JOT
JOURNAL OF
OBJECT TECHNOLOGY

[KRW22] O. Kautz, B. Rumpe, L. Wachtmeister:
Semantic Differencing of Use Case Diagrams.
In: Journal of Object Technology, Volume 21, pp. 3:1-14, AITO - Association Internationale pour les Technologies Objets, Juli 2022.
www.se-rwth.de/publications/

Semantic Differencing of Use Case Diagrams

Oliver Kautz[*], **Bernhard Rumpe**[†], **and Louis Wachtmeister**[†]
[*]CBC Cologne Broadcasting Center GmbH, Germany
[†]RWTH Aachen University, Germany

ABSTRACT Use case diagrams (UCDs) are widely used for describing how different users use the functionalities of a system to achieve their goals. As today's software systems offer an ever-increasing number of functionalities, it becomes more and more difficult for software engineers to decide how the behaviors of different systems, various versions, or system variants differ from the stakeholders' points of views. Notwithstanding the importance of UCDs in addressing this challenge, previous work neglected the development of a definition of an UCD language with a precise semantics that goes beyond a high-level representation of stick figures with bubbles. Since these representations neither have a well-defined syntax nor a formal semantics, clearly distinguishing the meanings of two diagrams becomes difficult. To tackle this challenge, this paper defines a formal syntax and semantics for a UCD variant and presents a semantic differencing operator that we evaluated experimentally with a set of example diagrams.

KEYWORDS Use Case, Use Case Diagram, Semantic Differencing, Semantics, Analysis, Evolution, Difference, Model, UML

1. Introduction

Use case diagrams (UCDs) describe the functionalities of a system in terms of how its various users achieve their goals by using the system (Friedenthal et al. 2014). Because of their simplicity and clarity, UCDs are not only used in the development of complex software systems, but also for system development in general.

As the complexity of software systems continues to increase (France & Rumpe 2007; Pretschner et al. 2007; Hölldobler et al. 2019), it is a growing challenge for software engineers to maintain an overview of the functionality of the software systems under development. This also means that deciding to what extent two software systems differ from the stakeholders points of views is becoming increasingly complicated. This is especially important when comparing two different versions or two different variants of the same system.

Apart from comparing variants of the same systems, it is also essential to draw conclusions about the compatibility and suitability of system specifications. Imagine a vendor using UCDs to specify the functionality of an offered system and an acquirer using UCDs to specify the properties of a required system. Both could use a semantic differencing approach to establish beyond doubt what the differences between the offered and required systems are and use the result as the basis for their further negotiations.

Another challenge that arises in the use case context is that small changes to a high-level description may have large effects on the overall system behavior. Knowing the semantic (Maoz et al. 2011c,d, 2012; Maoz & Ringert 2015, 2018; Kautz & Rumpe 2018a; Kautz 2021) difference between two UCDs supports software engineers in understanding the differences in the functionalities of the systems. It enables them to identify the use case and actor combinations that are possible in the one version but not in the other version.

Previous works neglected providing a well-defined syntax and precisely defined semantics that is required for semantic UCD differencing. For that reason, comparing use case diagrams is often still a syntactic comparison of stick figures and bubbles. Especially, when critical systems are developed with UCDs as a starting point, a precisely defined semantics for UCDs is crucial to guarantee a successful development and safe operation of the system under development. Without a precisely

JOT reference format:
Oliver Kautz, Bernhard Rumpe, and Louis Wachtmeister. *Semantic Differencing of Use Case Diagrams.* Journal of Object Technology. Vol. 21, No. 3, 2022.
http://dx.doi.org/10.5381/jot.2022.21.3.a5

defined semantics, the meaning of UCDs can be misinterpreted or differently interpreted by different developers, which can ultimately result in crucial errors in the developed product. Further, a precisely defined semantics enables the development of a semantic differencing operator. Such an operator supports developers in comparing different (versions of) UCDs and reveals the differences in terms of the differences in the meanings of the UCDs (Maoz et al. 2011c, 2012; Kautz & Rumpe 2018a; Kautz 2021).

To tackle the challenges described above, the contributions of this paper are:

- A mathematically precise definition of a UCD modeling language similar to (Harel & Rumpe 2004; Maoz et al. 2011c, 2012; Maoz & Ringert 2018; Kautz & Rumpe 2018a; Kautz 2021) including (1) a reduced abstract syntax, (2) a semantic domain based on scenarios relating use cases to actors, and (3) a precisely defined semantics mapping each UCD to the set of scenarios that it describes.
- Algorithms to compute scenarios contained in the semantics of UCDs.
- A semantic differencing operator that takes two UCDs as input and outputs all scenarios contained in the semantics of the one UCD that are not contained in the semantics of the other UCD.
- An experimental evaluation revealing that the performance of the operator suffices for several examples.

In the following, Section 2 motivates semantic differencing of UCDs with examples. Section 3 introduces the abstract syntax and formal semantics of UCDs. Then, Section 4 presents the semantic differencing operator. Afterwards, Section 5 presents the results of an experimental evaluation of the operator. Section 6 highlights related works and Section 7 discusses observations and mentions possible future work directions. Finally, Section 8 concludes.

2. Motivating Examples

This section motivates the semantic differencing operator for UCDs introduced in this paper with four example UCDs.

2.1. Car Charging Station Example

Figure 1 depicts two UCDs modeling possible scenarios in the context of charging an electrical car at a charging station.

In `CarCharging1`, there are drivers, cars, charging stations, and fast charging stations. Drivers can request charging processes at charging stations. Drivers can pay for the provided services at charging stations. A charging station can calculate the required charging quantity of a car, check whether the charging station is correctly connected to a car, and charge a car. Fast charging stations are special charging stations. They provide all functionalities that are also provided by charging stations and additionally provide a functionality for the fast charging of cars.

During the development of the system, managers decide which specific payment methods can be used. Developers also realize that the request of charging a car always leads to the calculation of the charging quantity and that calculating the charging quantity always includes checking the cable connection. A modeler thus changes the UCD `CarCharging1` to its successor version `CarCharging2`.

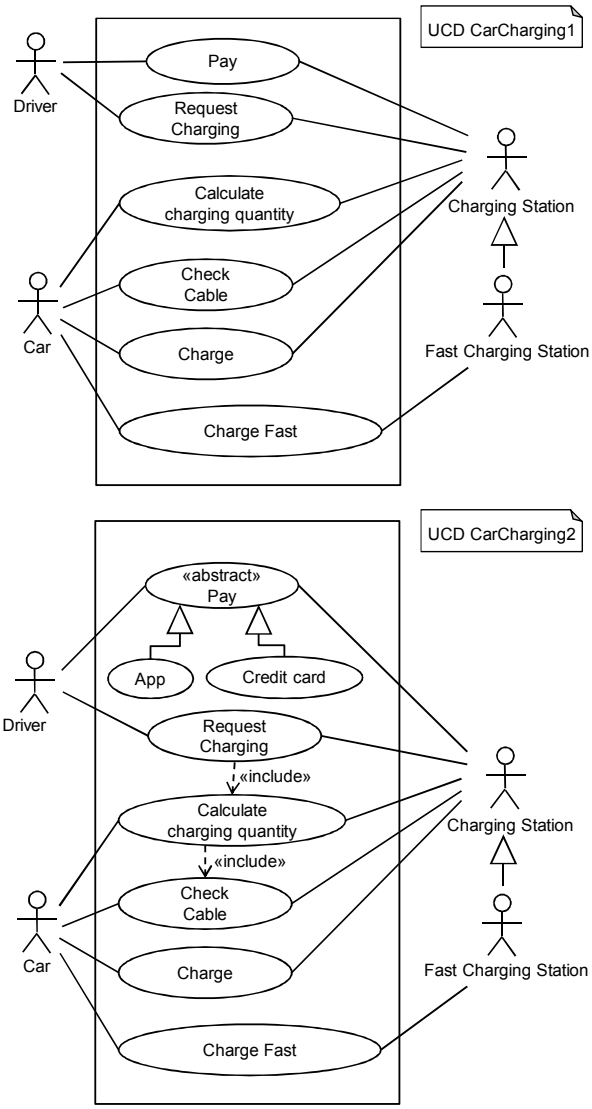

Figure 1 Two UCDs modeling usage scenarios for electric charging stations.

A developer wants to understand the semantic differences between the two versions. Therefore, she uses our semantic differencing operator. The semantic differencing operator reveals that there are possible scenarios in the new version `CarCharging2` that are not possible in the old version version `CarCharging1` and vice versa. Hence, new possible scenarios have been added and some of the previously possible scenarios have been removed. Among others, the developer gets presented the scenarios depicted in Figure 2.

The scenarios `diff1` and `diff2` are possible in the UCD `CarCharging1` and not possible in the UCD `CarCharging2`. From the first scenario, the developer understands that the use case `Pay` cannot be directly executed in the new version of the UCD anymore. The scenario is not possible in the

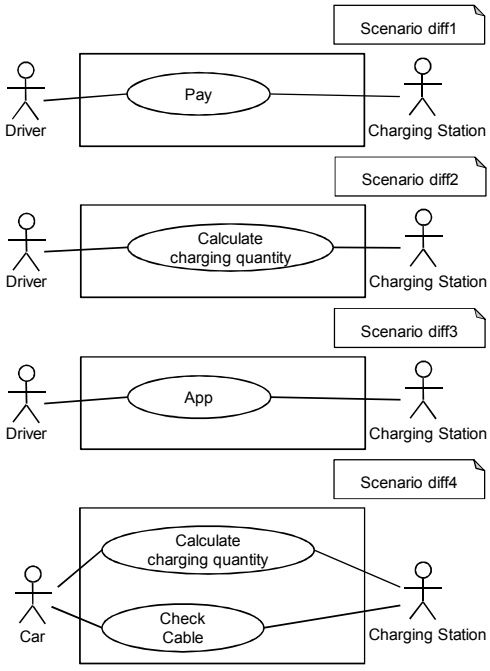

Figure 2 Scenarios contained in the semantic differences between the UCDs `CharCharging1` and `CarCharging2`.

UCD `CarCharging2` because the use case `Pay` is abstract in `CarCharging2`. The second scenario reveals that the charging quantity can be calculated without performing another use case in `CarCharging1`, which is not possible in `CarCharging2` anymore. The scenario is not possible in the UCD `CarCharging2` because the execution of the use case `Calculate charging quantity` in `CarCharging2` necessarily includes the execution of the use case `Check Cable`.

The scenarios `diff3` and `diff4` are possible in the UCD `CarCharging2` and not possible in the UCD `CharCharging1`. From `diff3`, the developer understands that payments can be executed via the app in the new version, which has not been possible before. The scenario is not possible in the UCD `CharCharging1` because the use case `App` does not exist in `CharCharging1`. From the last scenario `diff4`, the developer understands that the calculation of the charging quantity includes the checking of the cable in the new version, which is not the case in the old version. The scenario is not possible in the UCD `CharCharging1` because the execution of the use case `Calculate charging quantity` never implies the execution of the use case `Check Cable` and vice versa.

2.2. SwimmyFish Game Example

Figure 3 depicts two UCDs modeling possible scenarios in the context of a game called Swimmy Fish.

In `SwimmyFish1`, players can play the game, pay to become premium users, and change their profile pictures. The payment process always includes checking whether the user is already a premium user. Profile pictures can only be changed by premium players. It can be paid by credit card or by bank transfer. Additional payment methods, which are not further specified

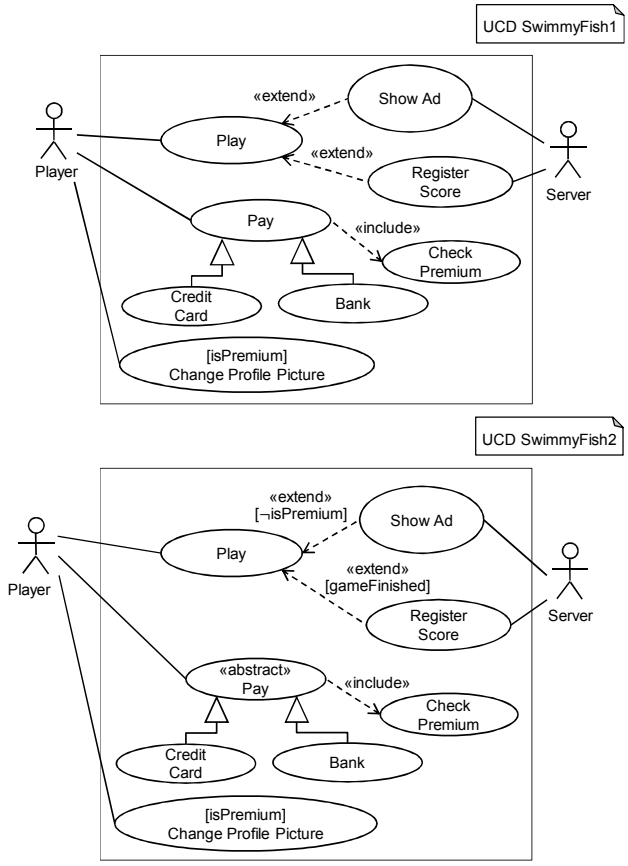

Figure 3 Two UCDs modeling usage scenarios of a game called SwimmyFish.

yet, should also be possible. Servers are involved in showing advertisements and registering the score of a game. When a player plays the game, she may be presented with advertisements and the achieved score may be registered to the server.

During the development, additional information about the requirements for the game become available. Thus, a developer refines the UCD `SwimmyFish1` with additional information. The resulting UCD is `SwimmyFish2`. Paying by credit card and by bank transfer should be the only payment methods. Thus, the developer marked the use case `Pay` as abstract. Further, advertisements should only be presented to non-premium users. To achieve this, the developer adds a guard to the extend relation between `Play` and `Show Ad`. Similarly, scores should only be registered if the player finished playing the game. Therefore, the developer adds a guard to the extend relation between `Play` and `Register Score`.

Another developer, who has not performed the changes, is interested in the differences between the UCD versions. She thus uses our semantic differencing operator for understanding the changes in the meanings of the UCDs. The semantic differencing operator reveals that the UCD `SwimmyFish2` is indeed a refinement of `SwimmyFish1`: Every scenario of `SwimmyFish2` is also a scenario of `SwimmyFish1`. Thus, the semantic difference from `SwimmyFish2` to `SwimmyFish1` is empty. On the other hand, there are scenarios possible in `SwimmyFish1` that are not

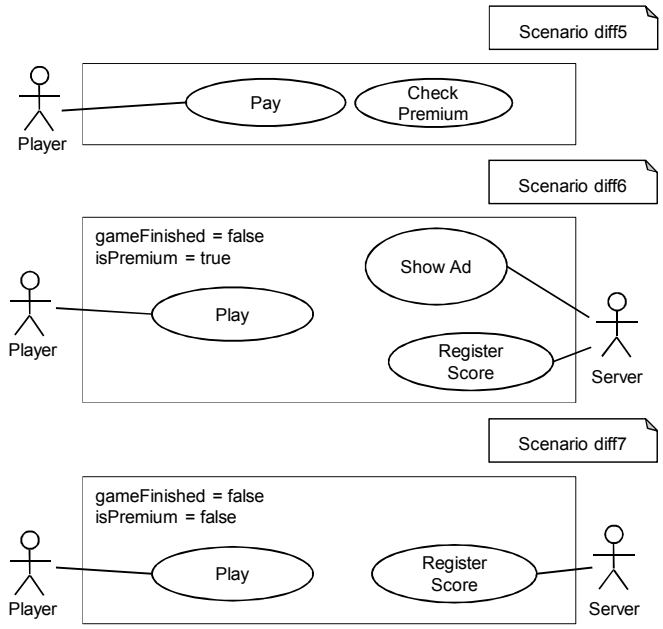

Figure 4 Scenarios contained in the semantic differences between the UCDs `SwimmyFish1` and `SwimmyFish2`.

possible in `SwimmyFish2`. These scenarios have been removed during the evolution from `SwimmyFish1` to `SwimmyFish2`.

Inter alia, the semantic differencing operator outputs the three scenarios depicted in Figure 4, which are possible in `SwimmyFish1` and not in `SwimmyFish2`. From the scenario `diff5`, the developer understands that the use case `Pay` cannot be executed anymore. The scenario is not possible in the UCD `SwimmyFish2` because the use case `Pay` is abstract in `SwimmyFish2`. From the scenario `diff6`, the developer understands that advertisements should not be shown and scores should not be registered when playing the game if the game is not finished and the user is not a premium user. The scenario is not possible in the UCD `SwimmyFish2` because the use cases `Play`, `Show Ad`, and `Register Score` can only be executed in the same execution of `SwimmyFish2` if `isPremium` equals `false` and `gameFinished` equals `true`. From the scenario `diff7`, the developer understands that scores should not be registered when playing the game and the game is not finished and the user is not a premium user. The scenario is not possible in the UCD `SwimmyFish2` because the use cases `Play` and `Register Score` can only be executed in the same execution of `SwimmyFish2` if `gameFinished` equals `true`.

3. Use Case Diagrams

This section introduces an abstract syntax and a semantics for UCDs. a UCD models the relation between actors and use cases, depending on the assignment state of variables. The assignments of variables influence the possible scenarios via preconditions of use cases and conditions used in the extend relation of a UCD. For notational simplicity, we assume that all variables are of type Boolean. A straight-forward generalization to variables of arbitrary finite types is possible.

3.1. Actors, Use Cases, Variables, Boolean Expressions

In the following, let $\mathcal{V}$ be an infinite set of variables, $\mathcal{U}$ be an infinite set of use cases, and $\mathcal{A}$ be an infinite set of actors where $\mathcal{U} \cap \mathcal{A} = \emptyset$. We denote by $\mathbb{B} = \{t, f\}$ the set of Boolean values. A variable assignment is a function $val : \mathcal{V} \to \mathbb{B}$. It maps each variable to a Boolean value. We denote by $\mathcal{V}^{\to}$ the set of all variable assignments. We denote by $Expr$ the set of all well-formed (finite) Boolean expressions over the variables contained in $\mathcal{V}$. The function $eval : Expr \times V^{\to} \to \mathbb{B}$ maps each expression $e \in Expr$ to its truth value $eval(e, v)$ under the variable assignment $v \in V^{\to}$ as usual.

3.2. Use Case Diagram Syntax

The following defines a reduced abstract syntax for UCDs.

Definition 1. *A use case diagram is a tuple* $d = (U, A, Abs, R, G_U, G_A, E, G, Con)$ *where*

- $U \subseteq \mathcal{U}$ *is a finite set of use cases,*
- $A \subseteq \mathcal{A}$ *is a finite set of actors,*
- $Abs \subseteq U \cup A$ *is a set of abstract use cases and actors,*
- $R \subseteq U \times A$ *are associations between use cases and actors,*
- $G_U \subseteq U \times U$ *is a transitive, reflexive generalization relation over the set of use cases* U*,*
- $G_A \subseteq A \times A$ *is a transitive, reflexive generalization relation over the set of actors* A*,*
- $E \subseteq U \times U$ *is an extend relation between use cases,*
- $G \subseteq U \times U$ *is a guarded extend relation between use cases, and*
- $Con : G \cup U \to Expr$ *maps each guarded extend to its guard and each use case to its precondition.*

The set U contains the set of all use cases modeled in the UCD. Similarly, the set A contains all modeled actors and the set Abs contains all abstract use cases and actors. The set R contains the associations between use cases and actors. The set G_U is the use case generalization relation where $(u, v) \in G_U$ represents that the use case u is a specialization of the use case v, respectively that the use case v is a generalization of the use case u. Likewise, the set G_A is the actor generalization relation where $(a, b) \in G_A$ represents that the actor a specializes the actor b, respectively that the actor b generalizes the actor a. The set E is the extend relation between use cases where $(u, v) \in E$ represents that the use case u extends the use case v. The extend relation modeled by E is unguarded. In contrast, the set G is the guarded extend relation between use cases where $(u, v) \in G$ represents the the use case u extends the use case v. The function Con maps each guarded extend $e \in G$ to its guard $Con(e)$, which is a Boolean expression. The syntax definition abstracts from include associations as each include association (v, u) can easily represented by a guarded extend $e = (u, v) \in G$ with $Con(e) = t$, *i.e.*, the guard is always satisfied. Thus, whenever v is executed, u must also be executed. The function Con also maps each use case $u \in U$ to its precondition $Con(u)$.

3.3. Reduced Abstract Syntax vs. UML Abstract Syntax

Figure 5 depicts a class diagram defining the abstract syntax of a UCD language. The diagram is strongly inspired by the UML

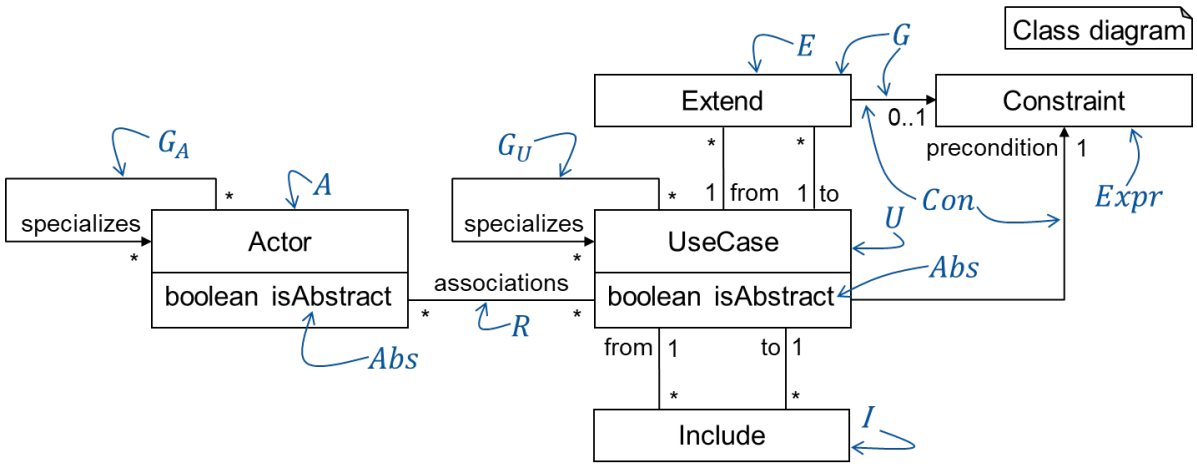

Figure 5 Class diagram defining the abstract syntax of a UCD language.

definition of the abstract syntax of UCDs ("OMG Unified Modeling Language (OMG UML)" 2017). It includes the entities of the UML abstract syntax definition that correspond to elements contained in the reduced abstract syntax (`Actor`, `UseCase`, `Extend`, `Include`, `Constraint`). Figure 5 indicates the relation between the reduced abstract syntax defined in Section 3.2 and the abstract syntax defined by the class diagram. For example, the class `Actor` corresponds to the set of actors A and the association `associations` corresponds to the set of associations R. The class diagram does not contain any entity of the UML abstract syntax definition that belongs to the common structure of UML diagrams (*e.g.*,`BehavioredClassifier`, `Classifier`, `DirectedRelationship`). The class diagram contains additional associations for elements that are not explicitly modeled in the UML abstract syntax definition or that do not exist in the UML abstract syntax definition (*e.g.*,`specializes`, `associations`, `precondition`).

In the UML, UCDs can have extension points and system boundaries. Further, associations between actors and use cases can have cardinalities on the association ends. The reduced abstract syntax defined in Section 3.2 does not support these elements. Section 7 discusses the addition of these syntactic elements into the syntax defined in this paper and the implications of the addition on the definition of the semantic mapping.

3.4. Semantic Domain

The semantics of a UCD is defined as a set of scenarios.

Definition 2. *A scenario is a tuple* (val, Use, Rel) *where*

- $val \in \mathcal{V}^{\rightarrow}$ *is a variable assignment,*
- $Use \subseteq \mathcal{U}$ *is a finite set of use cases, and*
- $Rel \subseteq Use \times \mathcal{A}$ *is a finite set of links between use cases of the scenario and actors.*

A scenario (val, Use, Rel) represents that the use cases contained in Use are executed by actors according to the relation Rel under the circumstances defined by the variable assignment val. A scenario contains a set of use cases Use but no dedicated set of actors because use cases can be executed without any actors, whereas actors cannot exist in scenarios without being linked with use cases.

3.5. An Intuitive View on the Semantics

The following intuitively explains which scenarios are contained in the semantics of a UCD. This paper uses a closed-world view on the semantics of UCDs. Section 7 discusses an alternative open-world semantics.

In this paper's semantics, each use case of a UCD can be executed individually if its precondition is satisfied. The execution of an individual use case introduces a scenario. If a use case is executed in a scenario, then each use case that extends the use case where the guard of the extend relation and the precondition of the use case are satisfied must also be executed in the scenario. This rule applies recursively to all executed use cases in the scenario. If there is an unguarded extend from a use case to a use case of the scenario, then the former use case may be executed in the scenario but is not required to be executed. No other use cases may be executed in the scenario. This includes uses cases that are not connected to the scenario's use cases via the extend relations as well as use cases that are connected via the guarded extend relation where the corresponding guards are not satisfied.

The actors participating in an executed use case are the actors that are explicitly and implicitly associated to the use case. An actor is explicitly associated to a use case if the UCD contains an association between the actor and the use case. An actor is implicitly associated to a use case u if the UCD contains an association between the actor and another use case that generalizes the use case u.

In each scenario, each use case u can be replaced by a use case that specializes the use case u and has a satisfied precondition. With this, the specialization also inherits the properties corresponding to the extend relation of its generalization. The actors participating in the specialization are determined as explained above. Further, all use cases that must be recursively executed when executing the specialization (because of the extend relations) must also be executed.

Each actor associated to a use case can always be replaced by one of its specializations. Thus, each actor can execute all use cases that can be executed by its generalizations.

3.6. Formalization of the Semantics

For precisely defining the semantics of UCDs, we first introduce the auxiliary notions of execution, closure, and scenarios of a use case in a UCD under a variable assignment. In the remainder of this section, let $d = (U, A, Abs, R, G_U, G_A, E, Con)$ be an arbitrary but fixed UCD. Further, we use the following auxiliary functions:

- For $u \in U$, we define the set of actors that are explicitly or implicitly (via generalization) associated to u by $Act_d(u) = \{a \in A \mid \exists v \in U : (u,v) \in G_U \wedge (v,a) \in R\}$.
- For $V \subseteq U$, we define the set of (explicit or implicit) associations between the use cases in V and actors in A by $R_d(V) = \{(v,a) \in V \times A \mid a \in Act_d(v)\}$.
- To avoid notational clutter, we write Act instead of Act_d and R instead of R_d if d is clear from the context.

The execution of a use case $u \in U$ under a variable assignment $val \in \mathcal{V}^{\rightarrow}$ includes all use cases that must be (transitively) executed in addition to u because of extend associations with guards that are satisfied under val. Executions abstract from the actors associated to the use cases and from unguarded extends between use cases.

Definition 3. *Let $u \in U$ be a use case and let $val \in \mathcal{V}^{\rightarrow}$ be a variable assignment. The execution of u in d under val is the smallest set $exec(u, val)$ satisfying the following rules:*

1. *If $eval(Con(u), val) = t$ then $u \in exec(u, val)$.*
2. *If $w \in exec(u, val)$ and $(v,w) \in G$ and $eval(Con(v), val) = t$ and $eval(Con((v,w)), val) = t$, then $v \in exec(u, val)$.*

The first rule states that if the precondition of the use case u is satisfied under the variable assignment v, then the use case u is contained in the execution. The second rule states that whenever a use case w is included in the execution, another use case v extends the use case w, and the guards of the extend association (v,w) and the precondition of v are satisfied under the assignment val, then v is also included in the execution.

The notion of closure closes the set of possible executions of an use case under the generalization relation of use cases and under the unguarded extend relation between use cases. The basic intuition behind the closure is (1) that we can replace a generalized use case by all of its specializations in all of its executions and (2) that we may (but are not required to) execute use cases that are associated with use cases from an execution via the unguarded extend relation. When replacing the generalization by the specialization, we execute the specialization. Thus, we also have to include all the use cases contained in the execution of the specialization under the variable assignment. Similarly, when we execute a use case associated via the unguarded extend relation, then we also have to include all use case contained in the execution of this use case under the variable assignment. The closure still abstracts from actors.

Definition 4. *Let $u \in U$ be a use case and let $val \in \mathcal{V}^{\rightarrow}$ be a variable assignment. The closure of u in d under val is the smallest set $closure(u, val)$ satisfying the following rules:*

1. *If $exec(u, val) \neq \emptyset$, then $exec(u, val) \in closure(u, val)$.*
2. *If $C \in closure(u, val)$ and $w \in C$ and $(v,w) \in G_U$ and $eval(Con(v), val) = t$, then $(C \setminus \{w\}) \cup exec(v, val) \in closure(u, val)$.*
3. *If $C \in closure(u, val)$ and $w \in C$ and $(v,w) \in E$ and $eval(Con(v), val) = t$, then $C \cup exec(v, val) \in closure(u, val)$.*

The first rule states that the execution of the use case under the assignment is an element of the closure (if it can be executed because its precondition is satisfied). The second rule states that if (1) there is an execution C in the closure, (2) there is a use case $w \in C$ in the execution, (3) the use case v is a specialization of the use case w, (4) and the precondition of the use case v is satisfied under the assignment val, then the execution obtained from replacing w by v and adding the use cases contained in the execution of v under val is also an execution contained in the closure.

The third rule states that if (1) there is an execution C in the closure, (2) there is a use case $w \in C$ in the execution, (3) the use case v extends the use case w, (4) and the precondition of v is satisfied under the assignment val, then the execution obtained by adding the execution of the use case v to the execution C is also an element of the closure.

The notion of scenario adds the variable assignments and the actors associated with use cases to the closure of a use case under a variable assignment. If a use case v is a specialization of a use case w, then all actors associated with w are also implicitly associated with v.

Definition 5. *Let $u \in U$ be a use case and let $val \in \mathcal{V}^{\rightarrow}$ be a variable assignment. The scenarios of u in d under val are defined by the smallest set $scn(u, val)$ satisfying:*

1. *If $C \in closure(u, val)$, then $(val, C, R(C)) \in scn(u, val)$.*
2. *If $(val, Use, Rel) \in scn(u, val)$ and $(v,a) \in Rel$ and $(b,a) \in G_A$, then $(val, Use, (Rel \setminus \{(v,a)\}) \cup \{(v,b)\}) \in scn(u, val)$.*

The first rule states that the scenarios obtained from the executions contained in the closure by adding the actors associated to the use cases are contained in the set of scenarios. The second rule states that actors can be replaced by their specializations.

The semantics of a UCD is defined as the set of all scenarios of the use cases in the UCD under all possible variable assignments where the scenarios neither contain abstract actors nor abstract use cases.

Definition 6. *The semantics of the UCD d is defined as $[\![d]\!] = \{(val, Use, Rel) \in scn(u, val) \mid val \in \mathcal{V}^{\rightarrow} \wedge u \in U \wedge Use \subseteq U \setminus Abs \wedge Rel(Use) \subseteq A \setminus Abs\}$.*

4. Semantic Differencing of Use Case Diagrams

The semantic difference from a UCD d to a UCD d' is defined as the set $\delta(d,d') = [\![d]\!] \setminus [\![d']\!]$ of all scenarios that are contained in the semantics of d that are not contained in the semantics of d'. It effectively reveals the scenarios that are possible in the one UCD and not possible in the other UCD. The remainder of this section presents a semantic differencing operator for UCDs. The semantic differencing operator is an automatic procedure for computing at least one element contained in the semantic difference if at least one exists. Thus, the operator can also be used for UCD refinement checking. a UCD d is a refinement of a UCD d' iff $\delta(d,d') = \emptyset$, *i.e.*, the semantic difference is empty. Then, every scenario of d is also a scenario of d'.

The semantic differencing operator is based on two observations. First, it suffices to consider a finite set of variable valuations for determining whether the semantic difference from a UCD to another UCD is empty. The second observation is that it is possible to automatically check whether a scenario is contained in the semantics of a UCD.

Formally, the first observation can be precisely stated by the following proposition:

Proposition 1. *Let d_1 and d_2 be two UCDs where V_i is the set of all variables used in the Boolean expressions of the preconditions and guards of the UCD d_i. Then, $\delta(d_1,d_2) \neq \emptyset$ iff there exists a scenario $s = (val, Use, Rel) \in [\![d_1]\!]$ with $s \notin [\![d_2]\!]$ and $val(v) = f$ for all $v \notin V_1 \cup V_2$.*

Proof. Let d_1, d_2, V_1, V_2 be given as above.

"⇒": Assume there exists a scenario $s = (val, Use, Rel) \in \delta(d_1,d_2)$. We define the scenario $s' = (val', Use, Rel)$ where $val'(v) = val(v)$ if $v \in V_1 \cup V_2$ and $val'(v) = f$ otherwise.

It holds that $s' \in [\![d_1]\!]$ because $s \in [\![d_1]\!]$, val' maps the variables occurring in d_1 to the same values as val, and the use cases, actors as well as links are the same in s and s'.

Suppose towards a contradiction that $s' \in [\![d_2]\!]$. Then, it also hold that $s \in [\![d_2]\!]$ because $s' \in [\![d_2]\!]$, val maps the variables occurring in d_2 to the same values as val', and the use cases, actors as well as links are the same in s and s'. This contradicts the assumption that $s \in \delta(d_1,d_2)$.

Thus, it holds that $s' \in [\![d_1]\!]$ and $s' \notin [\![d_2]\!]$ and $val(v) = f$ for all $v \notin V_1 \cup V_2$, which we needed to show.

"⇐": Directly follows from the assumption. □

By Proposition 1, it suffices to consider the finitely many variable valuations that map all but not necessarily the variables contained in the UCDs to false. The scenarios contained in the semantics of a UCD d must not contain use cases and actors not present in d and the sets of use cases and actors of each UCD are finite. Thus, it is possible to enumerate all scenarios contained in the semantics of d that contain one of the finitely many variable assignments that are relevant for semantic differencing. For each of these scenarios, it can be checked whether the scenario is also an element of the semantics of another UCD d'. If this is the case for all scenarios, then Proposition 1 guarantees that d is a refinement of d'. Otherwise, this procedure yields the computed scenarios that are not contained in the semantics of d' as diff witnesses, which reveal the semantic differences from d to d'.

4.1. Computing Elements Contained in the Semantics

Algorithm 1 Computing the subset of the semantics of an UCD $d = (U, A, Abs, R, G_U, G_A, E, G, Con)$ where exactly the variables in V may not be assigned to f.

```
1: procedure SEM(UCD d, finite set V ⊆ 𝒱)
2:     define res ← ∅ as empty set of scenarios
3:     for all val ∈ 𝒱^→ such that ∀x ∈ 𝒱 \ V : v(x) = f do
4:         for all u ∈ U do
5:             for all (v, Use, Rel) ∈ scn(d, u, val) do
6:                 if Use ⊆ U \ Abs then
7:                     if Rel(Use) ⊆ A \ Abs then
8:                         res ← res ∪ {(v, Use, Rel)}
9:     return res
```

Algorithm 1 can be used to compute the relevant elements contained in the semantics. It takes a UCD d and a finite set of variables V as inputs. The algorithm computes all scenarios contained in the semantics of the UCD d where the variables not contained in the set V must be assigned to f by the variable assignments of the scenarios. First, it initializes the set *res* as an empty set of scenarios (l. 2). Afterwards, it iterates over all variable assignments *val* where the variables not contained in V are assigned to f (l. 3) and over all use cases contained in the UCD (l. 4). In the loops, it adds all scenarios of the case u in the UCD d under the assignment *val* that do neither contain abstract use cases nor abstract actors to the set *res* (ll. 5-8). Finally, the algorithm returns the computed set *res* (l. 9). Thus, the algorithm returns all scenarios contained in the semantics of the UCD d according to Definition 6 where the variables not contained in the set V are assigned to f. The procedure *scn* used in the algorithm is defined in Algorithm 2.

The procedure *scn* takes a use case u, a UCD d, and an assignment *val* as input. It computes the scenarios of u in d under *val*. To this effect, the procedure first initializes the variable *scn* as the empty set (l. 2). Then, it iterates over all sets of uses cases contained in the closure of the use case u in the UCD d under the assignment *val* (ll. 3). For each of these sets, the algorithm constructs the scenario obtained from attaching the actors directly associated with the use cases to the use cases and adds it to the set *scn* (l. 4). These are the scenarios contained in $scn(u, val)$ according to the first rule in Definition 5. Thereafter, the procedure continues by computing the scenarios obtained by iteratively replacing the actors in the scenarios according to the generalization relation. These are the scenarios contained in $scn(u, val)$ according to the second rule in Definition 5. The procedure initializes an empty stack of scenarios *toProcess* that contains the scenarios where the actors still need to be replaced by their specializations (l. 5). Thus, the algorithm directly pushes all scenarios currently contained in the set *scn* to the stack (l. 6). Next, it initializes the empty set *processed* of scenarios. The set contains the scenarios where the actors have already been replaced. The procedure continues

by iteratively processing the scenarios contained in *toProcess* (ll. 8-17). In the loop, the algorithm pops the first scenario from the stack and adds it to the processed scenarios (ll. 9-10). Then, it iterates over the links (v, a) of the scenario (l. 11) and over the elements (b, a) contained in the actor generalization relation of the UCD (l. 12) where the more general actor in the generalization is also the actor in the link. It replaces the generalization by its specialization (ll. 13-14). Afterwards, the procedure adds the resulting scenario to the set of scenarios *scn* (l. 16) and to the stack *toProcess* (l. 17) if it has not been processed before (l. 15). Finally, the procedure returns the set of computed scenarios *scn* (l. 18).

Algorithm 2 Computing scenarios of an use case $u \in U$ in a UCD $d = (U, A, Abs, R, G_U, G_A, E, G, Con)$ under the variable assignment $val \in \mathcal{V}^{\rightarrow}$.

```
1:  procedure SCN(UCD d, u ∈ U, val ∈ V→)
2:      define scn ← ∅ as empty set of scenarios
3:      for all C ∈ closure(d, u, val) do
4:          scn ← scn ∪ {(val, C, R(C))}
5:      define toProcess as empty stack of scenarios
6:      toProcess.pushAll(scn)
7:      define processed ← ∅ as empty set of scenarios
8:      while toProcess not empty do
9:          (val, Use, Rel) ← toProcess.pop()
10:         processed ← processed ∪ {(val, Use, Rel)}
11:         for all (v, a) ∈ Rel do
12:             for all (b, a) ∈ G_A do
13:                 newRel ← (Rel \ {(v, a)}) ∪ {(v, b)}
14:                 new ← (val, Use, newRel)
15:                 if new ∉ processed then
16:                     scn ← scn ∪ {new}
17:                     toProcess.push(new)
18:     return scn
```

Algorithm 3 depicts the procedure *closure*, which takes an use case u, a UCD d, and an assignment *val* as input. It computes the closure of u in d under *val*. It first initializes the variable *cls* as an empty set of sets of use cases (l. 2). This variable stores the closure. Afterwards, the procedure adds the execution of the use case u in d under *val* to the set *cls* if the execution is not empty (l. 3-4). This corresponds to the first rule in Definition 4.

Then, the procedure continues by computing the sets of use cases obtained by iteratively (1) replacing the use cases in the already computed elements of the closure according to the use case generalization relation and (2) by adding executions resulting from adding use cases that may be added to executions because of the unguarded extend relation. These are the scenarios contained in $closure(u, val)$ according to the second and third rules in Definition 4.

To this effect, the procedure initializes an empty stack *toProcess* (l. 5) and pushes the execution contained in the set *cls* (l. 6). The stack contains the sets of use cases that still need to be processed for computing the complete closure. Then, it initializes the set *processed* as an empty set of sets of use cases

Algorithm 3 Computing the closure of an use case $u \in U$ in a UCD $d = (U, A, Abs, R, G_U, G_A, E, G, Con)$ under the variable assignment $val \in \mathcal{V}^{\rightarrow}$.

```
1:  procedure CLOSURE(UCD d, u ∈ U, val ∈ V→)
2:      define cls ← ∅ as empty set of sets of use cases
3:      if exec(u, val) ≠ ∅ then
4:          cls ← cls ∪ {exec(u, val)}
5:      define toProcess as empty stack
6:      toProcess.push(cls)
7:      define processed ← ∅ as empty set
8:      while toProcess not empty do
9:          C ← toProcess.pop()
10:         processed ← processed ∪ {C}
11:         for all w ∈ C do
12:             for all (v, w) ∈ G_U do
13:                 if eval(Con(v), val) = t then
14:                     new ← (C \ {w}) ∪ exec(d, v, val)
15:                     if new ∉ processed then
16:                         cls ← cls ∪ {new}
17:                         toProcess.push(new)
18:             for all (v, w) ∈ E do
19:                 if eval(Con(v), val) = t then
20:                     new ← C ∪ exec(d, v, val)
21:                     if new ∉ processed then
22:                         cls ← cls ∪ {new}
23:                         toProcess.push(new)
24:     return cls
```

(l. 7). This set contains the sets of use cases that have already been processed for computing the complete closure. The procedure continues by iteratively processing the scenarios contained in *toProcess* (ll. 8-24). In the loop, the procedure pops the first set of use cases contained on the stack and adds it to the set stored in the variable *processed* (ll. 9-10). Then, it iterates over all use cases w contained in the currently processed set (l. 11). Then, the outer loop contains two inner loops, which realize the second and the third rules of Definition 4.

First, the procedure iterates over all elements (v, w) contained in the use case generalization relation where the more general use case is w (l. 12). If the precondition of the use case v is satisfied (l. 13), the use case w is replaced by v in the currently processed set and stored in the variable *new* (l. 14). If the set stored in *new* has not already been processed (l. 15), the set is added to the computed closure (l. 16) and to the stack of sets of use cases to be processed (l. 17).

In the second inner loop, the procedure iterates over all elements (v, w) of the unguarded extend relation where the extended use case is equal to w. If the precondition of v is satisfied (l. 19), then the algorithm constructs the execution *new* obtained from adding the execution of v to the currently processed execution C (l. 20). If this execution has not already been processed (l. 21), the algorithm adds it to the closure (l. 22) and pushes it on the stack (l. 23). Finally, the procedure returns the computed closure (l. 24).

Algorithm 4 presents the procedure *exec*. It takes a use case

Algorithm 4 Computing the execution of an use case $u \in U$ in a UCD $d = (U, A, Abs, R, G_U, G_A, E, G, Con)$ under the variable assignment $val \in \mathcal{V}^{\rightarrow}$.

```
1:  procedure EXEC(UCD d, u ∈ U, val ∈ V^→)
2:      define exec ← ∅ as empty set of use cases
3:      if eval(Con(u), val) = t then
4:          exec ← exec ∪ {u}
5:      define toProcess as empty stack
6:      toProcess.pushAll(exec)
7:      define processed ← ∅ as empty set
8:      while toProcess not empty do
9:          cur ← toProcess.pop()
10:         processed ← processed ∪ {cur}
11:         for all (v, w) ∈ G with w = cur do
12:             if eval(Con((v, w)), val) = t then
13:                 if eval(Con(v), val) = t then
14:                     if v ∉ processed then
15:                         exec ← exec ∪ {v}
16:                         toProcess.push(v)
17:     return exec
```

u, a UCD d, and an assignment v as input and computes the execution of u in d under val. First, it initializes the set $exec$ as an empty set of use cases (l. 2). This set contains the execution of u. If the precondition of the use case u is satisfied (l. 3), it is added to the set $exec$ (l. 4). This corresponds to the first rule in Definition 3. The procedure continues with adding the use cases contained in the execution according to the second rule of Definition 3. It initializes the stack $toProcess$ (l. 5) and pushes the elements contained in $exec$ on the stack (l. 6). Thus, the stack is initially either empty or contains the use case u. The stack contains the use cases that still need to be processed for computing the execution. Then, the procedure initializes the variable $processed$ as an empty set (l. 7). The variable stores the set of use cases that have already been processed. In the following loop (ll. 8-16), the procedure iteratively computes the closure by processing the elements contained on the stack of use cases that still need to be processed. To this effect, it pops the first element cur from the stack (l. 9) and adds it to the set of processed use cases (l. 10). Afterwards, the procedure iterates over all elements (v, w) of the UCDs guarded extend relation where the extended use case w is equal to the currently processed use case cur (l. 11). If the condition of (v, w) is satisfied (l. 12), the precondition of the extending use case v is satisfied (l. 13), and the use case v has not already been processed (l. 14), the use case is added to the computed execution $exec$ (l. 15) and the stack of use cases that still need to be processed (l. 16). Finally, the algorithm returns the computed execution stored in the variable $exec$ (l. 17).

4.2. Semantic Differencing Operator

For checking whether the semantic difference from a UCD d_1 to a UCD d_2 is not empty, it suffices to compute the set $sem(d_1, V) \setminus sem(d_2, V)$, where V is the set of variables used in at least one of the UCDs d_1 or d_2, and to check whether it is empty. If it is not empty, each element contained in the set is a diff witness contained in the semantic difference from d_1 to d_2. If the set is empty, then d_1 is guaranteed to be a refinement of d_2 (cf. Proposition 1). Algorithm 1 depicts a procedure that can be used for the computation.

4.3. Semantic Differencing Operator Complexity

The semantic differencing operator described in Section 4.2 has exponential time and space complexities in the numbers of use cases and variables used in the UCDs. The exponential complexity is a consequence of the fact that there may be exponentially many scenarios (in the number of use cases and variables) contained in the set $sem(d, V)$ for a UCD d and a set of variables V. Thus, semantic differencing with the operator is practically infeasible for very large and complex UCDs.

The exponential time complexity is not surprising as the syntax of UCDs (cf. Definition 1) allows arbitrary propositional formulas to be used as preconditions of use cases. Thus, checking whether a propositional logic formula is satisfiable can be easily reduced to checking whether the semantics of a UCD is empty: The UCD only contains a single use case. The precondition of the use case is the formula that should be checked for satisfiability. Then, the semantics of the UCD is not empty iff the formula is satisfiable. As the satisfiability problem for propositional logic is NP-complete, the problem of checking whether the semantics of an UCD is empty is NP-hard. Further, semantic differencing of UCDs is at least as hard as checking whether the semantics of a UCD is empty because checking whether the semantics of a UCD d is empty can be reduced to checking whether $\delta(d, e) = \emptyset$ where e is the empty UCD not containing any use cases or actors. The semantics of e is trivially empty.

However, the application of the semantic differencing operator may be feasible for many relatively small UCDs used in practice as illustrated by the experimental evaluation presented in the following Section 5.

5. Experimental Evaluation

This section presents the results of experimental evaluations of the computation times of a prototype implementation.

We implemented a textual UCD modeling language using the language workbench MontiCore (Hölldobler et al. 2021). For this language, we implemented the semantic differencing operator presented in Section 4. The implementation is purely written in Java and together with test cases covering the algorithms presented in this paper publicly available in a GitHub repository of the MontiCore group (https://github.com/MontiCore/ucd).

A MontiCore grammar (MCG) implements the reduced abstract syntax presented in Section 3.2. A simplified excerpt of the grammar is depicted in Figure 6. Detailed information can be found in the documentation on GitHub. The relation between the reduced abstract syntax and the MCG is straightforward: Use cases diagrams are defined by using the nonterminal `UseCaseDiagram`, use cases are defined by using the nonterminal `UCDUseCase`, and actors are defined by using the nonterminal `UCDActor`. The implementation of the semantic

```
UseCaseDiagram =                                          MCG
  "usecasediagram" Name "{"
    (UCDUseCase | UCDActor)*
  "}";

UCDUseCase =
  ["abstract"]? Name ("[" Expression "]")?
  ("specializes" sup:(Name || ",")+)?
  ("extend" (UCDExtend || ",")+ )?
  ("include" incl:(Name || ",")+ )?
  ";";

UCDExtend =
  Name ("[" Expression "]")?;

UCDActor =
  ["abstract"]? "@" Name
  ("specializes" sup:(Name || ",")+)?
  ("--" uc:(Name || ",")+)?
  ";";
```

Figure 6 Simplified MontiCore grammar for UCDs.

differencing operator directly operates on the abstract syntax tree instances instantiated by the MontiCore parser generated from the MCG for the UCD language.

The implementation is a straight-forward translation of the algorithms presented in Section 4.1 into Java. To build confidence on the completeness and correctness of our implementation, we also added several test cases to the GitHub repository covering all relevant cases. Only minor error cases are not covered from these test cases.

The result of the Java method implementing the semantic differencing operator is of type `Set<Scenario>` where `Scenario` is a class with the attributes `val` of type `Set<String>`, `ucs` of type `Set<String>`, and `actor2uc` of type `Multimap<String,String>`. For an instance of type `Scenario`, the attribute `val` exactly contains the names of all variables that are assigned to `true` in the scenario. All other variables are assigned to `false`. The set `ucs` contains all use cases participating in the scenario. The map `actor2uc` defines the actors participating in the scenario (the key set of the map) and the relation between the actors and use cases. It maps each actor to all the use cases to which it is associated. Users of the UCD language can directly operate on the result computed by the Java method implementing the semantic differencing operator. Alternatively, users of the semantic differencing operator can use the command line interface (downloadable at http://monticore.de/download/UCDCLI.jar and documented at https://github.com/MontiCore/ucd) for computing diff witnesses. When using the command line interface, the diff witnesses are presented in a textual notation obtained from pretty printing the values of the attributes of the scenarios.

We ran experiments with twelve models from Section 2 and inspired by external sources (Friedenthal et al. 2014; Jäckel et al. 2020, 2021). Unfortunately, to the best of our knowledge, there exists no publicly available and easily automatically accessible database containing many UCDs suitable for automatic testing of use case diagrams. Even though, GenMyModel (https://www.genmymodel.com/) provides a large overview of user created diagrams, the API is currently not in the state to make automatic testing and comparison of models easily possible. It would be an interesting future work to test our approach with such a library in a larger scale or based on use case diagrams used in industry projects.

Table 1 summarizes the sizes of the UCDs used in the experiments in terms of the sizes of the individual components according to Definition 1. The names of the UCDs are abbreviated as follows: CarCharging1 (CC1), CarCharging2 (CC2), SwimmyFish1 (SF1), SwimmyFish2 (SF2), FeatureBroadcastPosition (FBP), FeatureNavigation (FN), OperateVehicle (OV), OperatePremiumVehicle (OPV), SecurityEnterprise (SE), and SecurityEnterpriseCorrection (SEC). The UCD `VTOL2`, for example, comprises 44 use cases, 14 actors, 37 associations, and 25 elements in the extend relations in total.

We executed the semantic differencing operator for each pair of example models. This yielded 144 experiments in total. The experiments were all executed on an ordinary laptop computer equipped with an Intel Core i7-8650U CPU@1.90GHz processor, 16GB RAM, and a Samsung PM981 512GB SSD hard drive using Windows 10 and Java 1.8.0_192. All experiments were executed in the same Java virtual machine run. Table 2 summarizes the computation times of the semantic differencing operator implementation. The computation times range from under 1 millisecond (corresponding to the 0 entries in the table) to 107 milliseconds.

As the computation times of all experiments were relatively small (smaller than 107 milliseconds), we conclude that the performance of the semantic differencing operator suffices for the example models. Although we cannot generalize the results to arbitrary UCDs (cf. Section 4.3), we believe that the performance of the operator is sufficient for many UCDs used in practice (especially for UCDs with sizes that are comparable to the sizes of the UCDs used in the experiments).

6. Related Work

This section presents related work concerning use case modeling and semantic differencing.

Use cases were introduced in (Jacobson et al. 1992) as a method to describe a system's functionality based on simple models. In general, UCDs can be differentiated from use case specifications. While UCDs present an overview about a system's use cases and their interactions with actors, use case specifications focus on the internal behavior of use cases. While the graphical syntax of UCDs is standardized in the Unified Modeling Language (UML) specification ("OMG Unified Modeling Language (OMG UML)" 2017) and in the Systems Modeling Language (SysML) specification ("OMG Systems Modeling Language" 2019), there exist various additional ways to formulate use case specifications such as tables as mentioned in (Fowler 2004; Cockburn 2001) or other UML/SysML diagrams (Friedenthal et al. 2014). Not all approaches clearly differentiate between UCDs, as a means to express use cases and their relation, and use case specifications, which specify the system behavior under consideration of a use case.

In general, there are two approaches for syntactically repre-

UCD	$\|U\|$	$\|A\|$	$\|Abs\|$	$\|R\|$	$\|G_U\|$	$\|G_A\|$	$\|E\|$	$\|G\|$
CC1	6	4	0	12	6	5	0	0
CC2	8	4	1	16	10	5	0	2
SF1	8	2	0	7	10	2	2	1
SF2	8	2	1	7	10	2	0	3
FBP	12	9	0	17	12	9	0	2
FN	9	5	0	7	9	5	1	4
OV	8	3	0	6	10	5	0	3
OPV	12	4	0	10	16	6	0	5
SE	6	4	0	6	8	4	0	1
SEC	6	5	0	7	8	5	0	1
VTOL1	43	13	0	35	43	13	2	20
VTOL2	44	14	0	37	44	15	3	22

Table 1 Sizes of the UCDs used in the experiments.

senting UCDs. First, the graphical UCD syntax from ("OMG Unified Modeling Language (OMG UML)" 2017) that we based our Definition 1 on. Second, tabular specification techniques such as the method proposed in (Cockburn 2001; Fowler 2004) that usually focus on use case specifications. As we focus on the semantics of UCDs and the OMG specification is the de facto standard in this area, our UCD syntax is mainly inspired by ("OMG Unified Modeling Language (OMG UML)" 2017) and complemented with a textual syntax used in our implementation for the evaluation.

UCDs are outside of the scope of the OMG's fUML (Object Management Group 2005), which partly specifies the semantics of the UML. An approach for formalizing the behavior of use cases is presented in (Shen & Liu 2003). However, the semantics in (Shen & Liu 2003) primarily focuses on the presentation of a use cases pre- and post-conditions and not on complete UCDs. More recently, an approach to define the semantics of UCDs based on the common Algebraic Specification Language (CASL) is presented in (Mondal et al. 2014). In contrast to our approach, the authors use CASL to specify their use case diagrams formally and do not differentiate between include and extend relationships. In (Whittle 2006), an approach to precisely specify use cases is presented. In contrast to our approach, which focuses on the specification of the UCD, this approach primarily focuses on use case specifications and uses a combination of sequence and activity diagrams to model use cases. While the approach defines a precise formal semantics, the syntax and semantics of the diagrams are more related to UML activity and sequence diagrams than to the UML UCDs. This is the case because the UCDs are modeled with use case charts based on activity diagrams and scenario diagrams based on sequence diagrams. Therefore, this semantics definition is more related to the semantics of sequence diagrams presented in (Harel & Maoz 2008; Eichner et al. 2005) and activity diagrams (Maoz et al. 2011a; Kautz & Rumpe 2018b; Kautz 2021) than to the use case diagram semantics presented in our work. Nevertheless, it could be an interesting future work to specify the behavior of a use case using a similar semantics in combination with our UCD semantics. Similarly, an approach to formally define the semantics of a use case specification without focusing on the diagram is presented in (Oliveira et al. 2014).

Moreover, (Sengupta & Bhattacharya 2006) describes a method to translate UCDs into a Z notation scheme to perform type checking techniques on the UCDs and finally present the results in an entity relationship diagram. In contrast to our approach, the presented method relies on a Z notation scheme and a type checking framework, while our approach is mostly tool independent. Additionally, (Sousa et al. 2017) presents an interesting approach to transform UCDs into Event-B. While this approach has its strengths in defining systems as contexts, in contrast to our approach, the authors decided to not consider generalization and extension points in the use case diagram variant they consider.

Other approaches for semantic differencing using the same definition of semantic difference have been developed for class diagrams (Maoz et al. 2011b; Kautz et al. 2017), statecharts (Drave, Eikermann, et al. 2019), feature diagrams (Acher et al. 2012; Drave, Kautz, et al. 2019; Kautz 2021), activity diagrams (Maoz et al. 2011a; Kautz & Rumpe 2018b; Kautz 2021), sequence diagrams (Kautz 2021), interactive automata (Butting et al. 2017, 2019; Kautz 2021), combinatorial models of test designs (Tzoref-Brill & Maoz 2017), and Alloy modules (Ringert & Wali 2020). A framework for semantic differencing operators that are based on behavioral semantics specifications is

	CC1	CC2	SF1	SF2	FBP	FN	OV	OPV	SE	SEC	VTOL1	VTOL2
CC1	2	4	6	8	1	1	3	7	2	1	3	5
CC2	1	0	1	3	0	1	1	7	1	0	2	1
SF1	1	1	1	4	2	0	2	11	6	2	4	3
SF2	2	1	1	1	1	1	1	8	2	2	4	4
FBP	0	1	0	1	0	0	1	3	1	1	2	2
FN	0	1	1	1	0	1	0	3	0	1	1	1
OV	0	0	1	1	1	0	0	3	1	4	3	4
OPV	3	3	7	13	4	5	3	6	5	4	13	18
SE	0	1	1	1	2	0	1	5	0	1	2	3
SEC	0	1	0	2	1	0	1	5	0	1	2	3
VTOL1	2	0	3	7	3	6	10	14	3	5	3	4
VTOL2	2	7	4	9	3	107	3	104	3	5	6	6

Table 2 Computation times of the semantic differencing experiments in milliseconds.

presented in (Langer et al. 2014). The framework is instantiated with activity diagrams, class diagrams, and petri nets. Non-enumerative semantic differencing operators (Langer et al. 2014) representing semantic differences by another model have been developed for automata (Fahrenberg et al. 2011), class diagrams (Fahrenberg et al. 2014), and feature diagrams (Fahrenberg et al. 2011). Advanced techniques relating semantic to syntactic differences are presented in (Maoz & Ringert 2015, 2018; Kautz & Rumpe 2018a; Kautz 2021). The techniques can also be instantiated with this paper's UCD language and differencing operator.

7. Discussion and Possible Extensions

This paper defines a closed-world semantics for UCDs. In the closed-world, a scenario contained in the semantics of a UCD only contains use cases that can or must be executed when a designated use case of the diagram is executed. This relationship between the use cases is modeled in the UCD by the extend and include relations. Use cases not modeled in a UCD cannot be part of any scenario contained in the semantics of the UCD.

Alternatively, it is possible to define an open-world semantics for UCDs similar to the open-world semantics for feature models defined in (Drave, Kautz, et al. 2019). With the open-world semantics, different unrelated use cases could be executed in the same scenario. This corresponds to merging multiple scenarios contained in the closed-world semantics as defined in this paper. Similarly, the open-world semantics could allow the occurrence of use cases not modeled in a UCD to be contained in the scenarios of the UCD. The assumption of the closed-world semantics is that exactly what has been explicitly modeled is possible. In contrast, the assumption of an open-world semantics would be that anything not explicitly constrained is possible.

The closed-world semantics has the advantage that only use cases of the diagram can be part of the scenarios. Thus, small changes in the model can already cause semantic differences. In late development stages, in which the development of the UCDs is nearly finished, this leads to the expected behavior of the semantic differencing operator. In contrast, models often change in early development stages. In these stages, the addition of model elements, such as use cases, is usually considered to be a refinement of the model. Then, an open-world semantics and a corresponding semantic differencing operator could be adequate. The development of an open-world semantics or even multiple variants and corresponding semantic differencing operators is interesting future work.

As mentioned in the introduction, our algorithm for semantic differencing of UCDs can be used in the acquisition phase of system development. For example, it is possible to investigate the use-case level differences between a required system and the (sub-)systems available by different vendors on the market. Using our current proposal, the system developers are enabled to formally check whether the UCDs specified by the acquirer are compatible to those defined by the vendor company. This is possible by checking whether the UCD of the required system is a refinement of the UCDs provided by the vendor company. Since the algorithm ultimately results in diff witnesses, these witnesses can be used to analyze and discuss the found mismatches and incompatibilities in future negotiations. Similarly, checking whether a new version of a UCD could safely replace the previous version without the clients experiencing problems can be reduced to refinement checking. Both scenarios seem to be promising for future evaluations.

Further, the syntax and semantics presented in this paper could be extended in future works with advanced concepts, such

as guarantees and triggers (Fowler 2004) or use case templates (Cockburn 2001; Fowler 2004) to additionally enable the definition of use case specifications. When additionally considering use case specifications, two UCDs that are syntactically equal in this paper's view can be different from a semantical point of view if the specifications of the use cases differ.

Other possible extension are the additions of system boundaries and extension points ("OMG Unified Modeling Language (OMG UML)" 2017). With system boundaries, the syntax and semantics of UCDs must be extended to be able to distinguish the systems in which use cases are executed. When adding extension points, the semantics has to distinguish whether a use case has been executed in the context of an extension point or not. In case a use case has been executed in the context of an extension point, the semantics also has to consider the name of the extension point. The semantics must distinguish whether a use case has been executed in the context of an extension point e or an extension point e' with $e \neq e'$.

The UML UCD definition also supports cardinalities on association ends. For adding cardinalities to this paper's approach, the semantic domain must be extended such that scenarios can contain multiple instances of actors and use cases. Then, the semantics of a UCD should only contain scenarios where the links between the instances of actors and use cases are compatible to the cardinalities defined in the UCD.

8. Conclusion

This paper presented an approach to automatic semantic differencing of UCDs. To this end, we defined an abstract syntax, a formal semantics, and a semantic differencing operator for a UCD variant. Our experimental evaluation has shown that although the semantic differencing operator has exponential time and space complexity, the performance of the semantic differencing operator may be feasible for many UCDs used in practice. Thus, it eases semantic UCD evolution analysis by automation. In a more general context, the contributions of this paper can be integrated into system modeling tools to enhance the process of modeling top-level requirements.

Acknowledgments

This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IS20092. The responsibility for the content of this publication is with the authors.

References

Acher, M., Heymans, P., Collet, P., Quinton, C., Lahire, P., & Merle, P. (2012). Feature Model Differences. In *Advanced information systems engineering* (pp. 629–645). Springer Berlin Heidelberg.

Butting, A., Kautz, O., Rumpe, B., & Wortmann, A. (2017). Semantic Differencing for Message-Driven Component & Connector Architectures. In *Ieee international conference on software architecture (icsa)* (p. 145-154). IEEE.

Butting, A., Kautz, O., Rumpe, B., & Wortmann, A. (2019). Continuously analyzing finite, message-driven, time-synchronous component & connector systems during architecture evolution. *Journal of Systems and Software*, *149*, 437–461.

Cockburn, A. (2001). *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc.

Drave, I., Eikermann, R., Kautz, O., & Rumpe, B. (2019). Semantic Differencing of Statecharts for Object-Oriented Systems. In *Proceedings of the 7th international conference on model-driven engineering and software development* (p. 272-280). SciTePress.

Drave, I., Kautz, O., Michael, J., & Rumpe, B. (2019). Semantic Evolution Analysis of Feature Models. In *Proceedings of the 23rd international systems and software product line conference* (pp. 245–255). ACM.

Eichner, C., Fleischhack, H., Meyer, R., Schrimpf, U., & Stehno, C. (2005). Compositional semantics for UML 2.0 sequence diagrams using Petri Nets. In *International sdl forum* (pp. 133–148).

Fahrenberg, U., Acher, M., Legay, A., & Wasowski, A. (2014). Sound Merging and Differencing for Class Diagrams. In *Fundamental approaches to software engineering* (pp. 63–78). Springer Berlin Heidelberg.

Fahrenberg, U., Legay, A., & Wasowski, A. (2011). Vision Paper: Make a Difference! (Semantically). In *Model driven engineering languages and systems* (pp. 490–500). Springer Berlin Heidelberg.

Fowler, M. (2004). *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional.

France, R., & Rumpe, B. (2007). Model-Driven Development of Complex Software: A Research Roadmap. In *Future of software engineering (fose '07)* (p. 37-54). IEEE.

Friedenthal, S., Moore, A., & Steiner, R. (2014). *A Practical Guide to SysML, Third Edition: The Systems Modeling Language* (3rd ed.). Morgan Kaufmann Publishers Inc.

Harel, D., & Maoz, S. (2008). Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software & Systems Modeling*, *7*(2), 237–252.

Harel, D., & Rumpe, B. (2004). Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, *37*(10), 64-72.

Hölldobler, K., Kautz, O., & Rumpe, B. (2021). *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag.

Hölldobler, K., Michael, J., Ringert, J. O., Rumpe, B., & Wortmann, A. (2019). Innovations in Model-based Software and Systems Engineering. *The Journal of Object Technology*, *18*(1), 1-60.

Jacobson, I., Christerson, M., Jonsson, P., & Övergaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley.

Jäckel, N., Granrath, C., Schaller, R., Grotenrath, M., Fischer, M., Orth, P., & Andert, J. (2020). Integration of VTOL air-taxis into an existing infrastructure with the use of the Model-Based System Engineering (MBSE) concept CUBE. In *76th annual forum and technology display 2020 the future of vertical flight.*

Jäckel, N., Granrath, C., Wachtmeister, L., Karaduman, A., Rumpe, B., & Andert, J. (2021). Feature-Driven Specification

of VTOL Air-Taxis with the Use of the Model- Based System Engineering (MBSE) Methodology CUBE. In *77th annual forum and technology display 2021 the future of vertical flight.*

Kautz, O. (2021). *Model Analyses Based on Semantic Differencing and Automatic Model Repair*. Shaker Verlag.

Kautz, O., Maoz, S., Ringert, J. O., & Rumpe, B. (2017, July). *CD2Alloy: A Translation of Class Diagrams to Alloy* (Technical Report No. AIB-2017-06). RWTH Aachen University.

Kautz, O., & Rumpe, B. (2018a, October). On Computing Instructions to Repair Failed Model Refinements. In *Proceedings of the 21th acm/ieee international conference on model driven engineering languages and systems* (pp. 289–299). ACM.

Kautz, O., & Rumpe, B. (2018b, October). Semantic Differencing of Activity Diagrams by a Translation into Finite Automata. In *Proceedings of models 2018 workshops.* CEUR.

Langer, P., Mayerhofer, T., & Kappel, G. (2014). Semantic Model Differencing Utilizing Behavioral Semantics Specifications. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, & E. Insfran (Eds.), *Model-driven engineering languages and systems* (pp. 116–132). Springer International Publishing.

Maoz, S., & Ringert, J. O. (2015). A Framework for Relating Syntactic and Semantic Model Differences. In *Acm/ieee 18th international conference on model driven engineering languages and systems (models)* (pp. 24–33). IEEE.

Maoz, S., & Ringert, J. O. (2018). A framework for relating syntactic and semantic model differences. *Software & Systems Modeling*, *17*(3), 753–777.

Maoz, S., Ringert, J. O., & Rumpe, B. (2011a). ADDiff: Semantic Differencing for Activity Diagrams. In *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering* (pp. 179–189). ACM.

Maoz, S., Ringert, J. O., & Rumpe, B. (2011b). CDDiff: Semantic Differencing for Class Diagrams. In *Ecoop 2011 – object-oriented programming* (p. 230-254). Springer Berlin Heidelberg.

Maoz, S., Ringert, J. O., & Rumpe, B. (2011c). A Manifesto for Semantic Model Differencing. In *Models in software engineering* (pp. 194–203). Springer Berlin Heidelberg.

Maoz, S., Ringert, J. O., & Rumpe, B. (2011d). Summarizing Semantic Model Differences. In *International workshop on models and evolution at acm/ieee 14th international conference on model driven engineering languages and systems.*

Maoz, S., Ringert, J. O., & Rumpe, B. (2012). An Interim Summary on Semantic Model Differencing. *Softwaretechnik-Trends*, *32*(4), 44–46.

Mondal, B., Das, B., & Banerjee, P. (2014). Formal Specification of UML Use Case Diagram–A CASL based Approach. *International Journal of Computer Science and Information Technologies*, *5*(3), 2713–2717.

Object Management Group. (2005, April). *Semantics for a foundational subset for executable UML models, Request for Proposal.*

Oliveira, M., Ribeiro, L., Cota, É., Duarte, L., Nunes, I., & Reis, F. (2014). Use case analysis based on formal methods: An empirical study. In *Wadt.*

OMG Systems Modeling Language (Version 1.6 ed.) [Computer software manual]. (2019, March).

OMG Unified Modeling Language (OMG UML) (Version 2.5.1 ed.) [Computer software manual]. (2017, December).

Pretschner, A., Broy, M., Kruger, I. H., & Stauner, T. (2007). Software engineering for automotive systems: A roadmap. In *Fose '07: 2007 future of software engineering* (pp. 55–71). IEEE Computer Society.

Ringert, J. O., & Wali, S. W. (2020). Semantic comparisons of alloy models. In *Proceedings of the 23rd acm/ieee international conference on model driven engineering languages and systems* (p. 165–174). ACM.

Sengupta, S., & Bhattacharya, S. (2006). Formalization of UML use case diagram-a Z notation based approach. In *2006 international conference on computing informatics* (p. 1-6).

Shen, W., & Liu, S. (2003). Formalization, testing and execution of a use case diagram. In *International conference on formal engineering methods* (pp. 68–85). Springer.

Sousa, T., Kelvin, L., Neto, C., & Carvalho, C. (2017, 04). A Formal Semantics for Use Case Diagram Via Event-B. *Journal of Software*, *12*, 189-200.

Tzoref-Brill, R., & Maoz, S. (2017). Syntactic and Semantic Differencing for Combinatorial Models of Test Designs. In *2017 ieee/acm 39th international conference on software engineering (icse)* (pp. 621–631). IEEE.

Whittle, J. (2006). Specifying precise use cases with use case charts. In *Satellite events at the models 2005 conference* (pp. 290–301). Springer Berlin Heidelberg.

About the authors

Oliver Kautz successfully finished his doctoral studies at RWTH Aachen University in 2021. He is currently working as a product owner at CBC Cologne Broadcasting Center GmbH, mainly working on software applications for planning the contents of TV channels and streaming platforms. His research interests include software engineering, software language engineering, model-driven development, and semantics of modeling languages. You can contact him at oliver.kautz@rtl.de.

Bernhard Rumpe is full Professor at RWTH Aachen University, where he leads the Software Engineering Research Group. His research interests include modelling, model languages, software model language engineering, code synthesis from models and model-based analysis. You can contact him at rumpe@se-rwth.de or visit https://www.se-rwth.de/.

Louis Wachtmeister is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software and systems engineering, software architectures, and model-driven development. You can contact him at wachtmeister@se-rwth.de or visit https://www.se-rwth.de/.