(An Example for) Formally Modeling Robot Behavior with UML and OCL

Martin Gogolla[1] and Antonio Vallecillo[2]

[1] University of Bremen, Germany gogolla@informatik.uni-bremen.de
[2] Universidad de Málaga, Spain av@lcc.uma.es

Abstract. One of the problems that the design and development of robotic applications currently have is the lack of unified formal modeling notations and tools that can address the many different aspects of these kinds of applications. This paper presents a small example of a chain of robotized arms that move parts in a production line, modeled using a combination of UML and OCL. We show the possibilities that these high-level notations provide to describe the structure and behaviour of the system, to model some novel aspects such as measurement uncertainty and tolerance of physical elements, and to perform several kinds of analyses.

1 Introduction

Robotic applications are difficult to design, develop and check because of the inherent properties of these kinds of systems and their multi-faceted characteristics. For example, they are composed of heterogeneous parts difficult to model at the same level of abstraction, and to describe with a single notation. Besides, the heterogeneity of the available hardware platforms for robots and the lack of hardware and software standardization severely hampers cross-product development. Having to deal with physical components also implies the need to incorporate some particular properties such as continuous flows, mechanical forces, tolerance and accuracy. Finally, their different nature from software systems hinders in theory their specification with traditional software modeling notations and tools.

This paper presents a small example of a chain of robotized arms that move parts in a production line, modelled using a combination of UML and OCL. We show how these high-level and platform-independent notations permit modeling the system in a formal manner, taking into account novel aspects such as tolerance and measurement uncertainty, and allow performing several interesting analyses on the system, such as visualization of the system in operation, simulation of its behaviour, and validation of several properties of interest to the designer. We claim the importance of having a unified language (e.g., UML and OCL) for describing the functionality of the different parts within a robotic

system in order to understand and analyse the system and its different parts, at least for the central functionalities. In this respect, our approach is different from the various domain-specific languages/proposals for robotic applications that combine specialized languages, since we want to explore the possibilities that UML and OCL offer to model robotic systems.

The rest of the paper is structured as follows. The next Sect. 2 describes the example and shows interesting properties using the USE modeling environment [6,7]. Section 3 discusses related work. The paper ends with a conclusion and future work.

2 A UML and OCL Model for a Production Line with Robotized Arms

This section will first discuss structural system elements before we turn to the behavioral aspects. After that various visual property analysis options and formal test aspects of our approach are debated. All codes and some additional material about the example can be found at `https://goo.gl/DdLivi`.

Structural Elements. To illustrate our approach suppose a system composed of producers and consumers, as shown in Fig. 1. `Producer` machines generate `Items`, which once finished are placed in `Trays`. When informed that there is an element ready in a tray, a `Consumer` takes the generated item from that tray, performs some further work units on it, and stores it in a second tray (the `storageTray`). Assuming we are in a robotized environment, each tray has a `RobotArm` in charge of physically moving the items around. When asked to perform a `put` operation, the tray asks the arm to go where the item is, `grasp` it, `move` it to the position of the tray, and `drop` it there. Similarly, a `get(c:Coordinate)` operation on a tray makes the arm grasp the item from the tray, move it to the position that the caller has indicated (therefore the parameter of this operation), and drop it there. Consumers and producers keep a `counter` with the items they have handled, and trays have a limit on their capacity (attribute `cap`).

An important characteristic of any system that deals with physical objects is the associated measurement uncertainty, due to tolerance of the mechanical parts and the lack of precision of the arm movements. In order to deal with this kind of uncertainty, we make use of an extension of OCL and UML type Real, called `UReal` [17] that permits expressing values of physical quantities as pairs (x, u) where x represent the value and u the associated uncertainty, following the International Guide to the Expression of Uncertainty in Measurement (GUM) [9]. The corresponding operations on this type take also into consideration the propagation of uncertainty when uncertain values are added, multiplied, or other arithmetical operations are performed [10,17]. This is why all coordinates are expressed by `UReal` numbers.

Each robot arm in this system also has an associated `tolerance` that represents the deviation that the arm may introduce when performing a movement.

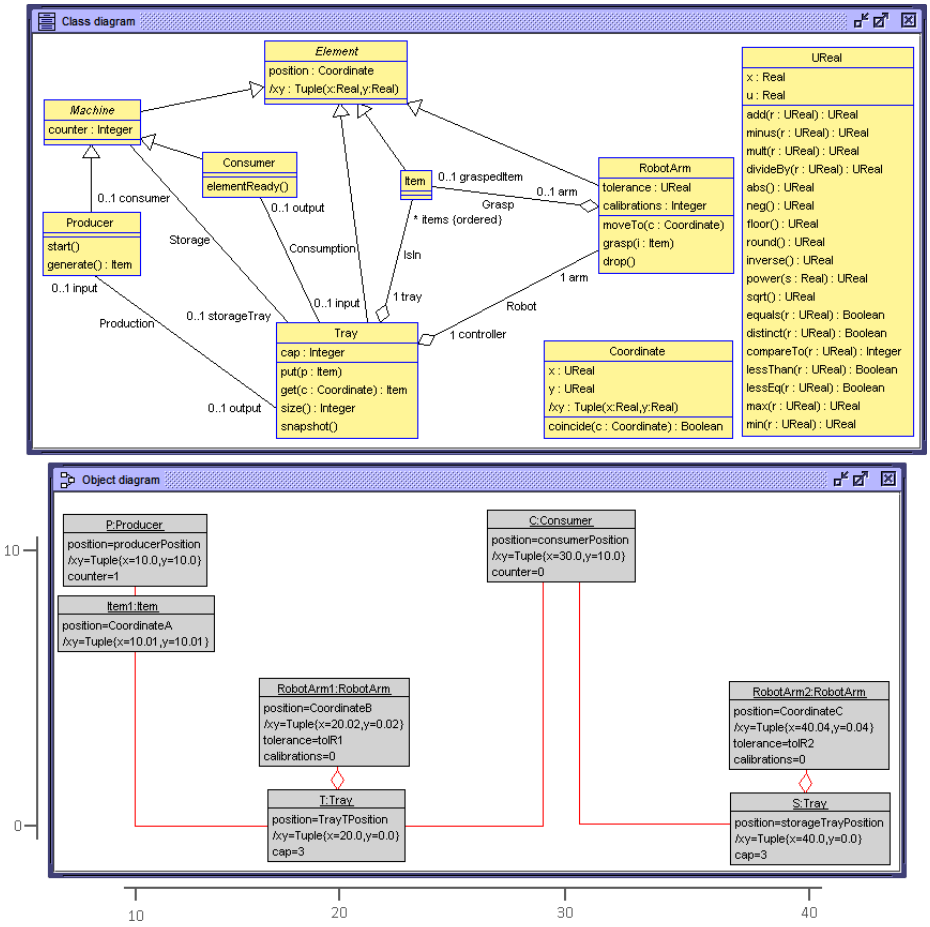


Fig. 1. Class and object diagram for robot arm example.

Besides, when the arm is asked to grasp an item, we check whether the position of the arm `coincides` with the position of the item. In case it does not (due to accumulated uncertainty or excessive tolerance), the arm needs to be calibrated and this is stored in an attribute that keeps track of how many `calibrations` each arm has already needed. Note that every calibration introduces a delay in the system and may have associated costs, and this is why it is important to know how often they occur. Finally, the derived attribute `/xy` is used to facilitate (an approximation of) the visual representation of the objects' coordinates in the real world and the UML diagrams.

Behavioral Elements. The behavior of the system can be expressed in UML and OCL by different means. First, pre and postconditions can be specified on the operations, as shown here for `grasp()` and `drop()` operations of a `RobotArm`.

```
  grasp(i:Item)
    pre  notWithItem: graspedItem.oclIsUndefined()
    post withItem: graspedItem=i
    post calibrationsCount: not self.position@pre.coincide(i.position)
           implies calibrations = calibrations@pre + 1
    post reposition: self.position.coincide(i.position)
  drop()
    post notWithItem: graspedItem.oclIsUndefined()
```

State machines can also be specified on objects. For example, the following listing shows the specification of the *state machine* of a `Tray`.

```
psm PutGet
states
  init:  initial
  Empty  [self.items->size()=0]
  Normal [0<self.items->size() and self.items->size()<self.cap]
  Full   [self.items->size()=self.cap]
transitions
  init   -> Empty  { create }
  Empty  -> Normal { [self.cap>1] put() }
  Normal -> Normal { [self.items->size()<cap-1] put() }
  Normal -> Full   { [self.cap>1 and self.items->size()=cap-1] put() }
  Empty  -> Full   { [self.cap=1] put() }
  Full   -> Empty  { [self.cap=1] get() }
  Full   -> Normal { [self.cap>1] get() }
  Normal -> Normal { [self.cap>1 and self.items->size()>1] get() }
  Normal -> Empty  { [self.items->size()=1] get() }
end
```

On top of that, USE also permits to specify the behavior of operations using a simple executable language called SOIL [4]. For instance, the behavior of `Tray::put()` and `RobotArm::moveTo()` operations can be specified as follows.

```
put(p:Item)
  begin
    insert(self,p) into IsIn;
    self.arm.moveTo(p.position);
    self.arm.grasp(p);
    self.arm.moveTo(self.position);
    self.arm.drop();
  end
  pre  notFull: self.items->size()<cap
  pre  armNotWithItemAtPre: arm.graspedItem=null
  post ElementAdded: self.items=self.items@pre->append(p)
  post armNotWithItemAtPost: arm.graspedItem=null
moveTo(c:Coordinate)
  begin
    declare aux:Coordinate;
    aux := new Coordinate;
    aux.x := c.x.add(self.tolerance);
    aux.y := c.y.add(self.tolerance);
    self.position := aux;
    if self.graspedItem->size() > 0 then
      self.graspedItem.position:=self.position;
    end
  end
```

Expressing and Proving Properties. Once we have the specifications, there are different kinds of analyses that we can perform on the system that show some of the potential advantages of developing model-driven robot descriptions with UML and OCL, such as:

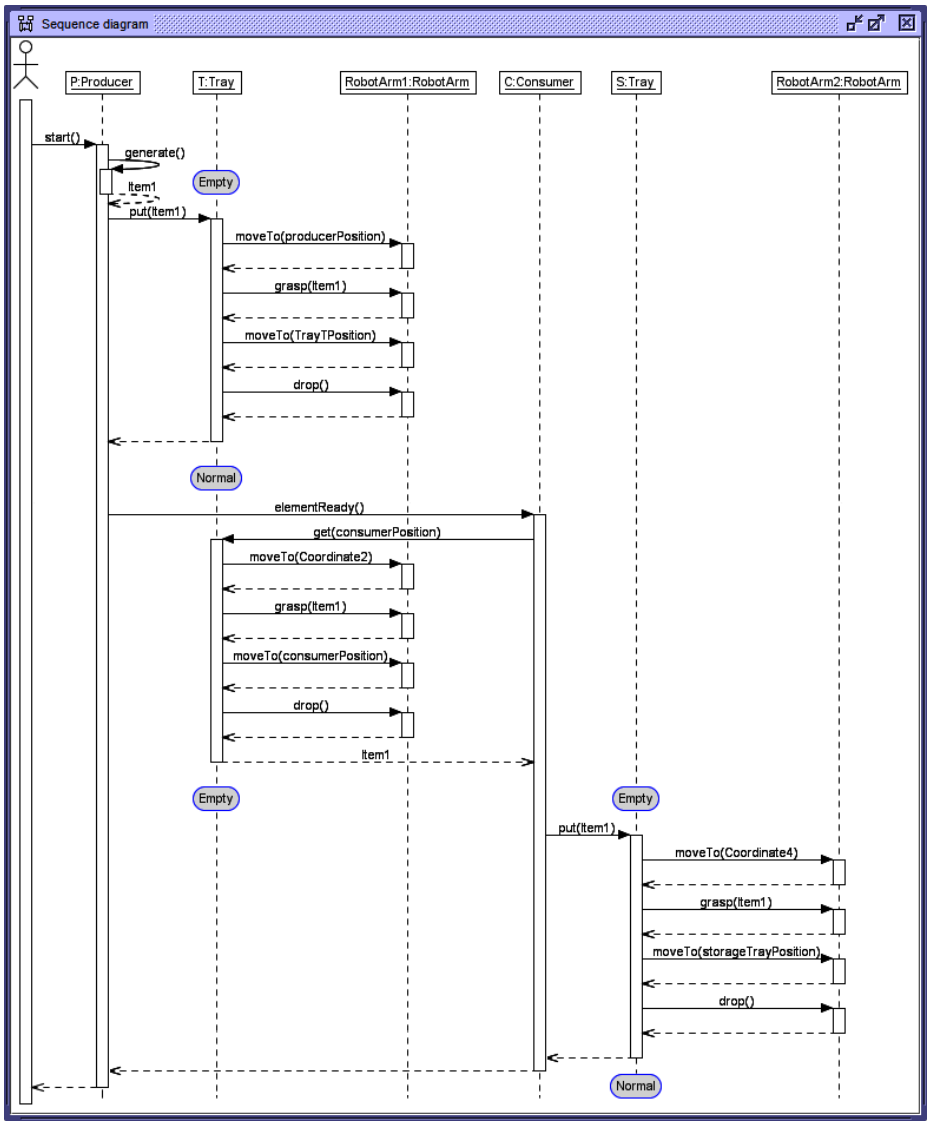


Fig. 2. Sequence diagram displaying the behavior of the system.

- visualization of complex structures and processes
- execution and simulation of scenarios (operation call sequences)
  - different scenarios with different structural properties e.g. trays with different capacities
  - variations of a single scenario with equivalence checking by analysing different operation call orders
- checking structural properties within states by OCL queries
  - e.g. calculating the number of currently produced items
- checking behavioral properties
  - e.g. testing the executability of an operation by testing its preconditions
- checking for weakening or strengthening model properties (invariants, contracts, guards) by executing a scenario with modified constraints
- proving general properties within the finite search space with the USE model validator [7]
  - structural consistency, i.e. all classes are instantiable
  - behavioral consistency, i.e. all operations can be executed
  - checking for deadlocks, e.g. construction of deadlock scenarios due to inadequate tray capacities

For example, based on the specifications above we are able to simulate the system, by creating an initial model of the system and invoking the `start()` operation to the producer. Then, if we have created a system with one producer and one consumer, and the producer just generates one item, the behavior of the system is recreated as shown using the UML sequence diagram in Fig. 2. The sequence diagram shows lifelines for objects and called messages. The evolution of a `Tray` object can also be traced by checking the statechart states that are placed on the lifelines. The behavior can also be displayed as a communication diagram (Fig. 3). As one detail, we emphasize that the `RobotArm1` with the second `moveTo` call moves to `Coordinate2` (displayed in the shown object diagram). `Coordinate2` is close to the `TrayTPosition` but not the exact `TrayTPosition`. This is possible in our approach that allows for uncertain real values.

Similarly, for every step we obtain the state machines of the Tray objects, which can be shown as depicted in Fig. 4.

Finally, the last object diagram in Fig. 6 shows the resulting system state after the system has gone through an iteration. We can see the final positions of the item and the arms. We can also see how the high tolerance that we have indicated for the two robot arms has caused two calibrations.

Figures 5 and 6 pictorially show a filmstrip of the behavior of the system as a sequence of snapshots after every robot arm operation. One can trace in the figures the movements of the `Item`, the `RobotArm1` and the `RobotArm2`. These two figures show two different aspects: a time dimension (through the sequence of diagrams) and a space dimension (within the single object diagrams). The physical placement of the objects is captured by their position in the diagrams: some objects have a fixed position (e.g. the producer, consumer and trays) while others 'move' from object diagram to object diagram as in the real process, e.g. the `Item1` and the two robot arms. Uncertainty is captured through `UReal` values.

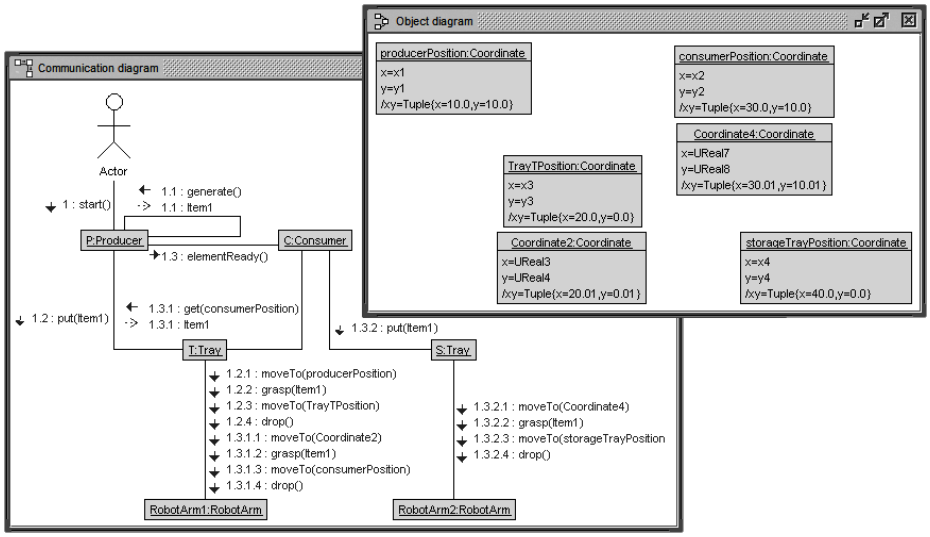


Fig. 3. Communication diagram.

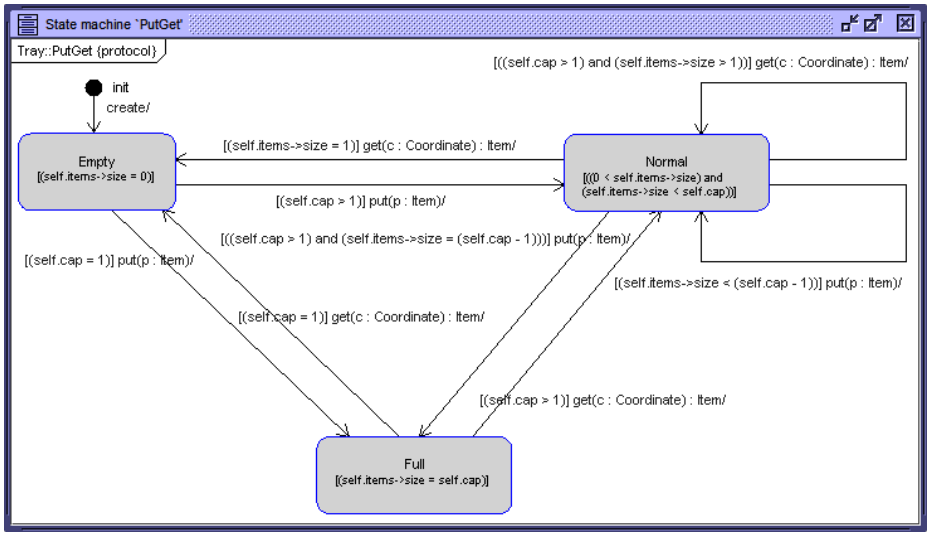


Fig. 4. State Machine for Tray objects.

Aggregation associations are used to visualize 'ownership' between objects (e.g. a robot arm has an item, or an item is placed on a tray).

Another interesting representation of the system behavior is shown in Fig. 7. It depicts a communication diagram showing the operation calls involving an `Item` object. We have also included the associations in which the `Item` engages during the execution of the system (`IsIn`, `Grasp`) as a result of the operations.

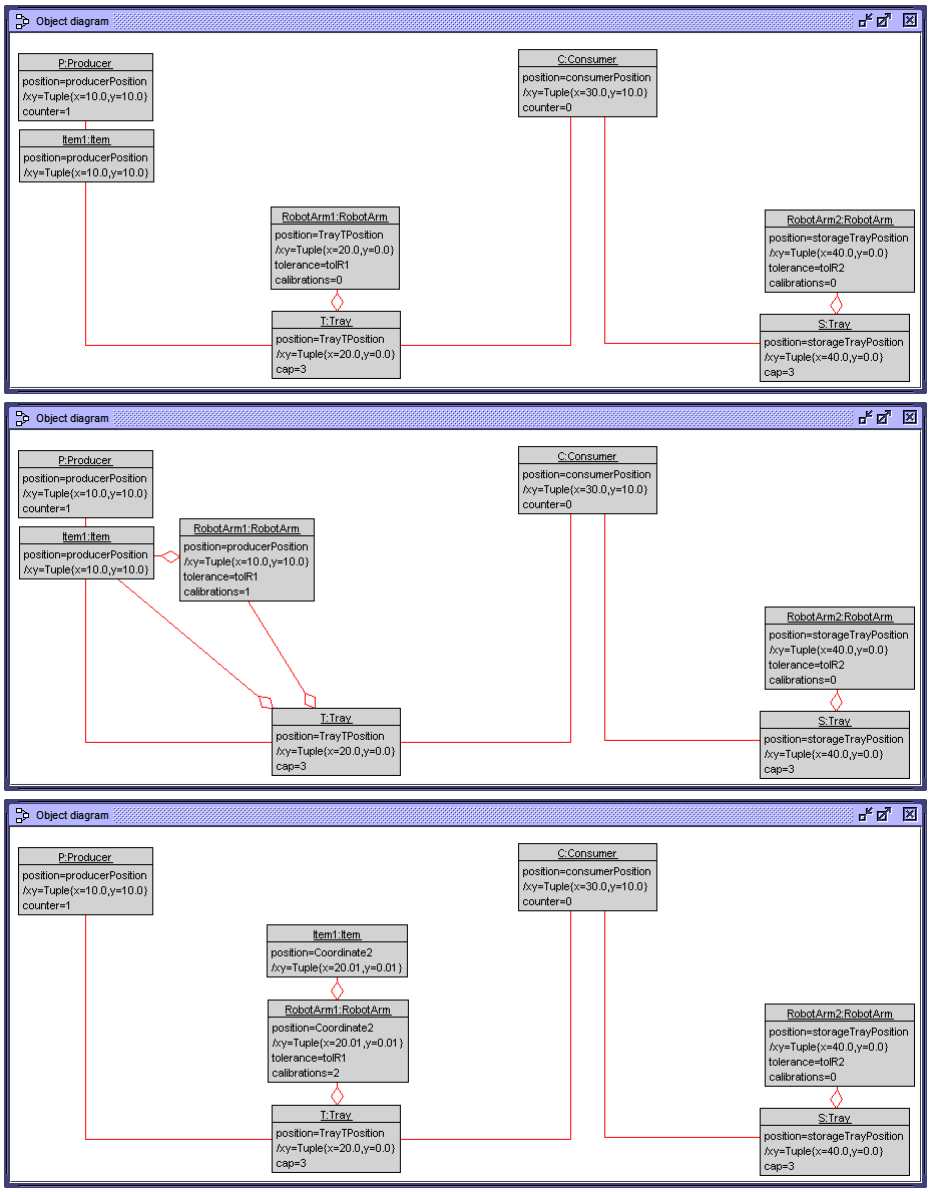


Fig. 5. Object diagram sequence displaying the behavior of the system (Part 1).

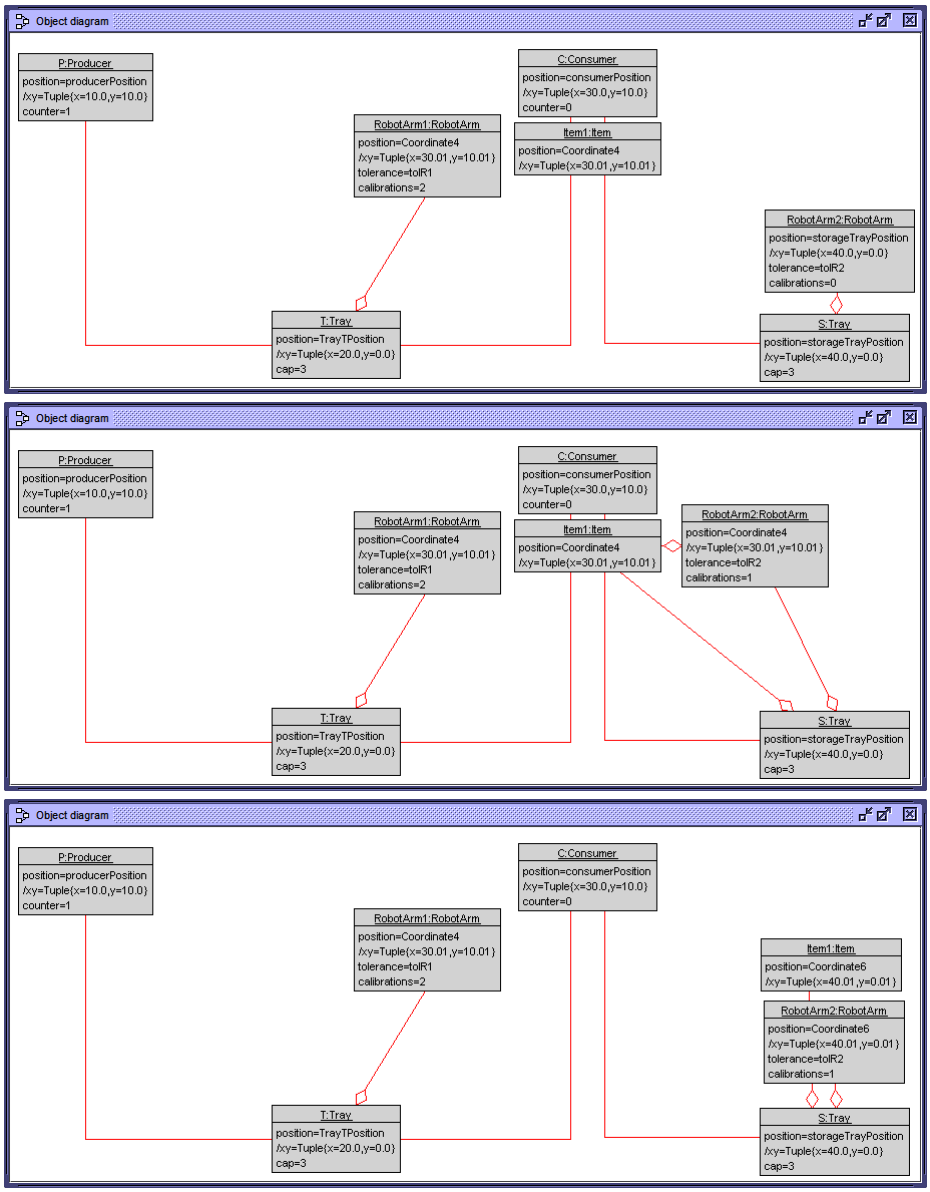


Fig. 6. Object diagram sequence displaying the behavior of the system (Part 2).

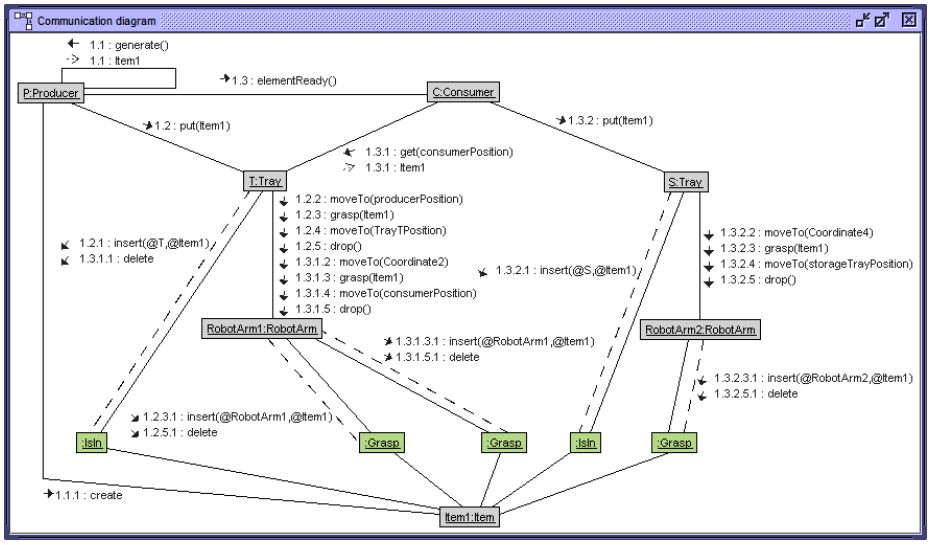
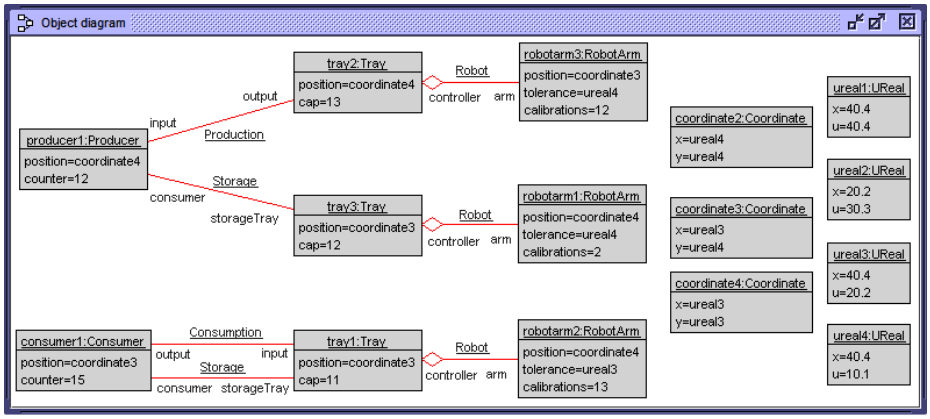


Fig. 7. Communication diagram showing operation calls involving an Item object.

This diagram is very useful to check the order in which operations are called, and their effects. One can trace that the item `Item1` is first created, then put in the first tray, and finally put in the second tray.

Validation and Verification through Testing. Finally, we want to highlight the importance of running structural tests on the metamodels. One of them concerns their instantiability and their ability to faithfully represent the application domain. For example, we decided to ask the USE model validator [7] to generate a producer-consumer-tray constellation using the system metamodel. The resulting object diagram is shown in Fig. 8, together with the model validator configuration that we employed (e.g., optional minimum and mandatory maximum number of instances per class; analogous specifications for associations and datatypes). Interestingly, the produced system is wrong! For example, the producer and consumer are disconnected, sharing no tray between them. This motivates the need to develop additional, currently missing invariants (on the structural system constellation level) and demonstrates the potential usefulness of this approach for validating this kind of properties which are normally overlooked for being considered obvious.

Figure 9 shows another generated test case indicating missing constraints. This time an additionally loaded invariant guarantees a proper Producer-Consumer-Tray constellation. However, erroneously `robotarm2` can grasp an item from the storage tray, and `robotarm1` can grasp an item from the producer output tray. Furthermore, the test case reveals that constraints on the coordinates of machines and trays are missing. The underlying model validator configuration basically looks like the one presented in Fig. 8.



Model element	*Model validator bounds*
Producer	1..1
Production (input:Producer, output:Tray)	1..*
Consumer	1..1
Consumption (input:Tray, output:Consumer)	1..*
Storage (consumer:Machine, storageTray:Tray)	1..*
RobotArm	3..3
Item	0..0
Grasp (arm:RobotArm, graspedItem:Item)	0..*
Tray	3..3
IsIn (tray:Tray, items:Item)	0..*
Robot (controller:Tray, arm:RobotArm)	1..*
Coordinate	0..4
UReal	0..4
Integer	0..15
Real	0.1, 0.2, 0.3, 0.4, 10.1, 20.2, 30.3, 40.4

Fig. 8. Generated test case for Producer-Consumer-Tray configuration.

Another analysis option in our approach with regard to behavioral aspects is, that the developer can check in a system state for the applicability of an operation, for example for the operation `Tray::get(c:Coordinate)`. One can check whether the preconditions of an operation when applied on a particular object together with appropriate parameters are satisfied. One can also construct a respective set of tuples with possible objects to apply the operation to and corresponding parameters.

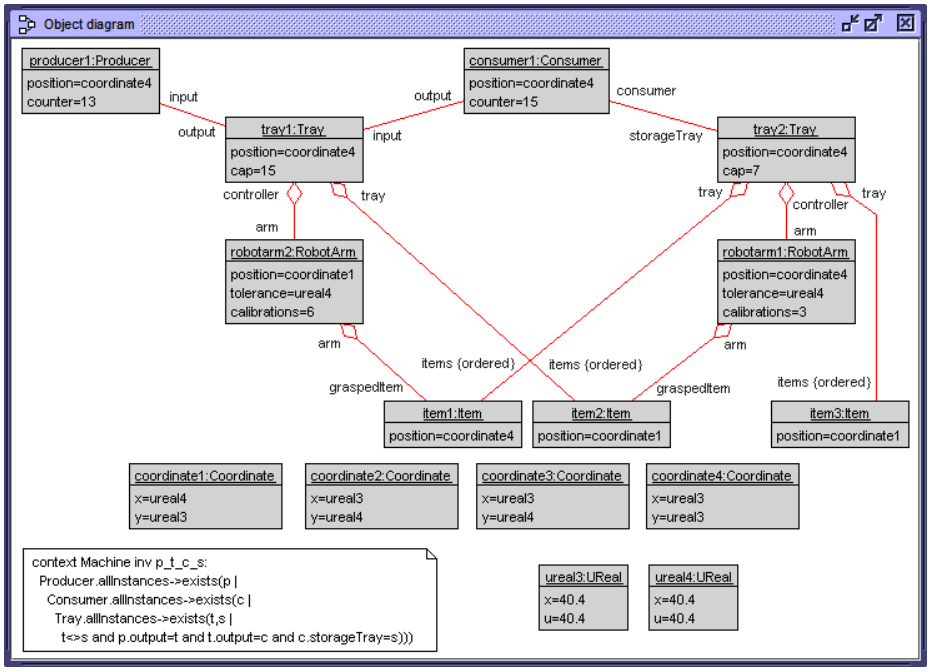


Fig. 9. Test case showing missing constraints for `IsIn` and `Grasp` associations.

The OCL query below retrieves from the last object diagram in Fig. 6 the possible operation calls by returning tuples with a tray `TRAY` whose item set is not empty and whose robot arm is not grasping an item (as required in the preconditions of the operation `get`) together with the coordinate `COOR` of the tray: the result is constructed in such a way that the preconditions of the operation call `TRAY.get(COOR)` would be satisfied for all elements of the returned tuple set, a singleton in this case. Concerning the object diagram in Fig. 9, the query would show that the operation `get` is not applicable there.

```
get(c:Coordinate):Item
pre notEmpty: self.items->size()>0
pre armNotWithItem: self.arm.graspedItem=null

Tray.allInstances->
  select(self|self.items->size()>0)->
    select(self|self.arm.graspedItem=null)->
      collect(t|Tuple{TRAY:t,COOR:t.position})->asSet()

Set{Tuple{TRAY=S,COOR=storageTrayPosition}}:
  Set(Tuple(TRAY:Tray,COOR:Coordinate))
```

3 Related Work

There are different kinds of works that use MDE techniques for modeling robotic applications, depending on their purpose. One set of works focuses on the au-

tomatic generation of components, control logic and other artefacts for the implementation of robotic systems [20,19,5]. Other works focus on transformations between models of different analysis tools that serve as bridges between the separate semantic domains [11,8]. And there are those works that propose models for describing at a high-level and in a platform- and technology-agnostic manner the algorithms and choreography that robotic systems composed of several cooperating agents have to perform to achieve their goals [22,16].

Our paper is more closely related to those works that focus on the specification of the robotic systems themselves. Here the discussion happens between those that propose the use of separate (related) views of the system, using independent domain-specific languages, and those that try to use general purpose modeling languages. One of the major problems with the former approach is the combination of the languages, both horizontally (i.e., at the same level of abstraction—see, e.g. [23]) and vertically (one example of this kind of vertical combination for robotic systems is [2], that uses deep metamodeling [21] to combine system descriptions at different levels). Among the latter, the most widely known ones use high-level component-based architectures with the functional decomposition of the robotic systems, using block-diagrams and/or UML components. Examples include SafeRobots [18], RobotML [3], SmartSoft [4], BCM [5], V3CMM [1] and HyperFlex [3]. Our approach sits at a higher level of abstraction, when not even the architecture of the system needs to be considered, just its basic functionality, and hence many of the details can be abstracted away for later consideration.

We have explored in this paper the option of using a widely used general-purpose modeling language, such as UML, augmented with OCL for the specification of integrity constraints, and pre- and postconditions of operations. On top of them we have used some extensions and tools: (*a*) to be able to execute the specifications we have used SOIL [4]; (*b*) the USE model validator has been employed to generate instances of the model; finally, we have shown how the UML/OCL type system can be easily extended to account for some specific features—namely measurement uncertainty, by defining type `UReal` as an extension of type `Real`. We wanted to follow this path to study its feasibility and expressive power, departing from other approaches that enrich UML with Profiles (such as MARTE [15] or SysML [13]) and make use of action languages like Alf [12] for executing fUML [14] specifications.

A comparison with the pros and cons of our approach with regard to those others is part of our future work, now that we have seen that we are able to get a relevant set of meaningful and workable specifications of these kinds of systems.

[3] `http://robotml.github.io/`
[4] `http://smart-robotics.sourceforge.net/`
[5] `http://www.best-of-robotics.org/bride/bcm.html`

4 Conclusions and Future Work

In this paper we have illustrated the possibilities that UML and OCL offer to model robotic systems at a high-level of abstraction but still providing some key benefits to the system designer. In particular, we are able to describe in a formal manner its structure and behaviour; to incorporate some physical characteristics such as measurement uncertainty; to validate of some of the structural and behavioural properties of the system, and to perform simulation.

There are several lines of work that we plan to address next. First, we want to explore the limitations of our approach due to the type of notations employed. For example, both UML and OCL can handle discrete quantities but are not naturally devised to deal with continuous variables. Some of them are difficult to overcome, but others could have relatively easy solutions. For example, we want to add randomness and other types of uncertainty into our OCL models—e.g., the fact that up to 5% of the generated parts can be defective. We also want to be able to conduct performance analyses about the production time of the system, using e.g. model attributes that specify the time each machine needs to process a part—adding probability distributions to the description of the types of the attributes.

Finally, given that our models just represent early prototypes of the system to study the feasibility of the solution, we want to connect our models to the different analysis and simulation tools currently used in industry, each one able to conduct more fine-grained and precise validations, but of a more heterogeneous nature. In this way, we expect our high-level models to play a pivotal and unifying role that permit connecting the modeling and simulation tools needed for the complete design and validation of these systems.

References

1. Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastor, J., Alvarez, B.: V3CMM: A 3-view component meta-model for model-driven robotic software development. Journal of Software Engineering for Robotics **1**(1) (2010) 3–17
2. Atkinson, C., Gerbig, R., Markert, K., Zrianina, M., Egurnov, A., Kajzar, F.: Towards a Deep, Domain Specific Modeling Framework for Robot Applications. In: Proc. of MORSE'14). Number 1319 in CEUR WS Proceedings (2014) 1–12
3. Brugali, D., Gherardi, L.: HyperFlex: A Model Driven Toolchain for Designing and Configuring Software Control Systems for Autonomous Robots. In: Robot Operating System (ROS): The Complete Reference. Volume 1. (2016) 509–534
4. Büttner, F., Gogolla, M.: On OCL-Based Imperative Languages. Science of Computer Programming **92** (2014) 162–178
5. Djukić, V., Popović, A., Tolvanen, J.P.: Domain-Specific Modeling for Robotics: From Language Construction to Ready-made Controllers and End-user Applications. In: Proc. 3rd WS Model-Driven Robot Software Engineering. MORSE'16, ACM (2016) 47–54
6. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Science of Computer Programming **69** (2007) 27–34

7. Gogolla, M., Hilken, F.: Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In Oberweis, A., Reussner, R., eds.: Proc. Modellierung (MODELLIERUNG'2016), GI, LNI 254 (2016) 203–218
8. Hinkel, G., Groenda, H., Vannucci, L., Denninger, O., Cauli, N., Ulbrich, S.: A Domain-Specific Language (DSL) for Integrating Neuronal Networks in Robot Control. In: Proc. 2015 Joint MORSE / VAO WS Model-Driven Robot Software Engineering and View-based Software-Engineering, ACM (2015) 9–15
9. JCGM 100:2008: Evaluation of measurement data – Guide to the expression of uncertainty in measurement (GUM). Joint Committee for Guides in Metrology. (2008) `http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf`.
10. Mayerhofer, T., Wimmer, M., Vallecillo, A.: Computing with Quantities. (2016) `https://github.com/moliz/moliz.quantitytypes`.
11. Morozov, A., Janschek, K., Krüger, T., Schiele, A.: Stochastic Error Propagation Analysis of Model-driven Space Robotic Software Implemented in Simulink. In: Proc. 3rd WS Model-Driven Robot Software Engineering. MORSE'16, ACM (2016) 24–31
12. Object Management Group: Action Language for Foundational UML (FUML), version 1.0.1. (October 2013) OMG Document formal/2013-09-01, `http://www.omg.org/spec/ALF/1.0.1/PDF/`.
13. Object Management Group: OMG Systems Modeling Language (SysML), version 1.4. (January 2016) OMG Document formal/2016-01-05.
14. Object Management Group: Semantics Of A Foundational Subset For Executable UML Models (FUML), version 1.2.1. (January 2016) OMG Document formal/2016-01-05, `http://www.omg.org/spec/FUML/1.2.1/PDF/`.
15. OMG: UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE). Object Management Group. (June 2008) OMG doc. ptc/08-06-08.
16. Opfer, S., Niemczyk, S., Geihs, K.: Multi-Agent Plan Verification with Answer Set Programming. In: Proc. 3rd WS Model-Driven Robot Software Engineering. MORSE'16, ACM (2016) 32–39
17. Orue, P., Morcillo, C., Vallecillo, A.: Expressing measurement uncertainty in software models. In: Proc. of QUATIC'16. (2016) 1–10
18. Ramaswamy, A., Monsuez, B., Tapus, A.: Model-driven software development approaches in robotics research. In: Proc. of MISE'14, ACM (2014) 43–48
19. Ringert, J.O., Rumpe, B., Wortmann, A.: Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In: Proc. 2015 Joint MORSE / VAO WS Model-Driven Robot Software Engineering and View-based Software-Engineering, ACM (2015) 41–47
20. Ringert, J.O., Roth, A., Rumpe, B., Wortmann, A.: Code Generator Composition for Model-Driven Engineering of Robotics Component and Connector Systems. In: Proc. of MORSE'14). Number 1319 in CEUR WS Proceedings (2014) 63–74
21. Rossini, A., de Lara, J., Guerra, E., Rutle, A., Wolter, U.: A formalisation of deep metamodelling. Formal Asp. Comput. **26**(6) (2014) 1115–1152
22. Ruscio, D.D., Malavolta, I., Pelliccione, P.: A Family of Domain-Specific Languages for Specifying Civilian Missions of Multi-Robot Systems. In: Proc. of MORSE'14). Number 1319 in CEUR WS Proceedings (2014) 13–26
23. Vallecillo, A.: On the combination of domain specific modeling languages. In: Proc. of ECMFA'10. Volume 6138 of LNCS., Springer (2010) 305–320