Goal-driven elaboration of OCL enriched UML class diagrams

Robert Darimont[1], Christophe Ponsard[2], and Michel Lemoine[3]

[1] Respect-IT S.A., Louvain-la-Neuve, Belgium
robert.darimont@respect-it.be
[2] CETIC Research Centre, Gosselies, Belgium
christophe.ponsard@cetic.be
[3] Independent Consultant, Toulouse, France
michel.lemoine@gmail.com

Abstract. UML class diagrams (or equivalent) are used to build a structural model of the system under design, including constraints captured using OCL. Being more focused on design, UML lacks support to make sure the model systematically captures the relevant domain concepts and properties in a traceable way, which is crucial for the subsequent design or testing steps. This paper shows how to apply goal-oriented requirements engineering techniques (GORE) to build UML class diagrams that are as complete as possible and consistent w.r.t. system goals and requirements captured using OCL. A mapping between both representations is defined and implemented based on the KAOS meta-model and USE textual syntax. In addition, the GORE process also enables to consider different alternatives before selecting a specific one and deriving specialised strengthenings of pre and post-conditions operations to enforce the system properties. The proposed approach is illustrated on excerpts of a railway system.

1 Introduction

Model-Driven Engineering (MDE) has become popular for developing software-based systems and is widely adopted to various extents in the industry [10]. It is primary used in the design phase and possibly in connection with other development and testing phases using model transformation, code generation, model-based testing. The Unified Modelling Language (UML) has become the common general purpose modelling language in software engineering [18] and has been further extended into Systems Modeling Language (SysML) for system engineering applications [17].

UML and SysML are not perfect, of course. Their latest evolutions have improved by providing better defined notations and related semantics using the UML superstructure. More formal semantics have also been defined for a number of diagrams. In addition, the Object Constraint Language (OCL) improved expressiveness to state richer properties that cannot be captured using the graphical syntax [16].

Nevertheless, a weak point of UML is its limited support for requirements engineering:

- about notations, UML has the concept of requirement. The Use Case diagram enables a partial capture of main functionalities with limited structuring and only coarse grained interactions between the system and its environment. The class Diagram enables to capture the domain model and a number of related properties, possibly with OCL. The dynamic behaviour of a system can be specified using more specific diagrams such the sequence, activity or state machine diagrams, but those are already directed towards solution design. SysML improves slightly over UML by providing a Requirements diagram with limited structuring capabilities.
- about the modelling process, a number of methods have been defined and can be driven by use cases, business processes or a domain analysis. However, such methods usually fail to precisely record the rationale that triggered the identification of a concept which should be related to the expression of some properties, either descriptive (i.e. domain properties) or prescriptive (i.e. requirements).

In the scope of this paper, we will show how to achieve systematic identification and specification of requirements using standard UML class diagrams. Such diagrams can be used to build a domain model (i.e. at problem level) and/or a conceptual design of the information system (i.e. at solution level). In both cases, a key point is to model the relevant concepts without unnecessary noise. Our focus is on building high quality UML class diagrams (or equivalent) integrating a rich set of descriptive and prescriptive properties in a consistent way with pre and post-conditions. For this purpose, we will rely on goal-oriented requirements engineering (GORE) defined in languages like KAOS [4], i* [25], or URN/GRN [11]. The key point of using such frameworks is that only the reference to goals make it possible to claim a specification is complete [26]. We will rely here on the KAOS variant which already includes a conceptual modelling close to UML class diagram and has a well documented process [13]. Our main contribution is to derive a more complete UML model using OCL to capture properties and pre/post conditions using the textual specification defined by the UML Specification Environment (USE) has target language [7].

The structure of our paper follows a model construction process. We assume the reader is familiar with UML/OCL. Section 2 first introduces the GORE modelling notations and presents the requirements of a small underground transportation system used as running example. Section 3 explains our mapping and implementation through USE. Section 4 and 5 respectively detail how to deal with operations and alternatives. Section 6 discusses the benefits and limitation of our approach in the light of related work. Finally Section 7 concludes and present our future work. The whole paper will rely on different excerpts of a railway system in order to illustrate the proposed approach.

2 Goal-oriented requirements engineering with KAOS

Goals capture, at different levels of abstraction, the various objectives the system under consideration should achieve. GORE is concerned with the use of goals for eliciting, elaborating, structuring, specifying, analysing, negotiating, documenting, and modifying requirements [12].

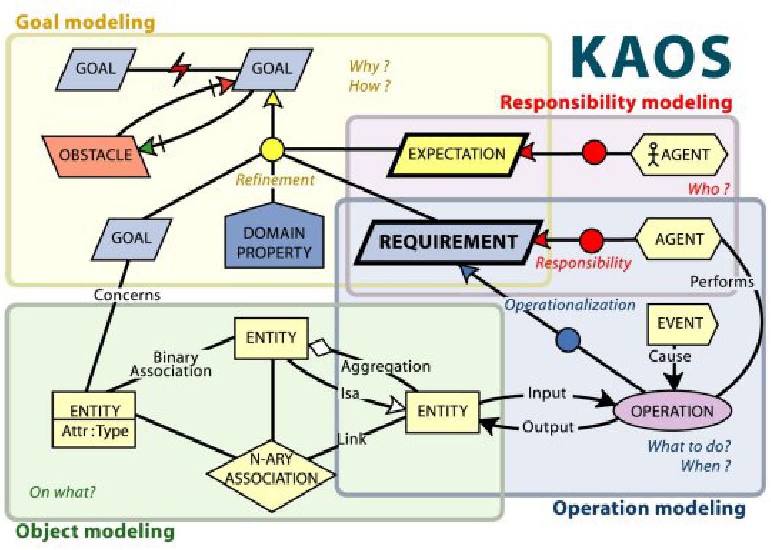

Fig. 1. KAOS meta-model together with its graphical syntax

.

The KAOS method is structured on the following four sub-models which are depicted in Figure 1 together with the graphical notations that will be used in the figures of this paper.

– The **goal model** structures functional and non-functional goals of the considered system. It is graphically represented as a directed goal graph. Goal-related concept are graphically represented as parallelograms. Requirements have thicker borders. A goal model also helps identify potential conflicts and obstacles and reason about their resolution [12]. Obstacles are represented with a red background and with an opposite tilt than goals.
– The **object model** defines and interrelates all the concepts involved in a goal specification. A conceptual object can be an entity, a relationship, an event or an agent depending on whether it is an autonomous, subordinate, instantaneous, or active object. Only the two former kinds are captured by the UML class diagram and are actually represented with similar notations. Note that the KAOS term object model should not be confused with a UML object model, which is an instance of a class diagram.

- The **agent model** identifies the agents of both the system and the environment as well as their interface and responsibilities. They can be shown as part of goal graphs or in more specific diagrams. Agents are represented using an hexagonal shape.
- The **operations model** describes how agents functionally cooperate to ensure the fulfilment of the requirements assigned to them and hence the system goals. Operations are graphically represented using ovals. Functional flow diagrams can be used between them.

The different abstraction levels to express goals can range from high-level strategic goals like "*Maintain[Underground Safety]*" down to more operational goals such as "*Avoid[travellers falling on tracks]*" or "*Avoid[train collisions]*" as depicted in Figure 2. High-level goals can be progressively refined into more concrete and operational ones through relationships linking a parent goal to several sub-goals, with different fulfilment conditions using either "AND-refinement" (all sub-goals need to be satisfied) or "OR-refinement" (a single sub-goal is enough, i.e. possible alternatives). For example, to avoid train collision different design can be used like fixed block systems or moving block systems. Later in the paper, we will also see how to address the problem of a train that is longer than the platform (modelled as an obstacle in Figure 2).

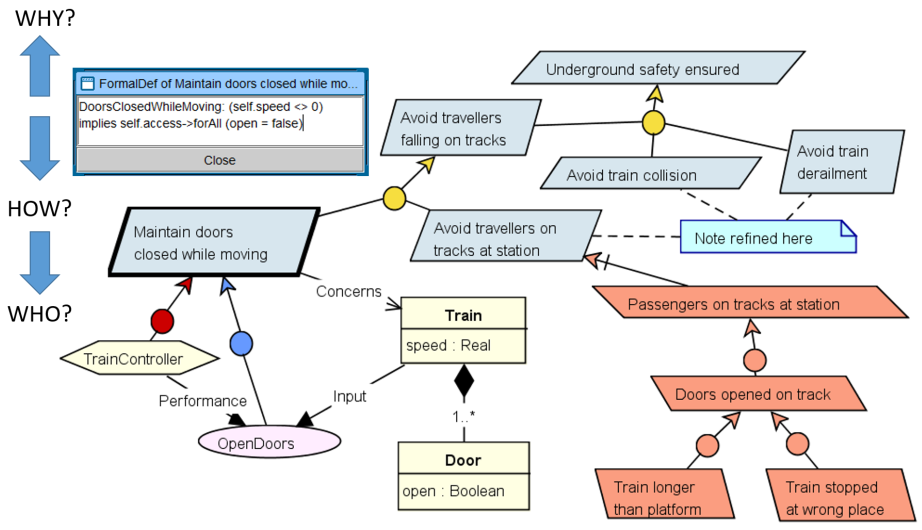
```
DoorsClosedWhileMoving: (self.speed <> 0)
implies self.access->forAll (open = false)
```
WHY?
HOW?
WHO?
FormalDef of Maintain doors closed while mo...
Close
Underground safety ensured
Avoid travellers falling on tracks
Avoid train collision
Avoid train derailment
Maintain doors closed while moving
Avoid travellers on tracks at station
Note refined here
Concerns
Train
speed : Real
Passengers on tracks at station
TrainController
Performance
Input
OpenDoors
1..*
Door
open : Boolean
Doors opened on track
Train longer than platform
Train stopped at wrong place

Fig. 2. Partial goal model with obstacles and related object model concepts

.

The "WHY" and "HOW" questions can be used to conveniently navigate to parent and sub-goals, respectively. The goal decomposition stops when reaching a goal controllable by an agent, i.e. answering the "WHO" questions about responsibility assignment. These goals are either requirements on software or hardware components to design, or expectations on the behaviour of agents in the environment (e.g. the train driver). Domain properties can also be involved in a refinement. Such properties are intrinsically valid, like "A train can only be at one location at a time" or be assumed like the fact that the train length is shorter than the length of all platforms. In this case they need to be challenged about their validity.

Based on properties expressed in the goal model, one can start identifying concepts and structuring them into the relevant concepts of the object model (i.e. UML class diagram). Analysing the requirement *Maintain[Door Closed While Moving]* will result in the identification of a *Train* entity. Moreover the Train entity will be bound to a *Door* entity through a composition relationship with obviously at least one door. The modelling approach is minimalist: only what is strictly needed is modelled. For instance, at this point there are no other concept identified and only two attributes are necessary the Door state (e.g. a Boolean to represent a Door is closed) and the Train movement state (e.g. a Real representing its speed). The resulting object model is depicted in Figure 2. Note that a trace link between the Train entity and requirements involving it is added using the "Concerns" link. This also allows to formalise that requirement in the context of that Entity using the following OCL statement (also depicted has pop-up property in Figure 2):

Requirement 1 (Maintain Doors Closed While Moving)
InformalDef: The train doors must be kept closed while it is moving
FormalDef: `(self.speed <> 0) implies self.doors->forAll (closed)`

An agent is a software, device, or human component of the system that plays some specific role in goal satisfaction. It controls behaviours by performing operations and run concurrently with each other. An agent is modelled by responsibility links to goals. As stated earlier requirements differ from expectation based on the fact the responsibility is on the system or the environment. In our case, the *Train Controller System* (TCS) will be responsible of the "*Maintain[Doors Closed While Moving]]*" requirement based on its capability to operate the *OpenDoors* operation. Operations control state transitions and can be specified using trigger/pre/post conditions either unconstrained or required to enforce a specific requirements. This will be further detailed in Section 4.

3 Generation UML class diagram with USE

UML Specification Design Environment (USE) was mainly designed to validate and verify specifications consisting of class diagrams together with OCL invariants and pre- and postconditions [7]. It provides support for instance of class diagrams (i.e. UML object diagrams), including the ability to generate model instances. It also has a nice textual syntax both for model and instance levels. Once loaded into the tool, those models can be inspected visually using different standard diagrams and with some automated layout support. The tool also provide a control over the instance level and a command line interface allowing one to load models and perform checks. The USE tool is implemented in Java and available in Open Source license [5].

Table 1 defines how to map a GORE object model into a UML class diagram. This mapping covers the various constructs like entities, associations, inheritance, different kind of goal concepts (Requirement, DomProp,...) and operations. It was implemented using a Java plug-in for the Objectiver tool that supports the KAOS method [21].

Table 1. Mapping GORE onto USE

GORE (KAOS)	UML (USE)
Entity/Agent	class
Obj1 isA Obj2 ; Obj1 isA Obj3	class Obj1 <Obj2, Obj3
Binary Association without attributes but with association type, ie. association, composition, aggregation	association composition aggregation
Binary Association with attributes	associationclass
N-ary Association rel without attributes	association
N-ary Association rel with attributes	associationclass
Operation (and related pre/post-condition)	Operation of the class associated to the agent performing the operation
FormalDomInvar of Entity/Agent	Clause inv: in the class of the entity/Agent
FormalDomInvar of N-AryAssociation	Clause inv: in the association class
FormalDef of Requirement/DomProp	Clause inv: in the constraints section

Applying the above mapping to the KAOS object model of Figure 2 yields the following UML model. At this point, traceability is ensured by the same naming convention, although more specific identifier could also be added later to guarantee a better and round-trip model synchronisation.

Model 1 *Translation of basic UML class Diagram*

```
Model UndergroundTransportationSystem
  class Train
    attributes
      speed : Integer
  end

  class Door
    attributes
      closed : Boolean
  end

  composition Rel_1 between
    Train[1]
    Door[1..*] role doors
  end
```

```
class TCS
  operations
    openDoors (tr: Train)
    -- <operation will be defined in section 4>
end

constraints
  context Train
    -- train doors must be kept closed while the train is moving.
    inv DoorsClosedWhileMoving:
      (self.speed <> 0) implies self.doors->forAll(closed)
```

4 Deriving pre and post-conditions

A KAOS operation is a state transition inside the system that can be controlled by some agent. The unconstrained operation is specified using domain preconditions (i.e. necessary condition) and post-conditions (resulting state). Note that we will not consider trigger conditions (i.e. sufficient conditions) because it is not supported by UML. For the *OpenDoors* operation, the domain pre and postconditions are: `tr.doors->forAll(closed)` and `tr.doors->forAll(not closed)`

We need to be sure that the operation fulfils all the invariant properties defined (through the requirements). Hence, each time we introduce an operation with its domain pre- and postconditions into a formal system, we need to be sure that the operation execution will preserve the system invariant in any circumstances. This can induce to specify complementary conditions that must be fulfilled before the operation execution (additional preconditions) and/or specify complementary conditions that must be fulfilled by the result of the operation to be compliant with the requirements. In those cases, the conditions are named required pre- and postconditions.

Such required conditions can actually be computed using specific reasoning techniques called goal regression detailed in [15]. Specific patterns are also available. To illustrate, let us start by the comparison of Requirement 1:

```
  tr.doors->exists (not closed) implies (tr.speed = 0)             (1)
```

As specified above, the OpenDoors operation opens all doors:

```
  post: tr.doors->forAll(not closed)                               (2)
```

As there is at least one door in the train, we can assert:

```
  tr.doors->forall(not closed) implies tr.doors->exist(not closed) (3)
```

Hence, when *TCS* performs an *OpenDoors* operation for a train `tr`, we get a state in which the following assertion is verified:

```
  tr.doors->forAll (not closed)                                    (4)
```

Due to (3), this implies that:

```
  tr.doors->exist (not closed)                                     (5)
```

And in order to preserve Requirement 1, we need to have:

```
  pre DoorsClosedWhileMoving: tr.speed = 0                         (6)
```

The resulting UML operation corresponding to the KAOS operation (as defined in the mapping of Table 1) is described below. It is bound to the TCS UML class which is mapped on the TCS agent in KAOS. Note that the rationale for this additional precondition is explicitly traced in the OCL specification.

Model 2 *Strengthened OpenDoors Operation*

```
openDoors (tr: Train)
  pre: tr.doors->forAll (closed)
  -- DoorsClosedWhileMoving: The train must be stopped.
  pre DoorsClosedWhileMoving: tr.speed = 0
  post: tr.doors->forAll (not closed)
```

5 Selecting among alternatives

Now let us focus on a more specific safety goal that our system should enforce in the case the train is stopped at a station.

Goal 1 (Avoid Passengers on Tracks At Station)
InformalDef: Passenger should never be allowed to step out of the train on the tracks when the train is stopped at the station.

Obstacles analysis can be applied to enforce this goal with a maximal level of safety, Figure 2 shows how when starting from the negation of the goal, different problems can be identified, such as the train being longer than the platform length or being stopped at the wrong place. In the rest of this section, we will reason about the platform length problem assuming the driving is stopping at the right place. We will explore two possible alternative designs that can address the problem. A few extra concepts are required and will enrich our KAOS object model and the corresponding USE/UML class diagram, like the association *at* between a train and a station as well as the *platformLength* of a station. Those are depicted in Figure 3 and 4.

5.1 Alternative 1 : train shorter than platform length

Let us assume the length is standard on all stations. A domain invariant could be the following one (both in natural language and in OCL):

DomInvar 1 (StandardPlatformLength)
InformalDef: Platforms are always 100m long
FormalDef: `Context Station`
```
inv: self.platformLength = 100
```

We could also assume the train length is compliant with this.

Assumption 1 (CompliantTrainLength)
InformalDef: Train length is always less than platform length
FormalDef: `Context Train`
```
inv: self.at<>NULL implies (self.length <= self.at.platformLength)
```

Such an assumption could become a requirement on another agent, i.e. the agent that is responsible for assembling the trains. For example, the enforced rule might be the following one:

Requirement 2 (TrainLength)
InformalDef: All trains are between 55m and 75m long
FormalDef: `Context Train`
```
    inv: self.length >= 55 and self.length <= 75
```

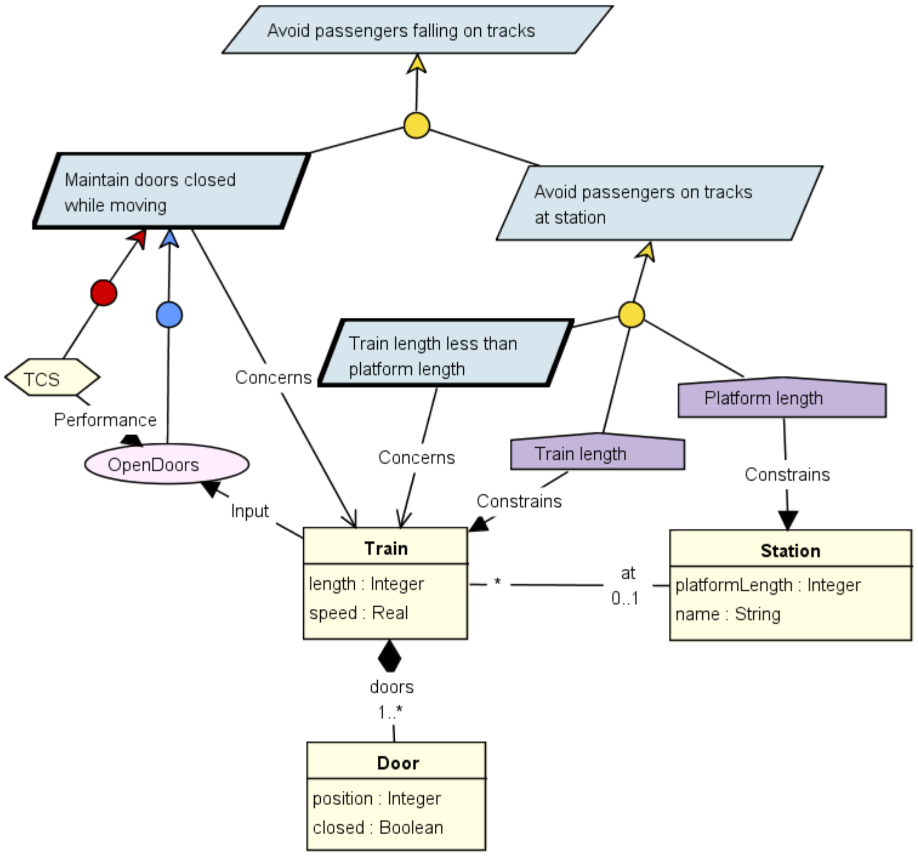

Fig. 3. Dealing with a first alternative design

.

The resulting model is depicted in Figure 3 without considering the "*No doors opening outside of platform*" at this point. We can see the object model was enriched with a few attributes like *Station.platformLength* and *Train.length*. The resulting (right) refinement is easy to manually prove correct as the maximum length is less than 100. The translation into the USE tool can also check the invariant but on a specific instantiation, thus acting more as a runtime monitor.

5.2 Alternative 2 : forbid doors opening outside of platform

The previous example is based on the hypothesis that all trains are between 55m and 75 m long. What happens if we rule out this hypothesis, by not requiring any maximal train length? As trains can now be more than 100m long, we need to find another way to guarantee the *No passenger going out on track at station* requirement. With such a train stopped in a station, according to the fact that, so far, all train doors are either all opened or closed, passengers being in the part of the train out of platform could go down on the tracks. We need therefore to forbid opening doors in the out of platform part of the train. In the Figure 4, an alternative refinement and a new requirement have been introduced to this aim.

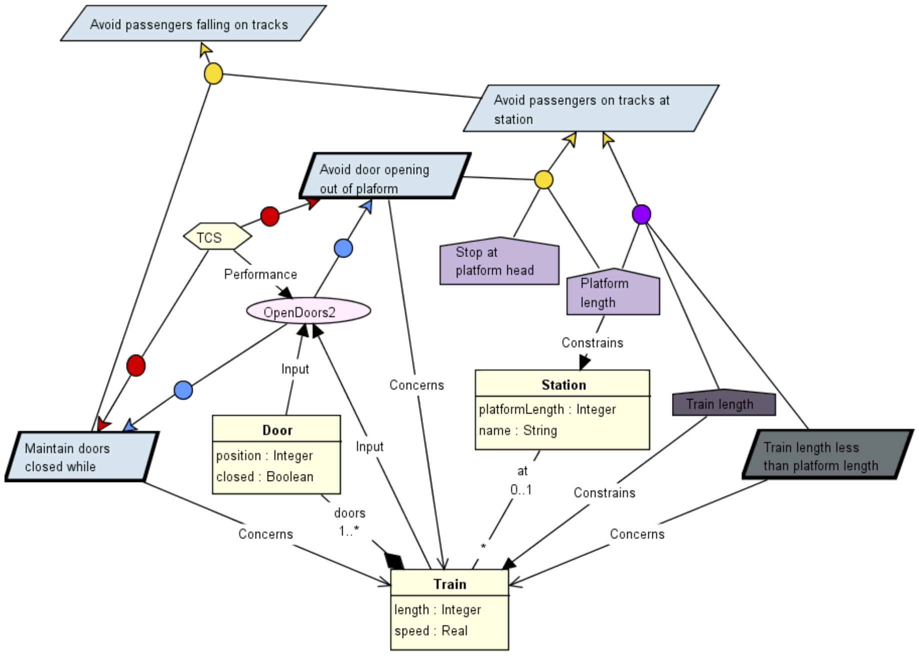

Fig. 4. Dealing with a second alternative design

.

This second design is more complex but can actually be stated in a single requirement:

Requirement 3 (NoDoorOpeningOutOfPlatform)
InformalDef: When a part of a train stopped at a platform is not within the platform length, then those doors should not open
FormalDef: `Context Train`

```
    inv: self.doors->forAll (d:Door |
        d.position > self.at.platformLength implies d.closed)
```

Using similar regression techniques than in section 3, one can show that in this case the strengthened operation is the following one:

Model 3 *Strengthened OpenDoors Operation for Alternative 2*

```
OpenDoors (tr: Train, selectedDoors: Set(Door))
  pre AllClosed: tr.doors->forAll(closed)
  pre Consistency: tr.doors->includesAll(selectedDoors)
  pre ZeroSpeed: selectedDoors->notEmpty implies tr.speed = 0
  pre OutOfPlatform:
    tr.doors->forAll(d:Door | not selectedDoors->includes(d)
                   implies d.position > tr.at.platformLength)
  post: tr.doors->forAll(d:Door | d.closed
                   implies not selectedDoors->includes(d))
```

Looking at the resulting model, one can see both designs are overlapping: the *Door.position* attribute of the second design is not required in the first design. This is not altering the behaviour but just introducing some noise. The problem is different if we consider the *OpenDoors* operation has other sets of additional preconditions involved. Generating a consistent UML diagram requires to make a decision about one of the alternatives. This process can be error prone but can be automated by computing the recursive closure of all related concepts of a design, except those that are shared. Such an alternative selection mechanism is supported by the developed plugin. Visually, the discarded concepts are greyed out. Actually in Figure 4, the first design presented in Figure 3 is greyed out except for shared concepts like the *PlatformLength* assumption and the *Station* entity.

6 Discussion w.r.t. related work

Previous work has already considered how to bridge the gap between requirements and UML. As stated in the introduction, use cases are quite popular and usually combine the *Use Case* diagram with more specific templates with approaches like Cockburn's explicitly capturing relevant goals [3]. However this approach is lightweight, mainly functional and does not cope with any formalisation of the domain.

Another approach is to try to develop a GORE approach directly inside UML through the stereotype-based extension mechanisms provided by UML. Such an approach was proposed by Heaven and Finkelstein [8]. It eases the translation with UML. However, the GORE notations can only partially be captured using such mechanisms. For example, AND-OR relationships for alternatives are difficult to represent, especially to capture the strengthened pre/post-conditions. It also comes at the price of developing the extension, unless it is already available, for example the RE tool provides GORE support for i* and KAOS inside StarUML [23]. Even in this case this still requires to develop more complex features such as the alternative selection mechanism.

Problem Frames, another famous requirements based approach, has also considered an UML mapping [14]. Problem Frames have less commonalities with UML than methods such as KAOS whose object model is a superset of the UML class diagram. As a result, they are rarely adopted in UML-based software development processes employing UML. A UML bridge has thus potentially a larger impact here but the adoption can be also more difficult so we believe an approach based on a closer language like KAOS, i* or GRL is probably better. Nevertheless the mapping implemented is interesting because it considers components diagrams which can lay good architectural grounds rather than going into conceptual modelling which usually result jumping too fast in an object-oriented development.

Interestingly, the problem frame mapping does not consider OCL for formalising properties but OTL Object Temporal Logic. This removes our limitation of only addressing invariants (i.e. safety requirements in our running example). However this extension is not standard; actually, other temporal extensions for OCL have been proposed, e.g. TOCL [27], OCL/RT [2], as part of ReqMon for goal monitoring [22]. They lack tool support which limits their usefulness. This limitation can be alleviated to some extend by translating such extension in standard OCL over an enhanced UML model using the so-called filmstripping technique, i.e. explicitly modelling state snapshots. It was applied to the USE framework [9]. However we did not try to consider it at this point because enforcing liveness properties would also require to introduce trigger-conditions which is not part of the standard UML specification of operations. As invariants are the most critical properties to consider, hence to model formally, we believe this limitation is acceptable. Actually other formal languages like Event-B [1] suffer from the same limitation. For this language, UML and GORE mappings are also available [19][24].

For fully dealing with formalised requirements, it is still possible to carry out the reasoning at the GORE level. A number of frameworks have developed specific tool support for temporal logic for example Formal Tropos [6] or the FAUST toolset for verifying mission critical systems [20]. Such early formalisation tools do not however seem to receive industrial adoption. Our approach to address most critical requirements with decent guidelines and support using standard UML seem to make sense. For system requiring high assurance, formal methods and tools will be applied by experts in that area starting from good requirements model consistently coupled with a UML design model as we are advocating for here.

7 Conclusion

Building high quality UML class diagrams is a difficult task because modelling needs to be carried out with a specific purpose in mind. Without analysing the requirements while building the model, it is easy to miss important concepts or introduce irrelevant ones. In this paper, we have shown how to conduct such a process guided by a specific goal-oriented requirements engineering method and relying on the USE framework for easing the mapping. However, we believe the methods and guidelines detailed here can be applied more widely with the limitation they only consider safety properties. A key point is that the explicit capture of properties helps to expose their degree of robustness and to decide how to improve the design. Using obstacle analysis, a systematic questioning can be applied, e.g. what is the doors would not close (due to mechanical failure), how should the system react in this case, what information is required, etc. This will result in an update of the UML model in terms of its structure and operations.

Our mapping has been implemented as a plugin for the Objectiver tool. However, it is currently limited to the export from a requirements to UML while it would be interesting to retro-engineer existing UML models or more tightly integrate both approaches for a full roundtrip engineering. This would also enable the use of verification and validation tools available at both levels. Dealing the larger set of progress properties would also be interesting but would require non-standard temporal extensions to OCL and add more complexity, reducing the interest of the approach. This work was also limited to functional requirements and dealing with non-functional requirements (NFR) can also be investigated, for example in the way they drive the identification of alternatives. Although less easy to manage in a model, OCL can be helpful to query model characteristics than can drive the evaluation of some categories of NFR. Last but not least, we plan to conduct a comparative validation experiment to assess the quality of the resulting model with and without the use of the proposed technique and plugin. This will take place in the scope of the training and coaching the authors are organising both at academic and industrial levels in Belgium and France.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York, NY, USA, 1st edn. (2010)
2. Cengarle, M.V., Knapp, A.: Towards ocl/rt. In: Proc. of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right. pp. 390–409. FME '02, Springer-Verlag, London, UK, UK (2002)
3. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn. (2000)
4. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Sci. Comput. Program. 20(1-2), 3–50 (Apr 1993)
5. Fabian Buttner and others: Use tool. https://sourceforge.net/projects/useocl (2002)
6. Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., Traverso, P.: Specifying and analyzing early requirements in tropos. Requirements Engineering 9(2), 132–150 (2004)
7. Gogolla, M., Büttner, F., Richters, M.: Use: A uml-based specification environment for validating uml and ocl. Science of Computer Programming 69(1), 27 – 34 (2007), http://www.sciencedirect.com/science/article/pii/S0167642307001608, special issue on Experimental Software and Toolkits
8. Heaven, W., Finkelstein, A.: A uml profile to support requirements engineering with kaos. In: IEE Proceedings - Software. p. 1 (2004)

9. Hilken, F., Gogolla, M.: Verifying linear temporal logic properties in uml/ocl class diagrams using filmstripping. In: 2016 Euromicro Conference on Digital System Design (DSD). pp. 708–713 (Aug 2016)
10. Hutchinson, J., Whittle, J., Rouncefield, M.: Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. Science of Computer Programming 89, 144 – 161 (2014), special issue on Success Stories in Model Driven Engineering
11. International Telecommunication Union: Recommendation Z.151 (10/12), User Requirements Notation (URN) Language Def. (2012)
12. van Lamsweerde, A.: Goal-oriented requirements engineering: a guided tour. In: Fifth IEEE Int. Symposium on Req. Eng. pp. 249–262 (2001)
13. van Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications. Wiley (2009)
14. Lavazza, L., Del Bianco, V.: Combining problem frames and uml in the description of software requirements. In: Proc. of the 9th International Conference on Fundamental Approaches to Software Engineering. pp. 199–213. FASE'06, Springer-Verlag, Berlin, Heidelberg (2006)
15. Letier, E., van Lamsweerde, A.: Deriving operational software specifications from system goals. In: Proc. of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering. pp. 119–128. ACM, New York, NY, USA (2002)
16. OMG: Object Constraint Language (OCL) - Version 2.4. https://www.omg.org/spec/OCL (2014)
17. OMG: System Modelling Language (SysML) - Version 1.5. http://www.omgsysml.org (2017)
18. OMG: Unified Modelling Language (UML) - Version 2.5.1. http://www.omg.org/spec/UML (2017)
19. Ponsard, C., Devroey, X.: Generating high-level event-b system models from KAOS requirements models. In: Actes du XXIXème Congrès INFORSID, Lille, France, 24-25 mai 2011. pp. 317–332 (2011)
20. Ponsard, C., Massonet, P., Molderez, J., Rifaut, A., van Lamsweerde, A., Van, H.T.: Early verification and validation of mission critical systems. Formal Methods in System Design 30(3), 233–247 (2007)
21. Respect-IT: Objectiver. http://www.objectiver.com
22. Robinson, W.N.: Extended ocl for goal monitoring. ECEASST 9 (2008)
23. Sam Supakkul and others: Re tool. http://www.utdallas.edu/ supakkul/tools/RE-Tools (2012)
24. Snook, C., Butler, M.: Uml-b: A plug-in for the event-b tool set1. In: ABZ2008 Conference (2008)
25. Yu, E.S.K., Mylopoulos, J.: Enterprise modelling for business redesign: The i* framework. SIGGROUP Bull. 18(1), 59–63 (Apr 1997)
26. Yue, K.: What Does It Mean to Say that a Specification is Complete? In: Fourth International Workshop on Software Specification and Design (IWSSD-4), Monterey, USA (1987)
27. Ziemann, P., Gogolla, M.: Ocl extended with temporal logic. In: Broy, M., Zamulin, A.V. (eds.) Perspectives of System Informatics. pp. 351–357. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)