Figure 5.6: Overview of elements involved in obtaining the database schema of a DCDS

To begin with, Figure 5.6 shows the elements involved in the translation process to obtain the database schema. All the elements in the BALSAs framework play a role in obtaining the database schema and Algorithms 2 to 4 are involved in this process. There are additional steps in the translation process that are outside the scope of the algorithms mentioned.

Figure 5.7 does the same for the condition-action rules. In this case, they are obtained from the state machine and activity diagrams, and the tasks. Algorithms 2 and 3 are in charge of this.

Finally, Figure 5.8 focuses on the actions. They are obtained from the state machine and activity diagrams, and the tasks. Algorithms 3 and 4 are the ones

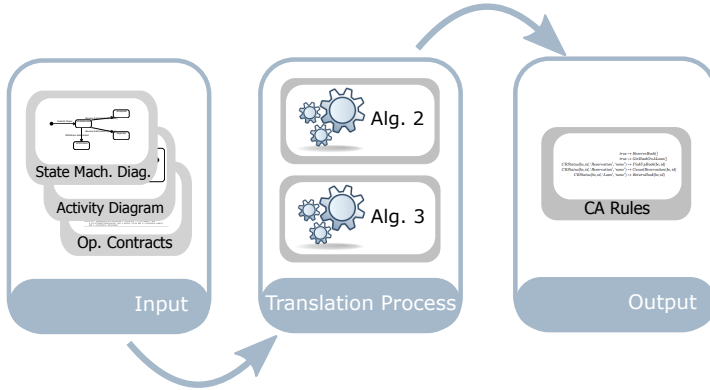


Figure 5.7: Overview of elements involved in obtaining the condition-action rules of a DCDS

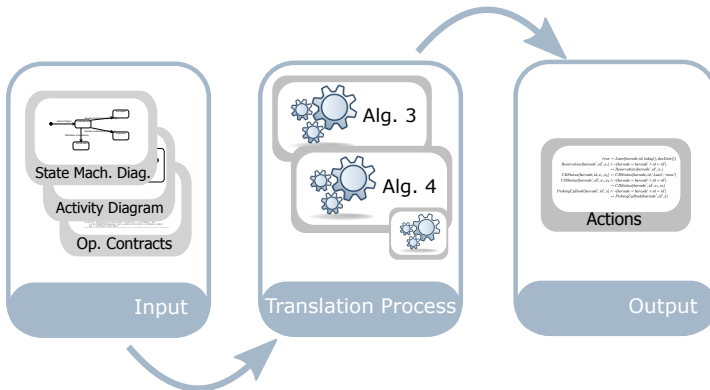


Figure 5.8: Overview of elements involved in obtaining the actions of a DCDS

responsible for performing this translation, together with other steps that are outside the scope of these algorithms.

5.3 Reasoning with the Resulting DCDS

Once we have obtained the DCDS, we may use it to reason about the original artifact-centric BAUML model. The reasoning we can perform is semantic: it is aimed at validating whether the BPM satisfies the business requirements. This is achieved by allowing the designer to ask questions about the DCDS (i.e. properties pertaining to the proper operation of the system) and check if the results correspond to those expected.

These properties have to be defined in logic to be able to check them. Ideally, they should be obtained automatically from the model. For this reason, after introducing the logic that we will use to define the properties, we present some generic properties that should be interesting to check in any model and could be generated automatically.

5.3.1 Verification Logic

First of all, we have to determine whether the services in the DCDS are deterministic or non-deterministic, as the type of logic that we will use to define the properties depends on this. Considering that all business process models require user intervention at some point, the services in the DCDS are non-deterministic. For this reason, we will have to define the properties that we wish to check using $\mu\mathcal{L}_p$, a fragment of μ -calculus.

μ -calculus distinguishes between properties that refer to the current state or its immediate successors, and properties that refer to arbitrarily far away states. $\mu\mathcal{L}_p$ restricts μ -calculus by assuming that quantification is limited to elements that are present in the current database, and these elements must continuously persist in the system in order for the quantification to take effect [11].

We summarize here the main aspects of $\mu\mathcal{L}_p$ [11], contextualizing it to the case of BAUML models. Given a BAUML model \mathcal{B} , the logic $\mu\mathcal{L}_p$ is defined as:

$$\begin{aligned} \Phi ::= & Q \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\text{LIVE}(x) \wedge \Phi \mid \\ & \text{LIVE}(\vec{x}) \wedge \langle \rightarrow \rangle \Phi \mid \text{LIVE}(\vec{x}) \wedge [-]\Phi \mid Z \mid \mu Z.\Phi \end{aligned}$$

where Q is a possibly open FO query, Z is a second order predicate variable, and the following assumption holds: in $\text{LIVE}(\vec{x}) \wedge \langle \rightarrow \rangle \Phi$ and $\text{LIVE}(\vec{x}) \wedge [-]\Phi$, the variables \vec{x} are exactly the free variables of Φ , once we substitute to each bounded predicate variable Z in Φ its bounding formula $\mu Z.\Phi'$ [11]. This

requirement expresses that $\mu\mathcal{L}_p$ quantifies only over those objects/artifacts that persist in the system, i.e., continue to stay in the active domain of the system.

We use the following abbreviations:

- $\Phi_1 \vee \Phi_2 = \neg(\neg\Phi_1 \wedge \neg\Phi_2)$,
- $[\neg]\Phi = \neg(\neg)\neg\Phi$,
- $\nu Z.\Phi = \neg\mu Z.\neg\Phi[Z/\neg Z]$,
- $\forall x.A(x) \rightarrow \Phi = \neg(\exists x.A(x) \wedge \neg\Phi)$,
- $\text{LIVE}(\vec{x}) \rightarrow \langle \neg \rangle \Phi = \neg(\text{LIVE}(\vec{x}) \wedge [\neg]\neg\Phi)$,
- $\text{LIVE}(\vec{x}) \rightarrow [\neg]\Phi = \neg(\text{LIVE}(\vec{x}) \wedge \langle \neg \rangle \neg\Phi)$.

The last two abbreviations show that $\mu\mathcal{L}_p$ allows one to “control” what happens when quantification ranges over a value that disappears from the current active domain: in the \rightarrow case the property trivializes to *true*, in the \wedge case it trivializes to *false*.

Formally, and knowing that the DCDS has nondeterministic services, the satisfaction of a certain property by a DCDS can be defined as follows. Given a DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$, with a data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, I_0 \rangle$ (notice that it includes an initial database instance I_0) and a process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \rho \rangle$, and a property Φ expressed in $\mu\mathcal{L}_p$, we say that \mathcal{S} verifies Φ if $\Upsilon_s \models \Phi$, where Υ_s corresponds to the transition system of the DCDS \mathcal{S} . A transition system represents all the possible executions of the process layer \mathcal{P} over the data layer \mathcal{D} in \mathcal{S} .

Although each model will have its own particular set of properties that need to be checked, there are also some generic properties that are applicable to any model. These properties could eventually be automatically generated. We will focus on checking the proper evolution of an artifact and the executability of services.

5.3.2 Evolution of an Artifact

Checking the correct evolution of an artifact is key, as this ensures that the artifact will be able to represent reality correctly. We can say that an artifact evolves properly when, after it is created, it can be deleted from the system (if the model allows it) or it reaches one of the last states in the state machine diagram (i.e. it reaches a state from which it cannot evolve anymore). In generic form, this property would be defined in the following way:

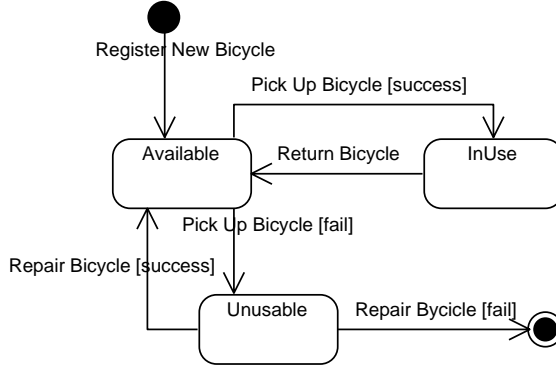


Figure 5.9: State diagram of *Bicycle* from our example on page 41.

$$\begin{aligned} & \nu X. (\forall id_i. (BAStatus(id_i, 'first', 'none') \wedge BA(id_i) \rightarrow \\ & \quad \mu Y. (BAStatus(id_i, 'last', 'none') \vee (BA(id_i) \wedge \langle - \rangle Y))) \wedge [-] X \end{aligned} \quad (5.49)$$

$$\begin{aligned} & \nu X. (\forall id_i. (BAStatus(id_i, 'first', 'none') \wedge BA(id_i) \rightarrow \\ & \quad \mu Y. (BAStatus(id_i, 'last', 'none') \vee (BA(id_i) \rightarrow \langle - \rangle Y))) \wedge [-] X \end{aligned} \quad (5.50)$$

The first property corresponds to the case in which artifacts are not destroyed, whereas the second considers the possibility that an artifact may be deleted from the system. Notice that any system that fulfills the first property will also fulfill the second; however it is best to use the most precise rule in each case.

Although this is an interesting property, in our example is not applicable, as the state machine diagram is not acyclic (see Figure 5.9) and there is no last state as defined above.

However, there is a similar property in our example that we could check: an *Unusable* bicycle should eventually be repaired successfully or destroyed. Below we show the definition of this property:

$$\begin{aligned} & \nu X. (\forall id. BicycleStatus(id, 'Unusable', 'none') \wedge Bicycle(id) \rightarrow \\ & \quad \mu Y. (BicycleStatus(id, 'Available', 'none') \vee (Bicycle(id) \rightarrow \langle - \rangle Y))) \wedge [-] X \end{aligned} \quad (5.51)$$

The result of model checking in this case would be a positive answer showing that this requirement is already satisfied by the UML artifact-centric

BPM specification of our example. Notice that the property refers to table *BicycleStatus* and not to the tables representing the artifacts themselves. This is necessary to guarantee that the whole activity diagram has been executed correctly; that is, to ensure that the property is evaluated when no transition is taking place.

However, checking that an artifact evolves correctly does not always guarantee that all the artifact's states are reachable. For instance, in the previous example, the property only ensures that an artifact that reaches state *Unusable* will eventually reach state *Available* or be destroyed, but it does not ensure that the artifact can become *InUse*. For this reason, it is also interesting to check the reachability of a particular state given a certain state. The generic form of this property is the following:

$$\begin{aligned} \mu X. (\exists id_i. BAStatus(id_i, 'source', 'none') \wedge BA(id_i) \wedge \\ \mu Y. (BAStatus(id_i, 'target', 'none') \vee (BA(id_i) \wedge \langle - \rangle Y))) \vee \langle - \rangle X \end{aligned} \quad (5.52)$$

The property above states that at least there is an artifact in state *SourceState* that eventually changes into state *TargetState*.

For instance, in our example, we may want to check if it possible to reach state *InUse* from *Available*. The property would be defined in the following way:

$$\begin{aligned} \mu X. (\exists id, c, t. BicycleStatus(id, 'Available', 'none') \wedge Bicycle(id) \\ \wedge \mu Y. (BicycleStatus(id, 'InUse', 'none') \vee \\ (Bicycle(id) \wedge \langle - \rangle Y))) \vee [-] X \end{aligned} \quad (5.53)$$

In this case, the model check would return also a positive answer, as state *InUse* can be reached by executing activity diagrams *RegisterNewBicycle* and *RequestBicycle*.

5.4 Summary & Conclusions

In this chapter we have presented a way to translate a BAUML model into a data-centric dynamic system (DCDS) [11], with the goal of checking its correctness. As DCDS are grounded on logic, they provide the ability to perform model checking on them for this purpose. DCDSs are able to represent both the static and the dynamic aspects of the initial UML models. Finally, we have outlined the definition of several generic properties that it would be desirable to check in any artifact-centric business process model.

This work helps to bring together a high-level specification, useful for business people, but which is not possible to check, and low-level specifications, impractical in the business world but whose correctness can be verified.

Chapter 6

Reasoning in Practice: AuRUS-BAUML

In the previous chapter we have seen a way to reason with our BAUML models by translating them into a DCDS and applying model checking techniques to perform the correctness tests.

Unfortunately, to the best of our knowledge, there is no tool that can perform the reasoning using DCDSs, although some work has been done on implementing the DCDSs [30]. For this reason, this chapter presents a tool which is able to perform several semantic tests to check the correctness of the BAUML model.

The modeler begins by defining the BAUML model graphically using an open-source tool, ArgoUML [7] and then exporting it as an XMI file. This XMI file is provided to *AuRUS-BAUML*, a tool that we have implemented and which is able to translate a BAUML model into logic and automatically perform several semantic tests. Internally, this tool uses another tool *SVTe*, which given a certain property over the logic schema (i.e. the result of the translation) can tell whether this property can be fulfilled. Following this workflow, the user can answer several questions which deal with the semantic correctness of the initial model.

It is important to point out that our reasoning approach works only with the specification of the model and does not need an initial instance of the model to obtain results. DCDSs, on the other hand, do require an initial instance of the database schema. Secondly, there are two possible outputs of the reasoning process. On the one hand, if the property can indeed be fulfilled by the initial

model, the result will be a sample instantiation. On the other hand, if it is not possible to fulfill the given property, the result will be the list of constraints that are preventing it. Logically, we consider all the elements in the BAUML framework when reasoning.

This chapter begins by introducing several properties of interest to ensure the semantic correctness of the BAUML model. Afterwards, we present the tool and its internal workings in more detail. We then specify the translation process which is performed by AuRUS-BAUML and formalize the properties introduced earlier. The chapter finishes by pointing out some conclusions and further work.

6.1 Checking the Semantic Correctness of BAUML Models

As we mentioned in the Introduction to this thesis, semantic correctness ensures that the model represents the domain correctly. More specifically, we distinguish between two types of semantic correctness: *verification* and *validation*. *Verification* looks for inherent errors in the model, answering the question “*Is the model right?*”, whereas *validation* ensures that the model represents the domain appropriately, answering the question “*Is it the right model?*”.

In terms of automation, the main difference between verification and validation is that verification can be performed without user intervention, as it looks for errors and contradictions within the model. In contrast, validation requires a user to ensure that reality is represented correctly in the model. As we shall see, there are some validation properties that can be generated and checked automatically, but it is the user’s decision to determine whether the fulfillment of these properties is actually right or not.

The verification and validation properties that we present in this section are based on or inspired by the following works: [105, 114, 129, 106]. This section is not meant to be an exhaustive list of all the necessary tests to ensure semantic correctness, but rather an illustrative overview of the kind of tests that can be performed.

6.1.1 Verification

There are several verification properties that can be checked in a BAUML model. We have divided these properties according to the dimension of the BAUML model they focus on, although all the dimensions are taken into consideration in the reasoning process.

The Class Diagram in a BAUML Model

We present three different properties that can be checked over the class diagram in a BAUML model: liveness of the classes and associations, the correctness of the cardinalities in the relationships, and the redundancy of integrity constraints.

Liveness of the classes and associations Checking that each class and association in the diagram is lively ensures that there can exist at least one instance of each of the classes and associations. Having a class or an association which cannot be instantiated implies that there is some mistake in the class diagram, as it does not make sense to have an element for which no instances can exist.

For instance, in the Bicing example presented in Chapter 3 on page 37, we could check if *Bicycle* or *BicycleRental* are lively.

Correctness of minimum and maximum cardinalities Cardinalities in the class diagram may contain errors. In the case of minimum cardinalities, it may be the case that the bound is actually higher than the one stated. The opposite may also hold for maximum cardinalities: the bound may be actually lower than the one that appears in the diagram.

For instance, in the class diagram corresponding to the Bicing example with two artifacts (see page 52), we could check if the maximum cardinality in the relationship between artifacts *Active* user and *InUse* bicycle is really three or is actually lower.

Redundancy of integrity constraints Although this is not strictly a semantic correctness property by itself, ensuring that the model avoids redundancy and is minimal are also correctness criteria. An integrity constraint is redundant with another when the fulfillment of the first constraint always implies the fulfillment of the second.

For example, we could check if any of the integrity constraints ensuring the correctness of the dates is redundant (on both versions of Bicing).

The State Machine Diagram in a BAUML model

There are three main properties of interest regarding state machine diagrams: state reachability, event applicability and event executability.

State reachability State reachability ensures that every state in which an artifact may be can eventually be reached. As each of the states in the state machine diagram has its corresponding subclass in the class diagram, it is equivalent to checking the liveness of each of the subclasses of the artifact.

For instance, we could check if states *Available*, *Unusable* and *InUse* are reachable in the state machine diagram of *Bicing* on page 41.

Transition applicability Transition applicability ensures that the required conditions are met for an external event or an effect to execute. Note that this property does not ensure that the transition executes successfully, but rather that the conditions can be met for it to begin its execution.

We will consider both external events and effects in the transition as black boxes; therefore, the conditions for applicability will only take into consideration the following elements:

- The source state of the transition
- The OCL condition in the transition

Logically, if the source state is the initial state, then there is no restriction in terms of the source state.

In our *Bicing* example, we could check if transitions *Register New Bicycle* or *Return Bicycle* meet the conditions for their execution (see the activity diagrams on pages 43 and 44).

Transition executability Transition executability will ensure that every transition in the state machine diagram can execute successfully. This will happen when the transition itself is applicable and after its execution it leaves the system in a state that fulfills all the integrity constraints.

Like in the previous case, we could check the executability for transitions *Register New Bicycle* or *Return Bicycle* in our example.

Activity Diagrams and Operation Contracts in a BAUML Model

We consider activity diagrams and the operation contracts representing the tasks together in this section as they are closely interrelated. The activity diagrams merely establish the order for the execution of the tasks, and it is the tasks themselves the ones that contain the semantic information.

Similarly to the case of the state machine diagram, in this instance we can also consider the applicability and the executability of the tasks.

Applicability of the tasks We consider that a task is applicable if the previous task has executed successfully and its precondition is met. If there is an OCL condition in one of the edges leading to the task, it will also have to be taken into consideration to study its applicability.

We could check the applicability of tasks *Confirm Pick-Up* and *Confirm Return* in the activity diagram in Figure 3.7 on page 44.

Executability of the tasks A task will be executable if it is applicable and its postcondition can be met. Please note that, since the integrity constraints are not checked until the end of the activity diagram execution, in most cases the tasks will be executable even if eventually (i.e. at the end of the activity diagram execution) they lead to an integrity constraint violation.

Like in the previous case, we could also check the executability of tasks *Confirm Pick-Up* and *Confirm Return*.

Some Thoughts on Verification Tests

Although we have classified the different tests according to the dimension that they check, in practice these tests are not completely independent from each other. This subsection gives an overview of how they are related (see Table 6.1) and how the results can be interpreted.

Test 1	Relationship	Test 2
State reachability	is equivalent to	Class liveness
Transition executability	implies	Transition applicability
Operation executability	implies	Operation applicability
Transition executability	implies	Event/Action executability

Table 6.1: Overview of the relationships between tests.

To begin with, the **state reachability** test is **equivalent** to the **class liveness** test. As we have explained in Chapter 3, each of the states in the state machine diagram corresponds to a subclass of the artifact in the class diagram. Therefore, checking if state s is reachable is actually the same as checking if there exists an instance of class c_s , where c_s is the subclass that corresponds to s .

On the other hand, the **executability tests** have an **implies** relationship with the **applicability tests** over the same element. That is, if a transition or an

operation is executable, then this implies that the transition or the operation (respectively) is applicable. By the logic equivalence rules, if the transition or operation is *not* applicable, then it is *not* executable either.

Moreover, notice that if a **transition is executable**, then the **external event or action** in the transition is also **executable**. If two different transitions $t1$ and $t2$ have the same event, it may be the case that one of them, let's say $t1$, is applicable (executable) and the other, $t2$, is not. This may be due to different conditions in the transition that in the case of $t1$ allow the execution of the event or action and in the case of $t2$ they do not.

6.1.2 Validation

Validation tests deal with the adequacy of the model in terms of representing the reality appropriately. Therefore, they can be used to check if the business process meets the requirements. However, due to this, it is more difficult to define tests that are valid for any BAUML model. What we do in this section is present some tests to detect potential errors, but it is ultimately the modeler's or user's responsibility to interpret the results.

User-defined Tests Allowing the user to define his or her own tests can be useful to ensure that the model fulfills the requirements elicited in the early stages of the process definition. This includes the ability to ensure that business rules, which are closely related to business goals [72, 71], have been incorporated correctly in the specification of the business process.

For instance, we could have the following requirements in our example:

- Blacklisted users cannot rent any bicycles.
- There cannot be two anchor points with the same number, even if they belong to different stations.

User-defined tests would then allow us to check that these properties are fulfilled.

Path Inclusion or Exclusion When there are two different associations which link exactly the same classes, in some cases one relationship should be subset of the other. In other cases the two paths should be mutually exclusive. Therefore, checking these properties can provide information to the user about a potential error.

A very intuitive example to illustrate these tests is based on a company with employees and departments. See Figure 6.1. Employees work in departments, every department has a manager and departments may be audited by one or several employees to ensure they work properly.

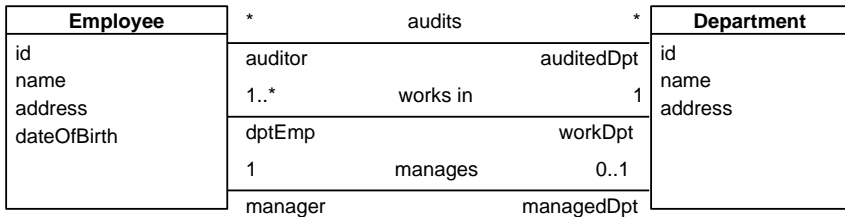


Figure 6.1: Class diagram showing an Employee and Department with several associations between these two classes.

Logically, the manager of a department should be one of its employees. On the other hand, an employee auditing a department should not work in it, to preserve the independence of the audit. The first example is a case of path inclusion, whereas the second corresponds to a case of path exclusion.

Missing irreflexive constraints For those associations which relate the same class to itself, in many instances there may be an irreflexive constraint missing: that is, one instance of a class cannot be related to itself.

A typical example in which an irreflexive constraint is necessary is that of marriages. See the small class diagram in Figure 6.2. A person may be married (or not) to another person, but a person cannot be married to himself or herself.

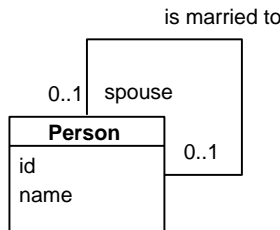


Figure 6.2: Class diagram showing the representation of marriages.

Full transition coverage The full transition coverage property tests that all possible combinations of transitions, as stated in the state machine diagram, can really take place.

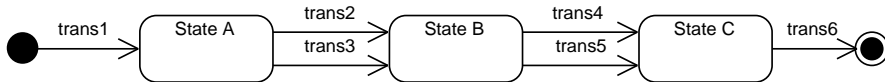


Figure 6.3: State machine diagram with multiple transitions between the same source and target states.

For example, Figure 6.3 shows a simple state machine diagram with 3 states and 6 transitions. Although there is only one way to reach State A, there are two different ways to reach state B and two ways to reach State C from State B. Therefore we have all these possible combinations for transition execution:

- trans1, trans2, trans4, trans6
- trans1, trans2, trans5, trans6
- trans1, trans3, trans4, trans6
- trans1, trans3, trans5, trans6

If one of these combinations does not execute successfully, then there is the possibility that something is wrong in the definition of the transitions or the external events/actions that make them up.

6.2 AuRUS-BAUML: The Tool & Its Workflow

After presenting several relevant tests, this section introduces AuRUS-BAUML. As we have already mentioned, AuRUS-BAUML allows us to check if a BAUML model fulfills some of the different properties we presented in the previous section. The following tools are used in the workflow:

- **ArgoUML**, to draw the UML diagrams and write the OCL operation contracts corresponding to the BAUML model. Its output is an XMI file containing the details of the model (Figure 6.4).
- **AuRUS-BAUML**, which checks the semantic correctness of the model. It receives as input an XMI file representing the BAUML model and the

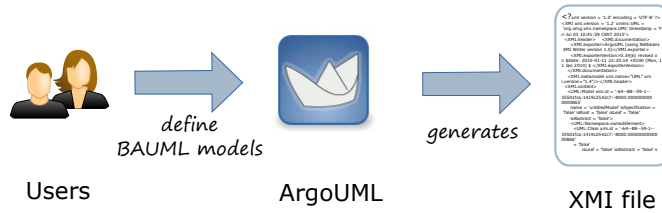


Figure 6.4: Process to generate the required XMI file using ArgoUML

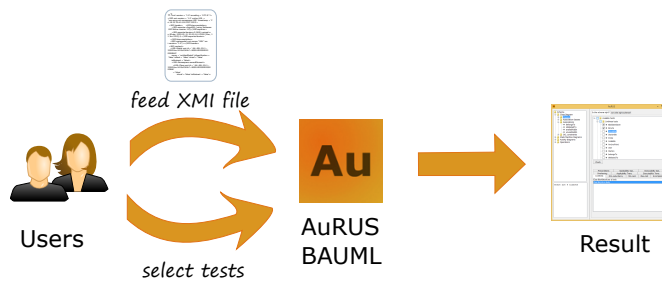


Figure 6.5: Process for using our tool.

properties that user wishes to check. Both the XMI file and the property are translated into logic. The resulting translation is then provided to another tool, SVTe, which performs the reasoning, and the result is then presented to the user. Note that the integration with SVTe is transparent from the users' point of view, as shown in Figure 6.5.

- **SVTe** performs the reasoning with the translated model. It receives as input a text file with a logic representation, together with the property that needs to be checked. As a result it informs AuRUS-BAUML of the success or failure of the test. In case of success, it provides a sample instantiation, otherwise it provides the restrictions which do not allow the achievement.

The remainder of this section presents and describes the tools involved in the process in more detail.

6.2.1 ArgoUML

As we have mentioned, we use ArgoUML to graphically represent BAUML models. ArgoUML [7] is a Java-based, open-source tool whose goal is to allow its users to draw UML diagrams. Although ArgoUML is free, lightweight and multi-platform, it also has some drawbacks.

The greatest issue when using ArgoUML to represent our BAUML models is the huge difference in semantics between UML 1.4 (implemented by ArgoUML) and UML 2.X for the UML activity diagrams. In UML 1.4 activity diagrams were a specialization of state machine diagrams, whereas in UML 2.0 they were reformalized and given token semantics, like Petri-nets.

In spite of this, we have opted to use it because we built AuRUS-BAUML on top of another tool, AuRUS-Operations, which was able to parse Argo's XMI file and load the class diagrams and the operation contracts into a metamodel. In any case, the notational differences are small and it works for our purposes. Figure 6.6 shows a screenshot of ArgoUML and an activity diagram drawn in it.

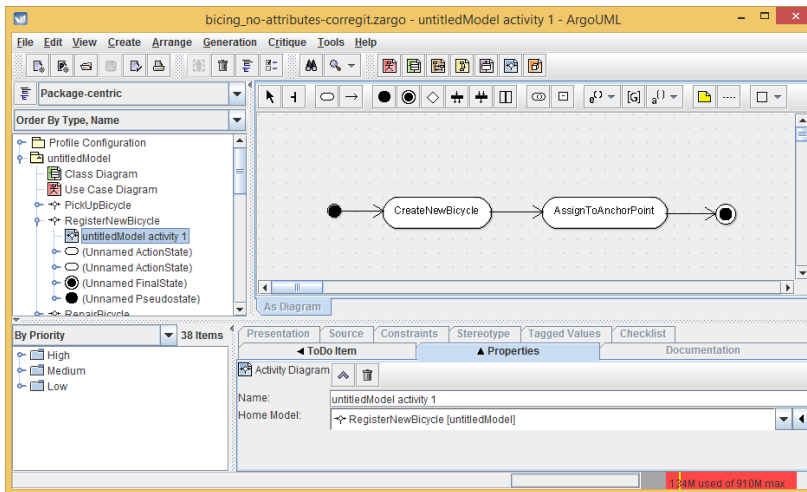


Figure 6.6: Screenshot of an activity diagram modeled using ArgoUML.

Another characteristic that needs to be considered is that the only graphical restrictions that can appear in the class diagram are the cardinality constraints and the disjointness and covering constraints in hierarchies. The rest of graph-

ical constraints need to be expressed as OCL restrictions.

Finally, OCL conditions in both the state machine and activity diagrams should have the following form, so that they can be processed properly:

```
context <ArtifactName> inv <ConditionName>: <OCL_Expression>
```

Once all the dimensions in the process have been represented in ArgoUML, then the diagrams can be exported into an XMI file that can be used by AuRUS-BAUML.

6.2.2 AuRUS-BAUML

We use AuRUS-BAUML to obtain the logic translation of the BAUML model contained in the XMI file. AuRUS-BAUML evolves from an already existing tool, AuRUS [114], that is capable of verifying and validating UML/OCL conceptual schemas specified using ArgoUML [7]. Initially it only dealt with structural schemas (without considering the dynamic part of a system), but the tool was expanded to incorporate operations in a master thesis [99], following the work in [105]. We will call this version of the tool that includes operations AuRUS-Operations.

Taking advantage of this, we decided to extend and adapt AuRUS-Operations to validate and verify our BAUML models. Figure 6.7 illustrates the internal workings of AuRUS-BAUML.

Given the XMI schema, AuRUS-BAUML loads it into two Java libraries which make up the metamodel. The first library, EinaGMC [128], is publicly available and covers both the UML Class Diagram and the OCL operation contracts. The second library, MetaSMADU (Meta State Machines and Activity Diagrams in UML), was created specifically for this thesis and has a simple but appropriate representation for the state machine and activity diagrams.

Once the models have been parsed and loaded, AuRUS-BAUML translates them into logic. To do so, we have made several important changes to AuRUS-Operations in order to deal with the particularities of BAUML models. In general terms, we have had to adapt the translation to incorporate the execution order of operations given by activity and state machine diagrams. Moreover, we no longer assume that all classes and associations have to be created by the operations in the model. The details of this process can be found on Section 6.3.

At the end of the process, AuRUS-BAUML shows in its graphical interface the available tests and allows the user to select those that he is interested in

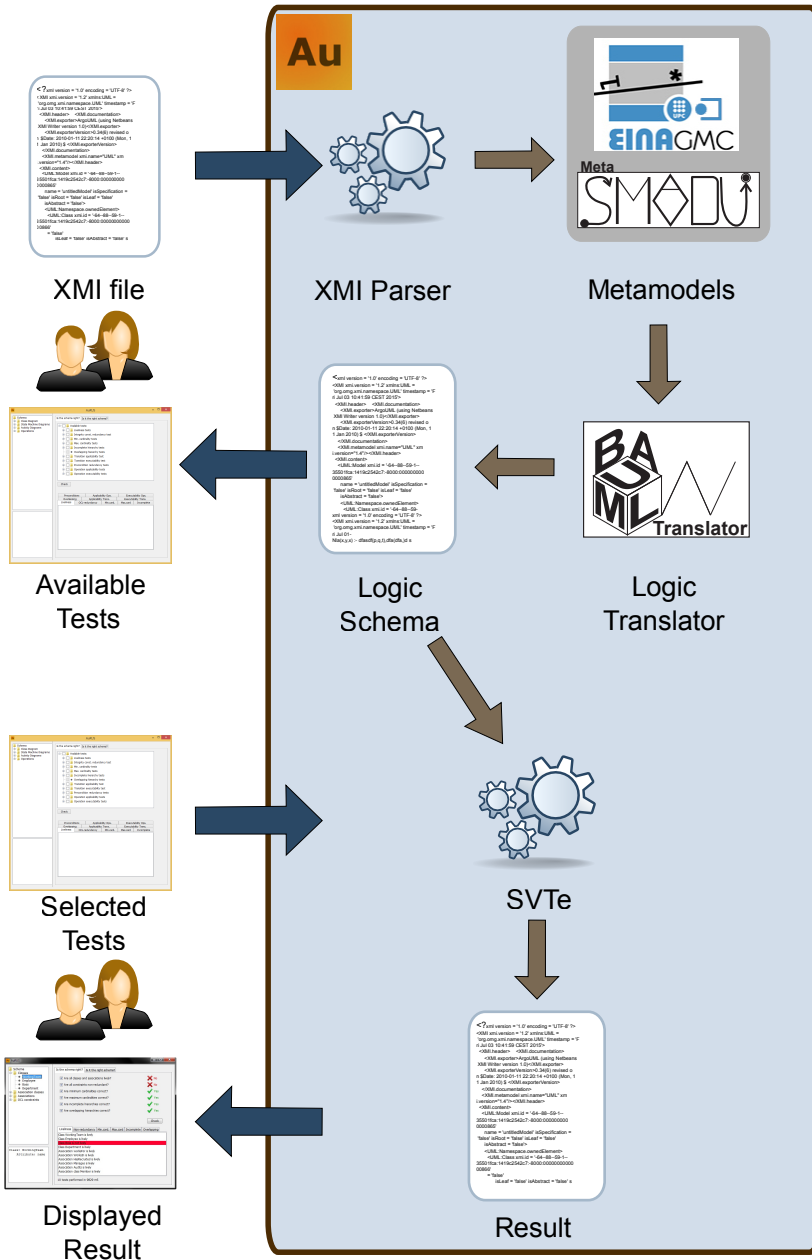


Figure 6.7: Internal workings of AuRUS-BAUML.

performing. These tests are generated automatically from the user's selection. For each of the selected tests type, the tool will inform the user of whether the tests have been performed successfully or not.

However, as it is still a prototype tool and a work-in-progress, only some of the verification and validation tests have been implemented and the tool requires more thorough testing. We expect to do this in future work.

SVTe

AuRUS-BAUML relies on an existing tool, SVTe, that is able to perform the tests described at the beginning of this chapter. SVTe deals with each test as a satisfiability problem. It uses the CQC_E method [115] which is aimed at building a consistent state of a database schema that satisfies a given goal, represented as a set of one or more literals. The method starts with an empty solution, and given the goal, the database schema, the constraints and the derivation rules, tries to obtain a set of base facts that satisfy the goal without violating any of the constraints. Notice that it does not require an initial instantiation of the database schema.

The CQC_E method is a semidecision procedure for finite satisfiability. This means that it does not terminate in the presence of solutions with infinite elements. However, termination is ensured if the model satisfies the conditions identified in the next chapter, Decidability.

To instantiate the variables during the inference process, the method uses Variable Instantiation Patterns (VIPs), which generate only the relevant facts that need to be added to the schema to satisfy the goal. If no instance that satisfies the database schema and the constraints is found, then the VIPs guarantee that the goal cannot be achieved with the given schema and constraints.

The CQC_E method executes in two stages. In the first stage, the method obtains an initial instance that satisfies the given goal. During the second stage, the method checks if the instance violates any of the given constraints, and if this is the case, tries to repair them by adding new facts. If it is not possible to perform any repair, it backtracks and tries a different initial database. This process goes on until it finds a solution or it determines that no solution exists.

SVTe provides the following results. If there exists a solution, it shows a sample instantiation to prove its existence. On the other hand, if no solution exists, the tool provides a list of the constraints that prevent it. As SVTe is transparently integrated in AuRUS-BAUML, these results are shown to the user through AuRUS-BAUML's interface.

6.3 Translation of BAUML into Logic

As we have seen, AuRUS-BAUML has a component, BAUML Translator, which translates the BAUML model contained in an XMI file provided by ArgoUML into a logic suitable for SVTe, which is in charge of the reasoning. AuRUS-BAUML then transparently shows the result of SVTe to the user.

This section focuses on describing the translation process which is carried out by BAUML Translator. It begins by presenting the background on logic formalization and then describes the translation process itself.

6.3.1 Background on Logic Formalization

For the formalization of our models, we use formulas in first-order logic. A term T is a variable or a constant. If p is a n -ary predicate and T_1, \dots, T_n are terms, then $p(T_1, \dots, T_n)$ or $p(\bar{T})$ is an atom. An ordinary literal is either an atom or a negated atom. A built-in literal has the form of $A_1 \theta A_2$, where A_1 and A_2 are terms. θ is either $<$, \leq , $>$, \geq , $=$ or \neq .

A normal clause has the form: $A \leftarrow L_1 \wedge \dots \wedge L_m$ with $m \geq 0$, where A is an atom and each L_i is an ordinary or built-in literal. All the variables in A , and in each L_i , are assumed to be universally quantified over the whole formula. A is the head and $L_1 \wedge \dots \wedge L_m$ is the body of the clause. A normal clause is either a *fact*, $p(\bar{a})$, where $p(\bar{a})$ is a ground atom, or a *deductive rule*, $p(\bar{T}) \leftarrow L_1 \wedge \dots \wedge L_m$ with $m \geq 1$, where p is the derived predicate defined by rule.

A condition is a formula of the (denial) form: $\leftarrow L_1 \wedge \dots \wedge L_m$ with $m \geq 1$. Finally, a schema S is a tuple (DR, IC) where DR is a finite set of deductive rules and IC is a finite set of conditions. All formulas are required to be *safe*, i.e. every variable occurring in their head or in negative or built-in literals must also occur in an ordinary positive literal of the same body. An instance of a schema S is a tuple (E, S) where E is a set of facts about base predicates. $DR(E)$ denotes the whole set of ground facts about base and derived predicates that are inferred from an instance (E, S) , and corresponds to the fixpoint model of $DR \cup E$.

6.3.2 Overview of the Translation Process

The translation process will map the classes and associations to predicates. Those that are read-only, will be base predicates, whereas those that are read-write will be derived from the execution of the tasks that create and/or delete them. These derivation rules will also have to take into consideration the

context in which the tasks execute, determined by the state machine diagram and the activity diagrams. Finally, the integrity constraints in the class diagram will be translated as logic formulas in the denial form, considering that they only need to be checked at the end of a transition.

The work we present here clearly differs from [105], where only class diagrams and operation contracts were considered. Note that in this case no restrictions were imposed on the execution of the tasks nor on the checking of the constraints.

To illustrate the translation process, we will use the Bicing example presented in Section 3.1, like in the previous section. The assumptions we make over the initial models, detailed in Section 5.1.3, also apply in this chapter.

Without loss of generality, and to keep the examples simpler, we will work with the class diagram in Figure 3.2 on page 37, but with no attributes. The integrity constraints that deal with attributes will also, logically, be omitted from the reasoning process. This will make the reasoning process easier for the tool, as there will be less restrictions to take into consideration.

6.3.3 Translation Algorithms

Our translation process is divided into four steps, shown in Algorithm 5. To begin with, we focus on the generic steps: obtaining derivation rules for classes and associations, translating the integrity constraints, generating the derivation rules from the tasks, and adding the required conditions to ensure that tasks execute properly, in the context given by state transition and activity diagrams.

The first step creates the derivation rules for the read-write set of classes and associations. To determine if a class or association is read-only or read-write, it is only necessary to examine the postcondition of all the tasks as described in [105], like in the previous chapter. The predicate corresponding to each read-write class and association will have a time component t indicating that the element exists at time t , whereas read-only elements will not include the time t and will be treated as base predicates.

The algorithm also takes into consideration if a class is *created* or *created and deleted* in the model. The general form of these rules is:

$$C(oid, \bar{p}, t) \leftarrow addC(\bar{p}, t_1) \wedge \neg deletedC(\bar{p}_j, t_1, t) \wedge t \geq t_1 \wedge time(t),$$

where \bar{p} corresponds to the attributes in the class (including its OID [unique object identifier]) or the participants in the association, \bar{p}_j represents the identifier of the class (its OID) or association (OID of the classes that participate

Algorithm 5 TranslateToLogic($\mathcal{B} = \langle \mathcal{M}, \mathcal{O}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$)

▷ Step 1: Creating rules for read/write classes and associations

```

r := ∅
for all c ∈ CLASSES( $\mathcal{M}$ ) do
  if c is created in  $\mathcal{P}$  ∧ c is not deleted in  $\mathcal{P}$  then
    r := r ∪ {C( $\bar{p}, t$ ) ← addC( $\bar{p}, t_1$ ) ∧ time(t) ∧ t ≥ t1}
  else if c is created in  $\mathcal{P}$  ∧ c is deleted in  $\mathcal{P}$  then
    r := r ∪ {C( $\bar{p}, t$ ) ← addC( $\bar{p}, t_1$ ) ∧ ¬deletedC( $\bar{p}_j, t_1, t$ ) ∧ t ≥ t1 ∧ time(t)}
    r := r ∪ {deletedC( $\bar{p}_j, t_1, t_2$ ) ← delC( $\bar{p}_j, t$ ) ∧ time(t1) ∧ time(t2) ∧ t ≤ t2 ∧ t > t1}
  end if
end for
for all a ∈ ASSOCIATIONS( $\mathcal{M}$ ) do
  if a is created in  $\mathcal{P}$  ∧ a is not deleted in  $\mathcal{P}$  then
    r := r ∪ {A( $\bar{p}, t$ ) ← addA( $\bar{p}, t_1$ ) ∧ time(t) ∧ t ≥ t1}
  else if a is created in  $\mathcal{P}$  ∧ a is deleted in  $\mathcal{P}$  then
    r := r ∪ {A( $\bar{p}, t$ ) ← addA( $\bar{p}, t_1$ ) ∧ ¬deletedA( $\bar{p}_j, t_1, t$ ) ∧ t ≥ t1 ∧ time(t)}
    r := r ∪ {deletedA( $\bar{p}_j, t_1, t_2$ ) ← delA( $\bar{p}_j, t$ ) ∧ time(t1) ∧ time(t2) ∧ t ≤ t2 ∧ t > t1}
  end if
end for

```

▷ Step 2: Translate integrity constraints

```

icSet := translateIC( $\mathcal{O}$ )
for all condition cond ∈ icSet do
  cond := cond + {∧validState(t)}
end for
taskRules := ∅

```

▷ Step 3: Generate rules for class and association creation and deletion for every task

```

for all t ∈  $\mathcal{T}$  do
  resRules := translateTask(t)
  taskRules := taskRules ∪ resRules
end for

```

▷ Step 4: Generate necessary rules and conditions to ensure correct execution order

```

taskRules := taskRules ∪ generateConstraintsTaskExecution( $\mathcal{B}$ )
return ⟨r, icSet, taskRules⟩

```

and identify it) C , and thus $\bar{p}_j \subseteq \bar{p}$, and t and t_1 represent the time. We will see how $addC(\dots)$ and $deletedC(\dots)$ are obtained later on.

The rule basically states that a class or an association will exist at time t if it has been created previously, at t_1 ($t_1 \leq t$), and it has not been deleted in the meantime. For instance, `Bicycle` is encoded as:

$$Bicycle(b, t) \leftarrow addBicycle(b, t_1) \wedge time(t) \wedge \neg deletedBicycle(b, t_1, t) \wedge t_1 \leq t,$$

whereas `User` is encoded as $User(u)$. `Bicycle` is a derived predicate created and deleted by some of the tasks. On the other hand, `User` is a base predicate as it is not created nor deleted by any task.

Step 2 of the algorithm translates the constraints \mathcal{O} into a set of formulas in denial form, following [106], but we need to add an atom $\wedge validState(t)$ to each

of them to ensure that they are only checked at the end of the execution of a state transition diagram transition, following the semantics of the framework.

For instance, the covering constraint in the hierarchy of *Bicycle* indicates that a *Bicycle* must have one of its subclasses' type. Then the condition:

$$\leftarrow Bicycle(b, t) \wedge \neg IsKindOfBicycle(b, t) \wedge validState(t)$$

states that there cannot be a bicycle which has not any of its subtypes (predicate *IsKindOfBicycle*), where *IsKindOfBicycle* is a derived predicate from *InUse*, *Available* and *Unusable* (see below). This condition only applies when there are no transitions taking place, indicated by predicate *validState*.

$$IsKindOfBicycle(b, t) \leftarrow InUse(b, t)$$

$$IsKindOfBicycle(b, t) \leftarrow Available(b, t)$$

$$IsKindOfBicycle(b, t) \leftarrow Unusable(b, t)$$

Step 3 is the most complex and it is decomposed into various algorithms. It generates the derivation rules that link the creation and deletion of the classes and associations with the tasks that perform these changes, and ensures that all tasks execute at the right time. This is done by calling Algorithms 6 and 7.

Finally, step 4 generates the remaining necessary constraints to ensure the correct execution of the tasks by calling Algorithm 8. For instance, if there is a sequence of tasks that execute in the activity diagram, it ensures that all of them execute and creates the derivation rules to generate predicate *validState* at the end of the execution of the activity diagram.

Algorithm 6 translateTask(*task*)

```

rules := ∅
prevRules := getContextPreviousTasks(task, t)
createList contains the classes and associations created by task
delList contains the classes and associations deleted by task
for all ruleFragment ∈ prevRules do
  for all el ∈ createList do
    r := addEl( $\bar{p}$ , t) ← task( $\bar{p}$ ,  $\bar{x}$ , t) ∧ pretask(t - 1) ∧ time(t) ∧ ruleFragment
    rules := rules ∪ r
  end for
  for all el ∈ delList do
    r := delEl( $\bar{p}$ , t) ← task( $\bar{p}$ ,  $\bar{y}$ , t) ∧ pretask(t - 1) ∧ time(t) ∧ ruleFragment
    rules := rules ∪ r
  end for
  rules := rules ∪ {task'(pa, t) ← task(pa,  $\bar{z}$ , t) ∧ pretask(t - 1) ∧ time(t) ∧ ruleFragment}
end for
return rules

```

► t represents a time term

We will now analyze the details of the remaining algorithms. Algorithm 6 is aimed at translating the atomic tasks. As they make changes to the instances of the class diagram, this translation will result in the derivation rules that generate predicates *addEl* and *delEl*, where *el* is a class or an association. In [105], these rules are generated by analyzing the postcondition of each task and determining if the task creates or deletes some instance. If the task has a precondition, then its translation (following [106]) is also added to the body of the derivation rule to ensure that it is true at time $t - 1$, where t represents the time the task executes.

However, this translation does not impose any restrictions over the order for task execution. In BAUML tasks execute following the restrictions and the order established by the state transition and activity diagrams. In particular, $task_k$ can only execute if pre_{task_k} is true and the previous task $task_{k-1}$ has executed at $t - 1$.

Algorithm 6 generates the creation and deletion rules as described, invoking Algorithm 7 to obtain the part of the rule that refers to the successful execution of the previous tasks. At the end, Algorithm 6 generates a rule of the form:

$$task'(p_a, t) \leftarrow task(p_a, \bar{z}, t) \wedge pre_{task}(t - 1) \wedge time(t) \wedge ruleFragment,$$

where p_a corresponds to the OID of the business artifact, which we use to ensure the proper evolution of the system, and \bar{z} corresponds to the remaining parameters or terms of *task*. The derived predicate of this rule, $task'(\dots)$, will be used as an indicator that *task* has executed properly by the next task.

Algorithm 7 is in charge of generating the part of the derivation rules that depends on the previous node(s) of a certain node. Its complexity lies in the fact that we consider not only linear activity diagrams, but that we also allow decision and merge nodes. We assume that control nodes do not add execution time to our diagrams and that they are traversed immediately. So, given a node n that belongs to an activity diagram P_ϵ and time t , the algorithm:

1. Obtains the previous nodes of n , stores them in *prevSet* and initializes *result* to the empty set.
2. For each $n_p \in prevSet$, it checks its type.
 - a) If n_p is a task, it then adds the $n'_p(\dots)$ predicate to the existing *result*, indicating that the task n_p will have executed successfully.
 - b) If n_p is a decision node, the algorithm needs to obtain the predicates corresponding to the tasks that may execute before n_p ; therefore

Algorithm 7 `getContextPreviousTasks(n,t)`

```

result := ∅
prevSet contains the previous nodes of n
for all  $n_p \in \text{prevSet}$  do
  if  $n_p$  is task then
    result := result  $\cup$   $n'_p(p_a, t - 1)$ 
  else if  $n_p$  is decision node then
    guard := getGuard( $n_p, n$ )
    res := getContextPreviousTasks( $n_p, t$ )
    for all  $el \in \text{res}$  do
      result := result  $\cup$   $\{el \wedge \text{guard}(t - 1)\}$ 
    end for
  else if  $n_p$  is merge node then
    res := getContextPreviousTasks( $n_p, t$ )
    result := result  $\cup$  res
  else if  $n_p$  is initial node then
    transitions contains the transitions in which the activity diagram appears
    for all  $t \in \text{transitions}$  do
       $s_s$  is the source state of  $t$ 
      cond is the translation of condition of  $t$ 
      if  $s_s$  is not initial pseudostate  $\wedge$  cond is not empty then
        result := result  $\cup$   $\{s_s(\bar{p}, t - 1) \wedge \text{cond}(t - 1)\}$ 
      else if  $s_s$  is not initial pseudostate then
        result := result  $\cup$   $\{s_s(\bar{p}, t - 1)\}$ 
      else if cond is not empty then
        result := result  $\cup$   $\{\text{cond}(t - 1)\}$ 
      end if
    end for
  end if
return result
end for

```

it invokes itself, but this time with n_p and t as input. As n_p is a decision node, there will be a guard condition in the edge between n_p and n . This guard will be translated as if it was a precondition and it will have to be true at $t - 1$ in order for the task to execute. Then, it will add the guard condition to each rule-part obtained by the self-invocation.

- c) If n_p is a merge node, it invokes itself with parameters n_p and t , and it adds the result of this invocation to variable *result*.
- d) If, on the other hand, n_p is an initial node, it adds the source state of the state transition diagram of the transitions in which the activity diagram appears. If there is an OCL condition, it also adds the translation of the condition.

3. The algorithm returns variable *result*, containing a set of rule fragments.

For instance, for task *Assign to Anchor Point*, we have the following rules:

$$\begin{aligned} \text{addAvailableIsIn}(b, a, t) &\leftarrow \text{assignToAnchPoint}(a, b, t) \wedge \text{AnchorPoint}(a) \\ &\quad \wedge \text{precondAssToAP}(a, t - 1) \wedge \text{Bicycle}(b, t) \wedge \text{createNewBicycle}'(b, t - 1) \\ \text{assignToAnchPoint}'(b, t) &\leftarrow \text{assignToAnchPoint}(a, b, t) \wedge \text{AnchorPoint}(a) \\ &\quad \wedge \text{precondAssToAP}(a, t - 1) \wedge \text{Bicycle}(b, t) \wedge \text{createNewBicycle}'(b, t - 1) \end{aligned}$$

The task creates an instance of the *available is in* association. It has a precondition which must be true at $t - 1$, and its translation appears in the derivation rule of *addAvailableIsIn*. In addition to this, the body of the rule includes the predicate *createNewBicycle'*, that guarantees that the previous operation (*Create New Bicycle*) has executed successfully.

Algorithm 8 generateConstraintsTaskExecution(\mathcal{B})

```

constr :=  $\emptyset$ 
for all task  $\in$  TASKS( $\mathcal{B}$ ) do
   $n_n$  is next node of task
  if  $n_n$  is task then
    constr := constr  $\cup$  { $\leftarrow$  task( $p_a, \bar{z}, t$ )  $\wedge$   $\neg n'_n(p_a, t + 1)$ }
  else if  $n_n$  is decision node  $\vee$   $n_n$  is merge node then
     $r := \leftarrow$  task( $p_a, \bar{z}, t$ )  $\wedge$   $\neg$ nextTask( $p_a, t + 1$ )
    res := generateConstraintsNextTasks( $n, \text{task}$ )
    constr := constr  $\cup$   $r \cup$  res
  else if  $n_n$  is final node then
    constr := {validState( $t$ )  $\leftarrow$  task'( $p_a, t$ )}
  end if
end for
return constr

```

With the algorithms that we have seen so far we have restricted the order for the tasks execution in one direction, ensuring that task $task_k$ can only execute if $task_{k-1}$ has taken place. We also need to ensure that, once an activity diagram begins execution, it finishes. Algorithm 8 generates the necessary constraints to do so. For each task, it obtains its next node and, if the next node n_n is a task, it creates a rule of the form: \leftarrow task(p_a, \bar{z}, t) \wedge $\neg n'_n(p_a, t + 1)$, where predicate n'_n corresponds to the derived predicate generated by Algorithm 6 to ensure that task n_n has executed properly. For instance, for the tasks *Create New Bicycle* and *Assign to Anchor Point* we have the following condition and derivation rule: \leftarrow createNewBicycle(b, t) \wedge \neg assignToAnchorPoint'($b, t + 1$).

On the other hand, if n_n is a decision node or a merge node, there is the possibility that there will be more than one task that can be executed. For this reason, the algorithm generates this rule: \leftarrow task(p_a, \bar{z}, t) \wedge \neg nextTask($p_a, t + 1$),

meaning that if *task* has executed at t one of its next tasks must have executed at $t + 1$. *nextTask* is a derived predicate resulting from the execution of any of the next tasks. These derivation rules are created in Algorithm 9 and have the following form: $nextTask(p_a, t) \leftarrow task'_n(p_a, t)$. The algorithm iterates over the nodes until the next task(s) are found. Guard conditions are not considered because they have already been translated by the other algorithms.

Finally, if a task is followed by a final node, we need to generate rule: $validState(t) \leftarrow task'(p_a, t)$. This rule will ensure that the restrictions of the model are checked at the end of the execution. For instance, in our example the successful execution of task *Assign To AnchorPoint* generates predicate *validState* as it is the last task in the activity diagram:

$$validState(t) \leftarrow assignToAnchorPoint'(b, t).$$

Algorithm 9 generateConstraintsNextTasks($n, task$)

```

result := ∅
nextSet contains the set of next nodes of n
for all  $n_n \in nextSet$  do
  if  $n_n$  is task then
    nextTask( $p_a, t$ )  $\leftarrow n'_n(p_a, t)$ 
  else if  $n_n$  is decision node  $\vee n_n$  is merge node then
    res := generateConstraintsNextTasks( $n_n, task$ )
    result := result  $\cup$  res
  else if  $n_n$  is final node  $\wedge n$  is decision node then
    guard contains the guard condition from  $n$  to  $n_n$ 
    nextTask( $p_a, t$ )  $\leftarrow task'(p_a, \bar{z}, t) \wedge guard(\bar{y}, t)$ 
    validState( $t$ )  $\leftarrow task'(p_a, \bar{z}, t) \wedge guard(\bar{y}, t)$ 
  end if
end for
return result

```

There is a special case, however. If there is a decision node n and one of the next nodes $n_n \in \text{FINAL}(P_\varepsilon)$ is a final node, then these rules are needed:

$$nextTask(p_a, t) \leftarrow task'(p_a, \bar{z}, t) \wedge guard(\bar{y}, t)$$

$$validState(t) \leftarrow task'(p_a, \bar{z}, t) \wedge guard(\bar{y}, t),$$

which will ensure that after the execution of *task*, the diagram terminates if the corresponding guard condition is met.

6.4 Formalization of Tests & Results

After showing how the BAUML models are translated into logic, we now formalize the tests presented in Section 6.1. All the tests are represented as checking the satisfiability of a derived predicate.

6.4.1 Verification Tests

Class diagram

Liveliness of a Class or Association The liveliness test of a class or an association will ensure that an instance of it can be successfully created and that it persists in the system until the transition that has created it ends. The general form of the test is the following, where el is the name of the class or association:

$$livelinessTestEl() \leftarrow el(\bar{p}, t) \wedge validState(t).$$

If the class or association is not created in the business process, then the time component t would be omitted from the derivation rule shown above. As during the execution of activity diagrams integrity constraints can be violated, $validState(t)$ ensures that the integrity constraint is only checked when no activity diagrams are in the middle of an execution, i.e. when the system is in a valid state.

Minimum and maximum cardinalities

Minimum cardinality Given a n -ary association $asso$, with m participants, and a minimum cardinality of x in the extreme of class C , we would define the test in the following way:

$$\begin{aligned} minCardCorrectTest() &\leftarrow asso(p_1, \dots, p_{m-1}, c_1, t) \wedge \dots \wedge asso(p_1, \dots, p_{m-1}, c_x, t) \\ &\quad \wedge \neg extraAsso(p_1, \dots, p_{m-1}, c_1, \dots, c_x, t) \\ &\quad \wedge c_1 \neq c_2 \wedge \dots \wedge c_1 \neq c_x \wedge validState(t) \\ extraAsso(p_1, \dots, p_{m-1}, c_1, \dots, c_x, t) &\leftarrow asso(p_1, \dots, p_{m-1}, c_{x+1}, t) \\ &\quad \wedge c_1 \neq c_{x+1} \wedge \dots \wedge c_x \neq c_{x+1} \end{aligned}$$

Maximum cardinality To avoid infinite loops, the maximum cardinality should be bounded before running this test. Given a n -ary association $asso$, with m participants, and a maximum cardinality of x in the extreme of class C , we would define the test in the following way:

$$\begin{aligned} maxCardCorrectTest() &\leftarrow asso(p_1, \dots, p_{m-1}, c_1, t) \wedge \dots \wedge asso(p_1, \dots, p_{m-1}, c_x, t) \\ &\quad \wedge c_1 \neq c_2 \wedge \dots \wedge c_1 \neq c_x \wedge \dots \wedge c_{x-1} \neq c_x \wedge \dots \wedge validState(t) \end{aligned}$$

For both tests, a satisfactory answer means that the cardinalities are correct, whereas a negative answer means that the cardinality should be greater, in the case of the minimum cardinality, or lower, for the maximum cardinality.

Redundancy of integrity constraints An integrity constraint ic is redundant if other constraints subsume it. To look for redundancy, what we do is remove the constraint from the schema and test if the model can fulfill the constraint:

$$icRedundant() \leftarrow ic(t) \wedge validState(t)$$

If the result is positive, it means that there is no other constraint restricting ic . Therefore, ic is not redundant. On the other hand, if the result is negative, then ic is redundant and can be deleted from the schema.

State Machine Diagram

State reachability As we have already mentioned, checking the reachability of a state is equivalent to checking the liveness of the corresponding class el :

$$stateReachabilityTest() \leftarrow el(\bar{p}, t) \wedge validState(t).$$

Transition Applicability Given a transition $t = \langle v_s, o, e, c, x, v_t \rangle$, checking its applicability means ensuring that v_s is reachable and that o (if any) is true:

$$transApplTest() \leftarrow predV_s(\bar{p}, t)[\wedge ocl(t)] \wedge validState(t)$$

$predV_s$ corresponds to the predicate representing the subclass that corresponds to state v_s , ocl the ocl condition o in the transition (if there is one). We have to ensure that these conditions are met on a $validState(t)$, as transitions begin their execution on this state.

Transition Executability There are many factors that should be considered for the executability of a transition $t = \langle v_s, o, e, c, x, v_t \rangle$:

1. The source state v_s .
2. Any OCL conditions that may appear in the transition: o .
3. The target state v_t .
4. The event e or effect x which is part of t
5. There cannot be any intermediate valid state ($validState(t)$) between the source state v_s and v_t . Otherwise, this would mean that the system has evolved through other transition(s) which have finished successfully.

Conditions 1 and 2 refer to the time, t_1 , before the execution of the transition begins. In contrast, condition 3 refers to the time, t_2 , at the end of the execution. Between t_1 and t_2 the tasks in e or x will execute.

Bearing all this in mind, the form of the test will be the following:

$$\begin{aligned} transExecTest() \leftarrow & \text{pred}V_s(oid, \dots, t_1)[\wedge ocl(t_1)] \wedge \text{validState}(t_1) \wedge \text{pred}V_t(oid, \dots, t_2) \\ & \wedge \text{validState}(t_2) \wedge \neg \text{validState}(t_3) \wedge \text{time}(t_3) \wedge t_1 < t_2 \wedge t_1 < t_3 \\ & \wedge t_3 < t_2 \wedge \text{execLastTask}'(oid, \dots, t_2) \end{aligned}$$

$\text{execLastTask}'(oid, \dots, t_2)$ corresponds to the last task in the activity diagram corresponding to event e or effect x . This will ensure that we are executing the right event or effect to perform the transition. $\text{pred}V_s$ and $\text{pred}V_t$ are the predicates representing the subclasses that correspond to the source and the target states, respectively.

Activity Diagram & Operation Contracts

We have defined the translation of the BAUML models into logic in a way in which activity diagrams cannot stop in the middle of an execution: they either execute successfully or they do not execute at all. Therefore, if we do not make some changes, the result of the executability and applicability tests of the tasks will depend, in the general case, on the executability of the event or effect they are part of. Therefore the tests would not provide any additional information.

What it would be interesting is to know if a task is applicable or executable considering only the context required up to the point of its execution. To achieve this, we could generate the logic schema without any of the predicates and rules produced by Algorithm 8 which force the execution to move forward.

Applicability Test This test will check whether a certain task can be executed, that is, if the necessary requirements for its execution are met. The test will have the following form, for task $task_i$:

$$\text{applicabilityTask}() \leftarrow \text{pre}_{task}(\bar{y}, t) \wedge \text{task}'_{i-1}(p_a, t).$$

Executability Test The executability test will check if a certain task can be executed. It is particularly useful for those activity diagrams with decision nodes to ensure that all paths can be taken. The test will have the following form:

$$\text{executabilityTask}() \leftarrow \text{task}'(p_a, t).$$

Notice that it is equivalent to checking if the predicate $task'$ can be generated, as $task'$ represents precisely the successful execution of $task$.

6.4.2 Validation Tests

Path Inclusion or Exclusion There are three tests (of two different types) that we can perform to check path inclusion and exclusion between two relationships that have the same participants. $asso1$ and $asso2$ are the associations which have the same participants p_1 to p_n .

$$pathIncExcTest1() \leftarrow asso1(p_1, \dots, p_n, t) \wedge asso2(p_1, \dots, p_n, t) \wedge validState(t)$$

$$pathIncExcTest2() \leftarrow asso1(p_1, \dots, p_n, t) \wedge \neg asso2(p_1, \dots, p_n, t) \wedge validState(t)$$

$$pathIncExcTest3() \leftarrow asso2(p_1, \dots, p_n, t) \wedge \neg asso1(p_1, \dots, p_n, t) \wedge validState(t)$$

The first test checks if it possible for $asso1$ and $asso2$ to have instances with exactly the same participants. Tests 2 and 3 check the opposite: *is it possible to have instances of $asso1$ ($asso2$) if there is not the corresponding instance in $asso2$ ($asso1$)?*

Test	Result	
	True	False
pathIncExcTest1	Allows path inclusion	Does not allow path inclusion → Path exclusion
pathIncExcTest2	$asso1$ does not depend on $asso2$	$asso1$ requires $asso2$ → Path inclusion
pathIncExcTest3	$asso2$ does not depend on $asso1$	$asso2$ requires $asso1$ → Path inclusion

Table 6.2: Table showing the interpretation of the different results for the path inclusion and exclusion tests.

Table 6.2 summarizes the meaning of the results of the tests. A positive result in the tests may indicate that there is a constraint missing, whereas a negative result will indicate that the paths are mutually exclusive ($pathIncExcTest1$) or that they are inclusive ($pathIncExcTest2$ and $pathIncExcTest3$).

Missing irreflexive constraints Given an association *asso* which relates the same class *c* to itself, we can check if it is reflexive by performing the following test:

$$\text{reflexTest}() \leftarrow \text{asso}(r_1, r_1, t) \wedge \text{validState}(t)$$

A positive result will indicate that the association is reflexive and that there might be an integrity constraint missing.

Full transition coverage The full transition coverage test checks if all potential combinations of transitions can actually take place.

To perform this test, we need to generate, first of all, a set of sequences of transitions that correspond to valid evolutions of the artifact through the state machine diagram. Then, for each of these sequences with transitions trans_1 to trans_n , we create the following test:

$$\begin{aligned} \text{transCovTest}() &\leftarrow \text{lastTask}_{\text{trans}_1}(\text{oid}, \dots, t_1) \wedge \dots \wedge \text{lastTask}_{\text{trans}_n}(\text{oid}, \dots, t_n) \\ &\quad \wedge t_1 < t_2 \wedge \dots \wedge t_{n-1} < t_n \wedge \neg \text{intValidState}(t_1, t_2) \wedge \dots \wedge \\ &\quad \neg \text{intValidState}(t_{n-1}, t_n) \\ \text{intValidState}(t_a, t_b) &\leftarrow \text{time}(t_a) \wedge \text{time}(t_b) \wedge \text{time}(t_c) \wedge t_a < t_c \wedge t_c < t_b \wedge \\ &\quad \text{validState}(t_c) \end{aligned}$$

Predicate *intValidState* is necessary to ensure that the only transitions executed in the diagram are the ones stated in the first derivation rule (i.e. there are no valid states in between transition executions, as this would imply that additional transitions have taken place). If the test is satisfiable, then the sequence of transitions is valid. On the other hand, if it is not, this may indicate an error in either the executability of the transitions themselves or in this particular combination of transitions. If the transitions have been checked satisfactorily for their executability, then the issue is in the combination of transitions.

6.4.3 Some Test Results

This subsection presents the results and the corresponding screenshots for some of the tests applied to our Bicing example.

Verification

Transition Applicability Test We will first show the results for the transition applicability test, for transition *Pick Up Bicycle*. In this case, the transition has

no conditions and its source state is *Available*, so the corresponding test will have the following form:

$$appTransTestPickUp() \leftarrow Available(b, t) \wedge validstate(t).$$

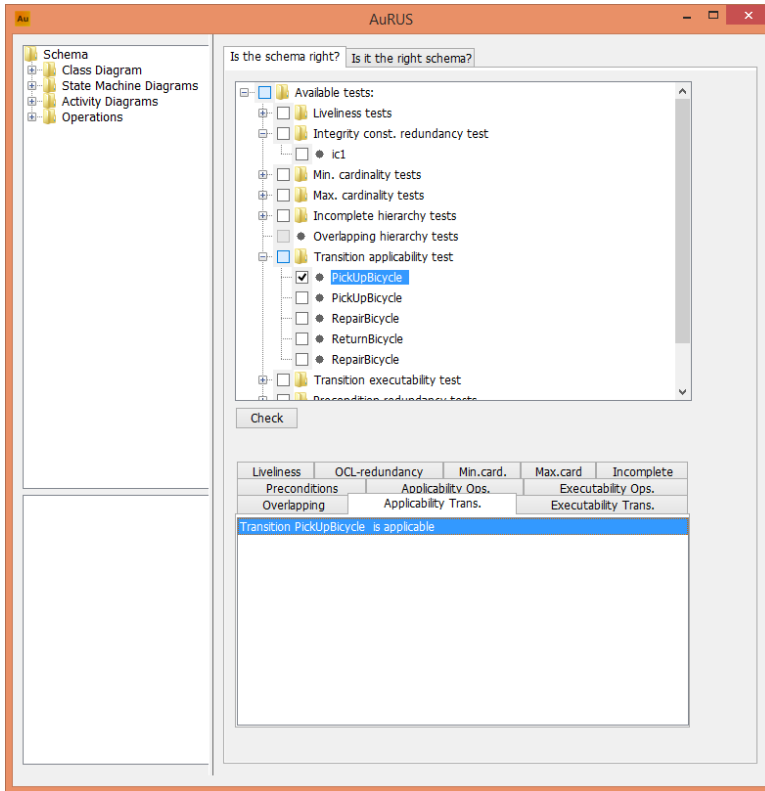


Figure 6.8: Result of the *Pick Up Bicycle* transition applicability test in AuRUS-BAUML.

Figure 6.8 shows the result of the test. As highlighted in the image, transition *Pick Up Bicycle* is applicable. By double-clicking on the result, the tool opens a new window detailing the base predicates required to achieve the goal: in this case, reaching a point where the transition can be applied.

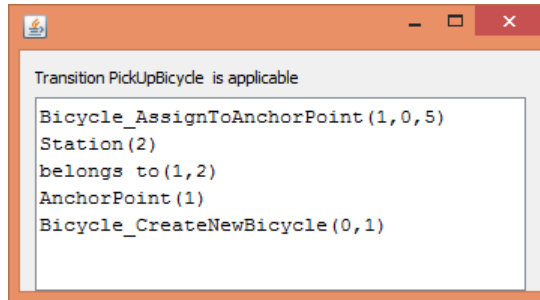


Figure 6.9: Details of the required base predicates to achieve the results

Figure 6.9 shows the required base predicates to achieve the results. Note that there are predicates corresponding to classes (e.g. *Station* or *AnchorPoint*) and to tasks (e.g. *CreateNewBicycle*, *AssignToAnchorPoint*). The predicates representing classes correspond to those which are not created by the operations in the model, and therefore they do not include a time component. On the other hand, the predicates corresponding to the operations have a component representing the time (the last term). In consequence, from the result we can see the order in which the tasks can execute to obtain the reach the necessary conditions for *Pick Up Bicycle* to execute.

Liveliness Test To test the liveliness of *BicycleRental*, we would define the following derivation rule:

$$livelinessTestBicycleRental() \leftarrow BicycleRental(br, b, a, t) \wedge validstate(t).$$

Figure 6.10 shows the result of applying the test to our example. As it can be seen in the result, the test executes successfully as there is a sample instantiation that satisfies the goal of the test.

Validation

User-defined tests In the context of our Bicing example, another validation test would be checking whether *Blacklisted* users are allowed to rent a bicycle. In this case, the test cannot be generated automatically and requires user intervention. The formalization of this property is as follows:

$$blacklistedUserRent() \leftarrow Blacklisted(u) \wedge BicycleRental(b, u, i, t) \wedge validState(t)$$

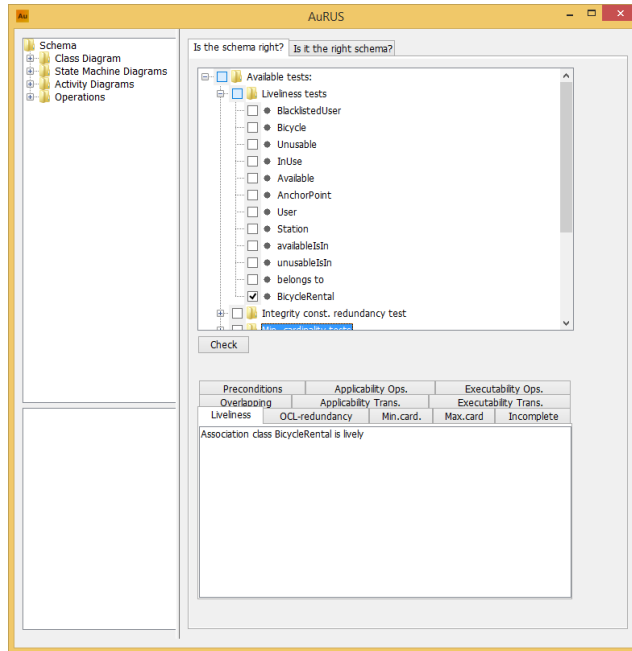


Figure 6.10: Figure showing the result of checking the liveness of *BicycleRental*.

Figure 6.11 shows the result of executing the test. It shows that it executes successfully: there is an instantiation proving that blacklisted users are allowed to rent bicycles. This is clearly a mistake, due to the fact that an integrity constraint is missing in the class diagram. Figure 6.12 shows the required predicates to achieve the result.

6.5 Summary & Conclusions

As we have explained, checking the correctness of BAUML models as early as possible is important to avoid the propagation of errors to the implementation of the process. Unfortunately, there were no tools to reason with our BAUML models.

To solve this and to show the feasibility of our approach, we have imple-

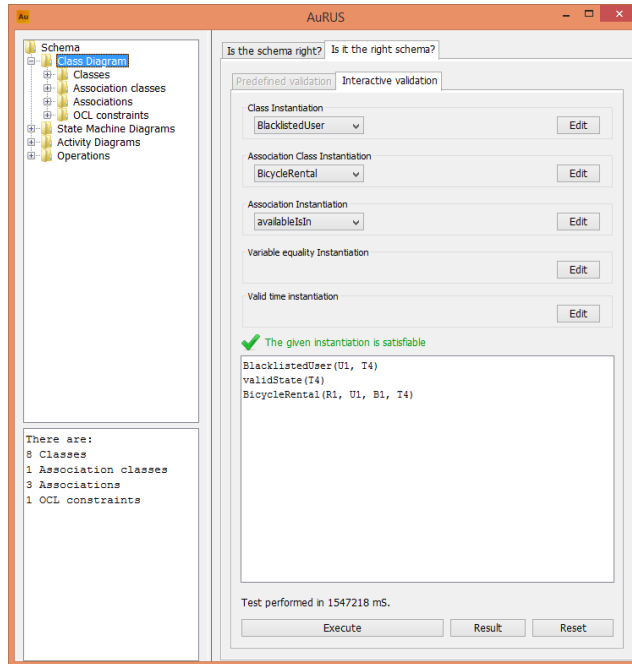


Figure 6.11: Result that answers the question *Can blacklisted users rent a bicycle?*.

mented a tool, AuRUS-BAUML, that given a BAUML model, it automatically translates it into logic. It then provides a list of available tests to the user, and the user is able to select the tests he is interested in performing. After this the tool reasons with the logic schema and the goal that corresponds to the test, and provides a result.

This result will either be a sample instantiation proving the satisfiability of the test, or a list of the restrictions which prevent its fulfillment. Note that AuRUS-BAUML does not require an initial instantiation of the schema.

Despite the potential of this tool, at the time of writing it suffers from efficiency issues. The simpler tests are performed quite rapidly, but the most complex ones may take hours or even not finish due to the cost of the search for a solution.

According to [99], the cost of evaluating negative literals is estimated as being 75 times higher than positive ones. Unfortunately, the way we define the

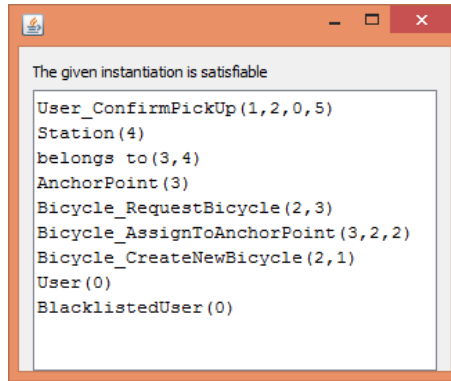


Figure 6.12: Details of the operations and classes required to obtain the result.

translation of the BAUML model requires the use of restrictions which include negative literals to ensure the proper execution of the activity diagram. This probably hinders the execution time of SVTe, the reasoner used in our tool.

Considering this, a clear area for further work would be working on the efficiency of the reasoner and even perhaps making changes to the translation to improve the overall response times.

Finally, it would be useful to make AuRUS-BAUML compatible with the XMI files generated by other diagramming or CASE tools other than ArgoUML, such as Visual Paradigm. Visual Paradigm [131] is a commercial tool which also offers a free community edition for non-commercial purposes. Although the community edition has some limitations, it allows exporting the UML diagrams into an XMI and is updated regularly.

Chapter 7

Decidability

After presenting two different ways of reasoning with our BAUML framework, this chapter deals with decidability. In particular, it studies whether it is decidable to check a property defined in a certain type of logic (a fragment of μ -calculus) over a BAUML model. To do so, it reduces the models to the halting problem of 2-counter machines, proving the undecidability, and from there it establishes restrictions over them to ensure their decidability. The goal is to single out the various sources of undecidability in terms of their verification and validation.

The first part of this chapter introduces the necessary concepts and background on 2-counter machines and a new running example. The second section focuses on the decidability analysis by incrementally restricting the models to ensure their decidability with the minimum number of restrictions. The third section discusses the implications of the analysis in our Bicing example. At the end of the chapter, we summarize our results and point out some conclusions.

7.1 Background

This section begins by giving a brief overview of 2-counter machines. We will use the halting problem of 2-counter machines to prove the undecidability of the various theorems presented in the chapter. After this, we introduce the running example that we will use to illustrate the theorems and the intuition behind the proofs in this chapter.

7.1.1 2-Counter Machines

Counter machines are used to model a computation. A counter machine has:

- A set of counters or registers
- A list of commands or instructions to be followed by the machine

The counters can only contain non-negative integers. An input for a counter machine will be a set of values to initialize its registers or counters. We say that the counter machine halts on input I if the execution of the instructions reaches the final command, HALT, which stops the machine.

Given a 2-counter machine, it is well-known that checking if it halts on a given input is undecidable [93]. Therefore, we can say the following:

Corollary 7.1.1. *It is undecidable to check whether a 2-counter machine halts on input $\langle 0, 0 \rangle$.*

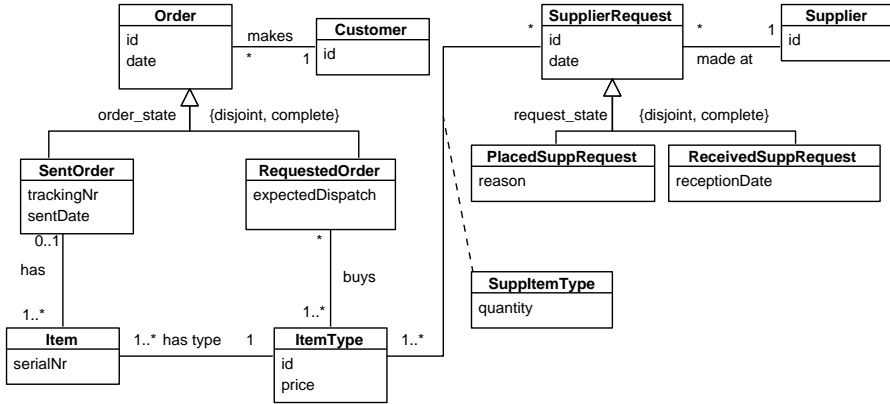
Corollary 7.1.1 will be the basis to prove the theorems in the following section.

7.1.2 Running Example: An Online-Retailer

This subsection presents a new example to better illustrate the variety of results of this chapter. It is based on a system for a company that registers orders from customers, and stores information about the orders made by the company to its suppliers. Our example is likely to specify a simplified version of the artifact-centric process models of an online shop like Amazon.

Figure 7.1 shows two business artifacts: `Order` and `SupplierRequest`. `Order` has two substates: `RequestedOrder` and `SentOrder`, that track the order's evolution. A `RequestedOrder` is related to various `ItemTypes`, indicating the products that the customer wishes to purchase. On the other hand, `SentOrder` is related to `Items`, which have a certain `ItemType`. That is, `SentOrders` are directly related to specific items identified by their serial number. Notice that apart from the artifact itself, the associations *makes*, *has*, and *buys* in which it takes part, are also created and deleted by the process.

Similarly, `SupplierRequest` represents the requests made to the supplier. It has two possible substates: `PlacedSuppRequest` and `ReceivedSuppRequest`, and it is related to `ItemType`, the association class that results from this relationship states information about the quantity of items of a certain type that have been requested to the supplier.



Key constraints: *serialNr* for *Item*, *id* for the other classes.

Figure 7.1: Class diagram for our online-retailer example

In this example, we assume that some classes/associations are *read-only*. This is the case, e.g., for *ItemType*.

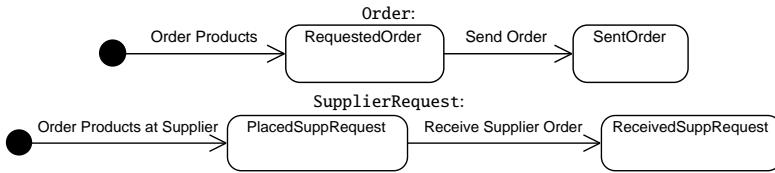


Figure 7.2: Artifact state machines for artifacts *Order* and *SupplierRequest*.

Both artifacts *Order* and *SupplierRequest* evolve independently from each other, with a lifecycle specified by the state machines of Figure 7.2. Their meaning is very intuitive. In the case of *Order*, when event *Order Products* takes place, the *RequestedOrder* is created. When we have a requested order and event *Send Order* executes, the order is sent to the customer and the artifact changes its state to *SentOrder*. The state machine diagram for *SupplierRequest* is analogous to that of *Order*.

Each of the events in the lifecycle transitions (*Order Products*, *Send Order*, *Order Products at Supplier* and *Receive Supplier Order*) are further defined using an activity diagram, which shows the units of work (i.e., the tasks) that are carried out, together with their execution order.

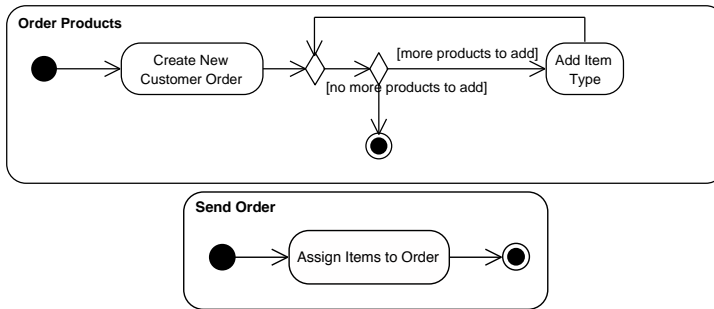


Figure 7.3: Activity diagrams for the events of Order

Figure 7.3 shows the activity diagrams for the events of Order. As for the *Order Products* event, the first task creates a new order, and the second task, which can be executed many times, adds an item type to the order that has been previously created. As for the *Send Order* event, its task adds the items to the order, marking it as sent.

Below we show the OCL operation contracts for the tasks in Figure 7.3.

Listing 7.1: Code for task *Create New Customer Order*

```
operation createNewCustomerOrder(orderId: String, date: Date, expDisp: Date,
    customerId: String): RequestedOrder
pre: not(RequestedOrder.allInstances()->exists(ro | ro.id=orderId)) and
    Customer.allInstances()->exists(c | c.id = customerId)
post: RequestedOrder.allInstances()->exists(ro | ro.ocIsNew() and ro.id=orderId
    and ro.date=date and ro.expectedDispatch=expDisp and
    ro.customer.id=customerId and result=ro)
```

CreateNewCustomerOrder receives as input the necessary parameters to create a new instance of the artifact *RequestedOrder*. Its precondition makes sure that no other order with the same identifier exists and that the customer ID is valid. It returns the *RequestedOrder* that has been created with the input parameters.

Listing 7.2: Code for task *AddItemType*

```
operation AddItemType(idItemType: String, ro: RequestedOrder )
pre: not (ro.itemType.id->includes(idItemType))
post: ro.itemType.id->includes(idItemType)
```

AddItemType adds an *ItemType* to the order that has been created in the previous operation. Its precondition checks that the item type has not been

already added to the order, and the postcondition creates the relationship between the given order and the right item type.

Notice that we assume that the artifact instance that is returned by the first operation, *CreateNewCustomerOrder*, is reused in the following operations. This assumption is necessary to ensure that we are always dealing with the same artifact instance.

Listing 7.3: Code for task *AssignItemsToOrder*

```
operation AssignItemsToOrder(o:RequestedOrder, date:Date)
pre: o.itemType->forall(it | it.item->exists(i | i.sentOrder->isEmpty()))
post: o.oclsTypeOf(SentOrder) and not o.oclsTypeOf(RequestedOrder) and
      o.oclsType(SentOrder).sentDate=date and o.itemType->forall(it |
      o.oclsType(SentOrder).item-> includes(it.item@pre->select(i |
      i.sentOrder->isEmpty()).asOrderedSet()->first()))
```

AssignItemsToOrder checks whether for the given *RequestedOrder* *o* it is the case that there are available items (i.e., that have not been assigned to a *SentOrder*) for each of the requested item types. If so, *o* becomes a *SentOrder* that is associated to an available item for each of the requested item types.

These operation contracts show that the only elements that are created are the artifact itself and its relationships to other objects. Notice again that class *ItemType*, which is shared by *Order* and *SupplierRequest*, is never modified by the tasks, and is in fact read-only. Moreover, all the actions that are not attached to the initial transition take as input an instance of the artifact type whose evolution is being modeled in the corresponding state machine, as required by our methodology. Notice that the navigation of all the OCL expressions in the pre and postconditions starts from the instance of the artifact in the corresponding state machine diagrams: an *Order* (or one of its subclasses) in our example.

7.2 Results of Our Decidability Analysis

After presenting the background on 2-counter machines and introducing our example, the purpose of this section is to carefully analyze the interaction between the dynamic and static component of BAUML models, in order to identify the various sources of undecidability when it comes to their verification. We show in particular that all the restrictions we introduce towards decidability of verification are required: by relaxing just one of them, verification becomes again undecidable.

Among the properties of interest for BAUML models, we consider in particular the fundamental requirement of *artifact termination*. Intuitively, this

property states that in all possible evolutions of the system, whenever an artifact instance of a certain type is present in the system, it must persist in the system until it eventually reaches (in a finite amount of computation steps) a proper termination state. Remember that such a state will have a counterpart in the UML model of \mathcal{B} , which will contain a subclass for that specific state. By denoting with $\text{TERM}_A \in \text{A-STATES}(A)$ the proper termination state of artifact $A \in \text{ARTIFACTS}(\mathcal{B})$, and by considering the standard FOL encoding of UML classes as unary predicates, the artifact termination property can be formalized in $\mu\mathcal{L}_p$ as follows:

$$\nu Z. \left(\bigwedge_{A \in \text{ARTIFACTS}(\mathcal{B})} (\forall x. A(x) \rightarrow \mu Y. \text{TERM}_A(x) \vee (A(x) \wedge \langle \rightarrow Y)) \right) \wedge \text{[}Z$$

In the following, all the undecidability results we give do not only hold for the $\mu\mathcal{L}_p$ logic in general, but specifically for the artifact termination property. Furthermore, we do only consider data coming from a countably infinite unordered domain, and that can only be compared for (in)equality. We thus avoid any assumption on the structure of data domains, and consider only string and boolean attributes¹. In this light, our results witness that it is not possible to achieve meaningful restrictions towards decidability just by restricting the property specification logic, but that it is instead necessary to suitably restrict the expressiveness of BAUML models themselves.

The following subsections are structured according to the restrictions that the different theorems presented in each of them share in common: the first subsection deals with unrestricted models, the second subsection analyzes what we call “models with non-shared instances” and the last one deals with “models with shared instances”.

7.2.1 Unrestricted Models

Our analysis starts by showing that, if we do not impose restrictions on the shape of OCL queries used in the pre-/post-conditions of tasks and in the decision points of a BAUML model, then verification of artifact termination is undecidable. We say that a BAUML model is *unrestricted* if it does not impose any restriction on the shape of such queries.

Theorem 7.2.1. *Checking termination over unrestricted BAUML models is undecidable.*

¹A boolean attribute can be considered as a special string attribute that can only be assigned to the special strings true or false. This constraint can be easily expressed in OCL.

Proof. By reduction from the halting problem of 2-counter machines, which is undecidable (cf. Corollary 7.1.1). See page 204 for details. \square

The intuition behind the proof is that there is an unbounded number of artifacts and objects in the model, and that artifacts freely manipulate instances of other classes. Therefore, this can potentially lead to situations in which an infinite number of artifact and/or objects is created.

7.2.2 Models with Non-Shared Instances

As our goal is to determine the conditions over the initial models that guarantee decidability, the first restriction that we impose on this section is that the artifacts in business processes do not share objects which are created in the model. We emphasize this characteristic because the following section will give a brief analysis of business processes whose artifacts do share object instances.

Navigational and Unidirectional Models The proof of Theorem 7.2.1 relies on the fact that artifact instances freely manipulate (i.e., create, read, delete) instances of other classes. Towards decidability, we have therefore to properly control how artifact instances relate to other objects. For this reason, we suitably restrict OCL expressions, by allowing only so-called navigational expressions.

To define navigational queries over a BAUML model $\mathcal{B} = \langle \mathcal{M}, \mathcal{O}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$, we start by partitioning the associations and classes in \mathcal{M} into two sets: a read-only set \mathcal{M}_r , and a read-write set \mathcal{M}_{rw} . Intuitively, \mathcal{M}_r represents the portion of \mathcal{M} whose data are only accessed, but never updated, by the execution of tasks, whereas \mathcal{M}_{rw} represents the portion of \mathcal{M} that can be freely manipulated by the tasks. These two sets can either be directly specified by the modeler, or easily extracted by examining all postconditions of operations present in \mathcal{T} , marking a class C as read-write every time a sub-expression $obj.oclIsNew()$ appears in some operation, and obj is an instance of C . In this light, all artifacts present in \mathcal{M} are always part of the read-write set: $\text{ARTIFACTS}(\mathcal{M}) \subseteq \mathcal{M}_{rw}$.

Given an object obj , an OCL expression is *navigational from obj* if it is defined by means of the usual OCL operations like exists, select, . . . , but in which each subexpression is a boolean combination of expressions Q_i that obey to one of the following two types:

- Q_i only uses role and class names from \mathcal{M}_r ;

- Q_i has the form of a path $o.r_1 \cdots r_n$, which starts from o and navigates through roles r_1 to r_n , where each r_i is either a role or an attribute, and where o is either the original object *obj*, or a variable used in the current operation.

A BAUML model $\mathcal{B} = \langle \mathcal{M}, \mathcal{O}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$ is *navigational* if:

- For every operation in \mathcal{T} , with the exception of the *init* operation (i.e. the operation that creates the artifact), the OCL expressions used in its pre- and post-conditions are navigational from a , where a is (the name of) the artifact instance taken as input by the operation.
- Every condition in $\text{CONDITIONS}(\mathcal{B})$ is an OCL expression that is navigational from (the name of) the artifact instance present in the scope of the condition.

Navigational BAUML models do not allow artifact instances to *share* objects from read-write classes. Indeed, for an artifact instance to establish a relation with an object of class C previously created by another artifact instance, it is necessary to write an OCL query that selects objects of type C , but this query is not navigational.

In spite of this observation, we will see that restricting BAUML models to navigational queries is still not sufficient, but additional requirements are needed towards decidability. The first requirement is related to the way OCL expressions navigate the roles in \mathcal{M} . Given a navigational BAUML model $\mathcal{B} = \langle \mathcal{M}, \mathcal{O}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$, and given a role r in \mathcal{M} , if there exists an OCL expression in \mathcal{B} that mentions r , then we say that r is a *target role*, written $\text{TRG}_{\mathcal{B}}(r)$, otherwise we say that r is a *source role*, written $\text{SRC}_{\mathcal{B}}(r)$. We use this notion to define the notion of dependency between two classes.

Given classes C_1 and C_{n+1} in \mathcal{M} , we say that C_{n+1} *depends on* C_1 if there exists a tuple $\langle A_1, \dots, A_n \rangle$ of binary associations such that each A_i connects C_i and C_{i+1} , and the role of A_i attached to C_{i+1} is a target role. We then say that \mathcal{B} is *bidirectional* if it is navigational and there exists a class in \mathcal{M}_{rw} that depends on itself or on one of its super/sub-classes, *unidirectional* if it is navigational and there is no class in \mathcal{M}_{rw} that depends on itself or on one of its super/sub-classes. Intuitively, for a unidirectional BAUML model it is possible to mark each association in its UML model as directed (since no association can have both nodes as targets), and the resulting directed graph is acyclic. This property, in turn, can be tested in NLOGSPACE .

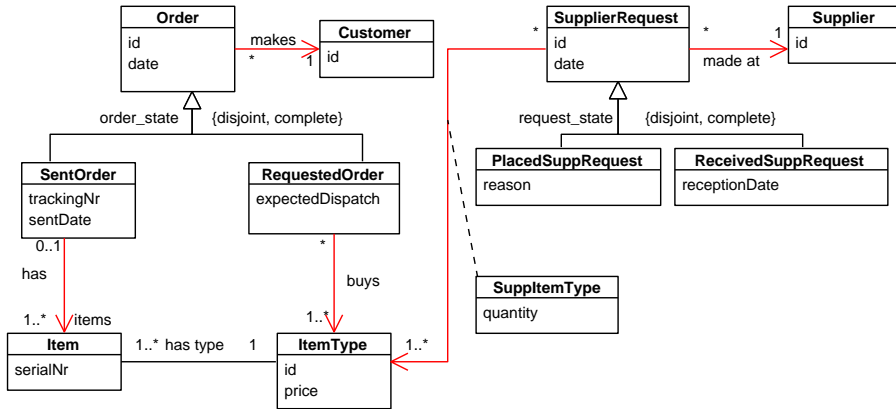


Figure 7.4: Class diagram for an online retailer example. The arrows indicate the direction in which the OCL queries navigate the class diagram.

Unfortunately, the following result shows that restricting BAUML models to be unidirectional is not sufficient to obtain decidability of checking termination properties.

Theorem 7.2.2. *Checking termination of unidirectional BAUML models is undecidable.*

In this case, the intuition behind the proof is that, despite having OCL queries that are navigational and unidirectional, still an unbounded number of objects can be created. The details of the proof can be found on page 207.

If we return to our example and examine the postconditions of the operations, we can see that the model fulfills the navigational and unidirectional restrictions. First of all, all the queries are navigational, because all the associations that are created in the OCL operation contracts select instances of classes which belong to the read-only set. Secondly, they are also unidirectional: Figure 7.4 shows the class diagram we had at the start, but now we have incorporated the information over the direction in which the diagram is navigated by the OCL queries. As it can be seen, there are no cycles, and thus it is unidirectional. Still, as we have seen, checking termination over this model is undecidable, as there could be an unlimited number of `ItemTypes` or `Items`, for instance.

Cardinality-Bounded Models To overcome the undecidability, which is due to the unbounded multiplicities in the target roles, we introduce the notion of cardinality-bounded BAUML model. A BAUML model $\mathcal{B} = \langle \mathcal{M}, \mathcal{O}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$ is *cardinality-bounded* if \mathcal{B} is navigational and each target role in \mathcal{M} has a bounded cardinality, i.e., is associated to a cardinality constraint whose upper bound is numeric. \mathcal{B} is *N-cardinality-bounded* if the maximum upper bound associated to a target role is N . If there exists at least a target role with unbounded cardinality, i.e., associated to a cardinality constraint whose upper bound is $*$, then \mathcal{B} is instead said to be *cardinality-unbounded*. Notice that no cardinality restriction is imposed, for cardinality-bounded models, on the cardinalities associated to roles that are not target roles.

With all these notions at hand, we are now able to state the main result of this chapter.

Theorem 7.2.3. *Let \mathcal{B} be an arbitrary unidirectional, cardinality-bounded BAUML model. Verifying whether \mathcal{B} satisfies a $\mu\mathcal{L}_p$ property navigationally compatible with \mathcal{B} is decidable, and reducible to finite-state model checking.*

The proof of this theorem can be found on page 209. The intuition behind the proof is the following. First of all, we have unidirectional navigation through the elements of the class diagram and artifact instances evolve independently one from the other. This leads to some kind of isolation property, which allows us to consider the artifact instances in the initial database plus an additional one. Secondly, the number of created elements in the diagram is bounded due to the bounded cardinalities in the target roles. In consequence, there is an upper bound on the total number of elements that can be created.

Bearing this in mind, we should bound the cardinalities in the target roles in our example to ensure decidability. Figure 7.5 shows the result of doing so.

An important open point is whether cardinality-boundedness is a sufficient restriction for decidability per se, i.e., without necessarily imposing unidirectionality. The following theorem provides a strong, negative answer to this question, witnessing that both restrictions are simultaneously required towards decidability.

Theorem 7.2.4. *Checking termination of 1-cardinality-bounded, bidirectional BAUML models is undecidable.*

The details of the proof of this theorem can be found on page 210. In this case, the source of undecidability is due to the fact that, although the cardinalities are bounded, the total number of instances is not. Therefore,

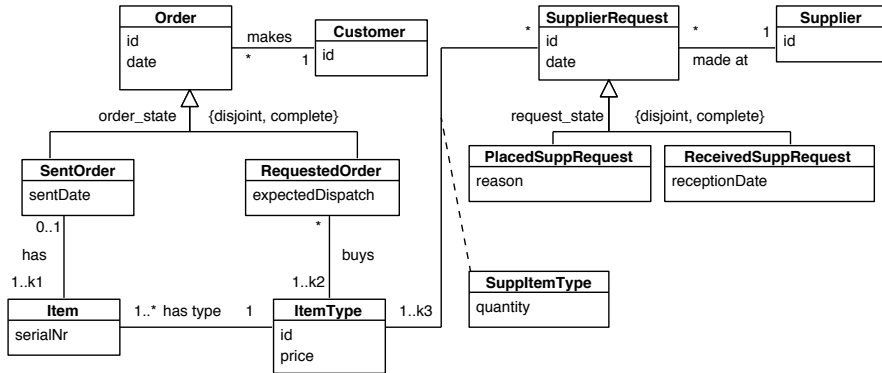


Figure 7.5: Class diagram for the online retailer example with bounded cardinalities.

the associations in the model could be navigated back and forth creating an unlimited number of class instances.

7.2.3 Models With Shared Instances

As argued in Section 7.2.2, unidirectional BAUML models are not able to make artifact instances share (read-write) objects. In this section, we study what happens if we relax unidirectionality so as to support this feature. An *unidirectional BAUML model with shared instances* $\mathcal{B} = \langle \mathcal{M}, \mathcal{O}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$ is a BAUML model in which, inside navigational expressions, it is possible to add free queries over \mathcal{M}_{rw} , as long as they *do not contain* the expression *oclIsNew()*. Intuitively, this means that new objects can only be created through standard navigational OCL expressions, but at the same time it is possible to establish associations with already existing objects that are not reachable by simply navigating from the artifact instance. The following theorem shows that this relaxation makes verification again undecidable.

Theorem 7.2.5. *Checking termination of 1-cardinality-bounded, unidirectional BAUML models with shared instances is undecidable.*

The proof of this theorem can be found on 212. The intuition for this proof relies on the fact the evolution of artifacts is no longer isolated, as they now share read-write instances.

We close this thorough analysis by showing that, if we introduce a bound on the number of artifact instances that are simultaneously active in the system, verification becomes decidable for this specific class of BAUML models. This technique cannot be applied to unrestricted nor unbounded BAUML models: by inspecting the proofs of Theorems 7.2.1 and 7.2.2 on pages 204 and 207, one can easily notice that undecidability holds even when there is just a single active artifact instance.

Theorem 7.2.6. *Verification of $\mu\mathcal{L}_p$ properties over cardinality-bounded, unidirectional BAUML models with shared instances of read-write classes is decidable and reducible to finite-state model checking when the number of simultaneously active artifact instances is bounded.*

The details of the proof of this theorem can be found on page 216. In this case, an artifact instance can only create a certain number of objects, and the number of simultaneously active artifact instance is also bounded. This leads to decidability.

Although our example falls in the case of Theorem 7.2.3, let's suppose for a moment that `ItemType` is a read-write class modified by one of the artifacts. In this case, both artifacts would be sharing a read-write relation. In consequence, we would require an *additional* bound on number of simultaneously active artifact instances, so as to fall into Theorem 7.2.6. It would be sufficient to add a bound to number of instances of a certain artifact. See Figure 7.6, classes `Order` and `SupplierRequest`, with multiplicities M and N , respectively.

It is important to observe that this bound still allows one to create an unbounded amount of artifact instances over time, provided that they do not accumulate in the same snapshot. In this light, Theorem 7.2.6 closely resembles the result given in [122] for business artifacts specified in the GSM notation.

7.2.4 Applicability of the Results to the Bicing Example

Given the results that we have obtained previously, we show their application to the Bicing example with two artifacts that we defined in Section 3.3 on page 50. In this example, users are allowed to rent more than one bicycle and the evolution of `User` and `Bicycle` is intertwined, e.g. when a user rents a bicycle, both the state of bicycle and user change.

By just looking at Figure 3.9 on page 52, we can see that the model does not fulfill the restrictions that ensure decidability. Artifacts `User` and `Bicycle` are directly connected to each other and they do not evolve independently.

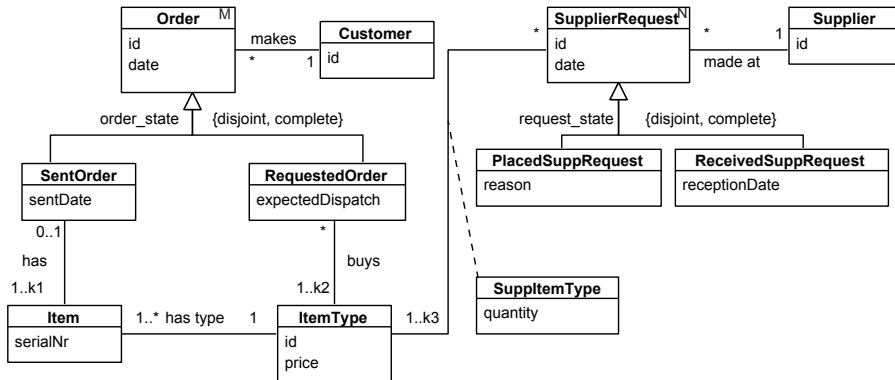


Figure 7.6: Class diagram with multiplicities in the artifacts' classes to ensure decidability.

Moreover, there is no bound on the number of artifact instances, which leads to undecidability.

7.3 Summary & Conclusions

This chapter has presented a decidability analysis for validating artifact-centric business process models defined using the BAUML framework. We have lifted the conditions for decidability from formal and low-level representations to the business level. Therefore, these conditions can be taken into consideration by the modeler of the process.

To sum up the results of the analysis, we have found that, in order to guarantee decidability when validating the models, *all* of the following conditions should be met:

- Artifacts should be related to a bounded number of objects.
- Two different artifacts can only share read-only objects.
- The OCL expressions should be navigational.
- There are no loops when navigating the associations between two classes, i.e. they are not navigated back and forth.

If the model contains shared read-write objects by the artifact-instances the validation becomes undecidable. To regain decidability, we need to establish a bound on the number of simultaneously active artifact instances.

Part IV

Closure

Chapter 8

Conclusions

The general aim of this PhD thesis was to contribute to the field of artifact-centric business process modeling. In particular, we had two main goals:

1. Find a way to model artifact-centric business process following the BALSA framework, using a high-level language that was easy to understand by the people involved in the business process and at the same time formal enough to avoid ambiguities.
2. Find a method to check the semantic correctness of the artifact-centric business process model as defined above.

This chapter summarizes the contributions in relation to these goals, and presents the conclusions associated to the results. After this, we point out possible ways of extending this work and give the details of the papers and articles that have been published and which are related to this thesis.

8.1 Contributions

This section presents our contributions and conclusions in relation to our research goals, which we presented in the Introduction. It is structured according to the two areas of our research goals: modeling business processes from an artifact-centric perspective and reasoning with them.

8.1.1 Modeling Artifact-centric Business Process Models

As we have explained, the first goal of this thesis was to find a way to model artifact-centric business processes using a high-level language that was understandable and with precise semantics to avoid ambiguities. The artifact-centric approach, in contrast to traditional process-centric modeling, specifies the data required by the business process, and in consequence, is also able to define precisely the meaning of the tasks.

One of the challenges in the artifact-centric world was to find the most appropriate model, depending on its purpose, to represent each of the dimensions of the BALSА framework. This framework defines four dimensions for artifact-centric business process models: the business artifacts, which show the relevant data for the business; the lifecycles, representing the evolution of the business artifacts; the services, which correspond to tasks, or units of work in which the business process is decomposed; and the associations, that establish restrictions over the services.

As we have explained in Chapter 2, most of the existing works either used representations which were intuitive but too informal to have a precise meaning or were grounded on logic, which made them very formal, but impractical from the point of view of business.

Therefore, we proposed in Chapter 3 a way to model business processes following an artifact-centric perspective which is based on the BALSА framework and uses a set of models grounded on UML and OCL. We call it the BAUML framework. To sum it up, we use a class diagram to represent the business artifacts, a state machine diagram to represent the lifecycle of these artifacts, a set of OCL operation contracts to define the details of the tasks or services, and activity diagrams to show the associations between the tasks. The chapter includes both an informal and a formal description of the framework, together with two examples to illustrate it.

Using UML and OCL provides us with a high-level (and graphical, in the case of UML) representation of business processes which is independent from their final implementation. In addition to this, they both are ISO standard languages and they avoid ambiguities. Finally, the two languages integrate naturally and can be used to represent all of the dimensions of the BALSА framework, as we have seen.

8.1.2 Reasoning on Artifact-centric Business Process Models

The second goal of the thesis was to find a way to determine the correctness of an artifact-centric business process model defined using the BAUML framework.

As we mentioned in the Introduction, there are several types of correctness that can be assessed. Because artifact-centric business process models define both the data and the details of the tasks, checking the semantic correctness of these models becomes possible. In contrast to checking only their syntactical or structural correctness, dealing with the semantic correctness allows us to check properties such as the executability of the tasks in the model or the liveness of the data. More importantly, it allows us to ensure that the model fulfills the business requirements.

Dealing with the semantic correctness of artifact-centric business process models has proved to be an important research topic, as we have already seen. However, the majority of works deal with models which are grounded on logic - thus their formality - but they have a low level of abstraction and are not practical from the point of view of the business. Therefore, our aim was to find a way to reason with the models in the BAUML framework.

We have contributed two different ways to do so. First of all, we have shown in Chapter 5 how to translate a BAUML model into a DCDS (Data-centric Dynamic System) in order to apply model checking techniques to ensure its semantic correctness. This shows that our approach is compatible with external frameworks.

However, as we have seen, the main drawback of DCDSs is that they have been proposed at a theoretical level and there is no tool that can perform the tests, although work has begun on that front.

Therefore, the second contribution of this part is proving the feasibility of our approach. As explained in Chapter 6, we have implemented a prototype tool, AuRUS-BAUML, which is able to translate a BAUML model into a first-order logic schema. AuRUS-BAUML then connects seamlessly to another tool, SVTe. Given a property or goal and a schema defined in logic, SVTe can tell us whether the goal can be achieved with the schema. This result is then presented to the user through AuRUS-BAUML. To make life easier for the user, he can define the BAUML models using ArgoUML, which can be exported as an XMI and then provided as input to AuRUS-BAUML.

Following this workflow, the user can answer several questions which deal with the semantic correctness of the initial model. This includes ensuring that the model fulfills the business requirements. We also provide a set of tests

that can be generated automatically from the models as a way to illustrate the potential of our proposal. We have also presented in Chapter 6 the translation process and the algorithms required to be able to obtain the logic translation of our BAUML models.

The last contribution in this field deals with the complexity of reasoning and is described in Chapter 7. We have proven that determining whether a BAUML model fulfills a certain property is undecidable. In order to guarantee decidability, the following conditions have to be met: the artifact should be associated to a limited number of objects, two artifacts can only share read-only objects, the OCL expressions must be navigational starting from the artifact instance which is manipulated, and the associations between two classes are not navigated back and forth.

If all these conditions are met, it is not necessary to bound the number of active artifact instances. On the other hand, if the two artifacts share read-write objects, in order to guarantee decidability we need to bound the number of active artifact instances.

8.2 Further Research

The research presented in this thesis can be further extended in several ways. We have applied the BAUML framework to various examples and a case study, and the next step would be to apply it to the corporate world. As we have mentioned, Léelo [82], a Spanish company, is interested in our approach to artifact-centric business process modeling, and on adapting our methodology to their needs. In fact, we have already begun work on that front.

In terms of modeling, incorporating other modeling notations such as BPMN for the associations or an ER diagram to represent the business artifacts would make our framework more flexible.

When it comes to reasoning with our BAUML models, we believe that there are many potential research contributions to be made. To begin with, we could deal with the simplifications we mentioned in Section 5.1.3 on page 81, e.g. by allowing action nodes in the activity diagrams for reasoning or incorporating subprocesses in the activity diagram. Although these simplifications do not alter the result of the reasoning, they would make life easier for the modeler.

Secondly, in order to broaden the applicability of our reasoning approaches, we could incorporate fork and join nodes in the translation process, in order to consider parallelism. In addition, we could also consider more than one artifact type for reasoning.

A further contribution would be to apply existing techniques to determine the syntactical and structural correctness of the BAUML models given as input before translating them into a DCDSs or into first-order logic for reasoning.

In terms of the tool itself, another possibility would be to optimize our tools (both AuRUS-BAUML and SVTe) to make them more efficient, improve the response time and to add compatibility with other modeling tools, other than ArgoUML, to represent the diagrams. And, last but not least, AuRUS-BAUML could be adapted to work with other modeling notations, as we mentioned previously.

Finally, when it comes to the study on decidability, further research could deal with the details on the interaction of read and write operations without weakening decidability. Another interesting aspect to study would be trying to control the exponentiality of data by partitioning it with the final goal of studying the practical application of the verification techniques.

8.3 Impact of the Thesis

The relevance of the work presented in this thesis is justified by the scientific publications that have been accepted in several international conferences and a book chapter. Moreover, it has also generated an interest in the industry: Léelo [82], a Spanish company based in Madrid, wished to adapt our methodology to their needs.

In this section we present the titles, authors and abstracts of the publications related to the thesis. It is divided into two subsections, one for each of the main contributions of this thesis. Within each subsection, the publications are listed in reverse chronological order.

8.3.1 Artifact-centric Business Process Modeling

Title: *Specifying Artifact-Centric Business Process Models in UML*
(ref. [48])

Authors: M. Estañol, A. Queralt, M.R. Sancho and E. Teniente
Published in: BMSD 2014 (Selected Papers), vol. 220 of LNBIP, Springer,
pp. 62-81
Year: 2015

– *Continues in the next page*

Title: Specifying Artifact-Centric Business Process Models in UML

– *Continued from previous page*

Abstract: In recent years, the artifact-centric approach to process modeling has attracted a lot of attention. One of the research lines in this area is finding a suitable way to represent the dimensions in this approach. Bearing this in mind, this paper proposes a way to specify artifact-centric business process models by means of well-known UML diagrams, from a high-level of abstraction and with a technology-independent perspective. UML is a graphical language, widely used and with a precise semantics.

Title: *Using UML to Specify Artifact-centric Business Process Models* (ref. [47])

Authors: M. Estañol, A. Queralt, M.R. Sancho and E. Teniente

Published in: BMSD 2014, SciTePress, pp. 84-93

Year: 2014

Abstract: Business process modeling using an artifact-centric approach has raised a significant interest over the last few years. One of the research challenges in this area is looking for different approaches to represent all the dimensions in artifact-centric business process models. Bearing this in mind, the present paper proposes how to specify artifact-centric business process models by means of diagrams based on UML. The advantages of basing our work on UML are many: it is a semi-formal language with a precise semantics; it is widely used and easy to understand; and it provides an artifact-centric specification which incorporates also some aspects of process-awareness.

Title: *Artifact-Centric Business Process Models in UML* (ref. [46])

Authors: M. Estañol, A. Queralt, M.R. Sancho and E. Teniente

– *Continues in the next page*

Title:	Artifact-Centric Business Process Models in UML – <i>Continued from previous page</i>
Published in:	BPM 2012 Workshops, vol. 132 of LNBIP, Springer, pp. 292-303
Year:	2013
Abstract:	Business process modeling using an artifact-centric approach has raised a significant interest over the last few years. This approach is usually stated in terms of the BALSAs framework which defines the four “dimensions” of an artifact-centric business process model: Business Artifacts, Lifecycles, Services and Associations. One of the research challenges in this area is looking for different diagrams to represent these dimensions. Bearing this in mind, the present paper shows how all the elements in BALSAs can be represented by using the UML language. The advantages of using UML are many. First of all, it is a formal language with a precise semantics. Secondly, it is widely used and understandable by both business people and software developers. And, last but not least, UML allows us to provide an artifact-centric specification for BALSAs which incorporates also some aspects of process-awareness.

8.3.2 Reasoning on Artifact-centric Business Process Models

Title:	<i>Verification and Validation of UML Artifact-centric Business Process Models</i> (ref. [51])
Authors:	M. Estañol, M.R. Sancho, E. Teniente
Published in:	CAiSE 2015, vol. 9097 of LNCS, Springer, pp. 434-449
Year:	2015

– *Continues in the next page*

Title: Verification and Validation of UML Artifact-centric Business Process Models

– *Continued from previous page*

Abstract: This paper presents a way of checking the correctness of artifact-centric business process models defined using the BAUML framework. To ensure that these models are free of errors, we propose an approach to verify (i.e. there are no internal mistakes) and to validate them (i.e. the model complies with the business requirements). This approach is based on translating these models into logic and then encoding the desirable properties as satisfiability problems of derived predicates. In this way, we can then use a tool to check if these properties are fulfilled.

Title: *Verifiable UML Artifact-Centric Business Process Models* (ref. [28])

Authors: D. Calvanese, M. Montali, M. Estañol, E. Teniente

Published in: CIKM 2014, ACM, pp. 1289-1298

Year: 2014

Abstract: Artifact-centric business process models have gained increasing momentum recently due to their ability to combine structural (i.e., data related) with dynamical (i.e., process related) aspects. In particular, two main lines of research have been pursued so far: one tailored to business artifact modeling languages and methodologies, the other focused on the foundations for their formal verification. In this paper, we merge these two lines of research, by showing how recent theoretical decidability results for verification can be fruitfully transferred to a concrete UML-based modeling methodology. In particular, we identify additional steps in the methodology that, in significant cases, guarantee the possibility of verifying the resulting models against rich first-order temporal properties. Notably, our results can be seamlessly transferred to different languages for the specification of the artifact lifecycles.

Title:	<i>Reasoning on UML Data-Centric Business Process Models</i> (ref. [50])
Authors:	M. Estañol, M.R. Sancho and E. Teniente
Published in:	ICSOC 2013, vol. 8274 of LNCS, Springer, pp. 437-445
Year:	2013
Abstract:	Verifying the correctness of data-centric business process models is important to prevent errors from reaching the service that is offered to the customer. Although the semantic correctness of these models has been studied in detail, existing works deal with models defined in low-level languages (e.g. logic), which are complex and difficult to understand. This paper provides a way to reason semantically on data-centric business process models specified from a high-level and technology-independent perspective using UML.

References

- [1] IEEE Standard for System and Software Verification and Validation. IEEE Std. 1012-2012 (2012)
- [2] Object Role Modeling: The official site for conceptual data modeling (2015), <http://www.orm.net/>
- [3] van der Aalst, W.M.P., Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing* 23(3), 333–363 (2011)
- [4] van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.): *Business Process Management, Models, Techniques, and Empirical Studies*, LNCS, vol. 1806. Springer (2000)
- [5] Aguilar-Savén, R.S.: Business process modelling: Review and framework. *International Journal of Production Economics* 90(2), 129–149 (2004)
- [6] April, J., Better, M., Glover, F., Kelly, J.P., Laguna, M.: Enhancing business process management with simulation optimization. In: Perrone, L.F., Lawson, B., Liu, J., Wieland, F.P. (eds.) *WSC 2006*. pp. 642–649. WSC (2006)
- [7] ArgoUML: ArgoUML (2015), <http://argouml.tigris.org/>
- [8] Awad, A., Decker, G., Lohmann, N.: Diagnosing and repairing data anomalies in process models. In: Rinderle-Ma, S., Sadiq, S.W., Leymann, F. (eds.) *Business Process Management Workshops*. LNBIP, vol. 43, pp. 5–16. Springer (2009)

-
- [9] Bagheri Hariri, B., Calvanese, D., De Giacomo, G., De Masellis, R., Felli, P.: Foundations of relational artifacts verification. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 379–395. Springer (2011)
- [10] Bagheri Hariri, B., Calvanese, D., Giacomo, G.D., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. CoRR abs/1203.0024 (2012)
- [11] Bagheri Hariri, B., Calvanese, D., Giacomo, G.D., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. In: Hull, R., Fan, W. (eds.) PODS. pp. 163–174. ACM (2013)
- [12] Bagheri Hariri, B., et al.: Verification of description logic knowledge and action bases. In: Raedt, L.D., et al. (eds.) ECAI. Frontiers in Artificial Intelligence and Applications, vol. 242, pp. 103–108. IOS Press (2012)
- [13] Bartsch, C., von Mevius, M., Oberweis, A.: Simulation of IT service processes with petri-nets. In: Feuerlicht, G., Lamersdorf, W. (eds.) ICSOC 2008 Workshops, ICSOC 2008 Revised Selected Papers. LNCS, vol. 5472, pp. 53–65. Springer (2008)
- [14] Basu, S., et al. (eds.): Service-Oriented Computing - 11th International Conference, ICSOC 2013, LNCS, vol. 8274. Springer (2013)
- [15] Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of deployed artifact systems via data abstraction. In: Kappel, G., Maamar, Z., Nezhad, H.R.M. (eds.) ICSOC 2011. LNCS, vol. 7084, pp. 142–156. Springer (2011)
- [16] Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of GSM-based artifact-centric systems through finite abstraction. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (eds.) ICSOC 2012. LNCS, vol. 7636, pp. 17–31. Springer (2012)
- [17] Bhattacharya, K., Caswell, N.S., Kumaran, S., Nigam, A., Wu, F.Y.: Artifact-centered operational modeling: lessons from customer engagements. IBM Syst. J. 46(4), 703–721 (Oct 2007)
- [18] Bhattacharya, K., Guthman, R., Lyman, K., Heath III, F.F., Kumaran, S., Nandi, P., Wu, F., Athma, P., Freiberg, C., Johannsen, L., Staudt, A.: A model-driven approach to industrializing discovery processes in pharmaceutical research. IBM Syst. J. 44(1), 145–162 (Jan 2005)

-
- [19] Bhattacharya, K., Gerede, C., Hull, R., Liu, R., Su, J.: Towards formal analysis of artifact-centric business process models. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 288–304. Springer (2007)
- [20] Bhattacharya, K., Hull, R., Su, J.: A Data-Centric Design Methodology for Business Processes. In: Handbook of Research on Business Process Management, pp. 1–28 (2009)
- [21] Boehm, B.W.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs (1981)
- [22] Böhmer, K., Rinderle-Ma, S.: A systematic literature review on process model testing: Approaches, challenges, and research directions. CoRR abs/1509.04076 (2015)
- [23] Borrego, D., Gasca, R.M., López, M.T.G.: Automating correctness verification of artifact-centric business process models. *Information & Software Technology* 62, 187–197 (2015)
- [24] Cabanillas, C., Knuplesch, D., Resinas, M., Reichert, M., Mendling, J., Ruiz Cortés, A.: Ralph: A graphical notation for resource assignments in business processes. In: Zdravkovic et al. [135], pp. 53–68
- [25] Cabanillas, C., Resinas, M., del-Río-Ortega, A., Ruiz Cortés, A.: Specification and automated design-time analysis of the business process human resource perspective. *Inf. Syst.* 52, 55–82 (2015)
- [26] Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL operation contracts. In: Leuschel, M., Wehrheim, H. (eds.) IFM. LNCS, vol. 5423, pp. 40–55. Springer (2009)
- [27] Calvanese, D., Giacomo, G.D., Lembo, D., Montali, M., Santoso, A.: Ontology-based governance of data-aware processes. In: Krötzsch, M., Straccia, U. (eds.) RR. LNCS, vol. 7497, pp. 25–41. Springer (2012)
- [28] Calvanese, D., Montali, M., Estañol, M., Teniente, E.: Verifiable UML artifact-centric business process models. In: Li, J., Wang, X.S., Garofalakis, M.N., Soboroff, I., Suel, T., Wang, M. (eds.) CIKM 2014. pp. 1289–1298. ACM (2014)

- [29] Calvanese, D., Montali, M., Estañol, M., Teniente, E.: Verifiable UML artifact-centric business process models (extended version). CoRR abs/1408.5094 (2014), <http://arxiv.org/abs/1408.5094>
- [30] Calvanese, D., Montali, M., Patrizi, F., Rivkin, A.: Implementing data-centric dynamic systems over a relational DBMS. In: Cali, A., Vidal, M. (eds.) Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management. CEUR Workshop Proceedings, vol. 1378. CEUR-WS.org (2015)
- [31] Cangialosi, P., Giacomo, G.D., Masellis, R.D., Rosati, R.: Conjunctive artifact-centric services. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 318–333. Springer (2010)
- [32] Chen, P.P.: The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.* 1(1), 9–36 (1976)
- [33] Choi, Y., Zhao, J.L.: Decomposition-Based Verification of Cyclic Workflows. In: Peled, D., Tsay, Y.K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 84–98. Springer (2005)
- [34] Choppy, C., Klai, K., Zidani, H.: Formal verification of UML state diagrams: a Petri net based approach. *ACM SIGSOFT Soft. Eng. Notes* 36(1), 1–8 (2011)
- [35] Curtis, B., Kellner, M.I., Over, J.: Process modeling. *Commun. ACM* 35(9), 75–90 (Sep 1992)
- [36] Damaggio, E., Deutsch, A., Hull, R., Vianu, V.: Automatic verification of data-centric business processes. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 3–16. Springer (2011)
- [37] Damaggio, E., Deutsch, A., Vianu, V.: Artifact systems with data dependencies and arithmetic. *ACM Trans. Database Syst.* 37(3), 22 (2012)
- [38] Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard-Stage-Milestone lifecycles. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 396–412. Springer (2011)
- [39] Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard – Stage –

- Milestone lifecycles. *Information Systems* 38(4), 561 – 584 (2013), special section on BPM 2011 conference
- [40] De Giacomo, G., Dumas, M., Maggi, F.M., Montali, M.: Declarative process modeling in BPMN. In: Zdravkovic et al. [135], pp. 84–100
- [41] Desel, J., Erwin, T.: Modeling, simulation and analysis of business processes. In: van der Aalst et al. [4], pp. 129–141
- [42] Deutsch, A., Hull, R., Patrizi, F., Vianu, V.: Automatic verification of data-centric business processes. In: Fagin, R. (ed.) *ICDT. ACM International Conference Proceeding Series*, vol. 361, pp. 252–267. ACM (2009)
- [43] van Dongen, B.F., van der Aalst, W.M.P., Verbeek, H.M.W.: Verification of EPCs: Using reduction rules and Petri nets. In: Pastor, O., Falcão e Cunha, J. (eds.) *CAiSE 2005. LNCS*, vol. 3520, pp. 372–386. Springer (2005)
- [44] Eshuis, R.: Symbolic model checking of UML activity diagrams. *ACM Trans. Softw. Eng. Methodol.* 15(1), 1–38 (2006)
- [45] Estañol, M., Queralt, A., Sancho, M.R., Teniente, E.: EU-Rent as an artifact-centric business process model: Technical report (2012), available at: <http://hdl.handle.net/2117/16928>, Ref: ESSI-TR-12-3
- [46] Estañol, M., Queralt, A., Sancho, M.R., Teniente, E.: Artifact-centric Business Process Models in UML. In: La Rosa, M., Soffer, P. (eds.) *Business Process Management Workshops 2012. LNBIP*, vol. 132, pp. 292–303. Springer (2013)
- [47] Estañol, M., Queralt, A., Sancho, M.R., Teniente, E.: Using UML to specify artifact-centric business process models. In: Shishkov, B. (ed.) *BMSD 2014 : Proceedings of the Fourth International Symposium on Business Modeling and Software Design*. pp. 84–93. SciTePress (2014)
- [48] Estañol, M., Queralt, A., Sancho, M.R., Teniente, E.: Specifying artifact-centric business process models in UML. In: Shishkov, B. (ed.) *BMSD 2014, Revised Selected Papers. LNBIP*, vol. 220, pp. 62–81. Springer (2015)
- [49] Estañol, M., Queralt, A., Sancho, M.R., Teniente, E.: Specifying artifact-centric business process models in UML: technical report (2015), available at: <http://hdl.handle.net/2117/28344>, Ref:ESSI-TR-15-2

- [50] Estañol, M., Sancho, M.R., Teniente, E.: Reasoning on UML data-centric business process models. In: Basu et al. [14], pp. 437–445
- [51] Estañol, M., Sancho, M., Teniente, E.: Verification and validation of UML artifact-centric business process models. In: Zdravkovic et al. [135], pp. 434–449
- [52] Fahland, D., Leoni, M.D., van Dongen, B.F., van der Aalst, W.M.P.: Behavioral conformance of artifact-centric process models. In: Abramowicz, W. (ed.) BIS 2011. LNBIP, vol. 87, pp. 37–49. Springer (2011)
- [53] Farré, C., Rull, G., Teniente, E., Urpí, T.: SVTe: a tool to validate database schemas giving explanations. In: Giakoumakis, L., Kossmann, D. (eds.) DBTest 2008. pp. 1–6. ACM (2008)
- [54] Fritz, C., Hull, R., Su, J.: Automatic construction of simple artifact-based business processes. In: Fagin, R. (ed.) ICDT. ACM International Conference Proceeding Series, vol. 361, pp. 225–238. ACM (2009)
- [55] Gerede, C.E., Su, J.: Specification and verification of artifact behaviors in business process models. In: Krämer, B.J., Lin, K.J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 181–192. Springer (2007)
- [56] Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.* 69(1-3), 27–34 (2007)
- [57] Gonzalez, P., Griesmayer, A., Lomuscio, A.: Verifying GSM-based business artifacts. In: Goble, C.A., Chen, P.P., Zhang, J. (eds.) 2012 IEEE 19th International Conference on Web Services. pp. 25–32. IEEE Computer Society (2012)
- [58] Gonzalez, P., Griesmayer, A., Lomuscio, A.: Model checking GSM-based multi-agent systems. In: Lomuscio, A., Nepal, S., Patrizi, F., Benatallah, B., Brandic, I. (eds.) ICSOC 2013 Workshops. LNCS, vol. 8377, pp. 54–68. Springer (2013)
- [59] Halpin, T.: *Conceptual Schema and Relational Database Design* (2nd Ed.). Prentice-Hall (1996)
- [60] Heath, F.T., et al.: Barcelona: A design and runtime environment for declarative artifact-centric BPM. In: Basu et al. [14], pp. 705–709

- [61] Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. *MIS Quarterly* 28(1), 75–105 (2004)
- [62] Hoch, R., Kaindl, H., Popp, R., Ertl, D., Horacek, H.: Semantic service specification for v&v of service composition and business processes. In: Bui, T.X., Jr., R.H.S. (eds.) *HICSS 2015*. pp. 1370–1379. IEEE (2015)
- [63] Hommes, B.J.: The Evaluation of Business Process Modeling Techniques. Ph.D. thesis, Technische Universiteit Delft (2004)
- [64] Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: Meersman, R., Tari, Z. (eds.) *OTM 2008*. LNCS, vol. 5332, pp. 1152–1163. Springer (2008)
- [65] Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath, F.T., Hobson, S., Linehan, M.H., Maradugu, S., Nigam, A., Sukaviriya, P., Vaculín, R.: Introducing the Guard-Stage-Milestone approach for specifying business entity lifecycles. In: Bravetti, M., Bultan, T. (eds.) *WS-FM 2010*. LNCS, vol. 6551, pp. 1–24. Springer (2011)
- [66] Hull, R., Damaggio, E., Masellis, R.D., Fournier, F., Gupta, M., Heath, F.T., Hobson, S., Linehan, M.H., Maradugu, S., Nigam, A., Sukaviriya, P.N., Vaculín, R.: Business artifacts with Guard-Stage-Milestone lifecycles: managing artifact interactions with conditions and events. In: Eyers, D.M., Etzion, O., Gal, A., Zdonik, S.B., Vincent, P. (eds.) *DEBS*. pp. 51–62. ACM (2011)
- [67] ISO: ISO/IEC 19505-2:2012 - OMG UML superstructure 2.4.1 (2012), available at: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=52854
- [68] ISO: ISO/IEC 19507:2012 - OMG OCL version 2.3.1 (2012), available at: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57306
- [69] ISO: ISO/IEC 19510:2013 Information technology – Object Management Group Business Process Model and Notation (2013), http://www.iso.org/iso/catalogue_detail.htm?csnumber=62652
- [70] ISO: ISO/IEC 19793:2015 - information technology – open distributed processing – use of UML for ODP system specifications (2015), available at: http://www.iso.org/iso/catalogue_detail.htm?csnumber=68641

- [71] Kardasis, P., Loucopoulos, P.: Expressing and organising business rules. *Information & Software Technology* 46(11), 701–718 (2004)
- [72] Kardasis, P., Loucopoulos, P.: A roadmap for the elicitation of business rules in information systems projects. *Business Proc. Manag. Journal* 11(4), 316–348 (2005)
- [73] Knuplesch, D., Ly, L.T., Rinderle-Ma, S., Pfeifer, H., Dadam, P.: On enabling data-aware compliance checking of business process models. In: Parsons, J., Saeki, M., Shoval, P., Woo, C.C., Wand, Y. (eds.) *ER 2010*. LNCS, vol. 6412, pp. 332–346. Springer (2010)
- [74] Künzle, V.: *Object-Aware Process Management*. Ph.D. thesis, Ulm University (2013)
- [75] Kucukoguz, E., Su, J.: On lifecycle constraints of artifact-centric workflows. In: Bravetti, M., Bultan, T. (eds.) *WS-FM 2010*. LNCS, vol. 6551, pp. 71–85. Springer (2011)
- [76] Kumaran, S., Liu, R., Wu, F.Y.: On the duality of information-centric and activity-centric models of business processes. In: Bellahsene, Z., Léonard, M. (eds.) *CAiSE 2008*. LNCS, vol. 5074, pp. 32–47. Springer (2008)
- [77] Kumaran, S., Nandi, P., Heath, T., Bhaskaran, K., Das, R.: ADoc-oriented programming. In: *SAINT*. pp. 334–343. IEEE Computer Society (2003)
- [78] Künzle, V., Reichert, M.: Philharmonicflows: towards a framework for object-aware process management. *Journal of Software Maintenance* 23(4), 205–244 (2011)
- [79] Küster, J.M., Ryndina, K., Gall, H.C.: Generation of business process models for object life cycle compliance. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) *BPM 2007*. LNCS, vol. 4714, pp. 165–181. Springer (2007)
- [80] Lankhorst, M.M., Proper, H.A., Jonkers, H.: The architecture of the archimate language. In: Halpin, T.A., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Ukor, R. (eds.) *BPMDS 2009 and EMMSAD 2009*. LNBP, vol. 29, pp. 367–380. Springer (2009)
- [81] Larman, C.: *Applying UML and Patterns*. Prentice Hall, 2nd edition edn. (2002)

- [82] Léelo: Léelo procesos documentales (2016), <http://leelo.es/>
- [83] Lin, H., Zhao, Z., Li, H., Chen, Z.: A novel graph reduction algorithm to identify structural conflicts. In: HICSS. p. 289. IEEE Computer Society (2002)
- [84] Linington, P.F., Milosevic, Z., Tanaka, A., Vallecillo, A.: The PhoneMob system: RM-ODP using ODP4UML (2011), available at: http://www.iso.org/iso/catalogue_detail.htm?csnumber=68641
- [85] Liu, R., Bhattacharya, K., Wu, F.Y.: Modeling business contexture and behavior using business artifacts. In: Krogstie, J., Opdahl, A., Sindre, G. (eds.) CAiSE 2007. LNCS, vol. 4495, pp. 324–339. Springer (2007)
- [86] Lohmann, N.: Compliance by design for artifact-centric business processes. *Inf. Syst.* 38(4), 606–618 (2013)
- [87] Lohmann, N., Wolf, K.: Artifact-centric choreographies. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 32–46. Springer (2010)
- [88] Lucas, F.J., Molina, F., Álvarez, J.A.T.: A systematic review of UML model consistency management. *Information & Software Technology* 51(12), 1631–1645 (2009)
- [89] Ly, L.T., Rinderle, S., Dadam, P.: Semantic correctness in adaptive process management systems. In: Dustdar, S., Fiadeiro, J., Sheth, A. (eds.) BPM 2006. LNCS, vol. 4102, pp. 193–208. Springer (2006)
- [90] March, S.T., Smith, G.F.: Design and natural science research on information technology. *Decis. Support Syst.* 15(4), 251–266 (Dec 1995)
- [91] Mendling, J., Verbeek, H.M.W., van Dongen, B.F., van der Aalst, W.M.P., Neumann, G.: Detection and prediction of errors in EPCs of the SAP reference model. *Data Knowl. Eng.* 64(1), 312–329 (2008)
- [92] Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 171–186. Springer (2013)
- [93] Minsky, M.L.: *Computation: Finite and Infinite Machines*. Prentice-Hall (1967)

- [94] Nigam, A., Caswell, N.S.: Business artifacts: an approach to operational specification. *IBM Syst. J.* 42(3), 428–445 (2003)
- [95] Moreno-Montes de Oca, I., Snoeck, M., Reijers, H.A., Rodríguez-Morffi, A.: A systematic literature review of studies on business process modeling quality. *Information & Software Technology* 58, 187–205 (2015)
- [96] Olivé, A.: *Conceptual Modeling of Information Systems*. Springer, Berlin (2007)
- [97] OMG: BPMN 2.0 by example (2010), <http://www.omg.org/spec/BPMN/20100601/10-06-02.pdf>
- [98] OMG: Business Process Model and Notation (BPMN) 2.0 (2013), <http://www.omg.org/spec/BPMN/2.0.2/PDF/>
- [99] Oriol, X.: Verificació i validació d'esquemes conceptuals UML/OCL amb operacions. Master's thesis, Universitat Politècnica de Catalunya (2012)
- [100] Pastor, O., Gómez, J., Insfrán, E., Pelechano, V.: The OO-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. *Inf. Syst.* 26(7), 507–534 (2001)
- [101] Popova, V., Dumas, M.: Discovering unbounded synchronization conditions in artifact-centric process models. In: Lohmann, N., Song, M., Wohed, P. (eds.) *BPM 2013 International Workshops, Revised Papers*. LNBI, vol. 171, pp. 28–40. Springer (2013)
- [102] Popova, V., Fahland, D., Dumas, M.: Artifact lifecycle discovery. *Int. J. Cooperative Inf. Syst.* 24(1) (2015)
- [103] Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng.* 73, 1–22 (2012)
- [104] Queralt, A., Teniente, E.: Specifying the semantics of operation contracts in conceptual modeling. In: *Journal on Data Semantics VII, LNCS*, vol. 4244, pp. 33–56. Springer (2006)
- [105] Queralt, A., Teniente, E.: Reasoning on UML conceptual schemas with operations. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) *CAiSE 2009*. pp. 47–62. LNCS, Springer (2009)

- [106] Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. *ACM Trans. Softw. Eng. Methodol.* 21(2), 13 (2012)
- [107] Ralyté, J., Khadraoui, A., Léonard, M.: Designing the shift from information systems to information services systems. *Business & Information Systems Engineering* 57(1), 37–49 (2015)
- [108] Redding, G., Dumas, M., ter Hofstede, A.H.M., Iordachescu, A.: Generating business process models from object behavior models. *IS Management* 25(4), 319–331 (2008)
- [109] Reggio, G., Leotta, M., Ricca, F.: "Precise is better than light" a document analysis study about quality of business process models. In: *EmpiRE 2011*. pp. 61–68. IEEE (2011)
- [110] Rinderle-Ma, S.: Data flow correctness in adaptive workflow systems. *EMISA Forum* 29(2), 25–35 (2009)
- [111] Rozinat, A., Mans, R.S., Song, M., van der Aalst, W.M.P.: Discovering simulation models. *Inf. Syst.* 34(3), 305–327 (2009)
- [112] Ruiz, M., Costal, D., España, S., Franch, X., Pastor, O.: Integrating the goal and business process perspectives in information system analysis. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) *CAiSE 2014*. LNCS, vol. 8484, pp. 332–346. Springer (2014)
- [113] Ruiz, M., Costal, D., España, S., Franch, X., Pastor, O.: Gobis: An integrated framework to analyse the goal and business process perspectives in information systems. *Inf. Syst.* 53, 330–345 (2015)
- [114] Rull, G., Farré, C., Queralt, A., Teniente, E., Urpí, T.: AuRUS: explaining the validation of UML/OCL conceptual schemas. *Software & Systems Modeling* 14(2), 953–980 (2015)
- [115] Rull, G., Farré, C., Teniente, E., Urpí, T.: Providing explanations for database schema validation. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) *DEXA*. LNCS, vol. 5181, pp. 660–667. Springer (2008)
- [116] Sadiq, W., Orłowska, M.E.: Analyzing process models using graph reduction techniques. *Inf. Syst.* 25(2), 117–134 (2000)

- [117] Scheer, A., Nüttgens, M.: ARIS architecture and reference models for business process management. In: van der Aalst et al. [4], pp. 376–389
- [118] Serral, E., De Smedt, J., Snoeck, M., Vanthienen, J.: Context-adaptive petri nets: Supporting adaptation for the execution context. *Expert Syst. Appl.* 42(23), 9307–9317 (2015)
- [119] Sidorova, N., Stahl, C., Trcka, N.: Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. *Inf. Syst.* 36(7), 1026–1043 (2011)
- [120] Snoeck, M., Michiels, C., Dedene, G.: Consistency by construction: The case of MERODE. In: Jeusfeld, M.A., Pastor, O. (eds.) *Conceptual Modeling for Novel Application Domains, ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM*. LNCS, vol. 2814, pp. 105–117. Springer (2003)
- [121] Software AG: ARIS business process analysis | Software AG, http://www.softwareag.com/corporate/products/aris_alfabet/bpa/products/
- [122] Solomakhin, D., Montali, M., Tessaris, S., Masellis, R.D.: Verification of artifact-centric systems: Decidability and modeling issues. In: Basu et al. [14], pp. 252–266
- [123] Störrle, H.: Semantics and verification of data flow in UML 2.0 activities. *Electr. Notes Theor. Comput. Sci.* 127(4), 35–52 (2005)
- [124] Straeten, R.V.D., Simmonds, J., Mens, T.: Detecting inconsistencies between UML models using Description Logic. In: Calvanese, D., et al. (eds.) *Description Logics*. CEUR Workshop Proceedings, vol. 81. CEUR-WS.org (2003)
- [125] Szwed, P.: Verification of archimate behavioral elements by model checking. In: Saeed, K., Homenda, W. (eds.) *CISIM 2015*. LNCS, vol. 9339, pp. 132–144. Springer (2015)
- [126] Teorey, T., Lightstone, S., Nadeau, T.: *Database Modeling and Design*. Morgan Kaufmann, San Francisco, fourth edn. (2006)
- [127] Trcka, N., van der Aalst, W.M.P., Sidorova, N.: Data-flow anti-patterns: Discovering data-flow errors in workflows. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) *CAiSE 2009*. LNCS, vol. 5565, pp. 425–439 (2009)

-
- [128] Universitat Politècnica de Catalunya & Universitat Oberta de Catalunya: EinaGMC, http://gui.fre.lsi.upc.edu/eina_GMC/
- [129] Utting, M., Legeard, B.: *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
- [130] Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal specification and testing of model transformations. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) *SFM 2012. Advanced Lectures*. LNCS, vol. 7320, pp. 399–437. Springer (2012)
- [131] Visual Paradigm International: *Visual Paradigm*, <http://www.visual-paradigm.com/>
- [132] Weber, I., Hoffmann, J., Mendling, J.: Beyond soundness: on the verification of semantic business process models. *Distributed and Parallel Databases* 27(3), 271–343 (2010)
- [133] Weske, M.: *Business Process Management: Concepts, Languages, Architectures*. Springer, Berlin Heidelberg (2007)
- [134] Yourdon, E.: *Just enough structured analysis* (2006), available at: <http://www.yourdon.com/jesa/JESA.pdf>
- [135] Zdravkovic, J., Kirikova, M., Johannesson, P. (eds.): *Advanced Information Systems Engineering - 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings*, LNCS, vol. 9097. Springer (2015)

Appendix A

Bicing: Full Example Specification

This appendix presents the full specification of our Bicing example, introduced in Chapter 3 in UML and OCL. The first part presents the example with one artifact (Section 3.1); the second part shows the specification for the example with two artifacts (Section 3.3).

For easier readability and reference, we include again the diagrams which appeared in Chapter 3.

A.1 One Artifact

A.1.1 Class Diagram

Figure A.1 shows its UML class diagram with the corresponding textual constraints stated in natural language. *Bicycle* is the only business artifact since we wish to track in the system the bicycle's evolution. A *Bicycle* may be in state *Available*, *InUse* or *Unusable* (we shortened the names for convenience; they should be called *AvailableBicycle*, etc.). The rest of the classes correspond to objects and specify the data required to rent a bicycle.

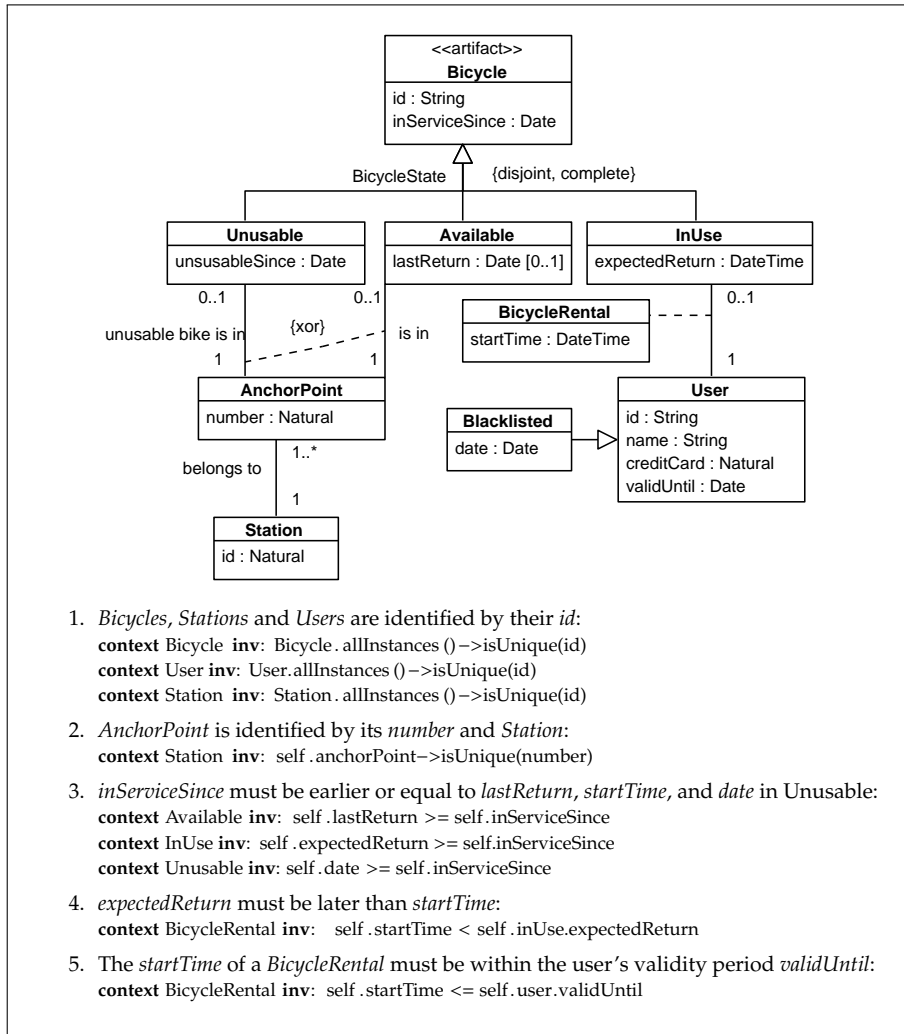


Figure A.1: Class diagram of our example with the corresponding integrity constraints.

A.1.2 State Machine Diagram

Figure A.2 shows the lifecycle of the artifact *Bicycle*. When a *Bicycle* is registered it is *Available*. When a *User* picks it up to rent it, he may return it to its anchor point if it is not in good shape and the bicycle is *Unusable*. Otherwise, it is *InUse*. When the user returns the bicycle, it is *Available* again. An *Unusable* bicycle may be repaired, so that it is again *Available*. Otherwise, it is destroyed.

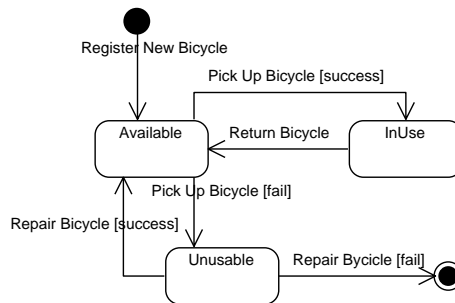


Figure A.2: State diagram of *Bicycle*.

A.1.3 Activity Diagrams & Operation Contracts

Register New Bicycle

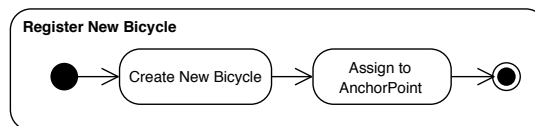


Figure A.3: Activity diagram of *Register New Bicycle*

```
operation createNewBicycle(): Bicycle
```

```
pre: -
```

```
post: Available.allInstances()->exists (b | b.oclIsNew() and
      b.oclIsTypeof(Available) and result=b)
```

```
operation assignToAnchorPoint(b: Bicycle, ap: AnchorPoint)
```

```
pre: -
```

```
post: ap.available = b.oclAsType(Available)
```

Pick Up Bicycle

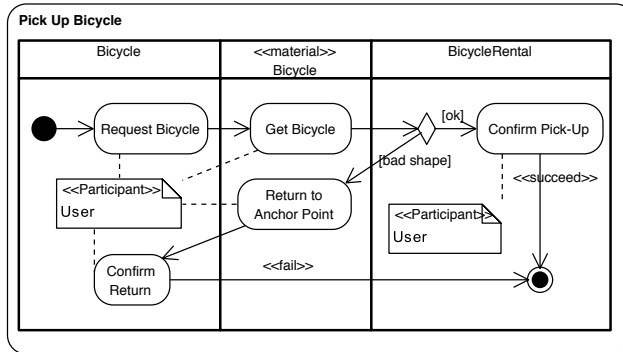


Figure A.4: Activity diagram for *Pick Up Bicycle*

```
operation requestBicycle(b: Bicycle)
```

```
pre: -
```

```
post: b.ocIsTypeOf(InUse) and not b.ocIsTypeOf(Available) and
      b.ocAsType(InUse).expectedReturn = now() + hour(3)
```

```
operation confirmPickUp(b: Bicycle , u: User)
```

```
pre: -
```

```
post: BicycleRental.allInstances()->exists(x | x.ocIsNew() and x.user=u and
      x.inUse = b.ocAsType(InUse) and x.startTime = now())
```

```
operation confirmReturn(b: Bicycle , ap: AnchorPoint)
```

```
pre: -
```

```
post: not b.ocIsTypeOf(InUse) and b.ocIsTypeOf(Unusable) and
      b.ocAsType(Unusable).anchorPoint=ap and
      b.ocAsType(Unusable).unusableSince = today()
```

Return Bicycle

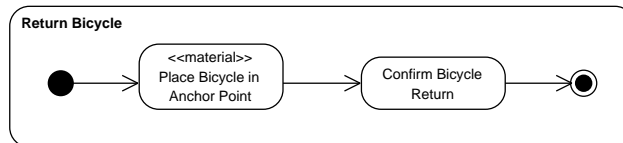


Figure A.5: Activity diagram for *Return Bicycle*

```

operation confirmBicycleReturn(b: Bicycle , ap: AnchorPoint)
pre: -
post: not b.ocIsTypeOf(InUse) and b.ocIsTypeOf(Available) and
      b.ocAsType(Available).anchorPoint = ap

```

Repair Bicycle

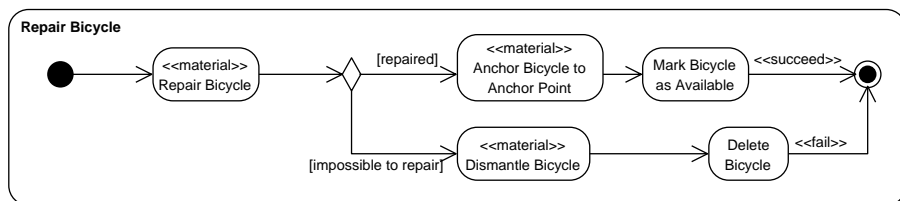


Figure A.6: Activity diagram for *Repair Bicycle*

```

operation markBicycleAsAvailable(b: Bicycle , ap: AnchorPoint)
pre: -
post: b.ocIsTypeOf(Available) and b.ocAsType(Available).anchorPoint=ap and not
      b.ocIsTypeOf(Unusable)

```

```

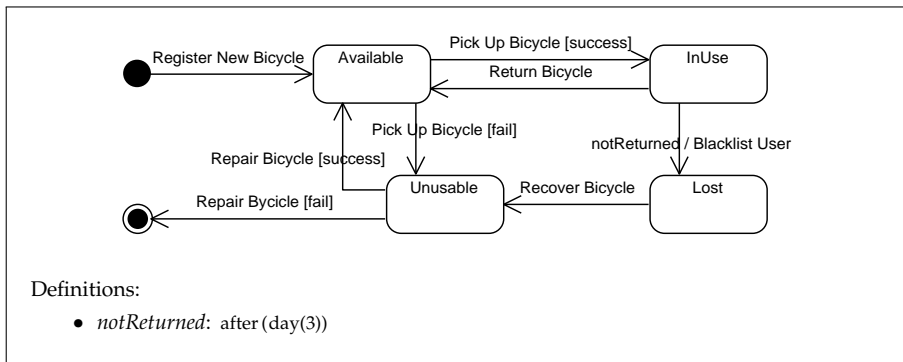
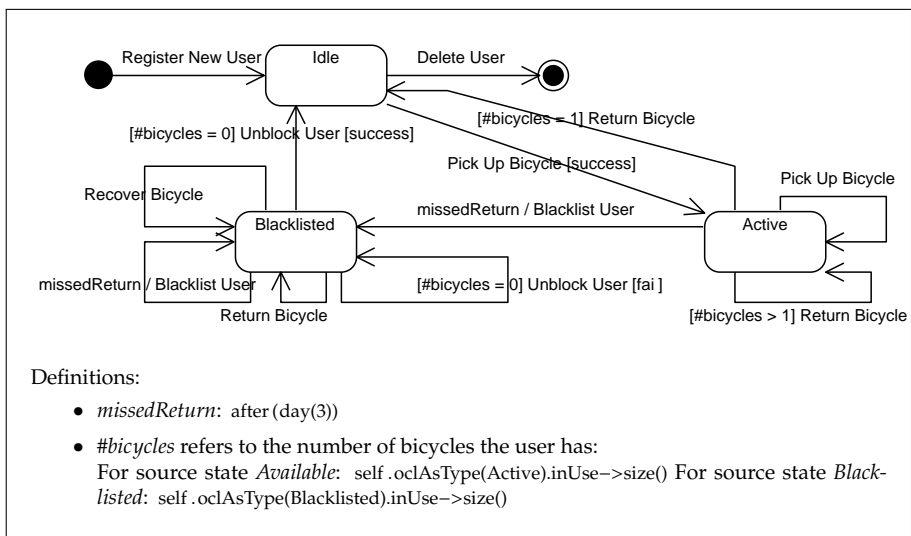
operation deleteBicycle(b: Bicycle)
pre: -
post: Bicycle.allInstances()->excludes(b)

```

A.2 Two Artifacts

This section presents the full specification for the Bicing example with two artifacts. Note that the diagrams and/or operation contracts which do not change from the example with one artifact are *not* included.

A.2.2 State Machine Diagram

Figure A.8: State machine diagram showing the evolution of the artifact *Bicycle*.Figure A.9: State machine diagram showing the evolution of the artifact *User*.

A.2.3 Activity Diagrams & Operation Contracts

Pick Up Bicycle

Listing A.1: Code for task *Confirm Pick Up*

```

operation confirmPickUp(b: Bicycle , u: User)
pre: -
post: BicycleRental.allInstances()->exists(br | br.ocIsNew() and
      u.ocIsTypeOf(Active) and br.active=u.ocAsType(Active) and not
      u.ocIsTypeOf(Idle) and br.startTime=now() and br.inUse =
      b.ocAsType(InUse))

```

Return Bicycle

Listing A.2: Code for task *Confirm Bicycle Return*

```

operation confirmBicycleReturn(b: Bicycle , ap:AnchorPoint)
pre: -
post: not b.ocIsTypeOf(InUse) and b.ocIsTypeOf(Available) and
      b.ocAsType(Available).lastReturn = today() and
      b.ocAsType(Available).anchorPoint = ap and
(u.ocIsTypeOf(Active) and u.ocAsType(Active).inUse@pre->size(>1 implies
      u.ocIsTypeOf(Idle) and u.ocAsType(Idle).lastRental=today())

```

Recover Bicycle

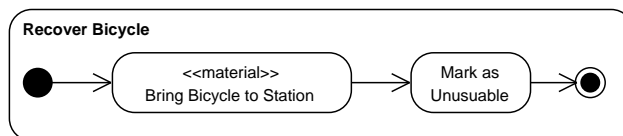


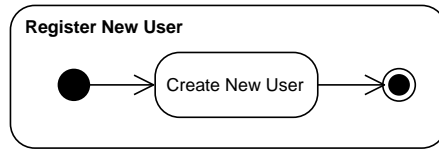
Figure A.10: Activity diagram for *Recover Bicycle*

Listing A.3: Code for task *Mark as Unusable*

```

operation markAsUnusable(b: Bicycle , ap:AnchorPoint)
pre: -
post: not b.ocIsTypeOf(Lost) and b.ocIsTypeOf(Unusable) and
      b.ocAsType(Unusable).anchorPoint = ap and
      b.ocAsType(Unusable).unusableSince = today()

```

Figure A.11: Activity diagram for *Register New User*

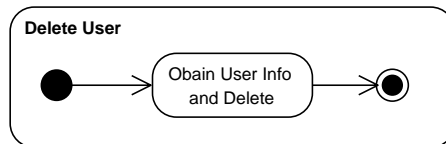
Register New User

Listing A.4: Code for task *RegisterNewUser*

```

operation registerNewUser(uId: String, uName:String, uMail: String, birth: Date,
    card: Natural, validity: Date)
pre: -
post: User.allInstances()->exists(u | u.ocllsNew() and u.id=uId and u.name=uName
    and u.email=uMail and u.dateOfBirth = birth and u.creditCard=card and
    u.validUntil = validity and u.ocllsTypeOf(Idle))
  
```

Delete User

Figure A.12: Activity diagram for *Delete User*Listing A.5: Code for task *DeleteUser*

```

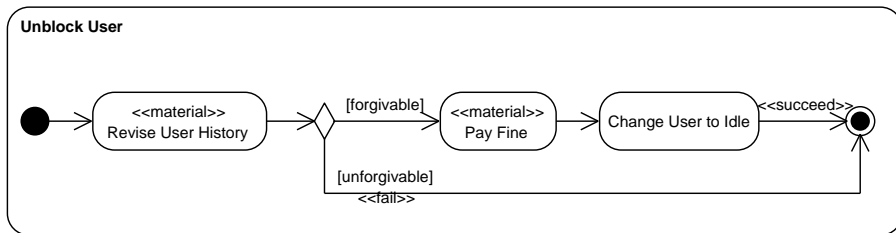
operation deleteUser(u: User)
pre: -
post: User.allInstances()->excludes(u)
  
```

Unblock User

Listing A.6: Code for task *Change User to Idle*

```

operation changeUserToIdle(u: User)
pre: -
post: not u.ocllsTypeOf(Blacklisted) and u.ocllsTypeOf(Idle)
  
```

Figure A.13: Activity diagram for *Unblock User*

Appendix B

Translation of Bicing into a DCDS

This appendix contains the translation into a DCDS of the Bicing example with one artifact, first introduced in section 3.1. The translation has been done following the process described in Chapter 5.

B.1 Data dimension

Figure B.1 shows the database schema required to store the information, although it is missing the tables that are used to store the data when tasks required to be translated into two different actions.

Notice also that we omitted the dates.

B.2 Condition-Action Rules

This section shows the condition-action rules that result from the translation of the Bicing example.

Lifecycle

These are the condition-action rules that are obtained from the state machine diagram.

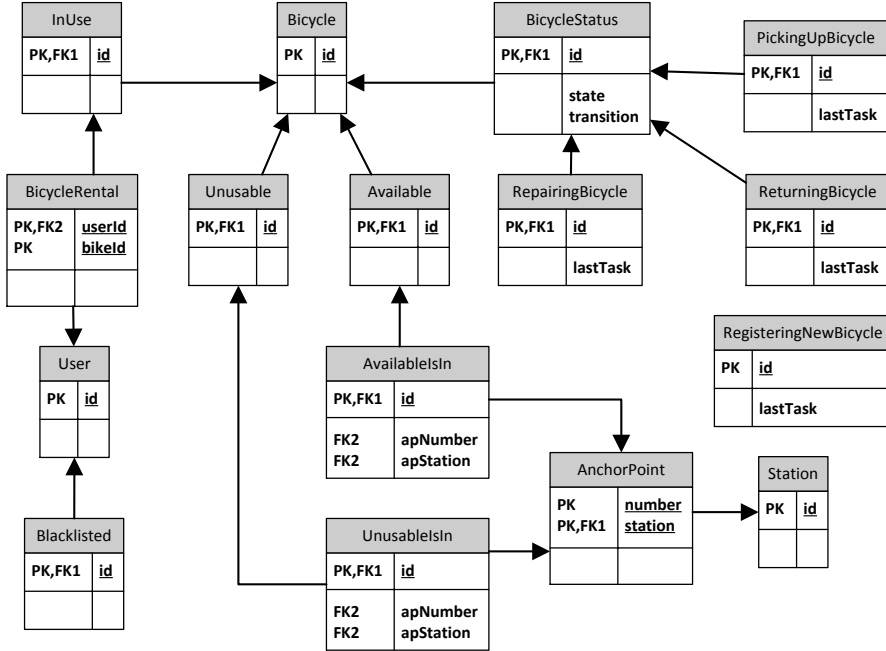


Figure B.1: Database schema containing the tables to store the information. We have not included tables *Busy*, *aux* nor the ones required for the splitting of the tasks, to keep it clearer.

$$\begin{aligned}
 & \neg \text{Busy}(id') \mapsto \text{RegisterNewBicycle}() \\
 & \neg \text{Busy}(id') \wedge \text{BicycleStatus}(id, 'Available', 'none') \mapsto \text{PickUpBicycle}(id) \\
 & \neg \text{Busy}(id') \wedge \text{BicycleStatus}(id, 'InUse', 'none') \mapsto \text{ReturnBicycle}(id) \\
 & \neg \text{Busy}(id') \wedge \text{BicycleStatus}(id, 'Unusable', 'none') \mapsto \text{RepairBicycle}(id)
 \end{aligned}$$

Associations

This subsection shows the condition-action rules that result from the translation of the activity diagram and the tasks/operation contracts.

$$\begin{aligned} \text{RegisteringNewBicycle}(id, 'none') \wedge \neg \text{Bicycle}(id) &\mapsto \text{CreateNewBicycle}(id) \\ \text{RegisteringNewBicycle}(id, 'CreateNewBicycle') &\mapsto \text{AssignToAnchorPoint}(id) \end{aligned}$$

$$\begin{aligned} \text{PickingUpBicycle}(id, 'none') &\mapsto \text{RequestBicycle}(id) \\ \text{PickingUpBicycle}(id, 'RequestBicycle') &\mapsto \text{ConfirmPickUp1}(id) \\ \text{PickingUpBicycle}(id, 'ConfirmPickUp1') &\mapsto \text{ConfirmPickUp2}(id) \\ \text{PickingUpBicycle}(id, 'RequestBicycle') &\mapsto \text{ConfirmReturn}(id) \end{aligned}$$

$$\text{ReturningBicycle}(id, 'none') \mapsto \text{ConfirmBicycleReturn}(id)$$

$$\begin{aligned} \text{RepairingBicycle}(id, 'none') &\mapsto \text{MarkBicycleAsAvailable}(id) \\ \text{RepairingBicycle}(id, 'none') &\mapsto \text{DeleteBicycle}(id) \end{aligned}$$

B.2.1 Actions

This section shows the details of the actions of the DCDS. We begin by looking at the actions from the state machine diagram, and afterwards we present the actions that correspond to the tasks.

Lifecycle

This first subsection shows the actions resulting from the translation of the state machine diagram, which are in charge of making explicit the implicit connection between the state machine and activity diagrams.

RegisterNewBicycle()

$$\begin{aligned} \text{true} &\rightsquigarrow \text{RegisteringNewBicycle}(\text{getBicycleId()}, 'none') \\ \text{true} &\rightsquigarrow \text{Busy}(\text{getBicycleId}()) \end{aligned}$$

PickUpBicycle(id)

$true \rightsquigarrow BicycleStatus(id, 'Available', 'PickingUp')$
 $true \rightsquigarrow PickingUpBicycle(id, 'none')$
 $true \rightsquigarrow Busy(id)$

ReturnBicycle(id)

$true \rightsquigarrow BicycleStatus(id, 'InUse', 'Returning')$
 $true \rightsquigarrow ReturningBicycle(id, 'none')$
 $true \rightsquigarrow Busy(id)$

RepairBicycle(id)

$true \rightsquigarrow BicycleStatus(id, 'Unusable', 'Repairing')$
 $true \rightsquigarrow RepairingBicycle(id, 'none')$
 $true \rightsquigarrow Busy(id)$

Services / Tasks

This section presents the translation of the tasks in the activity diagrams into DCDS actions.

Register New Bicycle

CreateNewBicycle(id):

$true \rightsquigarrow Bicycle(id)$
 $true \rightsquigarrow Available(id)$
 $true \rightsquigarrow RegisteringNewBicycle(id, 'RegisterNewBic')$

AssignToAnchorPoint(id):

$true \rightsquigarrow AvailableIsIn(id, getAPNr(), getStId())$
 $true \rightsquigarrow BicycleStatus(id, 'Available', 'none')$
 $BicycleStatus(id', x, y) \wedge id \neq id' \rightsquigarrow BicycleStatus(id', x, y)$
 $RegisteringNewBicycle(id', x) \wedge id \neq id' \rightsquigarrow RegisteringNewBicycle(id', x)$

PickUpBicycle**RequestBicycle(id):**

$$\begin{aligned} & true \rightsquigarrow \text{InUse}(id) \\ & \text{Available}(id') \wedge id \neq id' \rightsquigarrow \text{Available}(id') \\ \text{AvailableIsIn}(id', apNr, apSt) \wedge id \neq id' & \rightsquigarrow \text{AvailableIsIn}(id', apNr, apSt) \\ \text{PickingUpBicycle}(id, 'none') & \rightsquigarrow \text{PickingUpBicycle}(id, 'RequestBicycle') \end{aligned}$$
ConfirmPickUp1(id):

$$\begin{aligned} & true \rightsquigarrow \text{OutConfirmPickUp1}(id, \text{getUserID}()) \\ & true \rightsquigarrow \text{PickingUpBicycle}(id, 'ConfirmPickUp1') \\ \text{PickingUpBicycle}(id', x) \wedge id \neq id' & \rightsquigarrow \text{PickingUpBicycle}(id', x) \end{aligned}$$
ConfirmPickUp2(id):

$$\begin{aligned} & \text{OutConfirmPickUp1}(id, uid) \rightsquigarrow \text{BicycleRental}(id, uid) \\ & true \rightsquigarrow \text{BicycleStatus}(id, 'InUse', 'none') \\ \text{BicycleStatus}(id', x, y) \wedge id \neq id' & \rightsquigarrow \text{BicycleStatus}(id', x, y) \\ \text{PickingUpBicycle}(id', x) \wedge id \neq id' & \rightsquigarrow \text{PickingUpBicycle}(id', x) \end{aligned}$$
ConfirmReturn(id):

$$\begin{aligned} & true \rightsquigarrow \text{UnusableIsIn}(id, \text{getAPNr}(), \text{getStId}()) \\ & true \rightsquigarrow \text{BicycleStatus}(id, 'Unusable', 'none') \\ \text{BicycleStatus}(id', x, y) \wedge id \neq id' & \rightsquigarrow \text{BicycleStatus}(id', x, y) \\ \text{PickingUpBicycle}(id', x) \wedge id \neq id' & \rightsquigarrow \text{PickingUpBicycle}(id', x) \end{aligned}$$
Return Bicycle

ConfirmBicycleReturn(id):

$$\begin{aligned} & \text{true} \rightsquigarrow \text{Available}(id) \\ & \text{InUse}(id') \wedge id' \neq id \rightsquigarrow \text{InUse}(id') \\ & \text{true} \rightsquigarrow \text{AvailableIsIn}(id, \text{getAPNr}(), \text{getStId}()) \\ & \text{true} \rightsquigarrow \text{BicycleStatus}(id, \text{'Available'}, \text{'none'}) \\ & \text{BicycleStatus}(id', x, y) \wedge id' \neq id \rightsquigarrow \text{BicycleStatus}(id', x, y) \\ & \text{ReturningBicycle}(id', x) \wedge id' \neq id \rightsquigarrow \text{ReturningBicycle}(id', x) \end{aligned}$$
Repair Bicycle**MarkBicycleAsAvailable(id):**

$$\begin{aligned} & \text{true} \rightsquigarrow \text{Available}(id) \\ & \text{Unusable}(id') \wedge id' \neq id \rightsquigarrow \text{Unusable}(id') \\ & \text{true} \rightsquigarrow \text{BicycleStatus}(id, \text{'Available'}, \text{'none'}) \\ & \text{BicycleStatus}(id', x, y) \wedge id' \neq id \rightsquigarrow \text{BicycleStatus}(id', x, y) \\ & \text{RepairingBicycle}(id', x) \wedge id' \neq id \rightsquigarrow \text{RepairingBicycle}(id', x) \end{aligned}$$
DeleteBicycle(id):

$$\begin{aligned} & \text{Bicycle}(id') \wedge id' \neq id \rightsquigarrow \text{Bicycle}(id') \\ & \text{Unusable}(id') \wedge id' \neq id \rightsquigarrow \text{Unusable}(id') \\ & \text{UnusableIsIn}(id', \text{apNr}, \text{apSt}) \wedge id' \neq id \rightsquigarrow \text{UnusableIsIn}(id', \text{apNr}, \text{apSt}) \\ & \text{BicycleStatus}(id', x, y) \wedge id' \neq id \rightsquigarrow \text{BicycleStatus}(id', x, y) \\ & \text{RepairingBicycle}(id', x) \wedge id' \neq id \rightsquigarrow \text{RepairingBicycle}(id', x) \end{aligned}$$

Appendix C

Complexity: Proofs

This appendix contains the proofs of the main theorems presented in Chapter 7. It first begins by a formal introduction to counter machines, and in particular, to 2-counter machines. Afterwards it describes the proofs themselves.

C.1 Background on 2-Counter Machines

We follow the original formulation in [93]. A *counter* is a memory register that stores a non-negative integer. Given two positive integers $n, m \in \mathbb{N}^+$, an *m-counter machine* C with counters c_1, \dots, c_m is a program with n commands:

$$1 : \text{CMD}_1; \quad 2 : \text{CMD}_2; \quad \dots \quad n : \text{HALT};$$

where each CMD_k (for *index* $k \in \{1, \dots, n-1\}$) is either an increment command or a conditional decrement command.

Given $i \in \{1, \dots, m\}$, an *increment command* for counter i , written $\text{INC}(i)$, is a command that increases the counter c_i of one unit, and then jumps to the next instruction. Formally, for $k, k' \in \{1, \dots, n-1\}$,

$$k : \text{INC}(i, k') \quad \text{means} \quad k : c_i := c_i + 1; \text{GOTO } k';$$

Given $i \in \{1, \dots, m\}$, and $k, k', k'' \in \{1, \dots, n\}$, a *conditional decrement instruction* for counter i and instruction k , written $\text{CDEC}(i, k', k'')$, tests whether the value of counter i is zero. If so, it jumps to instruction k' ; otherwise, it decreases counter i of one unit, and then jumps to instruction k'' . Formally, for $k, k', k'' \in \{1, \dots, n-1\}$, command $k : \text{CDEC}(i, k', k'')$ means

$$k : \text{if } c_i = 0 \text{ then GOTO } k'; \text{else } \{c_i := c_i - 1; \text{GOTO } k'';\}$$

An *input* for an m -counter machine is an m -tuple $\langle d_1, \dots, d_m \rangle$ of values in \mathbb{N} initializing its counters. Given an m -counter machine C and an input I of size m , we say that C *halts on input* I if the execution of C with counter initial values set by I eventually reaches the last, HALT command.

It is well-known that checking whether a 2-counter machine halts on a given input is undecidable [93], and it is easy to strengthen this result as follows:

Corollary C.1.1. *It is undecidable to check whether a 2-counter machine halts on input $\langle 0, 0 \rangle$.*

In the following, we say that a 2-counter machine halts if it halts on input $\langle 0, 0 \rangle$.

C.2 Theorems' Proofs

C.2.1 Unrestricted Models

Theorem 7.2.1. *Checking termination over unrestricted BAUML models is undecidable.*

Proof. By reduction from the halting problem of 2-counter machines, which is undecidable (cf. Corollary 7.1.1). Specifically, given a 2-counter machine C , we produce a corresponding unrestricted BAUML model $\mathcal{B}_C = \langle \mathcal{M}^u, \emptyset, \{S_{2CM}^u\}, \{P_{init}^u, P_{run}^u\}, \{\text{init}, \text{inc}_1, \text{dec}_1, \text{inc}_2, \text{dec}_2, \text{halt}\} \rangle$, whose components are illustrated in Table C.1. The idea behind the reduction is as follows. \mathcal{M} contains a single artifact 2CM, which can be ready or halted, the latter being the termination state ($\text{TERM}_{2CM} = \text{Halted2CM}$), as it can be clearly seen in S_{2CM}^u . As specified in diagram S_{2CM} , the *init* operation is activated only if the extension of *Flag* is empty. In this case, a new artifact instance of *Ready2CM* and a new object of type *Flag* are simultaneously created. The creation of a *Flag* object has the effect of blocking the possibility of creating new instances of *Ready2CM*, in turn ensuring that only a single instance of *Ready2CM* will be created, and that only one execution of P_{run} will run. In fact, the only instance of 2CM that enters S_{2CM} will move to the *halted* state by executing the activity diagram P_{run} . In turn, P_{run} encodes the program of C , by combining the process fragments obtained by translating the single commands in C as specified in Table C.1. Two classes Item_1 and Item_2 are used to mirror the two counters. In particular, at a given moment in time, the number of instances of Item_i represents the value of counter i . In this light:

- incrementing counter i translates into the creation of a new instance of Item_i ;
- testing whether counter i is 0 translates into checking whether the extension of class Item_i is empty;
- decrementing counter i translates into the deletion of one of the current instances of Item_i .

Table C.1 shows how these three aspects can be formalized in terms of activity diagrams and OCL queries (focusing on counter 1). The diamond gateways at the beginning of each fragment are used to properly merge multiple incoming paths.

The claim follows by observing that C halts if and only if the unique instance of 2CM that enters $S_{2\text{CM}}$ also reaches the $\text{Halted}2\text{CM}$ state, i.e., properly terminates. □

C.2.2 Models with Non-Shared Instances

Navigational and Unidirectional Models Before presenting the proofs corresponding to the navigational and unidirectional models, we characterize navigation in $\mu\mathcal{L}_p$. Without loss of generality, we consider only binary relations¹. A *pseudo-navigational* $\mu\mathcal{L}_p$ property has the form:

$$\begin{aligned} \Phi ::= & \text{true} \mid \text{false} \mid A(x) \mid \neg A(x) \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \\ & Z \mid \mu Z.\Phi \mid \nu Z.\Phi \mid \\ & \exists x.A(x) \wedge \Phi(x) \mid \forall x.A(x) \rightarrow \Phi(x) \mid \\ & \exists y.R(x, y) \wedge \Phi(y) \mid \forall y.R(x, y) \rightarrow \Phi(y) \mid \\ & \exists y.R(y, x) \wedge \Phi(y) \mid \forall y.R(y, x) \rightarrow \Phi(y) \mid \\ & A(x) \wedge \langle \neg \rangle \Phi \mid A(x) \wedge [\neg] \Phi \mid A(x) \rightarrow \langle \neg \rangle \Phi \mid A(x) \rightarrow [\neg] \Phi \end{aligned}$$

where, in the last row, variable x is exactly the single free variable of Φ , once we substitute to each bounded predicate variable Z in Φ its bounding formula $\mu Z.\Phi'$ (resp., $\nu Z.\Phi'$). Notice that pseudo-navigational properties are in negation normal form, and that they constitute indeed a fragment of $\mu\mathcal{L}_p$. In fact, even if they do not make use of LIVE , they always guard quantification and next-state transitions with classes and/or relations, which imply the corresponding quantified objects to be in the current active domain.

¹Non-binary relations can be removed through reification.

Given a unidirectional BAUML model $\mathcal{B} = \langle \mathcal{M}, \mathcal{O}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$, we characterize the fact that a closed, pseudo-navigational $\mu\mathcal{L}_p$ property Φ is *navigationally compatible* with \mathcal{B} as:

- Φ contains a subformula of the form $\exists x.A(x) \wedge \Psi(x)$ or $\forall x.A(x) \rightarrow \Psi(x)$.
- The largest subformula of Φ of the form $\exists x.A(x) \wedge \Psi(x)$ or $\forall x.A(x) \rightarrow \Psi(x)$ is such that:
 - $A \in \text{A-CLASSES}(\mathcal{B})$, and
 - A and x are compatible with Ψ , written $\text{CMP}_A^x(\Psi) = \text{true}$, according to the notion of compatibility defined below.

Given a class C in \mathcal{M} , a variable x , and a pseudo-navigational open $\mu\mathcal{L}_p$ property $\Phi(x)$, we define $\text{CMP}_C^x(\Phi)$ as:

1. **true** if $\Phi \in \{\text{true}, \text{false}, Z\}$
2. $C \sqsubseteq_{\mathcal{M}} A \vee A \sqsubseteq_{\mathcal{M}} C$ if $\Phi \in \{A(x), \neg A(x)\}$
3. $\text{CMP}_C^x(\Phi_1) \wedge \text{CMP}_C^x(\Phi_2)$ if $\Phi \in \{\Phi_1 \wedge \Phi_2, \Phi_1 \vee \Phi_2\}$
4. $\text{CMP}_C^x(\Psi)$ if $\Phi \in \{\mu Z.\Psi, \nu Z.\Psi\}$
5. **false** if $\Phi \in \{\exists y.A(y) \wedge \Psi(y), \forall y.A(y) \rightarrow \Psi(y)\}$
6. $\text{TRG}_{\mathcal{B}}(R|_2) \wedge \text{CMP}_{C'}^y(\Psi) \wedge ((C \sqsubseteq_{\mathcal{M}} \exists R) \vee (\exists R \sqsubseteq_{\mathcal{M}} C))$
if $\Phi \in \{\exists y.R(x, y) \wedge \Psi(y), \forall y.R(x, y) \rightarrow \Psi(y)\}$
and $C' =_{\mathcal{M}} \exists R^-$
7. $\text{TRG}_{\mathcal{B}}(R|_1) \wedge \text{CMP}_{C'}^y(\Psi) \wedge ((C \sqsubseteq_{\mathcal{M}} \exists R^-) \vee (\exists R^- \sqsubseteq_{\mathcal{M}} C))$
if $\Phi \in \{\exists y.R(y, x) \wedge \Psi(y), \forall y.R(y, x) \rightarrow \Psi(y)\}$
and $C' =_{\mathcal{M}} \exists R$
8. $(C \sqsubseteq_{\mathcal{M}} A \vee A \sqsubseteq_{\mathcal{M}} C) \wedge \text{CMP}_C^x(\Psi)$
if $\Phi \in \{A(x) \wedge \langle \neg \rangle \Psi, A(x) \wedge [\neg] \Psi, A(x) \rightarrow \langle \neg \rangle \Psi, A(x) \rightarrow [\neg] \Psi\}$

Intuitively, the formulae above state that: (1) C and x are always compatible with non-first-order subformulae. (2) C and x are compatible with first-order components of the form $A(x)$ or $\neg A(x)$ if classes A and C belong to the same hierarchy according to \mathcal{M} ; this means that navigation through classes is only allowed in the context of the same hierarchy. (3) boolean connectives distribute the compatibility check to all their inner sub-formulae. (4) fixpoint constructs push the compatibility check to their inner sub-formulae. (5) compatibility is broken if new quantified variables over classes are introduced in the formula.

This means that at most one quantification over classes is allowed in a pseudo-navigational property to be navigationally compatible with \mathcal{B} . (6) and (7) deal with navigation along a binary relation, from the first to the second component in (6), and from the second to the first component in (7). In particular, (6) states that the formula can quantify over the second component of a relation R where x points to the first component if:

1. the second component of R is a target role in \mathcal{B} , witnessing that Φ agrees with the unidirectional navigation imposed by \mathcal{B} over R ;
2. class C belongs to the same hierarchy of the domain class for R , according to \mathcal{M} ;
3. C' and y are navigationally compatible with the inner formula Ψ , where y is the newly quantified variable, and C' is the image class for R according to \mathcal{M} .

(7) works in a similar way, by simply inverting the second and first components of R . (8) next-state transition formulae are compatible if the class used in the guard belongs to the same hierarchy of C , and C and x are compatible with the inner subformula.

Notice that termination properties are always guaranteed to be navigationally compatible with the corresponding BAUML model, since \mathbf{A} and TERM_A belong by definition to the same hierarchy.

Unfortunately, the following result shows that restricting BAUML models to be unidirectional is not sufficient to obtain decidability of checking termination properties.

Theorem 7.2.2. *Checking termination of unidirectional BAUML models is undecidable.*

Proof. Given a 2-counter machine C , we produce a corresponding unidirectional BAUML model $\mathcal{B}_C = \langle \mathcal{M}^*, \emptyset, \{S_{2\text{CM}}^*\}, \{P_{\text{init}}^*, P_{\text{run}}^*\}, \{\text{init}, \text{inc}_1, \text{dec}_1, \text{inc}_2, \text{dec}_2, \text{halt}\} \rangle$, whose components are illustrated in Table C.2. \mathcal{M}^* contains a single artifact 2CM , which can be ready or halted, the latter being the termination state ($\text{TERM}_{2\text{CM}} = \text{Halted}2\text{CM}$), as attested by $S_{2\text{CM}}^*$. When the `init` operation is applied, a new instance m of `Ready2CM` is created, attaching to it two dedicated objects of type `Counter`, using respectively role $c1$ and $c2$ of the associations `hasC1` and `hasC2`. Such `Counter` objects mirror the two counters of C . In particular, each of the two `Counter` objects attached to m has a 1-to-many association with `Item`: at a

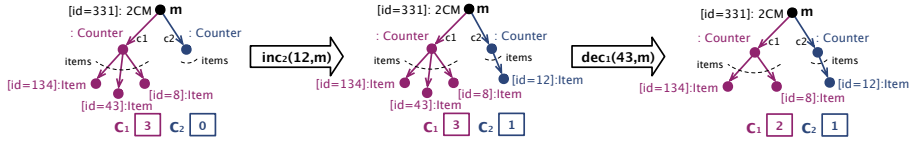


Figure C.1: Sample counter manipulation using the BAUML model in Table C.2

given time, the number of items attached to $m.c1$ ($m.c2$ resp.) represents the value of the first (second resp.) counter in C .

The artifact instance m then executes the process corresponding to the run event, which suitably encodes the program of C :

1. incrementing the first counter translates into the inclusion of a new `Item` to the items of $m.c1$, i.e., to the set $m.c1.items$;
2. testing whether the first counter is 0 translates into checking whether set $m.c1.items$ is empty;
3. decrementing the first counter translates into the removal of one item from set $m.c1.items$ (it is not important which).

Table C.2 shows how these three aspects can be formalized in terms of activity diagrams and OCL queries. The management of the second counter is analogous, with the only difference that it involves $m.c2.items$ in place of $m.c1.items$. Figure C.1 intuitively shows the evolution of a specific configuration of the system in response to the application of two operations.

Observe that, as graphically depicted in \mathcal{M}^* (consistently with the operations), \mathcal{B}_C is unidirectional: all OCL expressions (except from that in `init`) are navigational in m , and navigation unidirectionally flows from `2CM` to `Counter` to `Item`. Furthermore, no two objects of type `Counter`, nor two objects of type `Item`, are shared by different instances of `2CM`. This means that every instance of `Ready2CM` runs the process corresponding to the program of C in total isolation with other instances of `Ready2CM` and, consequently, either all halt or none halt. The claim follows by observing that C halts if and only if all instances of `Ready2CM` eventually reaches the `Halted2CM` state, i.e., properly terminate.

□

Cardinality-Bounded Models

Theorem 7.2.3. *Let \mathcal{B} be an arbitrary unidirectional, cardinality-bounded BAUML model. Verifying whether \mathcal{B} satisfies a $\mu\mathcal{L}_p$ property navigationally compatible with \mathcal{B} is decidable, and reducible to finite-state model checking.*

Proof. Let $\mathcal{B} = \langle \mathcal{M}, \mathcal{O}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$ be a cardinality-bounded, unidirectional BAUML model, and let Φ be a $\mu\mathcal{L}_p$ property navigationally compatible with \mathcal{B} . On the one hand, by inspecting the notion of navigational compatibility, one can notice that Φ is “rooted” in a single artifact class S , subject to the outermost subformula of the form $\exists x.S(x) \wedge \Psi(x)$ (or $\forall x.S(x) \rightarrow \Psi(x)$). Navigational compatibility then ensures that Φ only mentions relations and classes that can be reached by navigating \mathcal{M} using is-a relationships (in both directions), or associations, in a direction that is compatible with the unidirectionality imposed by \mathcal{B} .

On the other hand, as pointed out in Section 7.2.2, in a navigational model like \mathcal{B} it is impossible for artifact instances to share objects that belong to read-write classes. This means that the evolution of an artifact instance is completely independent from that of the other artifact instances of the same type ART_S , or other artifact types.

By combining this two observations, we obtain that Φ obeys to a sort of *isolation property*:

- Φ does not distinguish whether the system contains evolving artifact instances of types different than ART_S ;
- Φ does not distinguish whether the instances of ART_S evolve in isolation, or co-evolve in a concurrent way.

This isolation property is a data-aware variant of the free-choice property of Petri nets. Thanks to such property, instead of directly considering the whole concurrent evolution of the system, in which unboundedly many artifact instances could be created over time and evolved in parallel, one can consider a faithful, sound and complete abstraction of the system, which accounts only for the concurrent evolution of those instances of type ART_S present in the initial database of \mathcal{B} , plus an additional artifact instance of type ART_S , nondeterministically created and evolved in addition to the others.

Let b_i be the number of artifact instances of type ART_S present in the initial database of the system. From the fact that \mathcal{B} is unidirectional and cardinality-bounded, we have that each artifact instance can create only a bounded amount

of objects during its evolution. In fact, the number of objects that can be created by an artifact instance is bounded by $(k \cdot N)^{l+1}$, where:

1. k is the number of relations in the schema (which bounds the number of relations that are collectively attached to an artifact/class in the schema),
2. N is the maximum cardinality upper bound attached to a target role belonging to a path rooted in ART_S , and
3. l is the length of the longest navigational path rooted in ART_S .

As a consequence, by considering the aforementioned sound and complete abstraction, we have that at most $(b_i + 1) \cdot N^{l+1}$ objects and artifact instances are simultaneously present in a system snapshot. The claim then follows by:

1. applying the translation from BAUML models to data-centric dynamic systems (DCDSs) [11], explained in Chapter 5;
2. observing that the bound $(b_i + 1) \cdot N^{l+1}$ implies that the obtained DCDSs is state-bounded;
3. recalling that verification of $\mu\mathcal{L}_p$ properties over state-bounded DCDSs is decidable, and reducible to finite-state model checking [11].

□

Theorem 7.2.4. *Checking termination of 1-cardinality-bounded, bidirectional BAUML models is undecidable.*

Proof. Given a 2-counter machine C , we produce a corresponding 1-cardinality-bounded, bidirectional BAUML model $\mathcal{B}_C = \langle \mathcal{M}^b, \emptyset, \{S_{2\text{CM}}^b\}, \{P_{\text{init}}^b, P_{\text{run}}^b\} \rangle$, whose components are illustrated in Table C.2. \mathcal{M}^b contains a single artifact 2CM, which can be ready or halted, the latter being the termination state ($\text{TERM}_{2\text{CM}} = \text{Halted2CM}$), as attested by $S_{2\text{CM}}^b$. When the init operation is applied, a new instance m of Ready2CM is created, attaching a dedicated item that represents the *zero* point for both counters.

Intuitively, m mirrors the two counters in C as follows. Thanks to the fact that m can navigate and manipulate the association *hasNext* in both directions (i.e., from left to right and from right to left), the length of the right chain from the zero element $m.\text{zero}$ corresponds to the value of the first counter, whereas the length of the left chain from the zero element corresponds to the value of the second counter.

The artifact instance m suitably encodes the commands in C as follows:

- Incrementing the first counter requires to create a new `Item`, and to put this object between the zero element and the old right-successor of it (cf. `inc1`, which conveniently exploits notation “@pre” to query the configuration of objects in the last predecessor state). This has the effect of increasing the length of the right chain of one unit. The alternative operation `incZ1` handles the special case in which there is no right-successor from the zero element: in this case incrementing the counter just corresponds to add a new item on the right of the zero element.
- Testing whether the first counter is 0 translates into checking whether set `m.zero.r` is empty, i.e., whether it is true that the zero element does not have any right successor.
- Decrementing the first counter translates into the removal of one item from set right chain of the zero element. There are two possible cases. In the first case, there is just a single right-successor, i.e., the counter has value 1. In this case, operation `decS1` just ensures that `m.zero.r` does not have anymore this successor. If instead the right chain is longer than 1, then the decrement is handled by making the second right-successor of `m.zero` the new direct right-successor of it, at the same time isolating the old direct right-successor.

Table C.3 shows how these three aspects can be formalized in terms of activity diagrams and OCL queries. The management of the second counter is analogous, with the only difference that it navigates the left chain of the zero element, i.e., it exploits the *l* role of relation `hasNext` in place of the *r* role. Figure C.1 intuitively shows the evolution of a specific configuration of the system in response to the application of two operations.

Observe that, as clearly shown by \mathcal{M}^b , \mathcal{B}_C is 1-cardinality-bounded, and is bidirectional, because relation `hasNext` is navigated on both directions, making both *l* and *r* target roles. Furthermore, like for the reduction in Theorem 7.2.2, each artifact instance is created in state `Ready2CM`, and evolves completely independently from the other artifact instances. This means that either all instances of `Ready2CM` halt, or none halt. The claim follows by observing that `C` halts if and only if all instances of `Ready2CM` eventually reach the `Halted2CM` state, i.e., properly terminate.

□

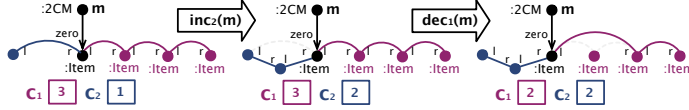


Figure C.2: Sample counter manipulation using the BAUML model in Table C.3

C.2.3 Models with Shared Instances

Theorem 7.2.5. *Checking termination of 1-cardinality-bounded, unidirectional BAUML models with shared instances is undecidable.*

Proof. Given a 2-counter machine C , we produce a corresponding 1-cardinality-bounded unidirectional BAUML model with shared instances $\mathcal{B}_C = \langle \mathcal{M}^{bu}, \emptyset, \{S_{2CM}^{bu}\}, \{P_{init}^{bu}, P_{run}^{bu}\} \rangle$, whose components are illustrated in Tables C.4 and C.5.

As shown in Tables C.4 and C.5, \mathcal{M}^{bu} contains a single artifact Conn , which can be ready or halted, the latter being the termination state ($\text{TERM}_{\text{Conn}} = \text{HaltedConn}$), as attested by S_{Conn}^{bu} . Due to cardinality boundedness and unidirectionality, a single instance of \mathcal{B}_C is not powerful enough to simulate C . Hence, differently from the previous undecidability proofs, the two counters are now simulated by unbounded chains of artifact instances. In this light, the main difficulty is to properly “synchronize” such different instances so as to ensure that they collectively implement the program of C , without interfering with each other. To realize such a synchronization, all instances of Conn share an instance of PC , which represents a “program counter” to keep track of the current instruction to be processed in C . Intuitively, each instance of Conn represents a connection between two items; a chain of three items is then built by using two instances of Conn , making sure that the first instance has on the right the same Item that the second instance has on the left. This structure constitutes the basis for simulating a counter.

Let us now go into the details of such a simulation. The initialization transition in S_{Conn}^{bu} consists now of a complex activity diagram P_{init}^{bu} , which consists of the following steps:

- Initially, if there is no instance of the program counter, one instance is created, setting its “position” (represented by a string attribute pos) to the constant string 1. If an instance of PC already exists, then this step is skipped.

- The second step consists of the creation of a new connection artifact instance (of type `Conn`), with a distinguished identifier. Upon creation, the *pc* role of this connection points to the only available instance of `PC`.
- The third step is applied only if no instance of class `Item` exists in the system. In this case, two special items are created so as to represent the zero elements for the two counters of *C*. This is done as follows:
 - The zero element for the first counter consists of a newly created instance i_0^R of `Item`, whose boolean attribute *startC1* is set to `true`. Item i_0^R is attached to the right of the just created instance of `Conn`. Since i_0^R is not on the left of any connection, also its boolean attribute *lastR* is set to `true`.
 - The zero element for the second counter consists of a newly created instance i_0^L of `Item`, whose boolean attribute *startC2* is set to `true`. Item i_0^L is attached to the left of the just created instance of `Conn`. Since i_0^L is not on the right of any connection, also its boolean attribute *lastL* is set to `true`.

The structure obtained when 4 instances of P_{init}^{bu} are executed in a row can be seen on the left of Figure C.3.

The idea behind the manipulation of counters starting from this structure is to extend (resp., reduce) the chain on the right of item i_0^R to increment (resp., decrement) the first counter, and to extend (resp., reduce) the chain on the left of item i_0^L to increment (resp., decrement) the second counter. Since the case of the second counter is obtained by just mirroring that of the first counter, we just concentrate on the first counter.

The first important observation, which is common to the case of counter increment and decrement, concerns the problem of synchronization. On the one hand, as already pointed out we want all instances of `Conn` to collectively realize the program of *C*. On the other hand, there is no control on when new instances of `Conn` are created. In particular, it could be the case that a new connection is created when the other active connections have already executed part of the program of *C*. Similarly, since there is no control on how the different active instances of `Conn` interleave with each other, when a connection executes the portion of P_{run}^{bu} corresponding to instruction number *k* in *C*, it must ensure that *k* is indeed the current instruction. More specifically, instruction number *k* always contains an initial choice, used to check whether the program counter is indeed *k* and, if so, whether the instance of `Conn` that is executing the process

is responsible for the execution of instruction k , or should instead just execute an “idle” loop and wait that the responsible connection executes step k . If the program counter stores in its *pos* attribute an instruction identifier different than k , then the process just “jumps” to the right step. If instead the program counter corresponds to k , then a different behavior is exhibited depending on whether the instruction number k corresponds to an increment or conditional decrement for the first counter.

In the case of increment:

- If the connection is not associated to any item on its left and its right (i.e., it is not part of any chain), then the connection becomes responsible for the increment, which is atomically executed using the operation $kInc_1$. The increment is realized as follows:
 - The unique item (called i) that has attribute *lastR* set to true is selected.
 - This item is attached on the left of the current connection, setting its *lastR* attribute to false. In this way, it is easy to see that an item has *lastR* = false if and only if there is no connection that has it on the left.
 - A new item is created and attached on the right of the current connection, setting its *lastR* attribute to true. This newly created item represents the increment of the first counter, and the current connection acts as the last connection of the chain simulating the first counter.
 - The program counter is updated, setting its *pos* attribute to the string that corresponds to the new instruction identifier k' . Since k' is a pre-defined string, each increment is different from the others, and this is why each specific increment is mapped to a separate operation in P_{run}^{bu} .

Considering e.g., the case of instruction $1 : INC(1,7)$, the central part of Figure C.3 represents the new data configuration after the execution of this step by one of the connections that are currently active but not associated to any item.

- If instead the connection is already attached to an item on the left or on the right, then it executes an idle step, going back to check whether the program counter is still k or has instead been updated.

In the case of conditional decrement:

- If the connection has on its right an item whose attribute *lastR* is true, then the connection becomes responsible for the conditional decrement. Two cases may then arise: either the first counter is 0, and consequently only the program counter must be updated, or the counter is positive, and consequently the counter must be decremented before updating the program counter. The test for zero can be easily captured in \mathcal{B}^{bu} by testing whether the item having *lastR* = true also has *startC1* = true: if so, then the first counter is zero, if not, then the first counter is positive. In the former case, captured by query Q_0^1 , the specific task **kPC** is executed, whose effect is simply to update the attribute *pos* of the program counter to the string corresponding to k'' ; since k'' is a pre-defined string, each program counter update is different from the others, and this is why each specific program counter update is mapped to a separate operation in P_{run}^{bu} . In the latter case, captured by query Q_1^1 , an atomic decrement and program counter update is executed using the operation **kDec₁**. The decrement is realized as follows:
 - The item that was previously on the right of the connection is updated making its *lastR* attribute equal to false.
 - The item that was previously on the left of the connection (i.e., on the right of the previous connection along the chain) is updated making its *lastR* attribute equal to true.
 - The connection is disconnected from both such items, hence reducing the chain of one item. This has also the indirect effect of making the connection eligible for being responsible of a successive increment.
 - The program counter is updated, setting its *pos* attribute to the string that corresponds to the new instruction identifier k' . Since k' is a pre-defined string, each decrement is different from the others, and this is why each specific decrement is mapped to a separate operation in P_{run}^{bu} .

Considering the case of instruction 7 : CDEC(1,2,9), the right part of Figure C.3 represents the new data configuration after the execution of this step by the connection that is currently at the end of the right chain.

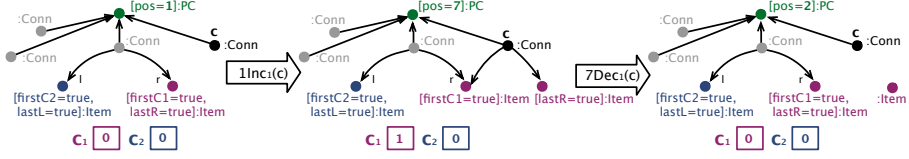


Figure C.3: Sample counter manipulation using the BAUML model in Tables C.4 and C.5

- If instead the connection does not have on its right the element whose $lastR$ attribute is true, then it executes an idle step, going back to check whether the program counter is still k or has instead been updated.

As soon as one of the active connection artifact instances sets the program counter to the constant n , all active connections move to the final part of P_{run}^{bu} , where they are moved from the ReadyConn to the HaltedConn state. If new instances of Conn are subsequently created, they immediately jump to execute this task as well (in fact, they all share the same program counter, whose pos attribute continues to be n). This means that either all instances of ReadyConn halt, or none halts. The claim follows by observing that C halts if and only if all instances of ReadyConn eventually reach the HaltedConn state, i.e., properly terminate. \square

Theorem 7.2.6. *Verification of $\mu\mathcal{L}_p$ properties over cardinality-bounded, unidirectional BAUML models with shared instances of read-write classes is decidable and reducible to finite-state model checking when the number of simultaneously active artifact instances is bounded.*

Proof. Let \mathcal{B} be a cardinality-bounded, unidirectional BAUML model. By combining unidirectionality and cardinality-boundedness, we have that an artifact instance can create only a bounded amount of objects during its evolution. In fact, the number of objects that can be created is bounded by $(k \cdot N)^{l+1}$, where k , N and l are as in the proof of Theorem 7.2.3. Since the number of simultaneously active artifact instances is bounded, say, by a number b , then at each time point the number of objects and artifact instances present in the overall system is bounded by $b \cdot (k \cdot N)^{l+1}$. The claim then follows by:

1. applying the translation from BAUML models to DCDSs, described in [50];

2. observing that the bound $b \cdot (k \cdot N)^{l+1}$ implies that the obtained DCDS is state-bounded;
3. recalling that verification of $\mu\mathcal{L}_p$ properties over state-bounded DCDSs is decidable, and reducible to finite-state model checking [11].

□

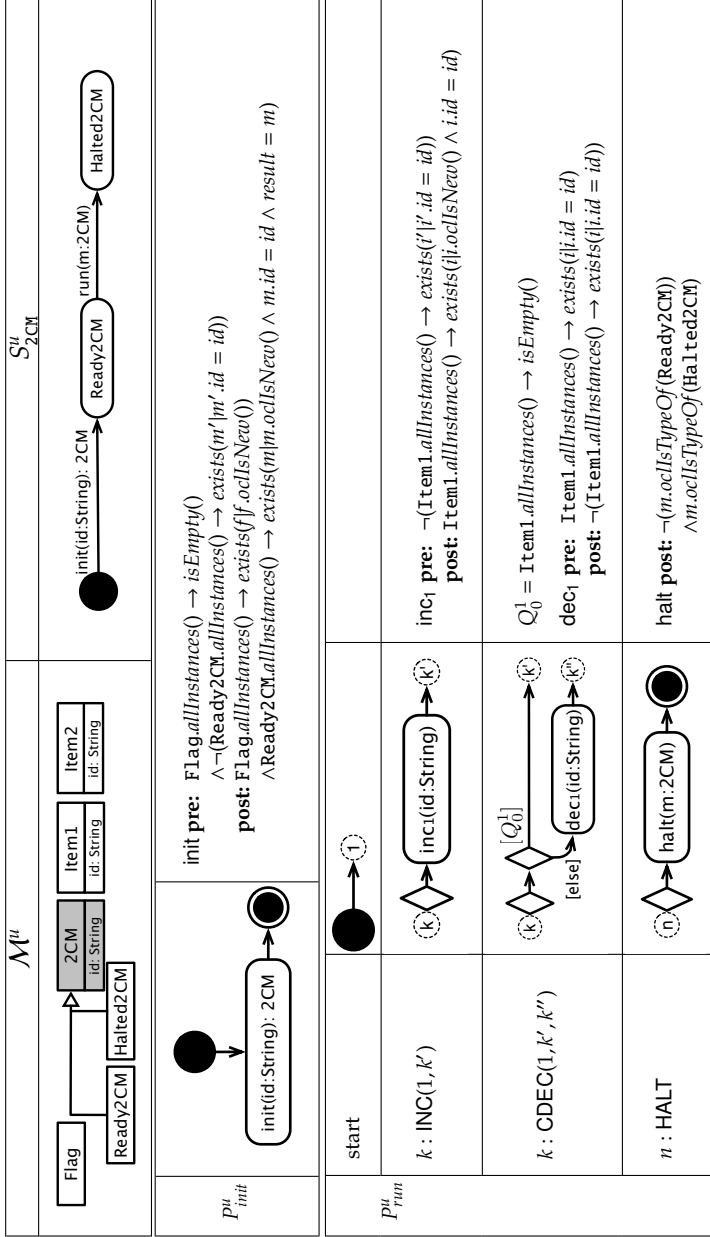


Table C.1: Unrestricted BAUML model simulating a 2-counter machine

\mathcal{M}^*		S_{2CM}^*
P_{init}^*		<p>pre: $\neg(\text{Ready2CM.allInstances}() \rightarrow \text{exists}(m' m'.id = id))$ post: $\text{Ready2CM.allInstances}() \rightarrow \text{exists}(m m.ocIsNew() \wedge m.id = id \wedge \text{result} = m \wedge (m.c1 \rightarrow \text{exists}(c1 c1.ocIsNew())) \wedge (m.c2 \rightarrow \text{exists}(c2 c2.ocIsNew()))$</p>
P_{run}^*	<p>start</p> <p>$k : \text{INC}(1, k')$</p> <p>$k : \text{CDEC}(1, k', k'')$</p> <p>halt</p> <p>$n : \text{HALT}$</p>	<p>inc₁ pre: $\neg(m.c1.items \rightarrow \text{exists}(i' i'.id = id))$ post: $m.c1.items \rightarrow \text{exists}(i i.ocIsNew() \wedge i.id = id)$</p> <p>$Q_0^1 = m.c1.items \rightarrow \text{isEmpty}()$ dec₁ pre: $m.c1.items \rightarrow \text{exists}(i i.id = id)$ post: $\neg(m.c1.items \rightarrow \text{exists}(i i.id = id))$</p> <p>halt post: $\neg(m.ocIsTypeOf(\text{Ready2CM}) \wedge m.ocIsTypeOf(\text{Halted2CM}))$</p>

Table C.2: Unidirectional BAUML model simulating a 2-counter machine

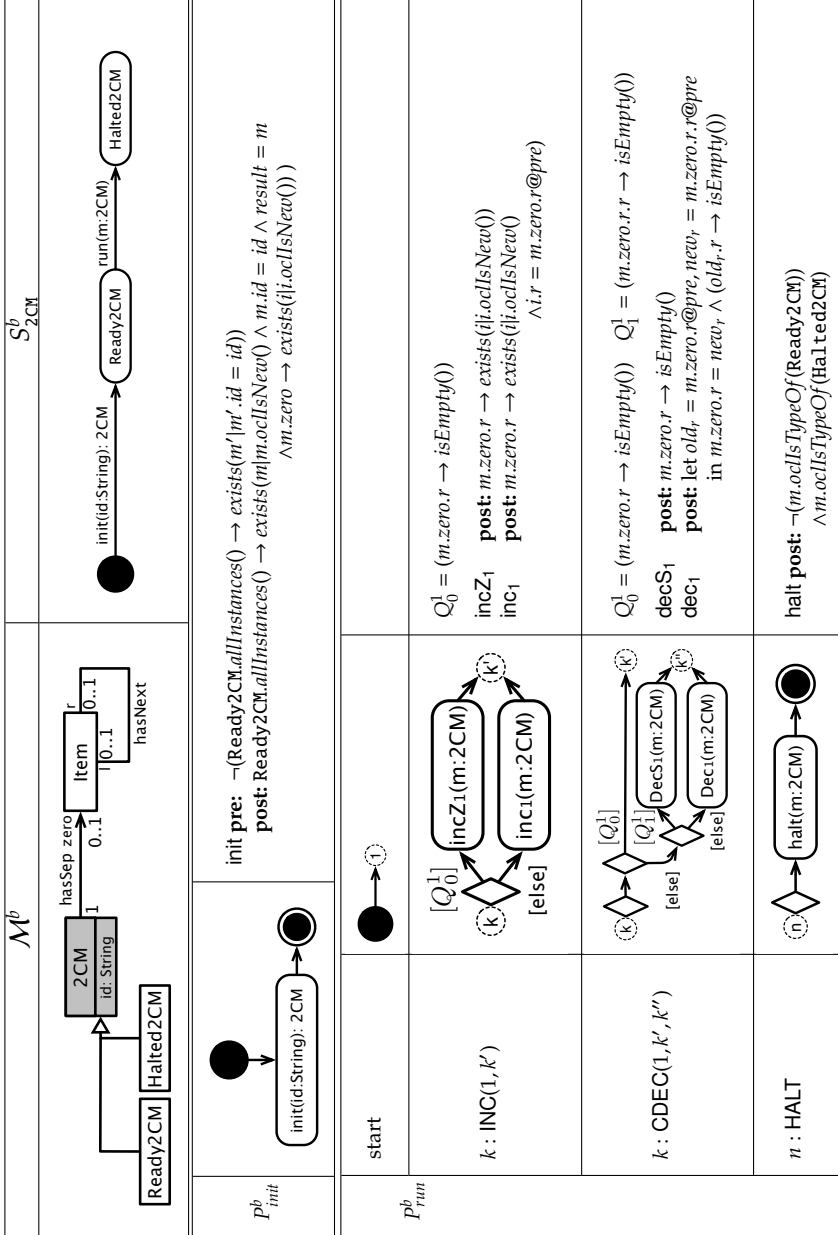


Table C.3: 1-cardinality-bounded, bidirectional BAUML model simulating a 2-counter machine

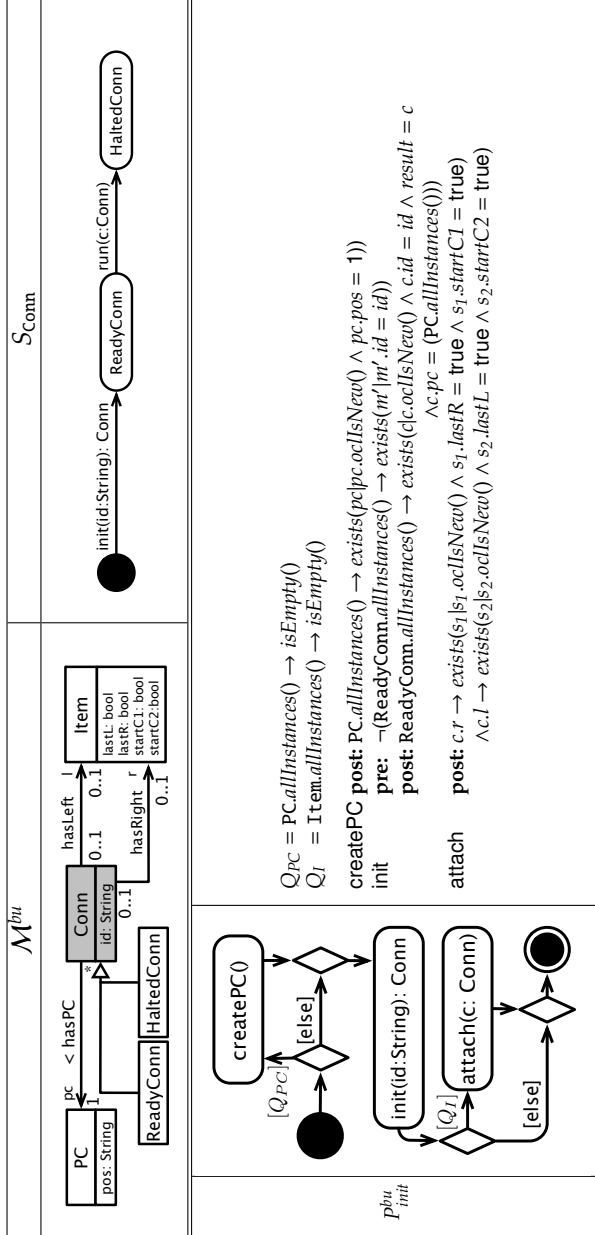


Table C.4: 1-cardinality-bounded, unidirectional BAUML model with shared objects simulating a 2-counter machine (Part 1)


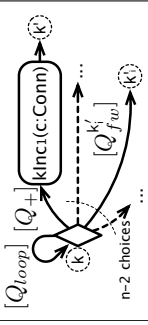
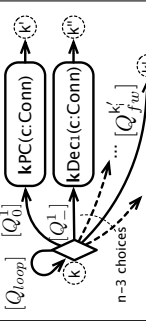

P_{run}^{bu}	start		
$k : \text{INC}(1, k')$		$Q_+ = (c.pc = k) \wedge (c.l \rightarrow \text{isEmpty}()) \wedge (c.r \rightarrow \text{isEmpty}())$ $Q_{loop} = (c.pc = k) \wedge \neg(c.l \rightarrow \text{isEmpty}()) \wedge (c.r \rightarrow \text{isEmpty}())$ $Q_{k'}^{fw} = (c.pc = k') \text{ for } k' \in \{1, \dots, n\} \setminus \{k, k'\}$ $\text{klnc}_1 \text{ post: let } i = (\text{Item.allInstances}() \rightarrow \text{select}(i'.lastR))$ $\text{ in } c.l = i \wedge i.lastR = \text{false}$ $\wedge c.r \rightarrow \text{exists}(i'' i''.\text{isOcNew}() \wedge i''.lastR = \text{true})$ $\wedge c.pc = k'$	
$k : \text{CDEC}(1, k', k'')$		$Q_0^+ = (c.pc = k) \wedge c.r.lastR \wedge c.r.startC1$ $Q_0^- = (c.pc = k) \wedge c.r.lastR \wedge \neg c.r.startC1$ $Q_{loop} = (c.pc = k) \wedge \neg c.r.lastR$ $Q_{k'}^{fw} = (c.pc = k') \text{ for } k' \in \{1, \dots, n\} \setminus \{k, k', k''\}$ $\text{kPC post: } c.pc = k''$ $\text{kDec}_1 \text{ post: let } i_r = c.r.@pre, i_l = c.l.@pre$ $\text{ in } i_r.lastR = \text{true} \wedge i_r.lastR = \text{false}$ $\wedge (c.l \rightarrow \text{isEmpty}()) \wedge (c.r. \rightarrow \text{isEmpty}())$	
$n : \text{HALT}$		$\text{halt post: } \neg(m.\text{oc}/\text{IsTypeOf}(\text{ReadyConn}))$ $\wedge m.\text{oc}/\text{IsTypeOf}(\text{HaltedConn})$	

Table C.5: 1-cardinality-bounded, unidirectional BAUML model with shared objects simulating a 2-counter machine (Part2)