Reasoning on UML Data-Centric Business Process Models

Montserrat Estañol[1], Maria-Ribera Sancho[1,2], and Ernest Teniente[1]

[1] Universitat Politècnica de Catalunya, Barcelona, Spain
[2] Barcelona Supercomputing Center, Barcelona, Spain
{estanyol,ribera,teniente}@essi.upc.edu

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.

1 Introduction

Modeling business processes correctly from their early stages is key to the success of an organization, in order to avoid the propagation of errors to the final service that is offered to the customer. At the same time, these models should be easy to understand for the people involved in the process. One way of modeling business processes is by means of the data-centric approach, in which data plays a key role. In a nutshell, business artifacts model key business-relevant entities which are updated by a set of services that implement the business process tasks.

Automated reasoning on data-centric BPM has attracted a lot of research in recent years and several promising techniques have been proposed [2, 5, 8]. However, all these proposals specify the business process model in some variant of logic, resulting in a specification that is low-level and complex, and therefore difficult to understand and unpalatable for business people.

Our work in this paper is aimed at providing semantic reasoning on data-centric business process models specified from a technology-independent and high-level perspective using UML/OCL [7]. We then translate the UML specification into a data-centric dynamic system (DCDS) [2] which can be reasoned with in order to determine the correctness of the original specification.

2 Basic Concepts

This section presents briefly presents the UML models that we use for the initial specification and the target language that provides the reasoning capabilities.

S. Basu et al. (Eds.): ICSOC 2013, LNCS 8274, pp. 437–445, 2013.

2.1 UML Data-Centric Business Process Models

We assume that each dimension of the BALSA framework used to model a data-centric BPM is represented by means of UML and OCL, as proposed in [7].

Business Artifacts. *Business artifacts* are intended to hold all the information needed to complete business process execution. We represent business artifacts by means of UML class diagrams. Figure 1 shows the UML class diagram of our running example which summarizes a generic process of lending a book from a library. *CopyRequest* is the key business artifact in this process since it records the information regarding copies of books requested by the users. A *CopyRequest* may be either a *Reservation* or a *Loan*. The rest of the classes in the diagram are those business artifacts required to hold all the information necessary to request a copy of a book: *User*, *Book* and *Copy* (of a book).

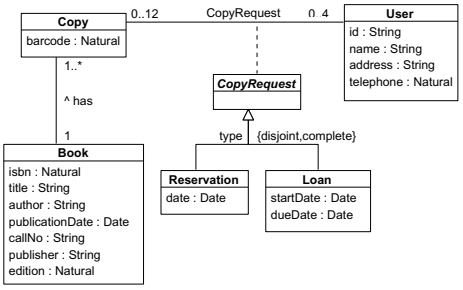

Fig. 1. Class diagram for a library

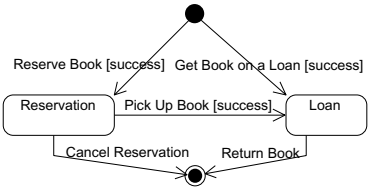

Fig. 2. State machine diagram of *CopyRequest*

The additional integrity constraints for Figure 1 are shown below.

1. Key constraints: *Book -> ISBN*, *User -> id*, *Copy -> barcode*.
2. A *Copy* may not be in more than three *CopyRequests* of subtype *Reservation*.
3. A *Copy* may not be in more than one *CopyRequests* of subtype *Loan*.

Lifecycle. The *lifecycle* of a business artifact states the key, business-relevant, stages in the possible evolution of the artifact. We represent lifecycles through UML state machine diagrams. Figure 2 shows the lifecycle of the main artifact in our example, *CopyRequest*. A *CopyRequest* is created as a *Reservation* or as a *Loan*, depending on whether the user has asked to reserve a copy of a book or to get it on a loan. When a *Reservation* is picked up by the user, then the *CopyRequest* becomes a *Loan*. Finally, a *CopyRequest* is deleted when the reservation is canceled or the book on loan is returned.

Associations. Transitions in the UML state machine diagram are decomposed into several tasks that must be performed in order for the transition to take place. *Associations* are used to establish the conditions under which these tasks can be executed and we represent them in activity diagrams. Therefore, we will have one such diagram for each transition. Figure 3 shows the activity diagram

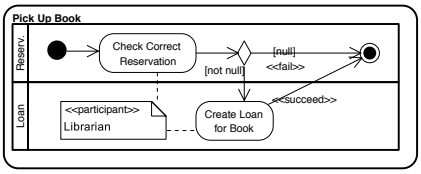

Fig. 3. Activity diagram of *Pick Up Book*

```
action CreateLoanForBook (r: Reservation)
localPost: r.oclIsTypeOf(Loan) and
     r.startDate=today() and
     r.dueDate=getDueDate() and not
     r.oclIsTypeOf(Reservation)
```

Listing 1. Code for service *CreateLoanForBook*

corresponding to *PickUpBook*. First of all, it must be checked whether the *Reservation* provided by the user is correct and that the copy is available. If it is, the *Loan* for the book is created and the transition ends successfully. Otherwise, the transition cannot take place.

Services. A *service* (or *task*) encapsulates an atomic unit of work meaningful to the business process. The execution of services makes business artifacts evolve. We use OCL operation contracts (defined by means of a precondition and a postcondition) to state their effect. Listing 1 shows the OCL operation contract for *CreateLoanForBook*, stating how the *Reservation* becomes a *Loan*.

2.2 Data-Centric Dynamic Systems

A relational data-centric dynamic system (DCDS) 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 $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$, such that $\mathcal{C}$ is a set of *values*, $\mathcal{R}$ is a *database schema* containing a finite set of tables, $\mathcal{E}$ is a finite set of *equality constraints* and $\mathcal{I}_0$ represents the *initial instance* of the database schema.

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 $\alpha(p_1, ..., p_n) : \{e_1, ..., e_m\}$, where:
  - α is the *action's name* and $p_1, ..., p_n$ represent *input parameters*.
  - $\{e_1, ...e_m\}$ is a set of *effects*. They take place simultaneously.
- 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. α is an *action* in $\mathcal{A}$.

3 Translating a UML Data-Centric BPM to a DCDS

This section presents the translation process from an initial model defined in UML/OCL into a Data-centric Dynamic System (DCDS). As it may be seen in Table 1, each of the UML models (columns) is translated into one or more elements in the DCDS (rows). Notice that the database schema is used to hold static information, not only from the class diagram, but also from the state machine and the activity diagram. Condition-action rules are used to represent the dynamic evolution in both the state machine and the activity diagram, whereas actions are the ones in charge of making the actual changes in the system.

Table 1. Overview of the elements involved in the translation

	Class Diagr.	State Mach. Diagr.	Act. Diagr.	Op. Contracts
DB Schema	DB_o	DB_i	DB_f	
CA Rules		CA_o	CA_i	CA_f
Actions			A_o	A_f

3.1 Business Artifacts in a Class Diagram

As the business artifacts hold static information and are represented in a class diagram, we will translate the diagram into a database schema, and the remaining integrity constraints will be translated into equality constraints. This translation is performed according to well-known techniques of database design [12].

Figure 4 shows the database schema corresponding to the class diagram in Figure 1. There is one table for each class; class identifiers correspond to the primary key[1] of the corresponding tables. Associations are represented by adding attribute(s) to the linked tables or by creating an additional table for the association itself. Finally, the class hierarchy between *CopyRequest* with *Reservation* and *Loan* in the original example has been implemented through foreign keys from the subclasses to the superclass. The remaining constraints (*disjoint* and *complete*, shown in the class diagram, and textual constraints numbers 2 and 3), should be translated into equality constraints. Due to space limitations, we cannot show their translation here.

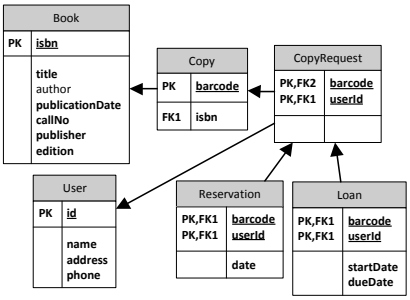

Fig. 4. Translation of the class diagram into a database schema

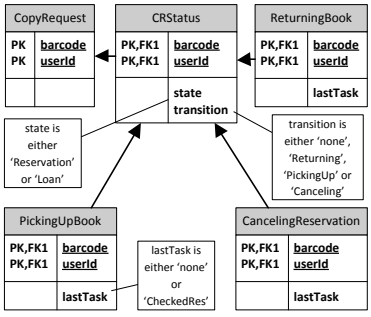

Fig. 5. Keeping track of the status of *CopyRequest* and *Pick Up Book*

3.2 Lifecycles in a State Machine Diagram

The translation of the state machine diagram into a DCDS requires the following: adding a table to the database schema, defining CA rules, and specifying the details of the actions in the previously defined CA rules. The table that is added

[1] Primary and foreign keys should be defined as equality constraints in the DCDS. However, for the sake of understanding, we show them graphically in the figure.

to the database schema is used to keep track of the status of the artifact *BA* associated to the state machine diagram. It will contain the primary keys of *BA* and two attributes: *state* and *transition*. Attribute *state* represents the current state of the artifact, and *transition* indicates, if any, the transition whose effects the artifact is under. Attribute *transition* is used to prevent the execution of two transitions simultaneously. See Figure 5 for an example.

For each transition in the state machine diagram we will define a CA rule. These rules will indicate the actions which *may* be carried out when the artifact is in a certain state by referring to the *BAStatus* table. As transitions that create the artifact have no source state, the left-hand side of the corresponding rules will indicate that they can be executed anytime using condition *true*. The action's parameters will be the primary key of the business artifact for which the state machine diagram is defined, except for actions that create the business artifact, which will have none.

For instance, rule 1 states that *GetBookOnALoan* can be executed anytime. Rule 2 states that, to execute the action *PickUpBook*, the artifact must be in state *Reservation* and it cannot be undergoing any transition.

$$true \mapsto GetBookOnALoan() \tag{1}$$

$$CRStatus(bc, id, \text{`}Reservation\text{'}, \text{`}none\text{'}) \mapsto PickUpBook(bc, id) \tag{2}$$

3.3 Associations in an Activity Diagram

In order to translate the associations, for each activity diagram we will need to define an additional DB schema table and several CA rules. The table will be used to keep track of the last task (or service) that has been executed in the activity diagram by means of attribute *lastTask*. In addition, it will reference *BAStatus* and will include *BAStatus*'s primary key. Figure 5 shows the additional tables we require in our example.

Now that we have added these tables, we are able to specify the effect of the DCDS actions required by the condition-action rules that we defined for the state machine diagram. First of all, they need to insert a new element in the table corresponding to the activity diagram of its transition to indicate that the execution of the activity diagram can begin. They should also update the information in *BAStatus* to indicate that a transition is taking place. Lastly, they should copy the rest of the contents of every table, as the semantics of the DCDS establish that content that is not explicitly copied is lost [2]. Action *PickUpBook(bc,id)* is specified below. Although not shown, it would also copy the contents of all tables except *CRStatus* with barcode *bc* and userId *id*:

$$true \rightsquigarrow PickingUpBook(bc, id, \text{`}none\text{'})$$

$$CRStatus(bc, id, \text{`}Res.\text{'}, \text{`}none\text{'}) \rightsquigarrow CRStatus(bc, id, \text{`}Res.\text{'}, \text{`}PickingUp\text{'})$$

Finally, condition-action rules will be used to establish when a service can be executed. The condition of each rule will indicate the service that must have

been executed for the next service to take place (referencing the table created earlier) and will include the precondition of the service that may be executed (a service cannot be executed if the precondition is false).

Moreover, activity diagrams may include decision nodes and guard conditions that restrict the execution of the next service. We will only deal with decision nodes that depend on the result of the previous service and in which one of the conditions causes the end of the execution of the diagram so that they do not require additional conditions on the CA rule.

According to these rules, the translation of transition *PickUpBook* results in the CA rules below. The two actions correspond to the atomic tasks of the activity diagram. The first rule indicates that no task has been performed yet, while the second one will only fire if the reservation has been correctly checked.

$$PickingUpBook(bc, id, \text{`}none\text{'}) \mapsto CheckCorrectReservation(bc, id)$$
$$PickingUpBook(bc, id, \text{`}CheckedRes\text{'}) \mapsto CreateLoanForBook(bc, id)$$

3.4 Services (Tasks) in Operation Contracts

The evolution of business artifacts in our framework is driven by the specification of the atomic services (tasks) that are carried out in an activity diagram, while this is achieved by actions in DCDS. Therefore, it naturally follows that we should translate our tasks into DCDS actions.

A DCDS action α has the form $\alpha(p_1, ..., p_n) : \{e_1, ..., e_m\}$. In our translation, $p_1, ..., p_n$ will correspond to the primary keys of the main business artifact which is being modified. DCDSs use functions to represent input from a user and they can only be a part of e_i. Because of this, we do not include the operation's input parameters as part of the action's parameters p_i. For instance, for service *CreateLoanForBook*, the action's signature will be the following: $CreateLoanForBook(barcode, id)$.

$e_1, ..., e_m$ are the effects of the DCDS action. Each e_i has the form $q_i^+ \wedge Q_i^- \rightsquigarrow E_i$. $q_i^+ \wedge Q_i^-$ represents the information that must be in the database schema in order for the E_i to take place. Those changes that are made regardless of any condition will have *true* as their $q_i^+ \wedge Q_i^-$.

In order to translate the OCL postconditions into logic we will base our work on [10]. The authors identify a set of OCL constructs that indicate when a class, relationship or attribute is created, deleted or modified. Bearing this in mind, the translation of these constructs into a set of effects e_i has the following form:

- **Insertion**:
  $true \rightsquigarrow Table_{INS_i}(...)$
- **Deletion**:
  $Table_i(pk_i', ...) \wedge \neg(pk_i = pk_i') \rightsquigarrow Table_i(pk_i', ...)$
  $Table_{REL_j}(pk_j', ...) \wedge \neg(pk_j = pk_j') \rightsquigarrow Table_{REL_j}(pk_j', ...)$, where $Table_{REL}$ refers to tables which are affected by the deletion of an element.

– **Attribute change**:
$Table_{ATT_i}(pk, ..., x_j) \rightsquigarrow Table_{ATT_i}(pk, ..., y_j)$
$Table_{ATT_i}(pk', ...) \wedge \neg(pk = pk') \rightsquigarrow Table_{ATT_i}(pk', ...)$

If any of the attributes/columns in these tables are given the value of input parameters in the operation contract, then the translated action will include the corresponding call to a function in their translation. We also need to make additional changes to the tables to reflect the evolution through the activity diagram. There are two cases: either the service is the last one in the activity diagram, or not.

If it is the last service in the activity diagram AD, it 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. It will also change table $BAStatus$: $BAStatus(pk, oldState, x) \rightsquigarrow BAStatus(pk, newState, \text{`}none\text{'})$. If it is not the last service in the activity diagram AD, it will change the corresponding row in the table that corresponds to activity diagram AD, indicating that the service has already been executed.

Finally, we need to add rules to copy the contents of all non-modified tables.

In our example, the translation of service *CreateLoanForBook* results in:

$$true \rightsquigarrow Loan(barcode, id, today(), dueDate()) \tag{3}$$

$$Reservation(barcode', id', x_1) \wedge \neg(barcode = barcode' \wedge id = id') \rightsquigarrow Reservation(barcode', id', x_1) \tag{4}$$

$$CRStatus(barcode, id, x_1, x_2) \rightsquigarrow CRStatus(barcode, id, \text{`}Loan\text{'}, \text{`}none\text{'}) \tag{5}$$

$$CRStatus(barcode', id', x_1, x_2 \wedge \neg(barcode = barcode' \wedge id = id') \rightsquigarrow CRStatus(barcode', id', x_1, x_2) \tag{6}$$

$$PickingUpBook(barcode', id', x) \wedge \neg(barcode = barcode' \wedge id = id') \rightsquigarrow PickingUpBook(barcode', id', x) \tag{7}$$

First of all, the translated action deletes the *Reservation* (4) and turns it into a *Loan* (3). Moreover, as *CreateLoanForBook* is the last service in its activity diagram it updates *CRStatus* (eqs. (5) and (6)) and deletes the corresponding row in *PickingUpBook* (7). Due to space limitations, we do not show the rules that copy the content of the rest of tables.

4 Reasoning on a UML Data-Centric BPM

After obtaining the DCDS, it can be used to reason semantically (i.e. check if the model satisfies the business requirements) about the original UML/OCL specification. Comparing the answers provided by the DCDS with those expected will help ensure that the model represents the required behavior. We assume that the model is structurally correct.

Given a DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$, with a data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ (notice that it includes an initial database instance $\mathcal{I}_0$) and a process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, and a

property Φ expressed in $\mu\mathcal{L}_P$ (a variant of μ-calculus), we can check whether $\mathcal{S}$ satisfies Φ by using the technique in [2]. For instance, in our example, we could check if a user is forbidden to have two simultaneous reservations of different copies of the same book. Formally[2]:

$$\nu X.(\forall bc_1, bc_2, id, isbn_1, isbn_2.Reservation(bc_1, id) \wedge Reservation(bc_2, id) \wedge \\ Copy(bc_1, isbn_1) \wedge Copy(bc_2, isbn_2) \wedge (bc1 \neq bc2) \\ \rightarrow (isbn_1 \neq isbn_2)) \wedge [-]X$$

In this case, we would get a negative answer, because the property is not guaranteed in the process's specification. Therefore, the designer should make the necessary corrections (he has probably forgotten to add an integrity constraint).

5 Related Work

When it comes to semantic reasoning on data-centric BPM, many of the existing approaches are close to [2], our target system of representation. The distinctive characteristic of [1] is that it uses a Knowledge and Action Base defined in a variant of Description Logic to represent the artifacts. [3] maps an ontology, representing the artifacts, to a DCDS in order to verify certain properties expressed in a variant of μ-calculus. [5] uses variables to represent artifacts, which are updated by services defined in first-order logic. The properties that the model should fulfill are defined in LTL-FO. [8] goes as far as to define a specification language, ABSL, based on CTL, to specify the artifacts' lifecycle behavior and checks the fulfillment of properties defined in ABSL. In contrast to our work, none of these proposals provide a higher level of representation for the dynamic aspects of the DCDS, and CTL and LTL-FO are not as powerful as μ-calculus.

On the subject of reasoning on UML diagrams, research has either focused on one particular type of diagram or on the consistency between some of them. Examples of approaches that fall in the first category are [10, 11] for the class diagram, [4] for the state-machine diagram or [6] for the activity diagram.

[9] reviews approaches in the second category. About half of the analyzed works deal with semantic consistency across different UML models; however, none of them deal with class, activity, state machine diagrams and operation specifications at the same time.

In summary, existing data-centric BPM proposals that deal with reasoning do not use high-level languages to represent the models. On the other hand, none of the proposals that reason on UML diagrams are able to handle at the same time the different diagrams we need for modeling the four BALSA dimensions.

6 Conclusions

Starting from the UML models we used in [7] to represent business process models from a data-centric perspective, in this paper we have shown a way

[2] In order to simplify the definition of the properties we have abused notation and included only the minimum number of variables representing each table's attributes.

to translate them into a data-centric dynamic system (DCDS) [2] in order to determine their semantic correctness before they are put into practice. UML is a standard, high-level and widely used language, and consequently it is easier to understand than logic. DCDSs, on the other hand, are grounded on logic, and therefore it is possible to perform automatic reasoning on them in order to check the correctness of the model. Moreover, DCDSs are able to represent and deal with the static and dynamic components, including non-deterministic services.

To the best of our knowledge, our proposal is the first to show a way to reason with data-centric process models defined at a high level of abstraction. Therefore, it bridges the gap between specifications defined in a high-level language but which are not possible to verify, and specifications which can be checked but are low-level and difficult to understand.

Acknowledgments. This work has been partially supported by the Ministerio de Ciencia e Innovación under projects TIN2011-24747 and TIN2008-00444, Grupo Consolidado, the FEDER funds and Universitat Politècnica de Catalunya.

References

1. Bagheri Hariri, B., Calvanese, D., Montali, M., De Giacomo, G., De Masellis, R., Felli, P.: Description logic knowledge and action bases. J. Artif. Intell. Res (JAIR) 46, 651–686 (2013)
2. Bagheri Hariri, B., et al.: Verification of relational data-centric dynamic systems with external services. In: PODS, pp. 163–174. ACM (2013)
3. Calvanese, D., De Giacomo, G., Lembo, D., Montali, M., Santoso, A.: Ontology-based governance of data-aware processes. In: Krötzsch, M., Straccia, U. (eds.) RR 2012. LNCS, vol. 7497, pp. 25–41. Springer, Heidelberg (2012)
4. 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)
5. Damaggio, E., Deutsch, A., Vianu, V.: Artifact systems with data dependencies and arithmetic. ACM Transactions on Database Systems 37(3), 1–36 (2012)
6. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)
7. Estañol, M., Queralt, A., Sancho, M.-R., Teniente, E.: Artifact-centric business process models in UML. In: Yao, S.B., Weldon, J.L., Navathe, S., Kunii, T.L. (eds.) Data Base Design Techniques 1978. LNCS, vol. 132, pp. 292–303. Springer, Heidelberg (1982)
8. 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, Heidelberg (2007)
9. 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)
10. Queralt, A., Teniente, E.: Reasoning on UML conceptual schemas with operations. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 47–62. Springer, Heidelberg (2009)
11. Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. ACM Trans. Softw. Eng. Methodol. 21(2), 13 (2012)
12. Teorey, T., Lightstone, S., Nadeau, T.: Database Modeling and Design, 4th edn. Morgan Kaufmann, San Francisco (2006)