On Computing Instructions to Repair Failed Model Refinements

Oliver Kautz, Bernhard Rumpe
Software Engineering
RWTH Aachen University, Aachen, Germany
www.se-rwth.de

ABSTRACT

A model refinement step is the process of removing underspecification from a model by applying syntactic changes such that the transformed model's semantics is subsumed by the semantics of the original model. Performing a refinement step is error-prone and thus needs automated and meaningful support for repair in case an intended refinement step yields an incorrect result. This paper introduces sufficient conditions on a modeling language that enable fully automatic calculation of syntactic changes, which transform one model to a refinement of another model. In contrast to previous work, this paper's approach is independent of a concrete modeling language, computes shortest syntactic changes to maintain the developer's intention behind the model as much as possible, and does not assume availability of powerful model composition operators. The method relies on partitioning the syntactic change operations applicable to each model in equivalence classes and on excluding syntactic changes that are not part of shortest changes leading to a refining model. This paper contains formal proofs for the modeling language independent results and shows the method's applicability and usefulness by instantiating the framework with three modeling languages. The results provide a language independent and fully automated method to repair refinement steps under intuitive assumptions as well as language independent foundational insights concerning the relation between syntactic changes and the impact of their application on a model's semantics.

CCS CONCEPTS

• **Software and its engineering** → *Software verification and validation*; *Formal software verification*; Software creation and management;

KEYWORDS

Model, Evolution Analysis, Modeling Language, Refinement

ACM Reference Format:
Oliver Kautz, Bernhard Rumpe. 2018. On Computing Instructions to Repair Failed Model Refinements. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239384

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MODELS '18, October 14–19, 2018, Copenhagen, Denmark
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239384

1 INTRODUCTION

Models are the primary development artifacts in model-driven software development (MDSD). Thus, managing their evolution is an important task during system development. Existing approaches mainly consider syntactic model evolution (e.g. [2, 12, 13, 15, 16, 27, 28]). Only a few approaches consider the changes of a model's semantics (e.g. [1, 3, 6, 7, 17, 18, 22, 24]) or try to relate syntactic changes to the impact of their application on a model's semantics [8, 20]. Research in syntactic differencing already produced well-accepted approaches abstracting from a modeling language's details [2]. Concrete modeling language independent approaches rarely exist in context of semantic differencing [17, 18]. This might be due to the high diversity and complexity of modeling language semantics. Even less approaches combine syntactic with semantic differencing while abstracting from a concrete modeling language [7, 20], although this produces general results that apply to multiple languages. This is surprising because the syntax and semantics of each modeling language is usually tailored towards a specific application domain. As the number of domains is rather infinite and permanently increasing, developing new approaches to evolution management for each newly emerging modeling language is costly and may be even redundant from a research perspective. Instead, employing a general definition of modeling language [10], stating assumptions for a modeling language to hold, and then developing methods for evolution analysis under the assumptions enables to provide methods for whole classes of concrete modeling languages that meet the assumptions.

This paper presents a modeling language independent approach relating syntactic with semantic differencing. A model is a refinement of another model if the semantics of the latter subsumes the semantics of the former [11]. A refinement step is the evolution process of changing a model such that the successor version is a refinement of the predecessor version. In MDSD, refinement steps are naturally performed in reaction to changing requirements and availability of additional information. The idea is to start with an underspecified model encoding the available information and to iteratively refine the model when additional information becomes available until ultimately obtaining a correct system implementation. Refinement steps are error-prone and the state of the art provides little support for repairing unsuccessful refinement steps. More specifically, a developer may introduce a bug to a successor model version such that it is no refinement of its predecessor. This bug needs to be detected and fixed. Semantic differencing approaches are able to detect whether a model is a refinement of another model. If refinement does not hold, semantic differencing usually provides a representation that can be used for (manually) analyzing the syntactic reasons causing the semantic difference. Such a representation is typically a diff witness [21] or another model summarizing the semantic difference [7]. A diff witness is

[KR18a] O. Kautz, B. Rumpe:
On Computing Instructions to Repair Failed Model Refinements.
In: Conference on Model Driven Engineering Languages and Systems (MODELS'18), pg. 289-299, Copenhagen, ACM, Oct. 2018.
www.se-rwth.de/publications/

an element of the semantics of one model, which is no element of the other model's semantics. It thus serves as concrete disproof for refinement. For repairing the intended refinement step, the developer may use the representation to analyze the evolved model's syntax for the reasons causing non-refinement. Based on this, she is possibly ultimately able to change the evolved model. However, this change may also fail, whereupon the developer needs to apply the same workflow again. The process of connecting the representation that reasons why refinement does not hold to the changes required to repair the evolved model is manually performed by the developer without tool support. Instructions in form of a plan containing syntactic changes that definitely transform the evolved model to a refinement of the original is missing. Our contribution bridges this gap: For any modeling language that meets our assumptions (cf. Sec. 5), it is possible to fully automatically calculate a shortest change sequence (syntactic changes) that definitely transforms the evolved model to a refinement of the original. As the developer intends the evolved model to be a refinement of the original, one can expect the number of required changes to be small. Further, applying smallest changes keeps a developer's intention as much as possible with respect to the syntax of original the model. Thus, calculating shortest syntactic changes is reasonable. The assumptions include that the concrete modeling language meets this paper's general definition of modeling language [10], there exists a (possibly infinite) set of well-defined change operations that can transform any valid model to any other valid model, consistency and refinement checking between models of the language is decidable, and change operations can be partitioned into finitely many equivalence classes characterized by some symmetry condition (cf. Sec. 4).

The next section presents three possible analyses enabled by instantiating our framework with an activity diagram (AD), a finite automaton, and a feature model (FM) language. Sec. 3 introduces the formal foundations that ground our approach. Sec. 4 formally defines the goal and reveals important properties relating change operations to the impact of their application on a model's semantics. Sec. 5 formalizes our assumptions that guarantee computability of shortest refinement repairing change sequences and presents an algorithm to compute them. Sec. 6 informally instantiates the framework with three concrete modeling languages. Sec. 7 discusses related work. Sec. 8 concludes.

2 EXAMPLES

This section provides examples illustrating potential use cases for our approach. To the best of our knowledge, there are no previous works that support the following analyses.

2.1 Shortest Repair of an Activity Diagram Refinement Step

Fig. 1 depicts two ADs inspired from [15, 16]. An insurance company manager aims at improving the efficiency of processing incoming claims. She thus formalizes the workflow to be executed by employees on receipt of incoming claims with ad_1. This AD is very underspecified as it models all possible executions that are reasonable from the manager's perspective. The manager thus hands the AD over to an employee, who should refine the workflow to exclude executions that are not reasonable from an operational perspective.

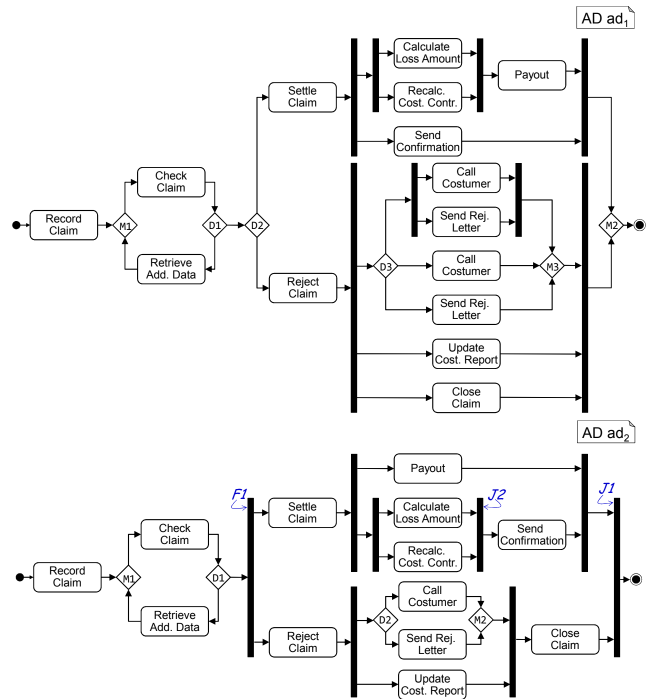
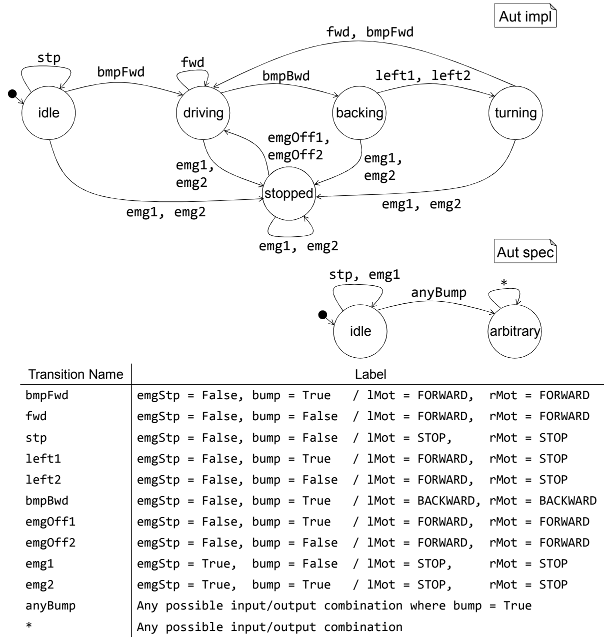


Figure 1: Two activity diagrams modeling workflows in context of an insurance company adapted from [15, 16].

The employee edits the workflow to ad_2 and informs the manager about the changes. The manager decides to review the changes, uses semantic differencing and identifies that the new version ad_2 is no refinement of the original model ad_1. She decides to first identify the error and to prepare a suggestion for repairing the model, before consulting the employee. Thus, she uses our framework and finds out that at least two changes are required. She gets presented that converting the concurrent fragment between nodes $\texttt{F}_1$ and $\texttt{J}_1$ to a branching fragment (converting $\texttt{F}_1$ to a decision node and converting $\texttt{J}_1$ to a merge node) and moving the action node labeled `Pay Out` between the join node $\texttt{J}_2$ and the node labeled `Send Confirmation` is a shortest sequence of change operations for transforming ad_2 into a refinement of ad_1. She considers the changes as reasonable. With this information, the manager consults the employee. It turns out, the employee has accidentally changed the fragment's type. Further discussions reveals that payouts should be definitely executed after calculating the exact loss amount and recalculating the costumer contribution. The manager applies the fully automatically calculated change operations to ad_2 and obtains the final AD, which is a refinement of ad_1.

2.2 Shortest Repair of a Finite Automaton Towards Requirement Satisfaction

Fig. 2 depicts a slightly adapted example from [25]. It shows two reactive finite automata describing the behavior of a mobile robot. In this example, we interpret the models as reactive finite safety automata. Thus, in the automata, all states are final and labels represent value assignments to communication channels (*e.g.,* `emgStop`,



Transition Name	Label
bmpFwd	emgStp = False, bump = True / lMot = FORWARD, rMot = FORWARD
fwd	emgStp = False, bump = False / lMot = FORWARD, rMot = FORWARD
stp	emgStp = False, bump = False / lMot = STOP, rMot = STOP
left1	emgStp = False, bump = True / lMot = FORWARD, rMot = STOP
left2	emgStp = False, bump = False / lMot = FORWARD, rMot = STOP
bmpBwd	emgStp = False, bump = True / lMot = BACKWARD, rMot = BACKWARD
emgOff1	emgStp = False, bump = True / lMot = FORWARD, rMot = FORWARD
emgOff2	emgStp = False, bump = False / lMot = FORWARD, rMot = FORWARD
emg1	emgStp = True, bump = False / lMot = STOP, rMot = STOP
emg2	emgStp = True, bump = True / lMot = STOP, rMot = STOP
anyBump	Any possible input/output combination where bump = True
*	Any possible input/output combination

Figure 2: Two reactive automata models adapted from [25].

bump, lMot). In such reactive automata, for each state and each input channel assignment, there must exists a transition with a label that subsumes the input channel assignment and originates from the state. In this example, the input channels are emgStp and bump of type Bool. Both channels can either be assigned to True or False. The robot is equipped with an emergency button and a touch sensor on its front. It is capable of moving forward as well as turning in any direction. The robot should drive forward until hitting a wall (indicated by an incoming signal on the touch sensor). After hitting a wall, the robot should drive a little backwards, turn by 90 degree in any direction, and then move forward, again. While the emergency button is pressed, the robot should stop moving and not perform any action until the button is released. The automaton *impl* in Fig. 2 models the robot's behavior.

During development, the engineering team receives a new requirement: when turned on, the robot must not start moving until its front touch sensor has been pressed. The engineering team decides to check whether the implementation already satisfies this property. If not, the team needs to change the implementation accordingly. The engineering team first formalizes the specification by creating the automaton *spec* as depicted in Fig. 2. With this, the implementation satisfies the property if, and only if, the traces accepted by the implementation's automaton *impl* is a subset of the traces accepted by the specification's automaton *spec*. Using semantic differencing, the team detects the automaton *impl* admits a trace that is not possible in the automaton *spec*. Therefore, refinement does not hold and the implementation does not satisfy the specification. The team decides to use our framework for calculating a shortest change sequence to transform the implementation such that it satisfies the requirement. It turns out that first adding a loop to state idle with label emg1 and then removing the transition

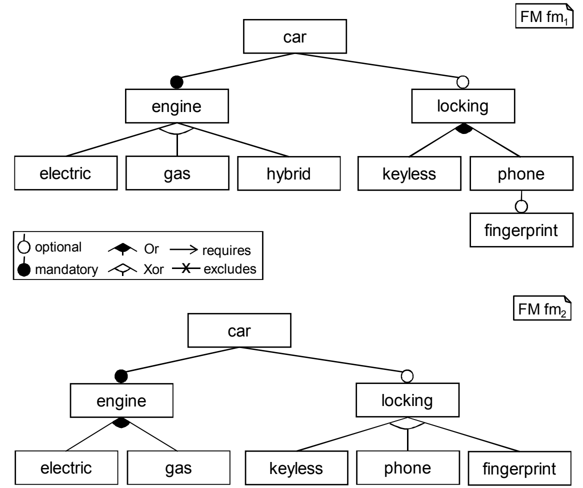


Figure 3: Two feature models adopted from [20].

from state idle to state stopped with label emg1 is such a shortest sequence (Adding the transition is required for reactiveness). The engineering team applies the fully automatically calculated change operations and obtains a new and correct implementation.

2.3 Understanding a Feature Model Evolution

Fig. 3 depicts an example from [20], which is inspired by a similar FM example from [5]. The models describe possible configurations for the engine and locking systems of a car. The FM describing the planned possible configurations is fm_1. During development, the engineering team decides to perform two changes to the FM, one to increase the model's understandability and another for adapting to changed requirements. The resulting model is fm_2. The change for increasing understandability is to remove the feature hybrid and to change the exclusive choice of a single engine system to an alternative. With this, simultaneously choosing the features electric and gas now models a hybrid engine system. The changed requirements state that exactly one locking system must be selected in each configuration and that the fingerprint locking system does not require the phone locking system anymore. Thus, the team moves the fingerprint feature to the group of the locking feature and makes the choice of a locking system exclusive.

Another engineer, who has not been involved in the changes, is informed that the number of possible locking system combinations has been decreased. With this information, she expects the evolved FM to be a refinement of the original. Using semantic differencing, the engineer detects that there are valid configurations of fm_2 that are no valid configurations of fm_1. The FM fm_2 is thus no refinement of fm_1. She tries to understand why this is the case and decides to identify the cause by investigating a shortest change sequence required to transform fm_2 into a refinement of fm_1. She uses our framework and finds out that at least two changes are required. She gets presented that changing the subgroup of feature engine to an exclusive alternative and moving feature fingerprint below phone transform fm_2 into a refinement of fm_1. With this information, she understands that the change to

engine's subgroup caused that the features electric and gas can now be chosen simultaneously, which was not possible before. She further understands that choosing feature fingerprint does not require choosing feature phone anymore.

3 PRELIMINARIES

This section presents a general and language independent notion of modeling language that explicitly captures the possibility to model underspecification via a set-based semantics mapping [9–11, 19, 20]. Inspired by the Diffuse framework [19, 20], we then describe syntactic differences between models by sequences of change of operations (cf. Sec. 3.2). Furthermore, Sec. 3.3 recapitulates details about semantic differencing.

Notation. Let A be an arbitrary non-empty set. We denote by A^* the set of all finite sequences (words) over the set A where $\varepsilon \in A^*$ denotes the empty sequence. We denote by $s \cdot t$ the concatenation of two sequences $s, t \in A^*$. The length of a sequence $s \in A^*$ is denoted by $|s|$. The prefix relation $\sqsubseteq$ over sequences is defined as usual by $\forall s, t \in A^* : s \sqsubseteq t \Leftrightarrow \exists u \in A^* : s \cdot u = t$. For every sequence $s \in A^*$ and for every $i \in \mathbb{N}$ with $0 \leq i < |s|$, the expression $s.i$ denotes the $(i+1)$-th element of the sequence s. Similarly, $s{\downarrow}i$ denotes the prefix of the sequence s with length $0 \leq i \leq |s|$ where $s{\downarrow}0 = \varepsilon$. The result from removing the first element of a sequence $s \in A^*$ with $|s| \geq 1$ is denoted by $rt(s)$. We sometimes treat an element $a \in A$ as a sequence of length one, *e.g.,* for all all $t \in A^*$, we define $t \cdot a$ as the sequence obtained from appending a to the end of t. Similarly, $a \cdot t$ denotes the sequence obtained from prepending a in front of t. For a function f, we write $f : X \to Y$ if f is a total function from X to Y. Similarly, $f : X \rightharpoonup Y$ denotes that f is a partial function from X to Y. We denote by $dom(f) \subseteq X$ the domain of a (partial) function $f : X \rightharpoonup Y$ and write $f(x) = \bot$ iff $x \notin dom(f)$.

3.1 Modeling Language

A *modeling language* $\mathcal{L}$ is a tuple $\mathcal{L} = (M, S, sem)$ where M is a countable set of syntactically correct models, S is a semantics domain, and $sem : M \to \wp(S)$ is a semantics mapping [10]. The semantics domain S is typically a well understood mathematical model. The semantics mapping sem maps each syntactically well-formed model $m \in M$ to its meaning $sem(m)$, which is a subset of the semantics domain. This set-based mapping enables to easily model underspecification when understanding each $s \in sem(m)$ as a possible realization of a model $m \in M$ [11]. A model $m \in M$ is called *consistent* iff $sem(m) \neq \emptyset$, *i.e.,* the model admits at least one realization [11], otherwise it is called *inconsistent.* An inconsistent model contains some contradictory constraints in itself and thus has no meaning (an empty semantics).

3.2 Change Operations

Let $\mathcal{L} = (M, S, sem)$ be a modeling language. A *change operation* for $\mathcal{L}$ is a partial function $o : M \rightharpoonup M$. The function is partial because applying a change operation may result in a syntactically not well-formed model. A sequence of change operations is called *change sequence.* A set of change operations O is called *change operation suite* for $\mathcal{L}$ iff each $o \in O$ is a change operation for $\mathcal{L}$. Let O be a change operation suite for $\mathcal{L}$. For all models $m \in M$ and all change sequences $t \in O^*$, the operator $\oplus : M \times O^* \rightharpoonup M$ for applying change sequences is defined by the following equation:

$$m \oplus t = \begin{cases} m, & \text{if } t = \varepsilon \\ \bot, & \text{if } |t| \geq 1 \land t.0(m) = \bot \\ t.0(m) \oplus rt(t), & \text{otherwise.} \end{cases}$$

Following [19, 20], for all $m \in M$ and $t \in O^*$, we write $m \oplus t \in M$ to denote that $m \oplus t \neq \bot$. Syntactic differencing is decidable for $\mathcal{L}$ with respect to change operation suite O iff there is a computable function $\Delta : M \times M \to O^*$ such that $\forall m, m' \in M : m \oplus \Delta(m, m') = m'$. The sequence $\Delta(m, m')$ is called *syntactic difference* between m and m'. As models are usually finite structures, Δ typically exists when using adequate change operations suites (cf. [2, 4, 20, 27]). There might be infinitely many functions Δ with the above property. Our approach only requires that an arbitrary and fixed Δ is given.

3.3 Semantic Difference and Refinement

The *semantic difference* [21] between two models is the set of elements in the semantics of the one model that are no members of the other model's semantics. For two models $m, m' \in M$, it is formally defined as $\delta(m, m') \stackrel{\text{def}}{=} sem(m) \setminus sem(m')$ [21]. Interpreting the model m as a successor version of the model m', the set $\delta(m, m')$ contains exactly the elements added to the semantics of m' during evolution to m. In this paper we are especially interested in refinements. A model $m \in M$ is called *refinement* of a model $m' \in M$ iff $\delta(m, m') = \emptyset$ [11], *i.e.,* all elements in the semantics of m are included in the semantics of m'. Refinement is said to be decidable iff $\delta(m, m') = \emptyset$ is decidable for all $m, m' \in M$.

4 CHANGE OPERATION PROPERTIES

In the following, let $\mathcal{L} = (M, S, sem)$ denote an arbitrary modeling language and O an arbitrary change operation suite for $\mathcal{L}$ such that refinement and syntactic differencing are decidable for $\mathcal{L}$. Let $\Delta : M \times M \to O^*$ be a syntactic differencing operator.

For two models, we study conditions enabling to compute a change sequence such that applying the sequence to the first model results in a refinement of the second model. Applying such a sequence may lead to an inconsistent model. For repairing a refinement step, computing such a model is often not desired as its semantics is empty, *i.e.,* it has no denotations and thus no useful meaning. Therefore, we are only interested in change sequences that lead to consistent models. This motivates the notion of change sequence that repairs a model towards refining another model:

Definition 1 (Repairs Towards Refining). *A change sequence $t \in O^*$ is said to repair a model $m \in M$ towards refining a model $m' \in M$ iff $m \oplus t \in M \land \emptyset \neq sem(m \oplus t) \subseteq sem(m')$.*

A change sequence $t \in O^*$ is a shortest change sequence that repairs $m \in M$ towards refining $m' \in M$ iff t repairs m towards refining m' and $\forall u \in O^* : (m \oplus u \in M \land \emptyset \neq sem(m \oplus u) \subseteq sem(m')) \Rightarrow |t| \leq |u|$.

Intuitively, the application of a change sequence that repairs a model towards refining another model to the former model results in a consistent model that is a refinement of the latter model.

Notation. Let $m, m' \in M$. We denote by $\mathcal{S}(m, m')$ the set of all shortest change sequences that repair m towards refining m'. Further, $\ell : M \times M \rightharpoonup \mathbb{N}$ is a partial function that maps all models $m, m' \in M$ to the length $\ell(m, m')$ of all shortest change sequences

that repair m towards refining m'. If no shortest sequence exists, *i.e.,* $\mathcal{S}(m, m') = \emptyset$, then $\ell(m, m') = \bot$. Otherwise, if a shortest change sequence exists, *i.e.,* $\mathcal{S}(m, m') \neq \emptyset$, then $\ell(m, m') \in \mathbb{N}$. There may be multiple shortest sequences. Nevertheless, $\ell(m, m')$ is well-defined because all such sequences have the same length:

Lemma 1. *For all models $m, m' \in M$ and all shortest change sequences $t, u \in \mathcal{S}(m, m')$, it holds that $|t| = |u|$.*

Proof. Let $m, m' \in M$ and $t, u \in \mathcal{S}(m, m')$. Suppose that $|t| \neq |u|$. Assume w.l.o.g. $|u| < |t|$. Then, by Def. 1, we have $t \notin \mathcal{S}(m, m')$ because $\emptyset \neq sem(m \oplus u) \subseteq sem(m')$ and $\neg(|t| \leq |u|)$. □

A change sequence that repairs refinement always exists if, and only if, the original model is consistent:

Lemma 2. *Let $m, m' \in M$ be two models. There exists a change sequence that repairs m towards refining m' iff m' is consistent.*

Proof. Let $m, m' \in M$ be two models.
$\Rightarrow$: Assume there exists a change sequence t that repairs m towards refining m'. Then, $\emptyset \neq sem(m \oplus t) \subseteq sem(m')$, which implies $sem(m') \neq \emptyset$. Thus, m' is consistent.
$\Leftarrow$: Assume m' is consistent. Then, it holds $sem(m \oplus \Delta(m, m')) = sem(m') \neq \emptyset$. Thus, $sem(m \oplus \Delta(m, m')) \subseteq sem(m')$ and $sem(m \oplus \Delta(m, m')) \neq \emptyset$, *i.e.,* $\Delta(m, m')$ repairs m towards refining m'. □

If a shortest sequence that repairs a model towards refining another model exists, then its length is bounded by the length of the syntactic difference between the input models:

Lemma 3. *If $t \in O^*$ is a shortest change sequence that repairs $m \in M$ towards refining $m' \in M$, then $|t| \leq |\Delta(m, m')|$.*

Proof. Let $m, m' \in M$ be two models. Assume $t \in O^*$ is a shortest change sequence that repairs m towards refining m'. Using Lemma 2, the existence of t implies that m' is consistent, *i.e.,* $sem(m') \neq \emptyset$. Thus, $sem(m \oplus \Delta(m, m')) = sem(m') \neq \emptyset$, which implies $sem(m \oplus \Delta(m, m')) \subseteq sem(m')$ and $sem(m \oplus \Delta(m, m')) \neq \emptyset$. Therefore, $\Delta(m, m')$ is a change sequence that repairs m towards refining m'. As t is a shortest change sequence that repairs m towards refining m', it holds by Def. 1 that $|t| \leq |\Delta(m, m')|$. □

Lemma 3 shows an upper bound for the length of shortest change sequences that repair refinement. However, there may still be infinitely many change sequences with length less than or equal to $|\Delta(m, m')|$, which hampers computability. In the following, we study a sufficient condition that enables to fully automatically calculate change sequences that repair refinement. To this effect, the following sections first introduce properties of change operations, before presenting a sufficient condition enabling computability of shortest sequences and an algorithm for their computation.

4.1 Operations Inducing Equally Quick Repair

Two syntactically different models $m, m' \in M$ with $m \neq m'$ may have the same semantics $sem(m) = sem(m')$. This is the case, for instance, if the models contain syntactically different elements that have the same meaning.

As a concrete example, let fm_1 be a FM and let p, c be two features in fm_1 such that c is a mandatory child of p. The FM fm_1 is equivalent to the FM fm_2 that contains exactly the elements of fm_1 except that c is an optional child of p in fm_2 and fm_2 contains an implies constraint between p and c. The FMs fm_1 and fm_2 are syntactically different but semantically equivalent. From a constructive viewpoint, adding an implies constraint in a FM fm from a feature p to a feature c where c is an optional child of p has the same effect on the semantics of fm as making c a mandatory child of p. Another example are finite automata: Let A be a finite automaton and let s be a state in A. The automaton A is equivalent to the automaton A' that is equal to automaton A except that all occurrences of state identifier s have been exchanged with a single other state identifier s' not already used in A. This is because A and A' are isomorphic with respect to their state labeling and state labels do not directly influence an automaton's semantics. From a constructive viewpoint, let A be a finite automaton and let s, s' be two state labels not already used in A. Adding the state s to A has the same impact on the semantics of A as adding the state s' to A. As a more complex behavior modeling example, let A be an activity diagram and let Act be an action in A with identifier I and label L. The identifier is used for connecting nodes within the activity diagram with another. A label describes the action executed when visiting an action. The activity diagram A is equivalent to the activity diagram A' that is syntactically equal to A except that all occurrences of identifier I have been exchanged with a single other node identifier I' not already used in A. From a constructive viewpoint, let A be an activity diagram, I, I' be two node identifier not already used in A and let L be a valid label. Then, adding an action with identifier I and label L at some position in A has the same impact on A's semantics as adding an action with identifier I' and label L at the same position in A.

When using adequate change operations suites, the above mentioned change operations are even stronger related to each other: Every further syntactic change to one of the altered models can be mimicked by a syntactic change of the same length to the other, in the sense that the further changed models have the same semantics.

For instance, let fm_1 denote the FM resulting from making the feature c a mandatory child of p and let fm_2 denote the FM resulting from adding the implies constraint from p to c in fm. The deletion of the previously added implies constraint in fm_2 can be mimicked by making the feature c an optional child of feature p in fm_1 to again obtain two equivalent feature models. Any syntactic change applied to fm_1 that does not affect any syntactic element referencing the features c or p can also be applied to the feature model fm_2, and vice versa, to again obtain two equivalent feature models. As another example, let A be a finite automaton and let s, s' be two state labels not already used in A. Let A_1 denote the automaton resulting from adding state s to the automaton A and let A_2 denote the finite automaton resulting from adding state s' to A. Every syntactic change applied to A_1 can be mimicked by a syntactic change to A_2 via modifying every syntactic change to state s such that it effects s' instead of s and vice versa. The further modified automata still remain semantically equivalent. For instance, adding a transition from state s to another state different from s' in A_1 can be mimicked by adding a transition from state s' to the same target state in A_2. Similarly, adding state s' to A_1 can be mimicked by adding state s to A_2 and adding a transition from s to s' in A_1 can be mimicked by adding a transition from s' to s in A_2. The situation is similar in the activity diagram example. As the changes in each example can

be mimicked by each other such that the resulting models have the same semantics, they induce equally quick repairs of the original model towards refining another model:

Definition 2 (Induce Equally Quick Repair). *Let $m, m' \in M$. Two change operations $o, o' \in O$ induce an equally quick repair of m towards refining m' iff $\ell(o(m), m') = \ell(o'(m), m')$.*

Notation. For all $m, m' \in M$ and all $o, o' \in O$, we write $o \sim_m^{m'} o'$ if o and o' induce an equally quick repair of m towards refining m'.

Intuitively, the above states that every shortest change sequence that repairs $o(m)$ towards refining m' has the same length as every shortest change sequence that repairs $o'(m)$ towards refining m'. Thus, for every shortest change sequence that repairs $o(m)$ towards refining m', we can find a shortest change sequence of the same length that repairs $o'(m)$ towards refining m' and vice versa.

Lemma 4. *For all models $m, m' \in M$, the relation $\sim_m^{m'}$ is an equivalence relation.*

Proof. (Sketch.) Let $m, m' \in M$ and let $a, b, c \in O$. Using the reflexivity, symmetry, and transitivity of $=$, we obtain $a \sim_m^{m'} a$, $a \sim_m^{m'} b \Leftrightarrow b \sim_m^{m'} a$, $(a \sim_m^{m'} b \wedge b \sim_m^{m'} c) \Rightarrow a \sim_m^{m'} c$. □

Notation. For all models $m, m' \in M$ and change operations $o \in O$, we denote by $[o]_m^{m'} \stackrel{\text{def}}{=} \{o' \in O \mid o \sim_m^{m'} o'\}$ the equivalence class of o with respect to $\sim_m^{m'}$. For $P \subseteq O$, we denote by $P/\sim_m^{m'} \stackrel{\text{def}}{=} \{[a]_m^{m'} \mid a \in P\}$ the quotient of P with respect to $\sim_m^{m'}$.

The next section introduces the notion of change operation that defers repair. Then, Sec. 5 shows that considering a single representative of each equivalence class is sufficient during iterative computation of shortest change sequences that repair refinement.

4.2 Change Operations That Defer Repair

Various change sequences with different lengths may transform a model to the same model. Different change sequences may also transform a model to syntactically different models that are semantically equivalent. Such change sequences may drastically differ in their lengths. As the goal is to identify shortest change sequences that repair refinement, the longer repairing sequences are not of interest. Thus, during stepwise computation of sequences that repair refinement, the identification and exclusion of the longer sequences is highly desired before completely computing them. In some cases, investigating a prefix of a change sequence already suffices to determine whether the complete sequence is no shortest sequence that repairs refinement. For example, the application of a change operation may add elements to a model's semantics that are not part of the semantics of the original model. These elements must be removed again from the semantics by subsequent change operations to ultimately obtain a refinement. In some cases, the application of such a change operation is necessary to enable the application of other change operations. However, when the application is not necessary, then applying the change operation *defers* the model's repair towards refining the other model.

As a concrete example, let fm_1 and fm_2 be two feature models and let f be a feature of fm_1. Assume the goal is to compute a shortest change sequence that repairs fm_1 towards refining fm_2. Let t be a change sequence that repairs fm_1 towards refining fm_2 and contains an operation o that solely adds an implies-constraint between f and f (*i.e.,* $f \Rightarrow f$). With this, the change operation adds a tautology. Therefore, under consideration of an adequate change operation suite, it is possible to shorten the sequence t by simply deleting the operation that adds the implies-constraint. The result is another change sequence that repairs fm_1 towards refining fm_2. Let fm be the feature model obtained from applying the prefix of t that ends directly before o to fm_1. For any change sequence that starts with the operation o and repairs fm towards refining fm_2, there exists another, shorter sequence such that the application of both sequences to the feature model fm result in semantically equivalent feature models. The operation o defers the repair of fm towards refining fm_2.

As another example, let A_1 and A_2 be two finite automata and let l be a transition label not occurring in A_2. Assume the goal is to compute a shortest change sequence that repairs A_1 towards refining A_2. Let t be a change sequence that contains an operation o solely adding a transition with label l to A_1. Assume t repairs A_1 towards refining A_2. Let A_3 denote the result from applying t to A_1. Then, the transition added by o is not part of any path from an initial state to a final state in A_3 because otherwise the result from applying t to A_1 would be no refinement of A_2. Thus, the semantics of the automaton obtained from deleting the transition from A_3 is equivalent to A_3. Therefore, under consideration of an adequate change operation suite, the change sequence obtained from deleting the change operation o from t is another sequence that repairs A_1 towards refining A_2. Thus, during iterative computation of a shortest change sequence, it is reasonable to not consider the operation o adding the transition with label l. The reason is similar to the feature model example above: Let A be the automaton obtained from applying the prefix of t that ends directly before o to A_1. For any change sequence that starts with the operation o and repairs A towards refining A_2, we can find another, shorter sequence such that the application of both sequences to the automaton A result in semantically equivalent automata. The operation o defers the repair of A towards refining A_2. More formally:

Definition 3. *Let $m, m' \in M$ be two models. A change operation $o \in O$ defers the repair of m towards refining m' iff*
$\forall t \in \mathcal{S}(m, m') : |t| \geq 1 \Rightarrow t.0 \neq o.$

Notation. $D_m^{m'}$ denotes the set of all change operations that defer the repair of $m \in M$ towards refining $m' \in M$.

Every change sequence that repairs refinement and starts with a change operation that defers the repair is never a shortest change sequence that repairs refinement. Vice versa, shortest change sequences that repair a model towards refining another model never start with operations that defer the repair of the former towards refining the latter.

5 REPAIRING REFINEMENT

The following shows that considering subsets of change operations that induce equally quick repairs during an iterative computation approach to shortest change sequences that repair refinement is sufficient. The requirement on each subset is that it contains at least one representative of each non-deferring equivalence class characterized by the "induce equally quick repair" equivalence relation:

In the following, we assume for all $m, m' \in M$, we are given a set $O(m, m') \subseteq O$ satisfying $O(m, m')/\sim_m^{m'} \cup D_m^{m'}/\sim_m^{m'} = O/\sim_m^{m'}$. Stated differently, $O(m, m')$ contains at least one element $e \in C$ of each equivalence class $C \in O/\sim_m^{m'}$ satisfying $C \cap D_m^{m'} = \emptyset$. Although O is typically an infinite set in practice, the finite sets $O(m, m')$ often exist (cf. Sec. 6 for three example modeling languages).

If there exists a shortest change sequence that repairs a model towards refining another model, then there also exists a shortest change sequence that only contains change operations from the sets $O(m, m')$ and repairs the model towards refining the other model:

Theorem 1. *Let $m, m' \in M$ be two models. If there exists a shortest change sequence that repairs m towards refining m', then there exists a shortest change sequence $t \in O^*$ that repairs m towards refining m' with $t.i \in O(m \oplus t{\downarrow}i, m')$ for all $i \in \mathbb{N}$ with $0 \leq i < |t|$.*

Proof. We show a more general property: For all change sequences $t \in O^*$ and all models $m, m' \in M$, if t is a shortest change sequences that repairs m towards refining m', then there exists a change sequence $u \in O^*$ such that $\emptyset \neq sem(m \oplus u) \subseteq sem(m')$ and $|u| = |t|$ and $u.i \in O(m \oplus u{\downarrow}i, m')$ for all $i \in \mathbb{N}$ with $0 \leq i < |u|$.

The proof is by induction over the lengths of change sequences.

$|t| = 0$: Let $m, m' \in M$ be two models. Assume ε is a shortest change sequence that repairs m towards refining m'. As $|\varepsilon| = 0$, the statement is trivially true for $u = \varepsilon$.

Let $n \in \mathbb{N}$. Assume the statement holds for all shortest change sequences t with $|t| \leq n$.

$|t| = n + 1$: Let $m, m' \in M$ be two models. Assume t is a shortest change sequence that repairs m towards refining m' and $|t| = n + 1$. Def. 3 guarantees that $t.0 \notin D_m^{m'}$. Thus, $[t.0]_m^{m'} \notin D_m^{m'}/\sim_m^{m'}$. Therefore, $[t.0]_m^{m'} \in O(m, m')/\sim_m^{m'}$ because $[t.0]_m^{m'} \in O/\sim_m^{m'}$ and $O(m, m')/\sim_m^{m'} \cup D_m^{m'}/\sim_m^{m'} = O/\sim_m^{m'}$. Now let $o \in O(m, m')$ such that $[o]/\sim_m^{m'} = [t.0]/\sim_m^{m'}$. As o and $t.0$ induce an equally quick repair of m towards refining m', there exists a shortest change sequence v that repairs $o(m)$ towards refining m' with $|v| = |t| - 1$.

Using the induction hypothesis, we obtain that there exists a change sequence w with $|w| = |v|$ such that $\emptyset \neq sem(o(m) \oplus w) \subseteq sem(m')$ and $w.i \in O(o(m) \oplus w{\downarrow}i, m')$ for all $i \in \mathbb{N}$ with $0 \leq i < |w|$.

To conclude, we have that $|o \cdot w| = 1 + |w| = 1 + |v| = |t|$ and $(o \cdot w).i \in O(m \oplus (o \cdot w){\downarrow}i, m')$ for all $i \in \mathbb{N}$ with $0 \leq i < |o \cdot w|$ and $\emptyset \neq sem(m \oplus o \cdot w) \subseteq sem(m')$. □

Our approach for computing shortest change sequences that repair refinement relies on iteratively computing shortest sequences that repair intermediate models towards refining the original. The search space is reduced by considering single representatives per equivalence class characterized by the "induce equally quick repair" relation and ignoring change operations that defer the repair of intermediate models. Formally, the condition requires $O(m, m')$ to be finite for all models m, m'. From a practical viewpoint, this requires an implementation to be able to calculate the set $O(m, m')$ for every two models $m, m' \in M$. It should be noted that the sets $O(m, m')$ are not unique and it suffices to compute arbitrary but fixed sets satisfying the described properties.

For all models $m, m' \in M$, let $C(m, m')$ denote the set of all change sequences $t \in O^*$ such that $|t| \leq |\Delta(m, m')|$ and $t.i \in O(m \oplus t{\downarrow}i, m')$ for all $i \in \mathbb{N}$ with $0 \leq i < |t|$. Combining Thm. 1 with Lemma 2 guarantees that the set $C(m, m')$ contains a shortest change sequence that repairs m towards refining m' iff m' is consistent. The following shows that assuming each set $O(m, m')$ to be finite implies that $C(m, m')$ is finite: We can represent the elements of $C(m, m')$ in a rooted tree $T(m, m') = (V, r, E)$ with root $r = \varepsilon$ where each node $v \in V$ corresponds to a sequence contained in $C(m, m')$, *i.e.,* $V = C(m, m')$. Two nodes $v, w \in V$ are connected in $T(m, m')$ iff the concatenation of the sequence v with a change operation of $O(m \oplus v, m')$ is equal to w, *i.e.,* $E = \{(v, w) \in V \times V \mid \exists o \in O(m \oplus v, m') : v \cdot o = w\}$.

Theorem 2. *Let $m, m' \in M$ be two models and let $T(m, m') = (V, r, E)$. For all $t \in O^*$ it holds that $t \in C(m, m')$ iff there exists a path from r to t in $T(m, m')$.*

Proof. Let $m, m', T(m, m')$, and t be given as above.

"$\Rightarrow$": Assume $t \in C(m, m')$. Then, $|t| \leq |\Delta(m, m')|$ and $t.i \in O(m \oplus t{\downarrow}i, m')$ for all $i \in \mathbb{N}$ with $0 \leq i < |t|$. It directly follows by definition of $C(m, m')$ that $t{\downarrow}i \in C(m, m')$ for all $0 \leq i \leq |t|$. Thus, by definition of V, it holds that $t{\downarrow}i \in V$ for all $0 \leq i \leq |t|$. Further, as $t.i \in C(m \oplus t{\downarrow}i, m')$ for all $0 \leq i < |t|$, by definition of E, we have $(t{\downarrow}i, t{\downarrow}(i + 1)) \in E$ for all $0 \leq i < |t|$. This shows there exists a path from $r = \varepsilon$ to t in $T(m, m')$.

"$\Leftarrow$": Assume there exists a path from r to t in $T(m, m')$. Thus, $t \in V$, which is equivalent to $t \in C(m, m')$. □

With this, if $T(m, m')$ is finite, it is possible to enumerate all change sequences in $C(m, m')$ by enumerating all states reachable from the root in $T(m, m')$. This is possible, for instance, by using a breadth-first search. The following shows, the assumption that $O(m, m')$ is finite for all $m, m' \in M$ guarantees that $T(m, m')$ is also finite for all $m, m' \in M$.

Theorem 3. *If for all models $n, n' \in M$ the set $O(n, n')$ is finite, then for all $m, m' \in M$ the tree $T(m, m')$ is finite.*

Proof. Assume for all models $n, n' \in M$ the set $O(n, n')$ is finite. Let $m, m' \in M$. Suppose $T(m, m') = (V, r, E)$ is infinite. As for all models $n, n' \in M$, the set $O(n, n')$ is finite, the tree $T(m, m')$ is finitely branched, *i.e.,* each node has only finitely many successors. Hence, by König's Lemma [14], the tree $T(m, m')$ contains an infinite branch. Let $p = v_0, e_0, v_1, e_1, \ldots$ be an infinite branch in $T(m, m')$. By definition of E it holds that $v_i \sqsubset v_{i+1}$ for all $i \in \mathbb{N}$. Thus, $|v_i| < |v_{i+1}|$ for all $i \in \mathbb{N}$. As p is infinite and the length of successively visited nodes in p is monotonically increasing, we have that for all $j \in \mathbb{N}$ there exists $i \in \mathbb{N}$ such that $j < |v_i|$. Hence, there exists a reachable state $v \in V$ such that $|v| > |\Delta(m, m')|$. This contradicts $V = C(m, m')$ as by definition of $C(m, m')$ it holds that $|v| \leq |\Delta(m, m')|$ for all $v \in C(m, m')$. □

This shows, if for all $n, n' \in M$ the set $O(n, n')$ is finite, then for all models m, m', a shortest change sequence that repairs m towards refining m' is computable by searching the finite tree $T(m, m')$.

Algo. 1 is a procedure for computing a shortest change sequence that repairs refinement. The assumption that guarantees the algorithm's termination is that $O(n, n')$ is finite for all $n, n' \in M$. The algorithm takes two models m and m' as input. It performs a breadth-first search on the tree $T(m, m')$. It returns $\bot$ iff no sequence that repairs refinement exists (ll. 1-3). This is the case if, and only if, the input model m' is not consistent (cf. Lemma 2). Variable

Algorithm 1 Computing a shortest change sequence for repairing refinement under assumption $O(n, n')$ is finite for all $n, n' \in M$.

Input: Two models $m, m' \in M$.
Output: Shortest change sequence $t \in O^*$ repairing m with respect to m', if one exists, and $\perp$ otherwise.

```
1:  if sem(m') = ∅ then
2:      return ⊥
3:  end if
4:  define T as empty queue of O*
5:  add ε to T
6:  while T not empty do
7:      t = T.dequeue()
8:      if m ⊕ t ∈ M ∧ ∅ ≠ sem(m ⊕ t) ⊆ sem(m') then
9:          return t
10:     end if
11:     if m ⊕ t ∈ M ∧ |t| < |Δ(m, m')| then
12:         for all o ∈ O(m ⊕ t, m') do
13:             T.enqueue(t · o)
14:         end for
15:     end if
16: end while
```

T queues the most recently explored nodes (l. 4). The algorithm starts in the root node ε (l. 5) and visits the tree's other nodes in increasing size of the lengths encoded by the nodes' change sequences (ll. 6-16). In each step of the outer loop, the algorithm first fetches the oldest recently explored node t (l. 7). In each step, this is the node labeled with the shortest recently explored change sequence. The algorithm returns the sequence in case it repairs m towards refining m' (ll. 8-10). Otherwise, in case the sequence's length is less than the upper bound for a shortest sequence of change operations repairing m towards refining m' (l. 11), which is given by $|\Delta(m, m')|$ (cf. Lemma 3), the algorithm proceeds as follows: It concatenates all operations relevant from the intermediate model $m \oplus t$ to the current sequence and enqueues the newly obtained sequences to the queue T (ll. 12-14). The algorithm terminates as soon as it finds a shortest change sequence that repairs m towards refining m'. The theorems above ensure the existence of this sequence in case m' is consistent (cf. Lemma 2) and thus the algorithm's termination. The algorithm's running time is exponential in the length of a shortest change sequence that repairs m towards refining m'.

6 INSTANTIATIONS OF THE FRAMEWORK

This section describes instantiations of the frameworks to a FM, an automaton, and an AD language. The section contains minimal explanations that show the applicability of this paper's approach. Sec. 2 presents examples of results produced by prototype implementations of the instantiations.

6.1 Feature Model Language

FMs are widely used for modeling possible product configurations. We use a similar syntax as defined in [5] and treat feature models as trees with the usual cross tree constraints but without propositional constraints. We denote the set of all valid FMs by FM. The semantics of a FM is the set of all possible configurations it describes. There exist translations from FMs to propositional logical formulas for determining the configurations (*e.g.,* [5]). Let Con denote the set of all possible configurations, *i.e.,* the set of all sets of features. Then, the language is defined as (FM, Con, sem_{FM}). Refinement checking of FMs is decidable [1]. We adapt the syntactic change operations defined in [28] for:

(1) creating/deleting a feature without children (creation of a feature in an empty model marks the feature as the root),
(2) creating/deleting cross tree constraints (implies/excludes),
(3) moving a feature to a new parent,
(4) changing the type (and/or/xor) of a group,
(5) making a feature optional or mandatory.

The operations described in 2–5 are uncritical in the sense that they are only partially defined for FMs that contain the referenced elements. With this, for every FM fm, only finitely many operations described by 2–5 are applicable to fm and the applicable operations can be calculated from fm. Similarly, the feature deletion operations (described in 1) are only defined for models containing the features. The calculation of the above mentioned change operations is straightforward as a FM is always a finite structure. For example, for each possible feature f there exists an operation $d_f : FM \rightharpoonup FM$ for deleting the feature f. The domain $dom(d_f)$ is the set of all FMs in which the feature f exists and has no sub-features. Thus, the set of feature deletion operations applicable to a FM fm contains exactly one operation for each feature in fm without children.

The set of operations for creating features (described in 1) is critical since there are infinitely many feature creation operations applicable to each feature model (assuming an infinite set of feature names). Specifically, for each feature f there exists an operation $a_f : FM \rightharpoonup FM$ for adding feature f. The domain of a_f is the set of all feature models in which f is not present. Let fm and fm' be two feature models. We argue that for all features f, g that do neither exist in fm_1 nor in fm_2, the operations a_f and a_g induce an equally quick repair of fm_1 towards refining fm_2: Assume there exists a shortest change sequence t that repairs $a_f(fm_1)$ towards refining fm_2. We define u as the change sequence of length $|t|$ obtained from replacing each change operation in t affecting f by the corresponding operation affecting g and vice versa. The addition of a cross tree constraint between f and g, for instance, is replaced by the same cross tree constraint between g and f, *e.g.,* "f implies g" is replaced by "g implies f". It is easy to see by construction that $fm_1 \oplus a_f \oplus t \neq \perp$ implies $fm_1 \oplus a_g \oplus u \neq \perp$. As the features f and g do not exist in fm_2, there are no configurations in $fm_1 \oplus a_f \oplus t$ and $fm_1 \oplus a_g \oplus u$ that contain f or g, *i.e.,* the features do either not exist or they are dead. It is easy to see that $fm_1 \oplus a_g \oplus u$ contains g iff $fm_1 \oplus a_f \oplus t$ contains f and vice versa. In case any of the features exists in one of the models, the induced propositional logical formula [5] is equivalent to the formula obtained by replacing all occurrences of the features' corresponding variables by the constant $false$. This holds because the features are dead. It is easy to verify that by definition of u, the transformed formulas for $fm_1 \oplus a_f \oplus t$ and $fm_1 \oplus a_g \oplus u$ are equivalent. Analogously, one can show that for each shortest change sequence t repairing $a_g(fm_1)$ towards refining fm_2, there exists a change sequence u with $|u| = |t|$ such that u repairs $a_f(fm_1)$ towards refining fm_2. We can conclude $\ell(a_f(fm_1), fm_2) = \ell(a_g(fm_1), fm_2)$. Hence, it

suffices to solely consider exactly one feature addition operation for each feature that does neither exist in fm_1 nor in fm_2. The above argumentation shows a finite set $O(fm_1, fm_2)$ is computable for all $fm_1, fm_2 \in FM$. Thus, our approach is applicable.

6.2 A Language for Reactive Safety Automata

A finite time-synchronous port automaton (TSPA) is a model for describing the behavior of a reactive system [3]. From this paper's viewpoint TSPAs can be interpreted as reactive finite safety automata where transition labels are finite functions. Each function models a channel assignment, *i.e.,* a mapping from communication channel identifiers to messages. Channels are partitioned into input and output channels [3]. In safety automata, all states are final. Reactivity requires that for each possible input channel assignment and each state, there exists at least one transition originating from the state such that the input channel assignment is a subset of the transition's label. The semantics of a TSPA are the possible behaviors (communication histories) that it describes. When interpreting the TSPA as a finite automaton, the semantics is the language accepted by the automaton. We use the syntax and behavior semantics for TSPAs as defined in [3]. We denote the set of all possible TSPAs by $TSPA$ and the set of all possible behaviors by Beh. Then, the TSPA modeling language is defined by $(TSPA, Beh, sem_{tspa})$. Refinement checking of TSPAs is decidable [3]. We use TSPA change operations that are inspired by [26] for:

(1) creating a new state,
(2) adding an input/output channel to a TSPA without states,
(3) creating a transition with specific label between states,
(4) deleting a transition if reactivity is preserved,
(5) deleting a state if reactivity is preserved,
(6) changing the initial state,
(7) deleting a channel from a TSPA without states.

The first operation creates a maximally underspecified state that contains a self-loop for each possible transition label (channel valuation). Changing a TSPA's set of channels (2, 7) completely changes the set of the automaton's transition labels (cf. [3]). New channels are only allowed to be added or deleted if the input TSPA does not contain any states (2, 7). This eliminates the necessity for changing all transition labels when adding a new communication channel to a non-empty TSPA. The constraint for transition deletion operations (4) is necessary because TSPA well-formedness requires reactivity. Deleting a state (5) also deletes all transitions originating from the state or leading to the state. Thus, states are only deletable if the resulting TSPA is reactive (5). Deleting the initial state is only possible if the TSPA contains no other states. Only existing states may be marked as initial (6).

The operations described in 3–7 are uncritical because they are only partially defined for TSPAs that contain the referenced elements. The set of change operations described by 3–7 that are applicable to a TSPA is finite and can be calculated from the TSPA. On the other hand, the set of operations for creating states and adding communication channels (1,2) are critical: Assuming infinitely many state and channel identifiers, there are infinitely many of such change operations that are applicable to a TSPA. A similar argumentation as in Sec. 4.1 shows that for all state identifiers s, t not already present in a TSPA A, the operations $addState_s$ and $addState_t$ induce an equally quick repair towards refining any other TSPA. Thus, for calculating a shortest change sequence for repairing refinement, it suffices to consider a single state addition operation in each computation step. A necessary condition for a TSPA A to refine a TSPA B is that A does not use channel identifiers that are not used in B. Thus, any channel addition operation (2) that adds a channel not present in B defers the repair of A towards refining B: Let t be a change sequence starting with a channel addition operation adding a channel c that is not present in B such that $A \oplus t$ is a refinement of B. Then, the channel c added by $t.0$ needs to be removed by a channel deletion operation contained in t later because otherwise $A \oplus t$ would be no refinement of B. Let $1 \leq i < |t|$ be the index of this channel deletion operation. We define the change sequence u: The sequence u starts with all channel addition and deletion operations occurring in t up to and excluding index i that reference channels different from c. The rest of u is equal to the suffix of t starting with index $i + 1$ (or ε in case $|t| = i + 1$). By definition, channel addition and removal operations are only applicable to TSPAs without states. Thus, A does not contain any states as $t.0$ is applicable to A. From this, it is easy to see that u is applicable to A and $A \oplus u = A \oplus t$. We further have that $|u| \leq |t| - 2$ as u does not contain the addition and deletion operations of c that are part of t. As $|u| < |t|$, we can conclude that t is no shortest change sequence that repairs A towards refining B. This shows that any change operation adding a channel not existing in B defers the repair of A towards refining B. Thus, it suffices to solely consider channel addition operations adding channels that exist in B, which are always finitely many. The above argumentation shows the applicability of this paper's approach.

6.3 Activity Diagram Language

Activity diagrams are widely used in the business process modeling domain for describing workflows. We use a similar syntax and operational semantics as defined in [23]. The following two well-formedness rules also apply [16]: each AD contains exactly one initial node and exactly one final node. Further, each node is always part of a path from the initial node to the final node. We reuse the semantic differencing operator for refinement checking as defined in [22]. We define the activity diagram language as (AD, T, sem_{AD}) where AD describes the set of all well-formed activity diagrams and T denotes the set of all execution traces. We adapt preexisting change operations for business process models [15, 16] for:

(1) inserting an action between two succeeding nodes,
(2) deleting an action,
(3) moving an action between two succeeding nodes,
(4) deleting a fragment (parallel, alternative, cyclic, etc.),
(5) inserting a fragment between two succeeding nodes,
(6) moving a fragment between two succeeding nodes,
(7) converting the type of a fragment to another type.

All the operations described in 1–6 also reconnect the control flow accordingly [15, 16]. Only finitely many of the operations described in 2–7 are applicable to each AD. Computation of the operations is straight-forward from the nodes and structure of an AD. The operations described in 1 are critical when assuming an infinite set of possible action labels. Then, infinitely many action insertion operations are applicable to each AD. However, for all ADs ad_1

and ad_2 and all action labels l not occurring in ad_2, each action insertion operation adding an action with label l between two nodes in ad_1 defers the repair of ad_1 towards refining ad_2. This is the case because adding such an action adds an execution trace to ad_1 that is no execution trace of ad_2. As all nodes are always on a path from the initial node to a final node, the action must be removed again to repair ad_1 towards refining ad_2. Omitting the operations that add and delete the action with label l leads to a shorter sequence that repairs ad_1 towards refining ad_2. Thus, the addition operation defers the repair of ad_1 towards refining ad_2. Therefore, it suffices to consider action insertion operations adding actions that are labeled as a node in ad_2. As activity diagrams are finite structures, ad_2 contains only finitely many action labels and thus there are only finitely many such action insertion operations. This argumentation shows the applicability of our approach.

7 RELATED WORK AND DISCUSSION

The related work described in [20] provides an overview of syntactic and semantic differencing approaches. These works are also related to ours. Syntactic differencing approaches (e.g. [2, 12, 13, 15, 16, 27, 28]) do not consider the impact of syntactic changes on a model's semantics. However, they provide a fundamental basis for frameworks combining semantic and syntactic differencing in form of change operations. Semantic differencing approaches (e.g. [1, 3, 17, 18, 22, 24]) provide the other required fundamental basis. They reveal the semantic differences of models but are not concerned with syntactic differences. If a semantic difference exists, semantic differencing approaches usually provide a concrete proof in form of a diff witness. However, semantic differencing does not reveal the syntactic differences that cause the semantic difference. It is further hard (often even impossible) to manually detect all syntactic differences responsible for the models' semantic differences from a single diff witness.

To the best of our knowledge, this paper is the first approach that aims at computing syntactic model changes such that one model becomes a refinement of another model while keeping the number of model changes as small as possible. The following summarizes related work combining syntactic and semantic differencing and approaches that are suited to support the repair of refinement steps.

Diffuse [19, 20] is the first language independent framework combining semantic with syntactic differencing. This paper and Diffuse share the same fundamental definition of modeling language [9]. Diffuse and this work have slightly different notions of syntactic difference. In this paper, syntactic differences are sequences of change operations, whereas Diffuse describes them with partially ordered sets of change operations [20]. This paper's description is simpler and easier to handle in formal proofs, whereas Diffuse's description is more compact in some circumstances. Our work is easily integrable into the Diffuse framework: Every linearization of a set of change operations is a change sequence. Vice versa, every change sequence is interpretable as a partially ordered set of change operations. Diffuse introduces the notions necessary, exhibiting, and sufficient sets of change operations [20]. Each of the three notions relates two models, a diff witness, and a concrete syntactic difference between the models to a subset of the syntactic difference. Combining our work with Diffuse may be interesting for lifting the three notions to abstract from a concrete syntactic difference. For instance, it may be interesting to compute whether there exists a generally necessary change operation for a witness. Diffuse enables to determine which change operation should not have been applied to avoid a specific witness. However, this does not reveal how to obtain a refinement. Further, omitting changes to avoid a single witness does not guarantee to obtain a refinement. The analysis is oriented backwards. In contrast, our approach is forward oriented and computes what needs to be done to definitely obtain a refinement.

There exist enumerative and non-enumerative approaches to semantic differencing [18]. The above mentioned approaches are all enumerative in the sense that they compute a single witness or a finite set of witnesses as concrete proof for semantic differences. Non-enumerative semantic differencing approaches [7] do not calculate witnesses. Instead, they compute an aggregated description that summarizes semantic differences (not necessarily all) between the input models. Non-enumerative approaches have been applied to feature models and automata [7] as well as to class diagrams [6]. As an aggregated description contains more information than a single witness, it is more suited to manually detect the syntactic elements causing the models' semantic differences than a single witness. Hence, the model describing the semantic difference better facilitates manual detection of syntactic changes required to repair refinement. However, also with existing non-enumerative approaches, the computation of syntactic changes that repair refinement is not automated as in our approach. The combination of this paper's method with non-enumerative approaches is interesting: One could first use this paper's approach to calculate a shortest change sequence that repairs refinement. Afterwards, one could investigate whether the changes are reasonable based on the output of a non-enumerative semantic differencing result.

8 CONCLUSION

This paper revealed requirements on a modeling language and its change operations that are sufficient to guarantee computability of change sequences that repair refinement. The requirements build on the notions of change operations that induce an equally quick repair towards refinement and change operations that defer repair. Both notions are defined in context of two concrete models, *i.e.,* in context of repairing the one model towards refining the other model. The "induce equally quick repair" relation is an equivalence relation. Operations that defer repair are not part of shortest change sequences that repair refinement. Based on this, the main result is the following: if for all pairs of models, there exists a finite set containing at least one element of each equivalence class that does not contain operations that defer repair, then computing a shortest change sequences that repairs refinement is possible. The result is a fully automated procedure that computes syntactic changes that definitely transform a model to a refinement of another model. This ultimately facilitates developers in identifying and fixing errors introduced in failed model refinement steps. The approach is limited to modeling languages where refinement is decidable. Further, the change operation equivalence classes and the operations that defer repair must be identified manually for each individual modeling

language. This task may be very challenging and requires the modeling language to exhibit a special kind of symmetry with respect to the models' syntax and the semantics mapping. We have applied the language independent framework to three concrete modeling languages. As the computational complexity for computing repairing sequences is high, language specific algorithm adjustments might be desired in practice. However, it is often not even clear whether automatic computation of repairing sequences is possible. In these cases, applying this paper's results to a concrete modeling language provides the evidence of feasibility.

REFERENCES

[1] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. 2012. Feature Model Differences. In *CAiSE - 24th International Conference on Advanced Information Systems Engineering - 2012*. Gdańsk, Poland.

[2] Marcus Alanen and Ivan Porres. 2003. Difference and Union of Models. In *Modeling Languages and Applications: 6th International Conference, San Francisco, CA, USA, October 20-24, 2003. Proceedings.*

[3] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*. IEEE, 145–154.

[4] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. 2007. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology* 6, 9 (2007).

[5] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *11th International Software Product Line Conference (SPLC 2007)*. 23–34.

[6] Uli Fahrenberg, Mathieu Acher, Axel Legay, and Andrzej Wąsowski. 2014. Sound Merging and Differencing for Class Diagrams. In *Fundamental Approaches to Software Engineering.*

[7] Uli Fahrenberg, Axel Legay, and Andrzej Wąsowski. 2011. Vision Paper: Make a Difference! (Semantically). In *Model Driven Engineering Languages and Systems.*

[8] Christian Gerth, Jochen M. Küster, Markus Luckey, and Gregor Engels. 2010. Precise Detection of Conflicting Change Operations Using Process Model Terms. In *Model Driven Engineering Languages and Systems.*

[9] David Harel and Bernhard Rumpe. 2000. *Modeling Languages: Syntax, Semantics and All That Stuff (Part I: The Basic Stuff)*. Technical Report MCS00-16. Mathematics & Computer Sience, Weizmann Institute Of Sience.

[10] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer* 37, 10 (2004), 64–72.

[11] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2007. An Algebraic View on the Semantics of Model Composition. In *Model Driven Architecture - Foundations and Applications (ECMDA-FA) (LNCS)*, D. H. Akehurst, R. Vogel, and R. F. Paige (Eds.). Springer, Haifa, Israel, 99–113.

[12] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2011. A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In *International Conference on Automated Software Engineering (ASE'11).*

[13] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2013. Consistency-Preserving Edit Scripts in Model Versioning. In *International Conference on Automated Software Engineering (ASE).*

[14] Dénes König. 1927. Über eine Schlussweise aus dem Endlichen ins Unendliche. *Acta litterarum ac scientiarum Regiae Universitatis Hungaricae Francisco-Josephinae : Sectio scientiarum mathematicarum* 3 (1927), 121–130.

[15] Jochen M. Küster, Christian Gerth, and Gregor Engels. 2009. Dependent and Conflicting Change Operations of Process Models. In *Model Driven Architecture - Foundations and Applications.*

[16] Jochen M. Küster, Christian Gerth, Alexander Förster, and Gregor Engels. 2008. Detecting and Resolving Process Model Differences in the Absence of a Change Log. In *Business Process Management.*

[17] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. 2014. A Generic Framework for Realizing Semantic Model Differencing Operators. In *PSRC@MoDELs (CEUR Workshop Proceedings)*, Vol. 1258. CEUR-WS.org.

[18] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. 2014. Semantic Model Differencing Utilizing Behavioral Semantics Specifications. In *Model-Driven Engineering Languages and Systems.*

[19] Shahar Maoz and Jan Oliver Ringert. 2015. A Framework for Relating Syntactic and Semantic Model Differences. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'15).*

[20] Shahar Maoz and Jan Oliver Ringert. 2016. A framework for relating syntactic and semantic model differences. *Software & System Modeling* (August 2016).

[21] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2010. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10) (LNCS 6627)*. Springer, 194–203.

[22] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 179–189.

[23] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. *An Operational Semantics for Activity Diagrams using SMV.* Technical Report AIB-2011-07. RWTH Aachen University, Aachen, Germany.

[24] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: Semantic Differencing for Class Diagrams. In *ECOOP 2011 – Object-Oriented Programming.*

[25] Jan Oliver Ringert. 2014. *Analysis and Synthesis of Interactive Component and Connector Systems.* Shaker Verlag.

[26] Bernhard Rumpe. 1996. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Doktorarbeit. TU Munich.

[27] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. 2014. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling* 13, 1 (Feb 2014).

[28] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning about Edits to Feature Models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, 254–264.