SN Computer Science (2023) 4:175
https://doi.org/10.1007/s42979-022-01577-0

ORIGINAL RESEARCH

Detection of Inconsistencies in SysML/OCL Models Using OWL Reasoning

Shan Lu[1] · Alexey Tazin[1] · Yanji Chen[1] · Mieczyslaw M. Kokar[1] · Jeff Smith[2]

Received: 19 May 2022 / Accepted: 16 December 2022 / Published online: 19 January 2023

Abstract
Requirement change management is a challenging issue in software development. One of the main objectives of the Intent-Defined Adaptive Software program is to verify the satisfaction of requirement changes during software development. In this paper, we develop an ontology-based method to detect inconsistencies in Systems Modeling Language (SysML) models with Object Constraint Language (OCL) constraints as a first step of requirement change management. Specifically, we map the SysML/OCL models to Web Ontology Language (OWL), so that the consistency of the corresponding ontology can be checked by OWL reasoners automatically. We propose a set of mapping rules to interpret the components of SysML state machine diagrams, along with OCL constraints, to OWL. Toward this objective, we demonstrate three consistency reasoning tasks over a state machine diagram using OWL reasoners. In each case, the result of reasoning is accompanied by an explanation of the logic behind the decision.

Keywords SysML models · OCL constraints · Consistency checking · OWL reasoning

Introduction

The DARPA-supported Intent-Defined Adaptive Software (IDAS) program [1] sought to develop technologies that enable rapid adaptation of software to changes in requirements and operating environments. The main objective of this program was to develop methods for the management of requirement changes in software development. The step in the change management process that is addressed in this paper is the identification of inconsistencies in requirements. Building an ontology as a common understanding of the structure of information between stakeholders is a widely used approach in software engineering for software requirements change management [2, 3]. It allows the reuse of domain knowledge, and making explicit domain assumptions that will allow assumptions to be changed easily when the domain knowledge is changed. In the work described in this paper, we focus on the verification of the satisfaction of changes in the requirements during software development. Our approach relies on the use of ontology for this purpose.

There are two main advantages of using ontology in software engineering. First, ontologies provide common vocabularies of given domains that can be shared among software developers working on different aspects of software applications. Second, once the model of the software and the user requirements are represented as an ontology in OWL [4], requirement satisfaction can be verified automatically using an inference engine. One of the advantages of using OWL is that OWL inference is decidable.

In this paper, we focus on the latter issue. However, instead of proving that user requirements are satisfied, we will show some "reasonable assurances" to the

This article is part of the topical collection "Advances on Model-Driven Engineering and Software Development" guest edited by Luís Ferreira Pires and Slimane Hammoudi.

✉ Shan Lu
lu.sha@northeastern.edu

Alexey Tazin
tazin.a@northeastern.edu

Yanji Chen
chen.yanj@northeastern.edu

Mieczyslaw M. Kokar
m.kokar@northeastern.edu

Jeff Smith
jeff.smith@sncorp.com

[1] Department of Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts 02115, USA

[2] Sierra Nevada Corporation, Sparks, Nevada 89434, USA

developer that the model is correct, reserving full verification of requirement satisfaction to future work. This paper is an extension of the work originally presented in MODELSWARD 2022 [5]. We will provide a more complete picture of the mapping from SysML/OCL models to OWL DL, and will show more detailed explanations for the OWL reasoning results.

Problem Statement

The Unified Modeling Language (UML) is a widely used industry standard language that provides graphical notation for software specification and design in the early phases of software development. The SysML is an extension of a subset of the UML. However, UML/SysML models alone are not expressive enough to represent constraints on the modeling concepts. The OCL is used to express constraints in UML/SysML models. Many UML/SysML tools support adding OCL constraints in UML class diagrams and SysML block diagrams. However, none of the tools supports the semantics checking of the constraints. Because UML/SysML are not formal languages (they have formal syntax but lack formal, computer-processable semantics), we cannot perform theorem proving in UML/SysML to verify the consistency of a model. In other words, the UML/SysML tools do not check if the model is correct according to these constraints.

To support software developers with automated reasoning capabilities when developing models of software, we develop an ontology-based method to reason about the correctness of SysML models that include OCL constraints. Specifically, we map SysML block diagrams, state machine diagrams, and OCL constraints to OWL, and check the consistency of the corresponding ontology by running an OWL inference engine. Although significant amount of research on mapping UML/SysML models to ontologies has been reported (e.g., [6–9]), almost all of the researchers limit their scope of investigation to class diagrams. There is a lack of widely accepted mapping rules for the mapping of UML/SysML behavior diagrams to OWL and thus more research in this area is needed.

In this paper, we propose a set of mapping rules to interpret the components of SysML state machine diagrams, along with OCL constraints, to OWL DL. The novelty of our approach is the identification and implementation of a more complete mapping of UML/SysML to OWL, than what is included with current CASE tools. There are a few papers on the mapping of the UML/SysML behavior diagrams to OWL DL, however, they either use a non-OCL logic, a restricted set of OCL operators and/or do not support formal proof. Our mapping rules have two advantages: (1) they show how to translate both basic state machine elements (including states, transitions, events, actions, guards, and triggers) and OCL constraints in the state machine diagrams (including the OCL invariants in states and guards) to OWL, and (2) the OCL to OWL translation rules cover relational operators (e.g equivalent, greater/less than) between variables.

The existing research on consistency checking of UML state machine diagrams primarily focuses on the detection of contradictions between: (1) the UML metamodel and state machine specifications, and (2) state machine diagrams and other types of diagrams. Verification that state machine diagrams are not contradictory with the requirements expressed as OCL constraints is also a very important issue in software development. In this paper, we check for contradictions within models that include OCL constraints on state machine diagrams. We demonstrate this capability on three exemplary inference tasks.

The rest of this paper is organized as follows. In "Related work", we review some of the existing literature that is related to our work. In "OWL reasoning about SysML block-diagrams", we show the OWL axiom usage in reasoning about SysML block diagrams. In "State machine diagram to OWL mappingrules", we propose a set of mapping rules to interpret the components of state machine diagrams along with OCL constraints to OWL DL, and demonstrate three reasoning tasks using the OWL reasoner Pellet. Finally, "Conclusions" summarizes our work.

Related Work

Consistency checking of UML/SysML models is an important step in MBSE-based system development. The definitions of types of consistency are still an open research topic, c.f., [10–12]. One of the UML consistency classifications is horizontal vs. vertical consistency. Horizontal consistency, also called intra-model consistency, means the lack of contradictions between different diagrams at the same level of abstraction. Vertical consistency, also called inter-model consistency, means the lack of contradictions between different diagrams at different levels of abstraction. Another basic classification of consistency in UML is syntactic vs. semantic consistency. Syntactic consistency refers to the relation between UML diagram specifications and a UML metamodel, whether the syntax of a given diagram is compatible with the syntax prescribed by the metamodel. Semantic consistency refers to the meaning of UML diagrams, i.e., to the notion of *truth* - whether any contradictions in the model do not exist and whether a concept can be instantiated. Other methods of consistency classification were also discussed in the literature, e.g., static versus dynamic consistency, multilevel consistency, and the nature of errors.

In this paper, we focus on the consistency of UML models that include requirements expressed in OCL. Since the

OCL constraints capture the semantics of the domain, our approach falls in the semantic validation category.

Consistency Checking via Mapping to Formal Languages

Many approaches rely on the mapping of UML/SysML models to a formal languages and the use of automatic proof engines for reasoning on these models. We looked at these methods with respect to the support of automatic proof, model analysis, structural and behavioral modeling, and checking for consistency of models that map requirements expressed as OCL constraints. While many recent papers propose mapping of UML to OWL, most of the papers seem to ignore the fact that UML and OWL have different semantics. This issue was first discussed in [13], where the authors identified similarities and differences between UML and DAML (DARPA Agent Markup Language). To close the gap between the two representations, the authors proposed extending UML by adding two metamodel elements called *Property* and *Restriction*, where a property is a grouping of association ends and a restriction is a classifier for objects. The recommendation from the [13] paper has not materialized primarily due to the fact that UML has not been modified as suggested in the paper. Moreover, DAML became OWL.

UML2Alloy [14, 15] maps UML/OCL models to Alloy notations, and then the Alloy model is automatically analysed by the Alloy Analyzer. Alloy is an object modeling language based on first-order logic (FOL), offering declaration syntax compatible with graphical object-oriented models, and state-based formulas. However, Alloy models do not provide semantic notations which are necessary during the analysis phase of software development [16].

USE [17, 18] includes an interpreter for a subset of UML and OCL. It provides its own UML/OCL user interface and lets one check constraints (invariants and pre- and post-conditions). That is, it checks model states (snapshots) making sure invariants are not contradictory. However, this is not a formal system and it uses a custom UI that does not support XMI import/export.

Rehman et al. [19, 20] modeled a smart parking system and a sewage system using UML activity diagrams, translating them to automata-based models, and then to temporal logic of actions. A (TLA)-based formal method was utilized to validate and verify system properties using the TLA+ toolbox [21]. The toolbox includes a proof system and a high level language to generate TLA code. TLA+ is a good tool for verifying code simulating state machines, but it is not clear how it could be used for model analysis, which was our objective.

Automated Reasoning on UML/OCL conceptual Schemas (AuRUS) [22] is a standalone application which allows verifying and validating UML/OCL conceptual schemas specified in ArgoUML. Verification consists in determining whether the schema satisfies a set of well-known desirable properties such as class liveliness or non-redundancy of integrity constraints. Validation consists of allowing the user to perform queries about reachable states of the schema. However, AuRUS only validates the structural part of a schema. Validation of the behavioral part of a schema is not supported.

There are many other methods surveyed in [10, 12] that are based on a formal language. In particular, the authors considered mapping UML models to DL. None of the methods reviewed in that paper check the consistency of the OCL constraints. Some researchers pursue verification of consistency, e.g., [23, 24], but not of UML/SysML models.

Ontology-Based Consistency Checking

In [25], the authors investigate the method, called TwoUse, to integrate a UML model and an OWL ontology. Since OWL classes cannot be exploited through OCL expressions, the authors propose an extension to the OCL basic library, called OCL-DL, to permit operations to call the OWL reasoner. In OCL-DL, the authors propose new operations which rely on reasoning engine services to extend the boundaries of OCL toward OWL. However, their method only focuses on UML class diagrams.

In [26], the authors represent class diagram operations using three ways: (1) FOL n-ary predicate that has to satisfy some FOL assertions. These are assertions that type the input parameters and the return value, and ensure the uniqueness of the return value; (2) DLR-ifd (variation of DL) operation is represented as an n-ary relation. The same assertions as in the previous case are used; (3) ALCQI (variation of DL) approach is based on reification. An operation is expressed as an atomic concept -the ALCQI role. There are also assertions that type the input parameters and the return value. There is no consideration of names of operation parameters.

The approach that we are using in this paper is in line with [26].

Mapping Behavior Models to OWL DL

The method in [9] translates UML state machines and OCL constraints to DLR—an expressive description logic (DL) that supports n-ary relations. The basic idea is that the states and transitions in a state diagram are mapped to primitive concepts in DL. We have not found tool support for the translation. Also, the translation of OCL to OWL does not allow relational operators between variables.

In [7], the authors describe a transformation of UML statechart primitives to OWL DL. In their transformation, a specific state is defined by a class expression constrained by transitions and state conditions. A specific transition is defined by an intersecting class expression standing for the source, target, event and guard of this transition. However, this work does not support translation of OCL constraints.

In [27, 28], the authors translate both the UML class and statechart diagrams of a model to a single ontology, and analyze the consistency and satisfiability of the model using OWL reasoners. However, the authors do not provide any translation of transitions in the statechart diagram. Moreover, translation of OCL to OWL does not allow relational operators between variables.

The method in [6] represents the state machine using OWL individuals. It is difficult to extend this method for OCL support, since the mapping of OCL to OWL that we use is class based.

The detailed comparison of these mapping rules is shown in Table 4.

OWL Reasoning about SysML Block Diagrams

Ontology Foundations

Ontology is a field of philosophy dealing with the nature of "being". In computer science, the same term has a somewhat different meaning: an ontology is an explicit, formal specification of a shared conceptualization [29]. To represent ontologies in computer science, the five basic constructs are used to capture knowledge about a specific domain: classes, properties, instances, constraints, and axioms.

- Classes: also called concepts or types, represent groups of things that share common characteristics. They could be either concrete objects of the real world or abstract concepts.
- Properties: are used to express relationships between two classes in a given domain. They are the ways in which classes and instances can be related to one another.
- Instances: are also known as individuals and are the things that the ontology describes or potentially could describe.
- Constraints: determine which values are allowed for properties or what relations should hold for specific classes of individuals.
- Axioms: classes, properties, and constraints can be put together to form logical statements or assertions. Axioms are formally stated descriptions of what must be true in order for some assertions to be accepted.

OWL is a family of standardized ontology languages with formal semantics that is used to formalize ontologies. To facilitate the representation of ontologies, many visualization tools have been developed. Protégé [30] is a popular open-source ontology editor and knowledge base framework. In this project, we used Protégé to develop our ontologies.

Mapping of SysML Block Diagram to OWL

The Cameo Enterprise Architecture tool is widely used by software engineering teams for modeling systems. The usage of this tool follows the Model-Based Software Engineering process. It includes many useful features; in particular, it is useful for the identification of potential design flaws early in the development stage, when they are easier to fix rather than fixing them in later stages. However, the capabilities of Cameo Enterprise Architecture are limited in detecting semantic inconsistencies in SysML/OCL models. For instance, when we "validate" a model known to be inconsistent, Cameo Enterprise Architecture shows "validation was successful". The reason for this is that Cameo Enterprise Architecture validation function is limited to checking the syntax of OCL constraints, but not their semantics. In our experiments, we used Cameo Concept Modeler plugin [31] for Cameo Enterprise Architecture to translate SysML block diagrams to OWL. Then we used rules to translate OCL contraints to OWL.

The mapping from UML class diagrams to OWL and DL has been studied by many papers, as described in "Related work". We summarize the widely accepted rules described in these papers in Table 1. The first column shows UML concepts, the second shows to what it is mapped in DL and the third shows a corresponding concept in OWL.

The OCL is used to express constraints on UML/SysML models, for example, the restrictions on the value of object attributes, and the restrictions on the existence of objects. In this paper, we only consider a subset of OCL constraints, the OCL invariants. The OCL invariant is associated with a Classifier (also referred to as a "type") in UML/SysML models. An OCL expression that is an invariant evaluates to be true if the condition is met. All invariants of a type must be true for all instances of that type at any time. Therefore, OCL invariants can help us to check for design contradictions of UML/SysML models. OCL invariants were manually translated into OWL 2 DL axioms based on [25] and [32]. Specifically, for a block diagram, the OCL invariants of a block are translated into OWL object property restrictions. In addition, our translation introduces relational operators between variables. Table 2 shows the mapping principles that we follow in this paper. Here, C_1, C_2, C are SysML blocks, p is either a block value or an association end, and V is a SysML value type. In the case that V is an instance of the SysML value type, the mapping of the third invariant in Table 2 should be $C \sqsubseteq \exists p.\{V\}$, and the mapping of the fourth invariant in Table 2 should be $C \sqsubseteq \neg\exists p.\{V\}$.

Table 1 The mapping of UML components to OWL components

UML Component		DL Formalization	OWL Component
Class diagram	Class	Concept	Class
	Property	Role	ObjectProperty
	Attribute	Data property	DatatypeProperty
	Operation	Role	ObjectProperty
	Association	Role	ObjectProperty with restriction
	Composition	Role	ObjectProperty with restriction
	Aggregation	Role	ObjectProperty with restriction
	Generalization	Subsumption	rdfs:subClassOf
Object diagram	Object	Individual	Individual
	Link	Role	ObjectProperty assertions

Table 2 Principles of mapping of OCL to OWL

Invariant	OWL DL
`Context C1 inv: p−> forAll(oclIsTypeOf(C2))`	$C_1 \sqsubseteq \forall p.C_2$
`Context C1 inv: p−> exists(oclIsTypeOf(C2))`	$C_1 \sqsubseteq \exists p.C_2$
`Context C inv: p = V`	$C \sqsubseteq (\forall p.V) \sqcap (\exists p.V)$
`Context C inv: p != V`	$C \sqsubseteq \neg\exists p.V$

The Use of OWL Axioms for Consistency Checking

The reason for mapping SysML models and OCL constraints to OWL is to support the software developers with automatic consistency checking by using OWL reasoners. Inconsistencies of the OWL axioms indicate that there are errors in the models. We could invoke OWL reasoners to show expectations when the ontology mapped from SysML/OCL models is inconsistent. OWL DL provides three types of class axioms to construct a class description—*SubclassOf*, *EquivalentClasses*, *DisjointClasses*—that can be used for checking consistency.

Generalization associations between two blocks are translated into *SubclassOf* axioms of the corresponding OWL classes. Moreover, OCL invariants of a block are translated into *SubclassOf* axioms of the corresponding OWL class. In DL, we represent this as $C_1 \sqsubseteq C_2$. When such a subclass axiom is part of an OWL model, for any individual x of C_1, the fact *(x rdf:type* C_2*)* is inferred by an OWL reasoner. If the class description of C_1 has conflicts with the class description of C_2, the OWL reasoner will detect this as an inconsistency.

Generalizations with the "Equivalient Class" stereotype in SysML are translated into *EquivalentClasses* axioms of the corresponding OWL classes. In DL, we represent this as $C_1 \equiv C_2$. If the class description of C_1 has conflicts with the class description of C_2 (i.e., the sets of the individuals from these two classes do not fully overlap), the OWL reasoner will detect this as an inconsistency.

Dependencies with the "Disjoint With" stereotype in SysML are translated into *DisjointClasses* axioms of the corresponding OWL classes. In DL, we represent this as $C_1 \equiv \neg C_2$. In such a case, if the class description of C_1 has any overlap with the class description of C_2, the OWL reasoner will detect this as an inconsistency.

An Example

Here, we consider an example based on the cloud agility baseline (CAB) model [1] that SNC and Northeastern developed to provide an adaptable framework which can be molded to meet typical logistics and cloud applications and changes to those requirements. This CAB model provided the flexibility for the various exercises to promote reuse of microservice components. For a logistics example, it might be hosted (initially) in AWS and provide a web-based interface for routing military supplies given a certain workflow and certain geopolitical realities. Figure 1 shows the SysML block diagram of the CAB model. A shipment requests the dispatcher to schedule an assignment for it. An assignment is a shipment–transporter pair to transit and deliver a shipment. Shipments have three possible states: pending, in transit, or delivered, and transporters have two possible states: busy or idle. The question we are investigating in this paper is - how do we verify the correctness of changes in the CAB model, e.g., are the state machines of the CAB correct?

Cameo can translate the five blocks in the CAB SysML model and the associations between them to the classes and properties in the CAB ontology. Figure 2 shows the main structure of the CAB ontology in Protégé. However, the Cameo mapping-to-OWL capability is very limited and the automatic translation loses some information in translation.

We extended the automatically generated CAB ontology by mapping the OCL constraints in the Dispatcher block and the Shipment block into OWL restrictions for the corresponding classes manually (shown in Figs. 3 and 4). In

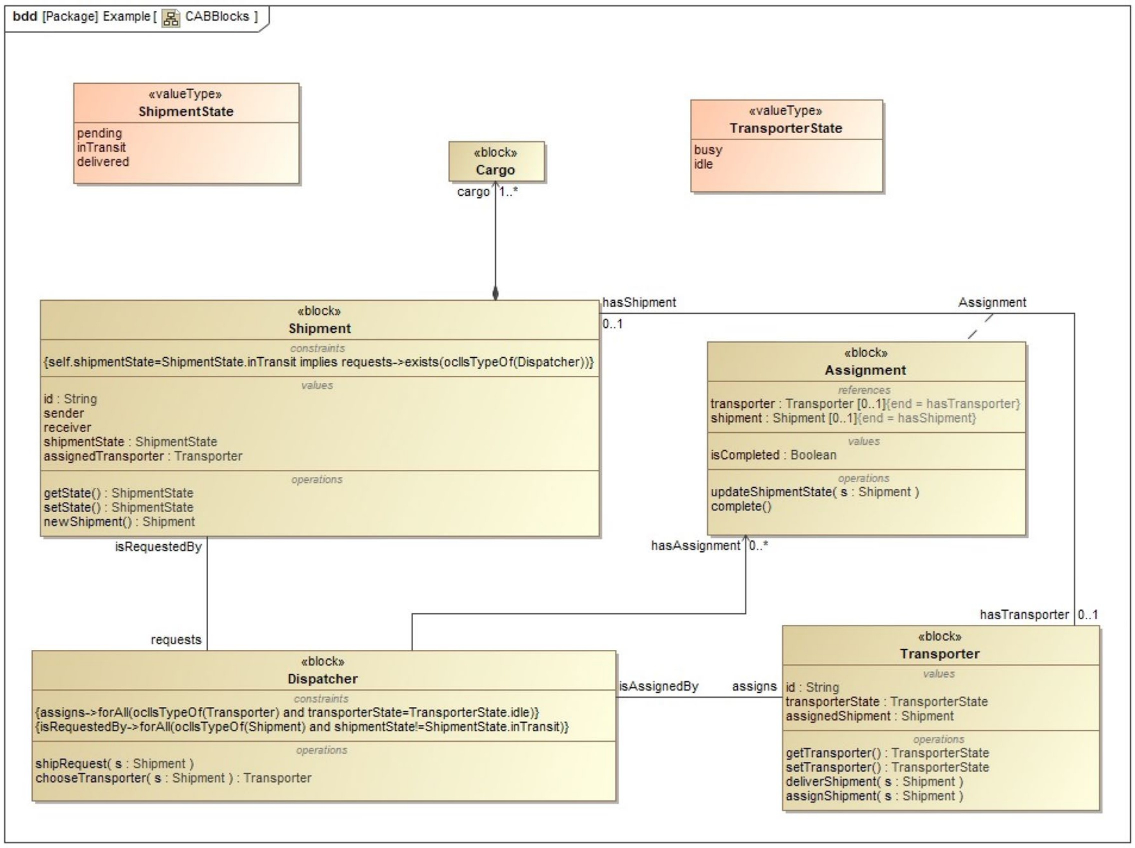

bdd [Package] Example [CABBlocks]
«valueType» ShipmentState
pending
inTransit
delivered
«block» Cargo
cargo 1..*
«valueType» TransporterState
busy
idle
«block» Shipment
constraints
{self.shipmentState=ShipmentState.inTransit implies requests->exists(oclIsTypeOf(Dispatcher))}
values
id : String
sender
receiver
shipmentState : ShipmentState
assignedTransporter : Transporter
operations
getState() : ShipmentState
setState() : ShipmentState
newShipment() : Shipment
hasShipment 0..1
Assignment
«block» Assignment
references
transporter : Transporter [0..1]{end = hasTransporter}
shipment : Shipment [0..1]{end = hasShipment}
values
isCompleted : Boolean
operations
updateShipmentState(s : Shipment)
complete()
isRequestedBy
hasAssignment 0..*
hasTransporter 0..1
requests
«block» Dispatcher
constraints
{assigns->forAll(oclIsTypeOf(Transporter) and transporterState=TransporterState.idle)}
{isRequestedBy->forAll(oclIsTypeOf(Shipment) and shipmentState!=ShipmentState.inTransit)}
operations
shipRequest(s : Shipment)
chooseTransporter(s : Shipment) : Transporter
isAssignedBy
assigns
«block» Transporter
values
id : String
transporterState : TransporterState
assignedShipment : Shipment
operations
getTransporter() : TransporterState
setTransporter() : TransporterState
deliverShipment(s : Shipment)
assignShipment(s : Shipment)

Fig. 1 SysML block diagram of the CAB model [5]

Fig. 2 Main structure of the CAB ontology [5]

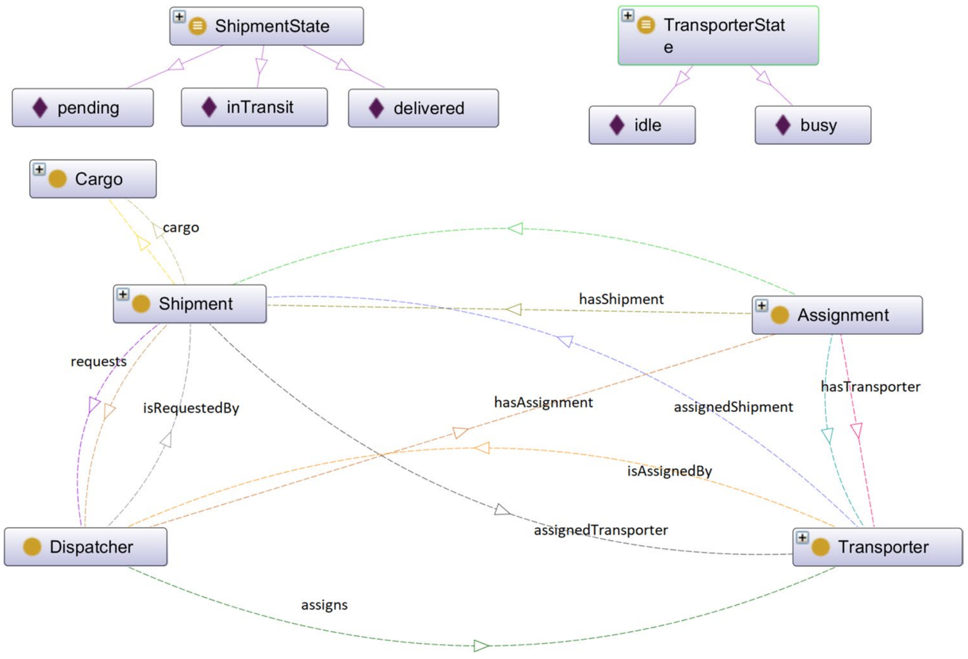


Table 3 Mapping of OCL invariants in CAB SysML model to OWL

OCL Invariants in CAB SysML model	OWL DL
`Context Shipment inv: self. shipmentState=Shipment.inTransit implies request->exists(oclIsTypeOf(Dispatcher))`	$Shipment \sqsubseteq \exists requests.Dispatcher \sqcap \exists shipmentState.\{inTransit\}$
`Context Dispatcher inv: assigns->forAll(oclIsTypeOf(Transporter) and transporterState=TransporterState.idle)`	$Dispatcher \sqsubseteq \forall assigns.(Transporter \sqcap \exists transporterState.\{idle\})$
`Context Dispatcher inv: isRequestedBy->forAll(oclIsTypeOf(Shipment) and shipmentState!=ShipmentState.inTransit)`	$Dispatcher \sqsubseteq \forall isRequestedBy.(Shipment \sqcap \neg\exists shipmentState.\{inTransit\})$

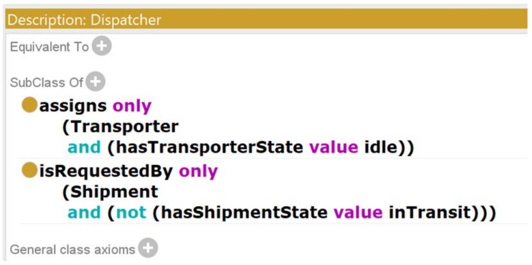

Description: Dispatcher
Equivalent To
SubClass Of
assigns only (Transporter and (hasTransporterState value idle))
isRequestedBy only (Shipment and (not (hasShipmentState value inTransit)))
General class axioms

Fig. 3 OWL restrictions for the Dispatcher class mapping from OCL

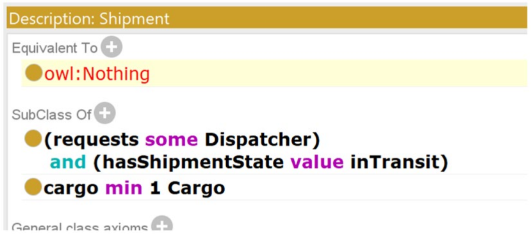

Description: Shipment
Equivalent To
owl:Nothing
SubClass Of
(requests some Dispatcher) and (hasShipmentState value inTransit)
cargo min 1 Cargo
General class axioms

Fig. 4 OWL restrictions for the Shipment class and the OWL reasoning results

this step, we followed the mapping rules shown in Table 3. Finally, we ran Pellet, the OWL reasoner embedded in Protégé. It identified the semantic inconsistency of the model (shown in Fig. 4): the Shipment block is equivalent to Nothing, i.e., the OWL class is not satisfiable (it cannot have any individuals).

State Machine Diagram to OWL Mapping Rules

The main concepts of state machines are *state, transition, event, guard* and *action*. These concepts are mapped to OWL following the rules shown in Table 4. A simple state is mapped to a class expression in OWL. The classes are constrained by transitions and *state invariants* expressed in OCL. All the state classes are disjoint. In OWL, each transition of a state machine diagram is represented by an intersection of class expressions for the source state, target state, event and guard of this transition. In addition, our translation introduces relational operators between variables.

A Mapping Example

The statechart diagram in Fig. 5 describes the behavior of the Dispatcher block from Fig. 1. Each state in the statechart diagram is annotated with a state invariant.

The six states are mapped to six OWL classes. The *Allocating* state is a composite state with three sub-states; it has a state invariant expressed as an OCL constraint (top right corner of the diagram). The OWL DL representation of the constraints on the states is shown in the following DL expressions.

$$Allocating \equiv Waiting \sqcup GettingTransporters \sqcup MakingAssignment \tag{1}$$

$$Allocating \sqsubseteq \exists isRequestedBy.Shipment \tag{2}$$

$$Complete \sqsubseteq \forall isRequestedBy.1Shipment \tag{3}$$

$$Waiting \sqsubseteq \forall hasAssignment.0Assignment \tag{4}$$

$$GettingTransporters \sqsubseteq \exists hasAssignment.(Assignment \sqcap \geq hasTransporter.1Transporter) \tag{5}$$

$$MakingAssignment \sqsubseteq \exists hasAssignment.(Assignment \sqcap \forall hasShipment.1Shipment \sqcap \forall hasTransporter.1Transporter) \tag{6}$$

The six transitions are also mapped to six OWL classes. The transition from *Waiting* to *GettingTransporters* is triggered by the event *shipRequest*. So the transition class *WaitingToGettingTrans* is defined as:

$$WaitingToGettingTrans \equiv \exists fsm.triggeredBy.\{shipRequest\} \tag{7}$$

Table 4 SysML state machine diagram to OWL mapping rules comparison [5]

State machine component		[7]	[9]	[27]	[6]	Our mapping rule
State	Simple state	Class	Class	Subclass of the class of the object	Individual	Class
	Composite state	Superclass of substate classes	Superclass of substate classes	Superclass of substate classes	Individual	Superclass of substate classes
	Initial state	Class	Class	Class	Individual	Class
	Final state	Class	Class	Class	Individual	Class
	OCL invariants	None	None	OWL object property restrictions	None	OWL object property restrictions
Transition		Class	Class	None	Individual	Class
Event		Class	Class	None	Individual	Class
Action		None	Class	None	Individual	Class
Guard		Class	Class	None	Individual	Class
Guard expressed in OCL invariants		None	None	None	None	OWL object property restrictions

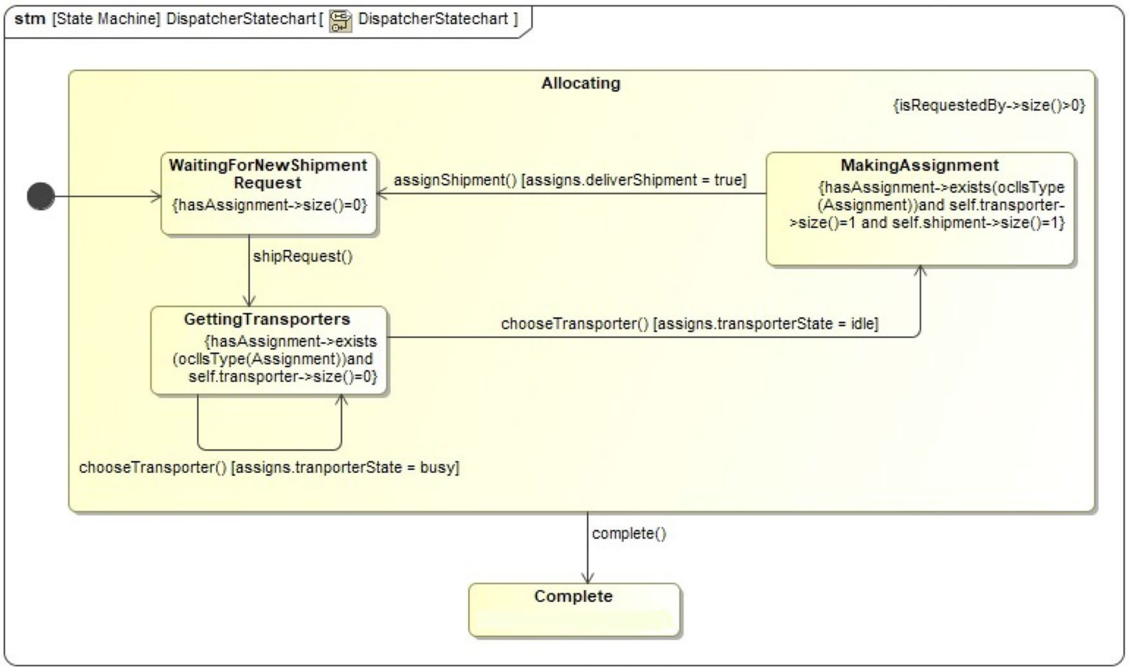

stm [State Machine] DispatcherStatechart [DispatcherStatechart]
Allocating
{isRequestedBy->size()>0}
WaitingForNewShipment Request
{hasAssignment->size()=0}
assignShipment() [assigns.deliverShipment = true]
MakingAssignment
{hasAssignment->exists(oclIsType (Assignment))and self.transporter->size()=1 and self.shipment->size()=1}
shipRequest()
GettingTransporters
{hasAssignment->exists (oclIsType(Assignment))and self.transporter->size()=0}
chooseTransporter() [assigns.transporterState = idle]
chooseTransporter() [assigns.tranporterState = busy]
complete()
Complete

Fig. 5 Dispatcher Statechart [5]

$$
\begin{aligned}
\mathit{WaitingToGettingTrans} \sqsubseteq & \exists \mathit{fsm.hasSource.Waiting} \\
& \sqcap \exists \mathit{fsm.hasTarget.GettingTransporters}
\end{aligned} \tag{8}
$$

GettingTransporters state has two outgoing transitions. After the operation *chooseTransporter* of the dispatcher is invoked, if the returned transporter state is busy, the state stays at *GettingTransporters*. Otherwise, the state transits to *MakingAssignment*. So the two transition classes are defined as:

$$
\begin{aligned}
\mathit{GettingTrans} \equiv & \exists \mathit{fsm.triggeredBy.chooseTransporter} \\
& \sqcap \exists (\mathit{fsm.hasGuard.fsm.EvalCmp.EQ} \\
& \sqcap \exists \mathit{fsm.hasLeftExpr.}\{\mathit{transporterState}\} \\
& \sqcap \exists \mathit{fsm.hasRightExpr.}\{\mathit{busy}\})
\end{aligned} \tag{9}
$$

$$
\begin{aligned}
\mathit{GettingTrans} \sqsubseteq & \exists \mathit{fsm.hasSource.GettingTransporters} \\
& \sqcap \exists \mathit{fsm.hasTarget.GettingTransporters}
\end{aligned} \tag{10}
$$

$$\begin{aligned} &\mathit{GettingTransToMakingAssignment} \\ &\quad \equiv \exists \mathit{fsm.triggeredBy.chooseTransporter} \\ &\quad \sqcap \exists (\mathit{fsm.hasGuard.fsm.EvalCmp.EQ} \\ &\quad \sqcap \exists \mathit{fsm.hasLeftExpr}.\{\mathit{transporterState}\} \\ &\quad \sqcap \exists \mathit{fsm.hasRightExpr}.\{\mathit{busy}\}) \end{aligned} \tag{11}$$

$$\begin{aligned} &\mathit{GettingTransToMakingAssignment} \\ &\quad \sqsubseteq \exists \mathit{fsm.hasSource.GettingTransporters} \\ &\quad \sqcap \exists \mathit{fsm.hasTarget.MakingAssignment} \end{aligned} \tag{12}$$

From the state *MakingAssignment*, after the dispatcher invokes the operation *assignShipment*, if the *deliverShipment* is true, then the state of the dispatcher goes back to *Waiting*.

$$\begin{aligned} \mathit{MakingAssignmentToWaiting} \equiv\ & \exists \mathit{fsm.triggeredBy.assignShipment} \\ & \sqcap \exists (\mathit{fsm.hasGuard.fsm.EvalCmp.EQ} \\ & \sqcap \exists \mathit{fsm.hasLeftExpr}.\{\mathit{deliverShipment}\} \\ & \sqcap \exists \mathit{fsm.hasRightExpr}.\{\mathit{true}\}) \end{aligned} \tag{13}$$

$$\begin{aligned} \mathit{MakingAssignmentToWaiting} \sqsubseteq\ & \exists \mathit{fsm.hasSource.MakingAssignment} \\ & \sqcap \exists \mathit{fsm.hasTarget.Waiting} \end{aligned} \tag{14}$$

OWL Reasoning with State Machine Diagrams

We check the consistency of OCL constraints in a state machine to check the consistency between the state machine and the requirements expressed as OCL constraints. For this purpose, we translate the state machine along with the OCL constraints to OWL (as shown in "A mapping example") and run the reasoner to detect inconsistency in the ontology. To show different types of inconsistencies of constraints, we developed three reasoning tasks.

The first OWL reasoning task is to check if the OCL invariants of all the states are consistent. The state invariants that may cause the object violate the constraints imposed on the state diagrams in the UML superstructure specification of state machine are considered to be inconsistent invariants. For example, for the composite state *Allocating*, the OCL constraint of the state invariant is:

$$\mathit{isRequestedBy}-> \mathit{size}() > 0, \tag{15}$$

which means the dispatcher is in *Allocating* state if it is requested by at least one shipment (Fig. 6). The *Waiting* state is a substate of *Allocating*. The substate should not have a conflicting invariant with the composite state. Adding the following invariant to the *Waiting* state:

$$\mathit{isRequestedBy}-> \mathit{size}() = 0 \tag{16}$$

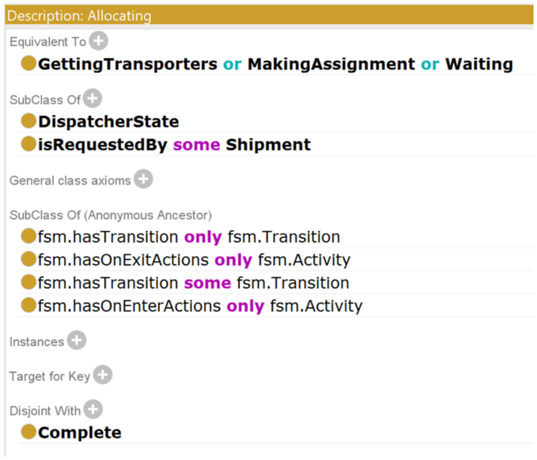

Description: Allocating
Equivalent To
GettingTransporters or MakingAssignment or Waiting
SubClass Of
DispatcherState
isRequestedBy some Shipment
General class axioms
SubClass Of (Anonymous Ancestor)
fsm.hasTransition only fsm.Transition
fsm.hasOnExitActions only fsm.Activity
fsm.hasTransition some fsm.Transition
fsm.hasOnEnterActions only fsm.Activity
Instances
Target for Key
Disjoint With
Complete

Fig. 6 OWL restrictions for Allocating state

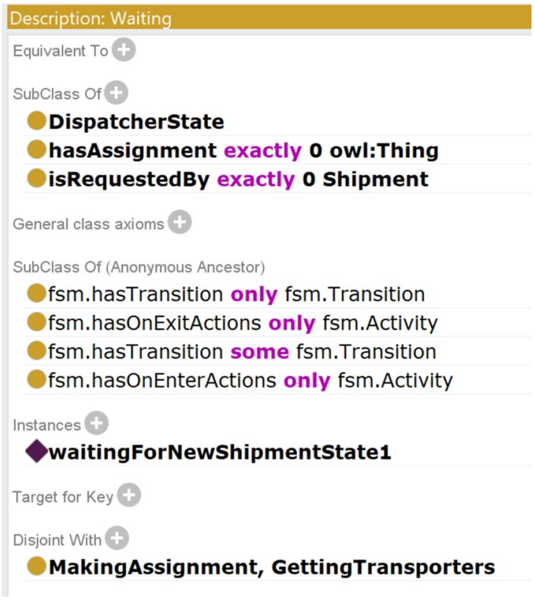
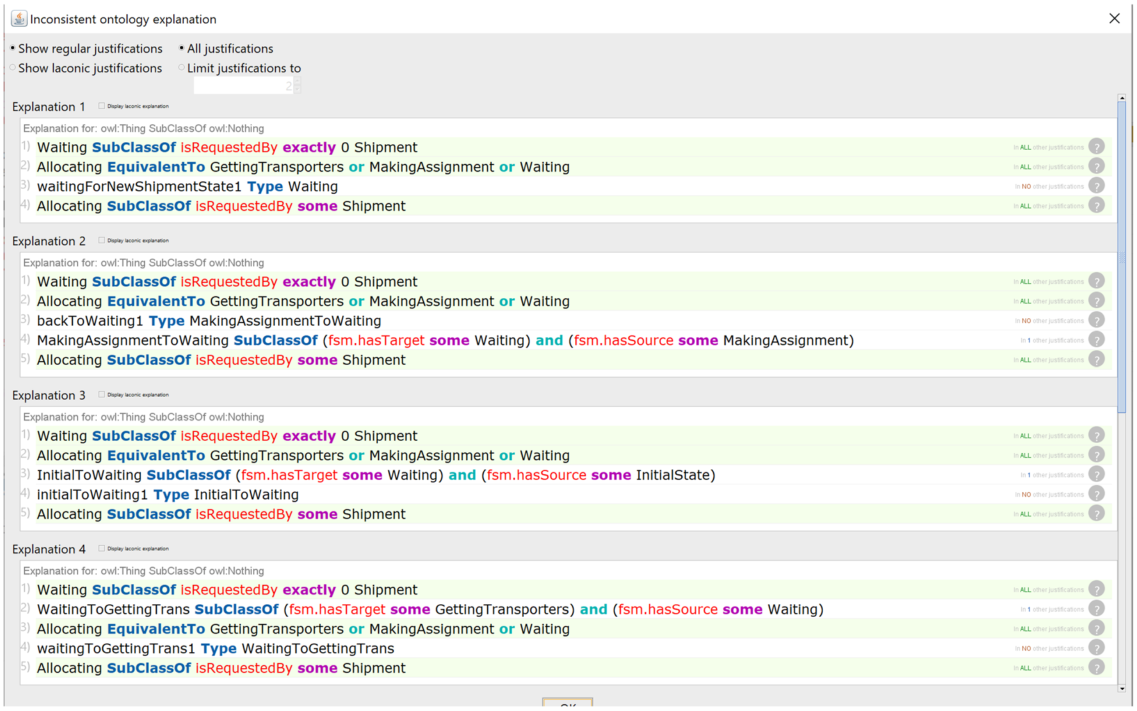

Description: Waiting
Equivalent To
SubClass Of
DispatcherState
hasAssignment exactly 0 owl:Thing
isRequestedBy exactly 0 Shipment
General class axioms
SubClass Of (Anonymous Ancestor)
fsm.hasTransition only fsm.Transition
fsm.hasOnExitActions only fsm.Activity
fsm.hasTransition some fsm.Transition
fsm.hasOnEnterActions only fsm.Activity
Instances
waitingForNewShipmentState1
Target for Key
Disjoint With
MakingAssignment, GettingTransporters

Fig. 7 OWL restrictions for Waiting state

would imply being in the *Waiting* state even if there was no request by any shipment (Fig. 7). Thus, such two OCL constraints are inconsistent, and thus there cannot be a state individual that can satisfy both the constraints of *Allocating* and the constraints of the *Waiting* states.

When we run the Pellet in Protégé, it will identify the semantic inconsistency of the OWL axioms in *Allocating* class and *Waiting* class. The inconsistency explanations are shown in Fig. 8.

The second OWL reasoning task is to check whether only one transition can be taken out of a state, i.e., whether the state machine is deterministic. For a state with more than one possible outgoing transition, e.g., *GettingTransporters*,



```
Inconsistent ontology explanation
• Show regular justifications    • All justifications
○ Show laconic justifications    ○ Limit justifications to  2

Explanation 1  ☐ Display laconic explanation
Explanation for: owl:Thing SubClassOf owl:Nothing
1) Waiting SubClassOf isRequestedBy exactly 0 Shipment                                   In ALL other justifications
2) Allocating EquivalentTo GettingTransporters or MakingAssignment or Waiting           In ALL other justifications
3) waitingForNewShipmentState1 Type Waiting                                             In NO other justifications
4) Allocating SubClassOf isRequestedBy some Shipment                                    In ALL other justifications

Explanation 2  ☐ Display laconic explanation
Explanation for: owl:Thing SubClassOf owl:Nothing
1) Waiting SubClassOf isRequestedBy exactly 0 Shipment                                   In ALL other justifications
2) Allocating EquivalentTo GettingTransporters or MakingAssignment or Waiting           In ALL other justifications
3) backToWaiting1 Type MakingAssignmentToWaiting                                        In NO other justifications
4) MakingAssignmentToWaiting SubClassOf (fsm.hasTarget some Waiting) and (fsm.hasSource some MakingAssignment)   In 1 other justifications
5) Allocating SubClassOf isRequestedBy some Shipment                                    In ALL other justifications

Explanation 3  ☐ Display laconic explanation
Explanation for: owl:Thing SubClassOf owl:Nothing
1) Waiting SubClassOf isRequestedBy exactly 0 Shipment                                   In ALL other justifications
2) Allocating EquivalentTo GettingTransporters or MakingAssignment or Waiting           In ALL other justifications
3) InitialToWaiting SubClassOf (fsm.hasTarget some Waiting) and (fsm.hasSource some InitialState)   In 1 other justifications
4) initialToWaiting1 Type InitialToWaiting                                              In NO other justifications
5) Allocating SubClassOf isRequestedBy some Shipment                                    In ALL other justifications

Explanation 4  ☐ Display laconic explanation
Explanation for: owl:Thing SubClassOf owl:Nothing
1) Waiting SubClassOf isRequestedBy exactly 0 Shipment                                   In ALL other justifications
2) WaitingToGettingTrans SubClassOf (fsm.hasTarget some GettingTransporters) and (fsm.hasSource some Waiting)   In 1 other justifications
3) Allocating EquivalentTo GettingTransporters or MakingAssignment or Waiting           In ALL other justifications
4) waitingToGettingTrans1 Type WaitingToGettingTrans                                    In NO other justifications
5) Allocating SubClassOf isRequestedBy some Shipment                                    In ALL other justifications
```

Fig. 8 Inconsistency explanations of the first reasoning task

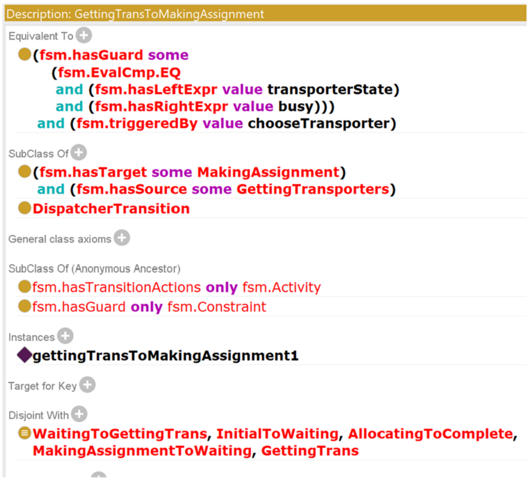
the state that has two outgoing transitions *GettingTrans* and *GettingTransToMakingAssignment*, adding the guard:

$$assigns.transporterState = busy \tag{17}$$

to both transitions would make the choice of a transition not unique. In OWL, two outgoing transitions are represented by two disjoint transition classes and should have disjoint guards. Otherwise, the Pellet will identify semantic inconsistency because an individual cannot belong to both a type and its complement. The inconsistent result is shown in red in Figs. 9 and 10 , and the reasoner explanations of inconsistency are shown in Fig. 11.

The third OWL reasoning task is to check if the state machine contains deadlocks. A deadlock can happen when two or more processes have conflicting resource needs. In this paper, we consider a simple deadlock case: for a state with only one outgoing transition, the guards on this outgoing transition of a state are mutually exclusive. In other words, if the guards on the only outgoing transition of a state cannot be satisfied at the same time, the state machine for the object will be stuck in this state. For example, for the transition from *MakingAssignment* to *Waiting*, if we add the guards as follows:

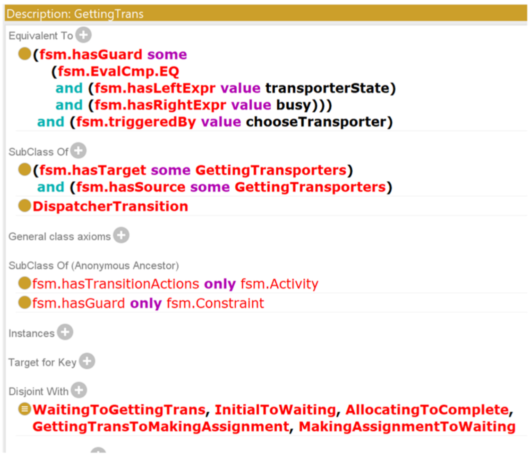
$$\begin{aligned} &assigns.deliverShipment = true, \\ &assigns.deliverShipment = false. \end{aligned} \tag{18}$$



```
Description: GettingTransToMakingAssignment
Equivalent To
  ● (fsm.hasGuard some
       (fsm.EvalCmp.EQ
        and (fsm.hasLeftExpr value transporterState)
        and (fsm.hasRightExpr value busy)))
     and (fsm.triggeredBy value chooseTransporter)

SubClass Of
  ● (fsm.hasTarget some MakingAssignment)
     and (fsm.hasSource some GettingTransporters)
  ● DispatcherTransition

General class axioms

SubClass Of (Anonymous Ancestor)
  ● fsm.hasTransitionActions only fsm.Activity
  ● fsm.hasGuard only fsm.Constraint

Instances
  ◆ gettingTransToMakingAssignment1

Target for Key

Disjoint With
  ● WaitingToGettingTrans, InitialToWaiting, AllocatingToComplete,
    MakingAssignmentToWaiting, GettingTrans
```

Fig. 9 OWL reasoning result in *GettingTransToMakingAssignment* transition

Once the state machine of the dispatcher object gets into *MakingAssignment*, it will not be able to get out of this state because the *deliverShipment* attribute cannot be both *true* and *false* at the same time. When we run the Pellet, it will identify the semantic inconsistency of the OWL axioms in the *MakingAssignmentToWaiting* class that represents the transition, as shown in red in Fig. 12. The reasoner explanations of inconsistency are shown in Fig. 13.



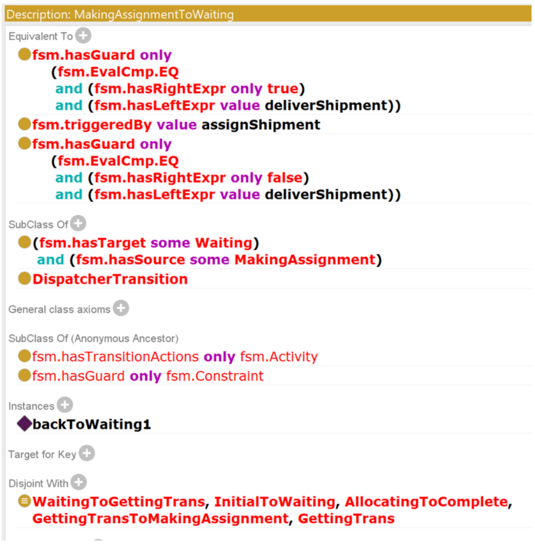
```
Description: GettingTrans
Equivalent To
  (fsm.hasGuard some
      (fsm.EvalCmp.EQ
       and (fsm.hasLeftExpr value transporterState)
       and (fsm.hasRightExpr value busy)))
    and (fsm.triggeredBy value chooseTransporter)

SubClass Of
  (fsm.hasTarget some GettingTransporters)
    and (fsm.hasSource some GettingTransporters)
  DispatcherTransition

General class axioms

SubClass Of (Anonymous Ancestor)
  fsm.hasTransitionActions only fsm.Activity
  fsm.hasGuard only fsm.Constraint

Instances

Target for Key

Disjoint With
  WaitingToGettingTrans, InitialToWaiting, AllocatingToComplete,
  GettingTransToMakingAssignment, MakingAssignmentToWaiting
```

Fig. 10 OWL reasoning result in *GettingTrans* transition

Conclusions

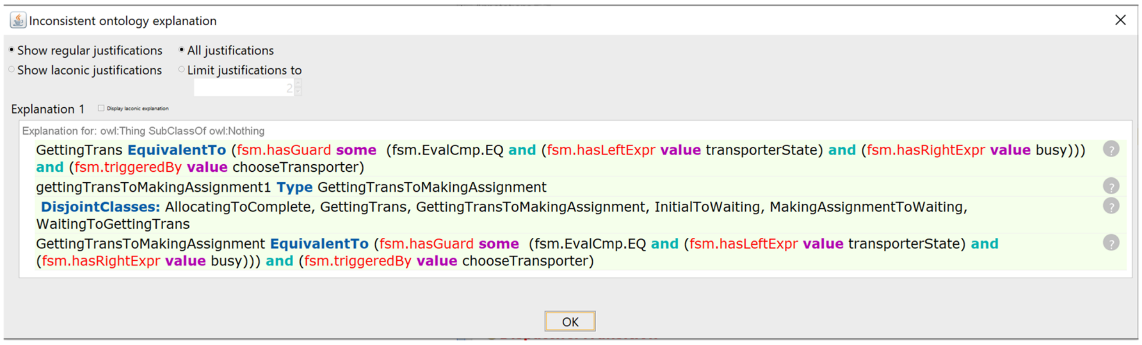
In this paper, we propose an ontology-based method to reason about the correctness of SysML models that include OCL constraints. Specifically, we map the SysML models with OCL constraints to OWL and check the consistency of the corresponding ontology by running an OWL inference engine. Although a number of mapping rules from UML class diagrams to OWL have been reported in the subject literature and accepted by the community, still, there is a lack of widely accepted mapping rules from UML behavior diagrams to OWL. In this paper, we propose a set of rules for mapping components of SysML state machine diagrams along with OCL constraints to OWL DL. We also demonstrate three examples of reasoning tasks in which OWL axioms are used for consistency checking of state machine diagrams. We recognize that our current approach is a combination of automated and manual steps, in part due to the limitations of the built-in concept modeler. In the future, we plan to develop fully automatic procedures for mapping SysML/OCL models to OWL DL.

Our work is aimed at providing the software developer with some reasonable assurances about the correctness of the model in the system modeling stage of software development. This is the first step of requirement change management. In the future, we plan to extend the scope of our approach to a more complete and automatic verification of SysML state machine specifications with OCL constraints as well as both theoretical analysis and experimental validation of the correctness of the mapping rules.



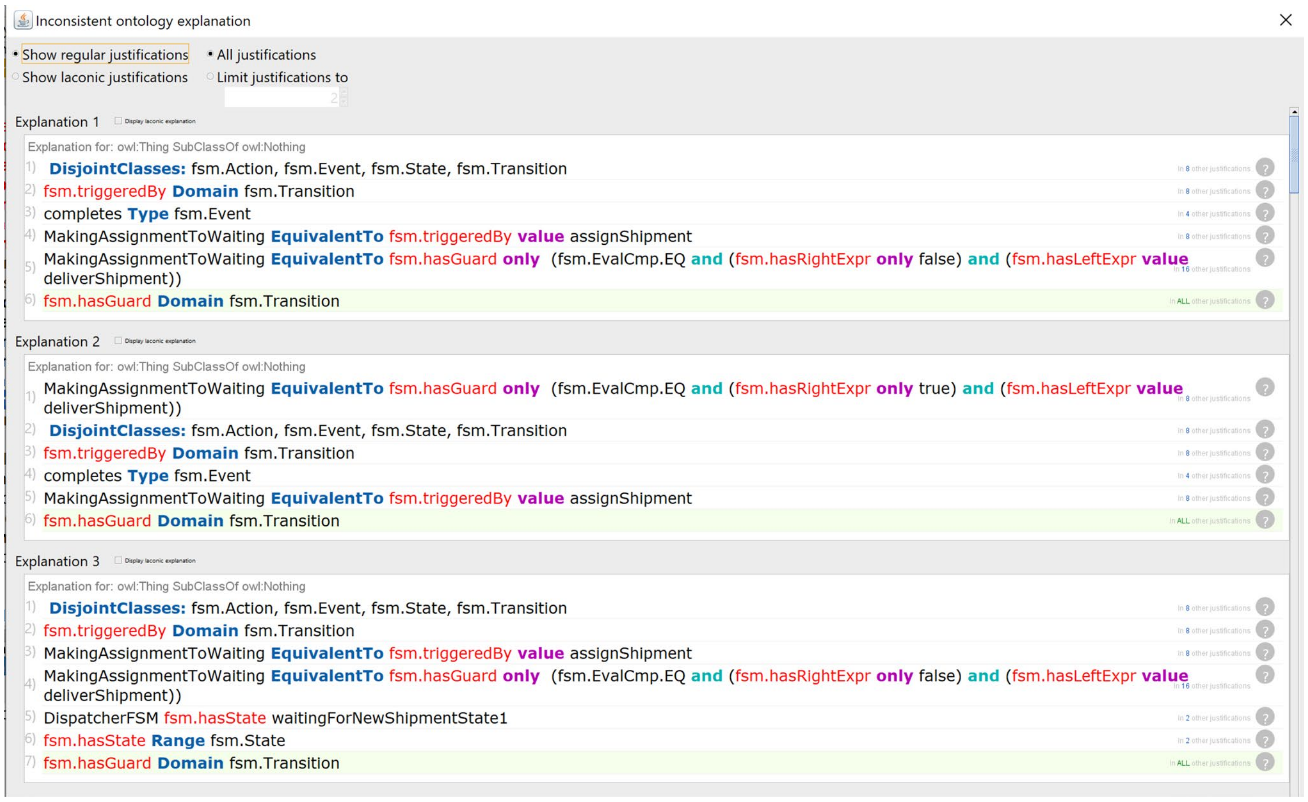
```
Description: MakingAssignmentToWaiting
Equivalent To
  fsm.hasGuard only
      (fsm.EvalCmp.EQ
       and (fsm.hasRightExpr only true)
       and (fsm.hasLeftExpr value deliverShipment))
  fsm.triggeredBy value assignShipment
  fsm.hasGuard only
      (fsm.EvalCmp.EQ
       and (fsm.hasRightExpr only false)
       and (fsm.hasLeftExpr value deliverShipment))

SubClass Of
  (fsm.hasTarget some Waiting)
    and (fsm.hasSource some MakingAssignment)
  DispatcherTransition

General class axioms

SubClass Of (Anonymous Ancestor)
  fsm.hasTransitionActions only fsm.Activity
  fsm.hasGuard only fsm.Constraint

Instances
  backToWaiting1

Target for Key

Disjoint With
  WaitingToGettingTrans, InitialToWaiting, AllocatingToComplete,
  GettingTransToMakingAssignment, GettingTrans
```

Fig. 12 OWL reasoning result in *MakingAssignmentToWaiting* transition



```
Inconsistent ontology explanation
• Show regular justifications     • All justifications
○ Show laconic justifications     ○ Limit justifications to  2

Explanation 1   ☐ Display laconic explanation
Explanation for: owl:Thing SubClassOf owl:Nothing
  GettingTrans EquivalentTo (fsm.hasGuard some  (fsm.EvalCmp.EQ and (fsm.hasLeftExpr value transporterState) and (fsm.hasRightExpr value busy)))
  and (fsm.triggeredBy value chooseTransporter)
  gettingTransToMakingAssignment1 Type GettingTransToMakingAssignment
   DisjointClasses: AllocatingToComplete, GettingTrans, GettingTransToMakingAssignment, InitialToWaiting, MakingAssignmentToWaiting,
  WaitingToGettingTrans
  GettingTransToMakingAssignment EquivalentTo (fsm.hasGuard some  (fsm.EvalCmp.EQ and (fsm.hasLeftExpr value transporterState) and
  (fsm.hasRightExpr value busy))) and (fsm.triggeredBy value chooseTransporter)

OK
```

Fig. 11 Inconsistency explanation of the second reasoning task



```
Inconsistent ontology explanation
• Show regular justifications    • All justifications
○ Show laconic justifications    ○ Limit justifications to  2

Explanation 1  ☐ Display laconic explanation
Explanation for: owl:Thing SubClassOf owl:Nothing
1) DisjointClasses: fsm.Action, fsm.Event, fsm.State, fsm.Transition                In 8 other justifications
2) fsm.triggeredBy Domain fsm.Transition                                             In 8 other justifications
3) completes Type fsm.Event                                                          In 4 other justifications
4) MakingAssignmentToWaiting EquivalentTo fsm.triggeredBy value assignShipment       In 8 other justifications
5) MakingAssignmentToWaiting EquivalentTo fsm.hasGuard only (fsm.EvalCmp.EQ and (fsm.hasRightExpr only false) and (fsm.hasLeftExpr value
   deliverShipment))                                                                 In 16 other justifications
6) fsm.hasGuard Domain fsm.Transition                                                In ALL other justifications

Explanation 2  ☐ Display laconic explanation
Explanation for: owl:Thing SubClassOf owl:Nothing
1) MakingAssignmentToWaiting EquivalentTo fsm.hasGuard only (fsm.EvalCmp.EQ and (fsm.hasRightExpr only true) and (fsm.hasLeftExpr value
   deliverShipment))                                                                 In 8 other justifications
2) DisjointClasses: fsm.Action, fsm.Event, fsm.State, fsm.Transition                In 8 other justifications
3) fsm.triggeredBy Domain fsm.Transition                                             In 8 other justifications
4) completes Type fsm.Event                                                          In 4 other justifications
5) MakingAssignmentToWaiting EquivalentTo fsm.triggeredBy value assignShipment       In 8 other justifications
6) fsm.hasGuard Domain fsm.Transition                                                In ALL other justifications

Explanation 3  ☐ Display laconic explanation
Explanation for: owl:Thing SubClassOf owl:Nothing
1) DisjointClasses: fsm.Action, fsm.Event, fsm.State, fsm.Transition                In 8 other justifications
2) fsm.triggeredBy Domain fsm.Transition                                             In 8 other justifications
3) MakingAssignmentToWaiting EquivalentTo fsm.triggeredBy value assignShipment       In 8 other justifications
4) MakingAssignmentToWaiting EquivalentTo fsm.hasGuard only (fsm.EvalCmp.EQ and (fsm.hasRightExpr only false) and (fsm.hasLeftExpr value
   deliverShipment))                                                                 In 16 other justifications
5) DispatcherFSM fsm.hasState waitingForNewShipmentState1                            In 2 other justifications
6) fsm.hasState Range fsm.State                                                      In 2 other justifications
7) fsm.hasGuard Domain fsm.Transition                                                In ALL other justifications
```

Fig. 13 Inconsistency explanations of the third reasoning task

Acknowledgements This research was supported by a grant from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the US Government. The authors wish to acknowledge contributions from the whole IDAS team, and in particular James Hove and Everett Pompeii of Sierra Nevada Corporation.

Conflict of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. DARPA.: Intent-defined adaptive software (IDAS). https://www.darpa.mil/program/intent-defined-adaptive-software.
2. Alsanad AA, Chikh A, Mirza A. A domain ontology for software requirements change management in global software development environment. IEEE Access. 2019;7:49352–61, 100223.
3. Eito-Brun R, Gómez-Berbís JM, de Amescua Seco A. Knowledge tools to organise software engineering data: development and validation of an ontology based on ECSS standard. Adv Space Res. 2022
4. W3C.: OWL web ontology language: overview. https://www.w3.org/TR/owl-features/.
5. Lu S, Tazin A, Chen Y, Kokar MM, Smith J. Ontology-based detection of inconsistencies in UML/OCL models. In: international conference on model-driven engineering and software development. 2022.
6. Belgueliel Y, Bourahla M, Brik M. Towards an ontology for UML state machines. Lect Notes Softw Eng. 2014;2(1):116.
7. Gröner G, Staab S. Specialization and validation of statecharts in OWL. In: international conference on knowledge engineering and knowledge management. Springer; 2010:360–370.
8. Mkhinini MM, Labbani-Narsis O, Nicolle C. Combining UML and ontology: an exploratory survey. Comput Sci Rev. 2020;35:100223.
9. Van Der Straeten R, Van R, Straeten D. Using description logic in object-oriented software development. 2002.
10. Ahmad MA, Nadeem A. Consistency checking of UML models using description logics: a critical review. In: 2010 6th international conference on emerging technologies (ICET). IEEE. 2010:310–315.
11. Elaasar M, Briand L. An overview of UML consistency management. Carleton University, Canada, Technical Report SCE-04-18. 2004.

12. Usman M, Nadeem A, Kim Th, Cho Es A. A survey of consistency checking techniques for UML models. Adv Softw Eng Appl IEEE. 2008;2008:57–62.
13. Baclawski K, Kokar MK, Kogut PA, Hart L, Smith J, Holmes WS, et al. Extending UML to support ontology engineering for the semantic web. In: international conference on the unified modeling language. Springer. 2001:342–360.
14. Anastasakis G Bordbar, Ray. On challenges of model transformation from UML to Alloy. Software & Systems Modeling. 2010;9(1):69–86.
15. Przigoda W Soeken, Drechsler. Verifying the structure and behavior in UML/OCL models using satisfiability solvers. IET Cyber-Phys Syst: Theor Appl. 2016;1:49–59.
16. Dwivedi AK, Rath SK. Transformation of alloy notation into a semantic notation. ACM SIGSOFT Softw Eng Notes. 2018;43(1):1–6.
17. Gogolla M, Bttner F, Kuhlmann M. System modeling with USE (UML-based Specification Environment). Genie Logiciel. 2008;85:57–8.
18. Gogolla M, Büttner F, Richters M. USE: A UML-based specification environment for validating UML and OCL. Sci Comput Program. 2007;69(1–3):27–34.
19. Latif S, Rehman A, Zafar NA. Modeling of Sewerage System Linking UML, Automata and TLA+. In: 2018 international conference on computing, electronic and electrical engineering (ICE Cube); 2018:1–6.
20. Latif S, Rehman A, Zafar NA. NFA based formal modeling of smart parking system using TLA+. In: 2019 international conference on information science and communication technology (ICISCT); 2019:1–6.
21. Lamport L. The TLA+ Toolbox. https://lamport.azurewebsites.net/tla/toolbox.html.
22. Rull G, Farré C, Queralt A, Teniente E, Urpí T. AuRUS: explaining the validation of UML/OCL conceptual schemas. Softw & Syst Model. 2015;14:953–80.
23. Filipovikj P. Automated approaches for formal verification of embedded systems artifacts. Mälardalen University. 2019.
24. Mahmud N, Seceleanu C, Ljungkrantz O. ReSA tool: structured requirements specification and SAT-based consistency-checking. In: proceedings of the federated conference on computer science and information systems. IEEE. 2016:1737 – 1746.
25. Parreiras FS, Staab S, Winter A. TwoUse: integrating UML models and OWL ontologies. University of Koblenz-Landau. 2007.
26. Berardi D, Calvanese D, De Giacomo G. Reasoning on UML class diagrams. Artif intell. 2005;168(1–2):70–118.
27. Khan AH, Porres I. Consistency of UML class, object and statechart diagrams using ontology reasoners. J Vis Lang & Comput. 2015;26:42–65.
28. Khan AH, Rauf I, Porres I. Consistency of UML class and statechart diagrams with state invariants. In: MODELSWARD. 2013:14–24.
29. Gruber TR. A translation approach to portable ontology specifications. Knowl Acquis. 1993;5(2):199–220.
30. Stanford University.: Prot*égé*. http://protege.stanford.edu/.
31. NoMagic.: Cameo concept modeler 2021x plugin documentation. https://docs.nomagic.com/.
32. Fu C, Yang D, Zhang X, Hu H. An approach to translating OCL invariants into OWL 2 DL axioms for checking inconsistency. Autom Softw Eng. 2017;24(2):295–339.