A Generic Construction for Crossovers of Graph-Like Structures

Gabriele Taentzer(✉), Stefan John(✉), and Jens Kosiol(✉)

Philipps-Universität Marburg, Marburg, Germany
{taentzer,johns,kosiolje}@mathematik.uni-marburg.de

Abstract. In model-driven optimization (MDO), domain-specific models are used to define and solve optimization problems with evolutionary algorithms. Models are typically evolved using mutations, which can be formally specified as graph transformations. So far, only mutations have been used in MDO to generate new solutions from existing ones; a crossover mechanism has not yet been elaborated. In this paper, we present a generic crossover construction for graph-like structures that can be used to implement crossover operators in MDO. We prove basic properties of our construction and show how it can be used to implement a whole set of crossover operators that have been proposed for specific problems and situations on graphs.

Keywords: Evolutionary Computation · Crossover · Model-driven optimization · Category Theory

1 Introduction

In software development, software engineers often make design decisions in the context of competing constraints ranging from requirements to technology. To efficiently find optimal solutions, Search-Based Software Engineering (SBSE) [16] attempts to formulate software engineering problems as optimization problems that capture the constraints of interest as objectives. By using meta-heuristic search techniques, good solutions can often be found with reasonable effort. Because of their generality, evolutionary algorithms, and in particular genetic algorithms [5,17] that use mutation, crossover, and selection to perform a guided search over the search space, are a technique of particular relevance. According to e.g. [13], the definition of an evolutionary algorithm requires a representation of problem instances and search space elements (i.e., solutions). It also includes a formulated optimization problem that clarifies which of the solutions are *feasible* (i.e., satisfy all constraints of the optimization problem) and best satisfy the objectives. The key ingredients of the optimization process are a procedure for generating a start population of solutions, a mechanism for generating new solutions from existing ones (e.g., by mutation and crossover), a selection mechanism that typically establishes the evolutionary concept of survival of the fittest, and

N. Behr and D. Strüber (Eds.): ICGT 2022, LNCS 13349, pp. 97–117, 2022.
https://doi.org/10.1007/978-3-031-09843-7_6

a condition for stopping evolutionary computations. Selecting these ingredients so that an evolutionary algorithm is effective and efficient is usually a challenge.

Model-driven optimization (MDO) aims at reducing the required level of expertise of users of meta-heuristic techniques. Two main approaches have emerged in MDO: the model-based approach [7,8] performs optimization directly on models, while the rule-based approach [1,4] searches for optimized model transformation sequences. In this paper, we focus on the model-based approach since it tends to be more effective [20] and refer to it as MDO for short. In MDO, optimization problems are specified as models that capture domain-specific information about a problem and its solutions. In that way, users can interact with a domain-specific formulation of their problem, rather than traditional encodings that are typically closer to implementation. While the search space consists of models, the mutation of search space elements is specified by model transformations. In sophisticated evolutionary algorithms, mutations typically perform local changes, while crossovers are used to generate offspring by recombining existing search space elements. For (the model-based approach to) MDO, no crossover mechanism has been worked out yet. This paper fills this research gap and presents a crossover construction for graph-based models.

Several graph-based approaches to crossover have been suggested in the literature, e.g. [27,29]. In most cases, these crossovers are not *generic* (in the sense of different kinds of graphs), but are designed with specific semantics of the underlying graphs in mind. We aim to develop a generic construction of crossovers that can be applied to different kinds of graph-like structures. Moreover, this construction of crossovers is applicable regardless of the semantics of the graphs of interest. We also prove the correctness and completeness of our crossover construction.

The paper is organized as follows: We start with an example MDO problem and discuss a possible crossover in this context in Sect. 2. Section 3 recalls preliminaries. The main contribution of this paper, a pushout-based crossover construction, is presented in Sect. 4. In Sect. 5, we explain how our new crossover construction encompasses important, more specific approaches to crossover (on graph-like structures) that have been suggested in the literature. We close with a discussion of related work and a conclusion in Sects. 6 and 7. All proofs are given in Appendix A.

2 Running Example

The CRA case [6] is an optimization problem from the domain of software design that has recently established itself as an easily understood use case in the context of MDO. Given a software product represented by a set of features (i.e., attributes and methods) and dependency relations between them, the task is to modularize the software by encapsulating its features into classes. Two well-known quality aspects are used to evaluate the quality of solutions: cohesion and coupling. Cohesion rewards classes in which features are highly interdependent, while coupling captures the interdependencies of features that exist between classes. A highly cohesive design with low coupling is considered easy to

understand and maintain. Therefore, maximizing cohesion and minimizing coupling are the opposing objectives of the CRA case.

The structure of models in the CRA case can be defined by the type graph shown in Fig. 1. A problem instance consists at least of the features and their dependencies. These elements form the invariant part of a concrete problem. Classes (and their relationships), on the other hand, can be added, modified and removed to explore the search space and create new solutions. Typical mutations for the CRA case include small changes like adding or removing a class, assigning a feature to a class, or changing the assignment of a feature from one class to another. Mutation usually does not consider already well optimized substructures that might be worth being shared with other solutions.

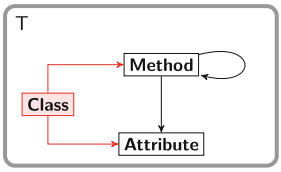


Fig. 1. Type graph of the CRA case. White solid elements specify invariant problem parts, the red colored class element and its relations are solution specific.

In the CRA case, a subset of features, along with their current assignment to classes, contains potentially valuable information. The exchange of this information between two solutions represents a promising crossover as we will see in the following example. Consider solutions $\overline{\mathsf{E}}$ and $\overline{\mathsf{F}}$ in Fig. 2, for a problem instance consisting of four methods and two attributes. Let a crossover choose to recombine them by exchanging their assignment information for the features 1:Method, 2:Attribute and 3:Method. This results in two offspring solutions. Solution $\overline{\mathsf{E}^1\mathsf{F}^2}$ keeps the original assignments of 4:Method, 5:Attribute, and 6:Method as found in solution $\overline{\mathsf{F}}$ and combines them with the assignments of $\overline{\mathsf{E}}$ for the exchanged features. The solution $\overline{\mathsf{E}^2\mathsf{F}^1}$ is constructed in the opposite way.

Note that combining 1:Method, 2:Attribute and 3:Method into one class (as done in solution $\overline{\mathsf{E}}$) seems a reasonable choice. Their pairwise dependencies promote cohesion, while splitting them would lead to coupling. The same is true for the features of class 12: Class in solution $\overline{\mathsf{F}}$. Consequently, the offspring $\overline{\mathsf{E}^1\mathsf{F}^2}$ combines the best of both worlds.

3 Preliminaries: $\mathcal{M}$-Adhesive Categories

In this section, we briefly recall our central formal preliminaries, namely $\mathcal{M}$-*adhesive categories* and $\mathcal{M}$-*effective unions* [12], which provide the setting in which we formulate our contribution. $\mathcal{M}$-adhesive categories with $\mathcal{M}$-effective unions are categories where pushouts along certain monomorphisms interact in a particularly nice way with pullbacks. This is of importance because our construction of crossovers is based on pushouts. Moreover, working in the framework of $\mathcal{M}$-adhesive categories allows us to easily abstract from the concrete choice of graphs used to formalize the models of interest (such as typed, labeled, and attributed graphs). We only use category-theoretic concepts that are common in the context of algebraic graph transformation, and refer to [11,12] for introductions.

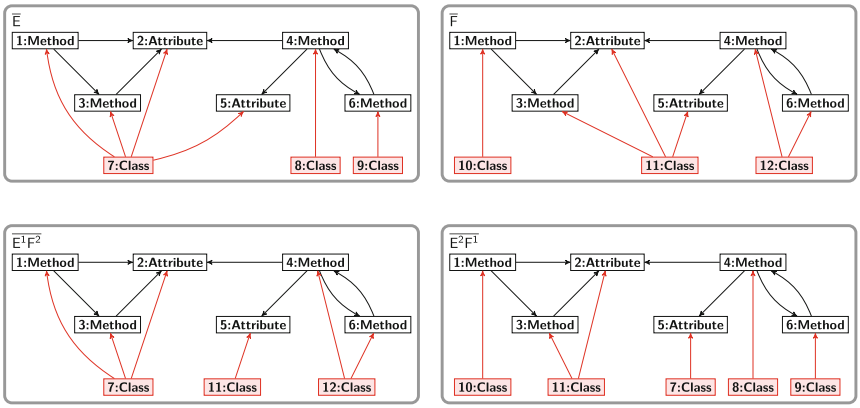


Fig. 2. Example crossover in the CRA case that creates the offspring $\overline{\mathsf{E}^1\mathsf{F}^2}$ and $\overline{\mathsf{E}^2\mathsf{F}^1}$ by exchanging the assignments of features 1:Method, 2:Attribute, and 3:Method between the solutions $\overline{\mathsf{E}}$ and $\overline{\mathsf{F}}$.

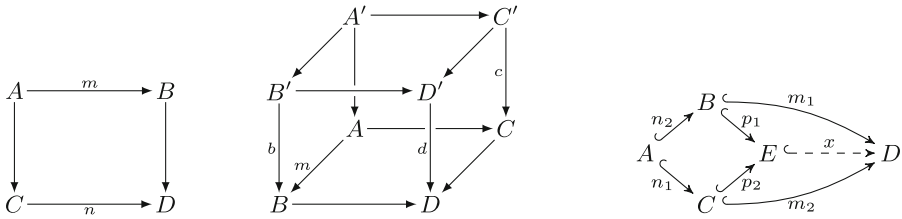
Definition 1 ($\mathcal{M}$-adhesive category). *A category $\mathcal{C}$ with a morphism class $\mathcal{M}$ is an* $\mathcal{M}$-adhesive category *if the following properties hold:*

- *$\mathcal{M}$ is a class of monomorphisms closed under isomorphisms (f isomorphism implies that $f \in \mathcal{M}$), composition ($f, g \in \mathcal{M}$ implies $g \circ f \in \mathcal{M}$), and decomposition ($g \circ f, g \in \mathcal{M}$ implies $f \in \mathcal{M}$).*
- *$\mathcal{C}$ has pushouts and pullbacks along $\mathcal{M}$-morphisms, i.e., pushouts and pullbacks where at least one of the given morphisms is in $\mathcal{M}$, and $\mathcal{M}$-morphisms are closed under pushouts and pullbacks, i.e., given a pushout like the left square in Fig. 3a, $m \in \mathcal{M}$ implies $n \in \mathcal{M}$ and, given a pullback, $n \in \mathcal{M}$ implies $m \in \mathcal{M}$.*
- *Pushouts in $\mathcal{C}$ along $\mathcal{M}$-morphisms are* vertical weak van Kampen squares*, i.e., for any commutative cube in $\mathcal{C}$ (as in the right part of Fig. 3a) where we have the pushout with $m \in \mathcal{M}$ in the bottom, $b, c, d \in \mathcal{M}$, and pullbacks as back faces, the top is a pushout if and only if the front faces are pullbacks.*

We speak of $\mathcal{M}$-adhesive categories $(\mathcal{C}, \mathcal{M})$ and indicate arrows from $\mathcal{M}$ as hooked arrows in diagrams. Examples of categories that are $\mathcal{M}$-adhesive include sets with injective functions, graphs with injective graph morphisms and various varieties of graphs with special forms of injective graph morphisms. In particular, typed attributed graphs form an $\mathcal{M}$-adhesive category (where the class $\mathcal{M}$ consists of injective morphisms where the attribute part is an isomorphism).

The existence of $\mathcal{M}$-*effective unions* ensures that the $\mathcal{M}$-subobjects of a given object form a lattice.

Definition 2 ($\mathcal{M}$-effective unions). *An $\mathcal{M}$-adhesive category $(\mathcal{C}, \mathcal{M})$ has* $\mathcal{M}$-effective unions *if for each pushout of a pullback of a pair of $\mathcal{M}$-morphisms the induced mediating morphism belongs to $\mathcal{M}$ as well, i.e., if in each diagram like*



(a) Vertical weak van Kampen square (b) $\mathcal{M}$-effective union

Fig. 3. Defining $\mathcal{M}$-adhesive categories with $\mathcal{M}$-effective unions

the one depicted in Fig. 3b where the outer square is a pullback of $\mathcal{M}$-morphisms and the inner one a pushout, the induced morphism x is an $\mathcal{M}$-morphism.

4 A Pushout-Based Crossover Construction

In this section, we develop our approach to crossover. We start with introducing the objects to which crossover will be applied.

In MDO, optimization problems are defined based on modeling languages, typically specified with meta-models. Various MDO approaches in the literature such as [7,8] have chosen to represent problem instances and solutions by models. Both can contain invariant problem parts as well as solution specific parts, a distinction typically embedded in the associated meta-model. In our formalization, this is reflected in the fact that a *computation element* is given by an object that conforms to a *computation type object*. The type object specifies which parts of a computation element are invariant and which parts contribute to the solution. A concrete problem to be optimized is given by a *problem instance*; every computation element can serve as such. The *search space* of a problem instance includes all computation elements with the same problem object as specified by the given problem instance. In MDO, problem instances and solutions are typically further constrained by additional conditions. We leave this refinement to future work.

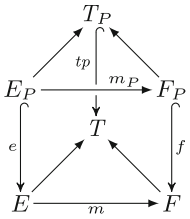


Fig. 4. Computation elements and ce-morphism

Definition 3 (Computation element. Problem instance. Search space). *Let $(\mathcal{C}, \mathcal{M})$ be an $\mathcal{M}$-adhesive category. A* computation type object *in $\mathcal{C}$ is an $\mathcal{M}$-morphism $tp\colon T_P \hookrightarrow T$; T_P is called the* problem type object*. A* computation element *$\overline{E} = (e\colon E_P \hookrightarrow E, t_{E_P}, t_E)$ over tp is an $\mathcal{M}$-morphism e together with* typing morphisms *$t_{E_P}\colon E_P \to T_P$ and $t_E\colon E \to T$ such that the induced square (over tp) is a pullback. The pair (E_P, t_{E_P}) is the* problem object *of $\overline{E}$. If defined, the initial pushout over e yields the* solution part *of $\overline{E}$, written $E \setminus E_P$.*

A computation-element morphism *$\overline{m} = (m_P, m)$, short* ce-morphism*, from computation element $\overline{E}$ to computation element $\overline{F}$ is a pair of morphisms $m_P\colon E_P \to F_P$ and $m\colon E \to F$ that are compatible with typing, i.e., $t_{F_P} \circ m_P =$*

t_{E_P} *and* $t_F \circ m = t_E$ *(see Fig. 4). A ce-morphism* $\overline{m}$ *is* problem-invariant *if* m_P *is an isomorphism between* E_P *and* F_P.

Given a computation type object $tp\colon T_P \hookrightarrow T$ *in* $\mathcal{C}$, *a* problem instance $\overline{PI}$ *of* tp *is a computation element* $\overline{PI} = (p\colon PI_P \hookrightarrow PI, t_{PI_P}, t_{PI})$ *over* tp. *It defines the* search space

$$S(\overline{PI}) := \{\overline{E} = (e\colon E_P \hookrightarrow E, t_{E_P}, t_E) \in CS \mid \text{there exists an isomorphism } a_P\colon PI_P \xrightarrow{\sim} E_P \text{ s.t. } t_{E_P} \circ a_P = t_{PI_P}\}.$$

Each element of the search space $S(\overline{PI})$ *is called* solution (object) *for* $\overline{PI}$.

Given a solution $\overline{E}$ *for* $\overline{PI}$, *a* subsolution *of* $\overline{E}$ *is a solution* $\overline{E^1}$ *from the search space* $S(\overline{PI})$ *such that there exists a problem-invariant ce-morphism* $\overline{s^1}$ *from* $\overline{E^1}$ *to* $\overline{E}$ *where* $s^1 \in \mathcal{M}$.

Before providing an example, some remarks with respect to the above definition and notation are in order. Since the typing of the problem object of a computation element is defined via a pullback, pullback decomposition implies that a ce-morphism is indeed a pullback square (compare Fig. 4). Thus, in abstract terms, we fix an $\mathcal{M}$-morphism $T_P \hookrightarrow T$ from a given $\mathcal{M}$-adhesive category $\mathcal{C}$. We then work in the category that has pullback squares over $T_P \hookrightarrow T$ as objects and pullbacks between such pullback squares as arrows. The results in [23, Theorem 1] ensure that this category is again $\mathcal{M}$-adhesive, provided that the original category $\mathcal{C}$ is also partial-map adhesive (as defined in [18]); a property that is satisfied by the category of attributed graphs; see, for example, [23, Corollary 1]. However, in this paper it will suffice to consider the arising diagrams as diagrams in the $\mathcal{M}$-adhesive category $\mathcal{C}$.

To shorten the presentation, we often only speak of computation elements $\overline{E}$ and ce-morphisms $\overline{m}$ and use their components (such as E_P, t_{E_P}, or m) freely without introducing them explicitly. Furthermore, we often let the typing be implicit; in particular, we omit it in almost all diagrams. In our examples, we use the category of graphs as the underlying $\mathcal{M}$-adhesive category $\mathcal{C}$. Finally, we specify problem instances in terms of the actual computation elements (and not just in terms of their problem objects) to account for the fact that in practice the problem of interest may be given as part of a (suboptimal) solution.

Example 1. The graph T in Fig. 1 can be viewed as a compact representation of a computation type graph where the black part marks the embedded problem type graph. Similarly, the typed graphs of Fig. 2 are interpreted as computation elements over T, with the black parts typed over the problem type graph; the typing is indicated by the names of the nodes. Since the typing morphisms form pullbacks, these black parts represent the problem graphs of the respective computation elements. Having identical problem graphs, all four graphs belong to the same search space, which can be defined using either of them. This reflects that a user might want to optimize an existing assignment of features to classes, rather than just specifying the features and their interdependencies.

Taking two computation elements (from the same search space) and splitting their solution parts, two offspring solutions are constructed by recombining the resulting subsolutions crosswise. In the following, we formally develop this intuition (based on the category-theoretic concept of pushouts) and prove basic properties of this construction of crossovers. We begin by defining the *split* of a given solution.

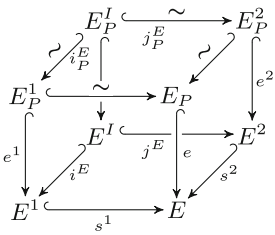


Fig. 5. Split of solution $\overline{E}$

Definition 4 (Split). *Given a problem instance $\overline{PI}$ and a solution $\overline{E}$ for $\overline{PI}$, a* split *of $\overline{E}$ is a commuting cube as depicted in Fig. 5 where the bottom square is a pushout, the vertical squares constitute ce-morphisms, all morphisms come from $\mathcal{M}$, and all problem objects (the objects in the square at the top) are isomorphic to PI_P. The bottom square is called* solution split *and $\overline{E^I}$ is a* split point *of $\overline{E}$. The subsolutions $\overline{E^1}$ and $\overline{E^2}$ of $\overline{E}$ are called* (solution) split objects *of $\overline{E}$.*

A solution can be split in several ways; the central idea is that each solution item of $\overline{E}$ occurs in (at least) one of the solution parts of $\overline{E^1}$ or $\overline{E^2}$. We next present a concrete construction that implements the above declarative definition.

Definition 5 (Split construction). *Given a solution $\overline{E}$, the* split construction *consists of the following steps:*

1. *Choose an $\mathcal{M}$-subobject $s^1 \colon E^1 \hookrightarrow E$ from E (in $\mathcal{C}$) such that when pulling back s^1 along e, the morphism s^1_P opposite to s^1 is an isomorphism (in particular, $E^1_P \cong E_P \cong PI_P$, where E^1_P is the object computed by this pullback). The typing morphisms $t_{E^1_P}$ and t_{E^1} are defined as $t_{E_P} \circ s^1_P$ and $t_E \circ s^1$, respectively.*
2. *Choose another such $\mathcal{M}$-subobject $s^2 \colon E^2 \hookrightarrow E$ from E such that s^1, s^2 are jointly epi (again, typing is defined by composition).*
3. *Complete the cube by constructing pullbacks. That is, determine E^I as the pullback of s^1 and s^2, E^I_P as the pullback of the isomorphisms at the top of the cube, and $e^I \colon E^I_P \hookrightarrow E^I$ as the morphism that is induced by the universal property of the bottom pullback. Again, when considered as computation element, the typing of $\overline{E^I}$ is defined by composition.*

Remark 1. While in general categories the above construction need not be constructive, it is when the underlying category is one of the familiar categories of graphs (being, e.g. typed, labeled, or attributed). Then, the choice of E^1 amounts to extending (an isomorphic copy of) E_P by a choice of solution elements from E; s^1 extends the isomorphism accordingly. Since pullbacks of injective morphisms compute intersections, the pullback of s^1 along e computes the chosen isomorphic copy (up to unique isomorphism). For the choice of E^2, one again extends an isomorphic copy of E_P by a choice of solution elements from E. To ensure that s^1 and s^2 become jointly epi (that is, jointly surjective in our case), one must include at least all solution elements of E not chosen in the construction of E^1.

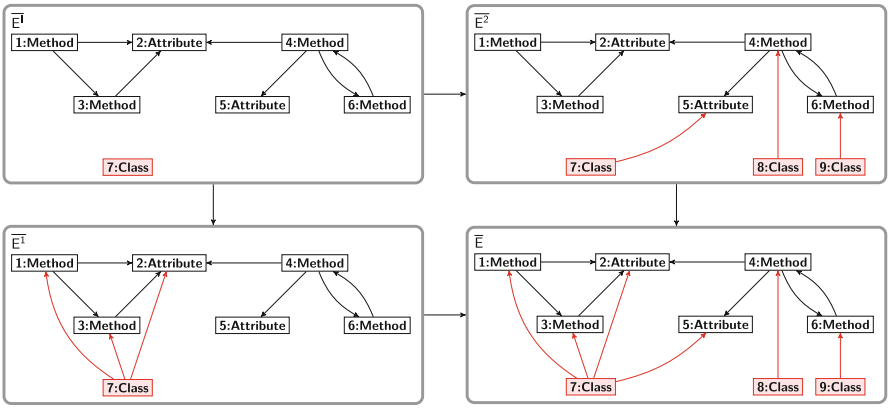


Fig. 6. A split of solution $\overline{\mathsf{E}}$

Example 2. Given the two degrees of freedom for a split, different splits can be constructed from solution $\overline{\mathsf{E}}$ shown in Fig. 2. In steps (1) and (2) we have all possibilities to extend its problem graph $\mathsf{E_P}$ (or an isomorphic copy) with solution parts that yield E^1 and E^2 as long as E^1 and E^2 form graphs and jointly cover E.

A possible split of the solution $\overline{\mathsf{E}}$ is shown in Fig. 6. Here, $\overline{\mathsf{E}}$ is split by first inserting the assignment relations of 1:Method, 2:Attribute, and 3:Method into E^1 along with the associated class 7:Class. The rest of the feature assignments and the necessary classes become part of E^2. The pullback E^{I} of E^1 and E^2 contains their common solution element 7:Class. To simplify the presentation, the problem graph $\mathsf{E_P}$ is reused in all four graphs. Note that the morphisms in Fig. 6 are indicated by equal numbers in the corresponding nodes. They uniquely induce the mapping of edges. We use these conventions in all of the following examples.

Proposition 1 (Correctness and completeness of split construction). *In an $\mathcal{M}$-adhesive category with $\mathcal{M}$-effective unions, the split construction in Definition 5 is* correct *and* complete*: it always yields a split of the given solution and every possible split can be realized through it. Moreover, for each choice of an $\mathcal{M}$-subobject $s^1\colon E^1 \hookrightarrow E$ there exists at least one possible split.*

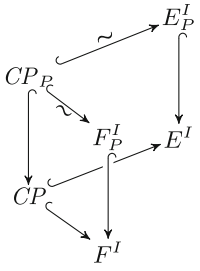


Fig. 7. Crossover point

Given a problem instance $\overline{PI}$ and two solutions $\overline{E}$ and $\overline{F}$ for it, a crossover of $\overline{E}$ and $\overline{F}$ can be performed. Their offspring are basically constructed by recombining solution split objects crosswise. Variations of recombinations are possible, since solution-split objects resulting from solution splits of $\overline{E}$ and $\overline{F}$ can be recombined with more or less overlap. To uniquely determine a crossover of $\overline{E}$ and $\overline{F}$, we define a crossover point that specifies the overlap of their solution split objects.

Definition 6 (Crossover point). *Given a problem instance $\overline{PI}$, two solutions $\overline{E}$ and $\overline{F}$ for $\overline{PI}$, with splits having split points $\overline{E^I}$ and $\overline{F^I}$, respectively, a* crossover point *$\overline{CP}$ is a common subsolution of $\overline{E^I}$ and $\overline{F^I}$. That is, a crossover point is a span of problem-invariant ce-morphisms as depicted in Fig. 7 (with bottom components coming from $\mathcal{M}$).*

We will explain crossover points later along with the crossover operation as such. Next we briefly mention that it is always possible to find a crossover point in a trivial way – the problem object of the given problem instance can always serve as such.

Lemma 1 (Existence of crossover points). *Given a problem instance $\overline{PI} = (p\colon PI_P \hookrightarrow PI, t_{PI_P}, t_{PI})$ over type object tp, two solutions $\overline{E}$ and $\overline{F}$ for $\overline{PI}$, and splits with split points $\overline{E^I}$ and $\overline{F^I}$, respectively, $\overline{CP} \coloneqq (id : PI_P \hookrightarrow PI_P, t_{PI_P}, tp \circ t_{PI_P})$ is always a crossover point for them. In particular, for each two splits of solutions for the same problem instance there always exists a crossover point.*

Taking two solutions $\overline{E}$ and $\overline{F}$ for a common problem instance and splitting them into subsolutions $\overline{E^1}, \overline{E^2}$ and $\overline{F^1}, \overline{F^2}$, we choose a crossover point for these splits and now define a crossover of these solutions. It basically recombines the subsolutions of $\overline{E}$ and $\overline{F}$ crosswise at the crossover point and yields the computation elements $\overline{E^1F^2}$ and $\overline{E^2F^1}$. We show in Proposition 2 that these two offspring are also solutions to the joint problem instance.

Definition 7 (Crossover). *Let a problem instance $\overline{PI}$, two solutions $\overline{E}$ and $\overline{F}$ for $\overline{PI}$, splits of these two solutions with split objects $\overline{E^1}, \overline{E^2}, \overline{F^1}, \overline{F^2}$ and split points $\overline{E^I}$ and $\overline{F^I}$, respectively, and a* crossover point *$\overline{CP}$ for these splits be given. Then, a* crossover *of solutions $\overline{E}$ and $\overline{F}$ (at $\overline{CP}$ and these splits) yields the two* offspring solutions *$\overline{O_1}$ and $\overline{O_2}$ of $\overline{E}$ and $\overline{F}$ that are shown in Fig. 8 and constructed as follows:*

1. *The ce-morphisms from $\overline{CP}$ to $\overline{E^1}$ and $\overline{E^2}$ are obtained by composing the ce-morphism from $\overline{CP}$ to $\overline{E^I}$ (given by the crossover point) with the ce-morphisms from $\overline{E^I}$ to $\overline{E^1}$ and $\overline{E^2}$ (given by the solution split of $\overline{E}$), respectively. The ce-morphisms from $\overline{CP}$ to $\overline{F^1}$ and $\overline{F^2}$ are obtained analogously.*
2. *The top and bottom squares of the cubes are computed as pushouts (in $\mathcal{C}$) yielding the objects $(E^1F^2)_P$, E^1F^2, $(E^2F^1)_P$, and E^2F^1. The typing morphisms for these objects are obtained from the universal properties of the respective pushout.*
3. *The morphisms $o_1\colon (E^1F^2)_P \hookrightarrow E^1F^2$ and $o_2\colon (E^2F^1)_P \hookrightarrow E^2F^1$ are also induced by the universal property of the pushout squares at the top of the cubes. These morphisms form the objects of $\overline{O_1}$ and $\overline{O_2}$.*

We illustrate the construction before establishing some of its basic properties such as its correctness.

Example 3. A split of solution $\overline{\mathsf{F}}$ (introduced in Fig. 2) is shown in Fig. 9. Again, the split point extends the problem graph by a Class element. Therefore, a

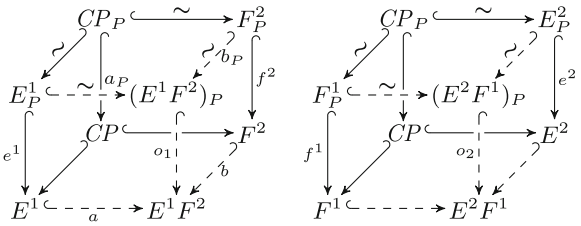


Fig. 8. Crossover of solutions $\overline{E}$ and $\overline{F}$

crossover point for $\overline{\mathsf{E}}$ and $\overline{\mathsf{F}}$ (with the splits given in Figs. 6 and 9) consists either of their common problem graph only, or of this problem graph extended by a single Class. Figure 2 already shows the two offspring graphs that result from applying crossover to $\overline{\mathsf{E}}$ and $\overline{\mathsf{F}}$ where the problem graph is chosen as crossover point. In contrast, adding a Class to the crossover point would merge 7:Class and 11:Class during the recombination and result in the offspring shown in Fig. 10.

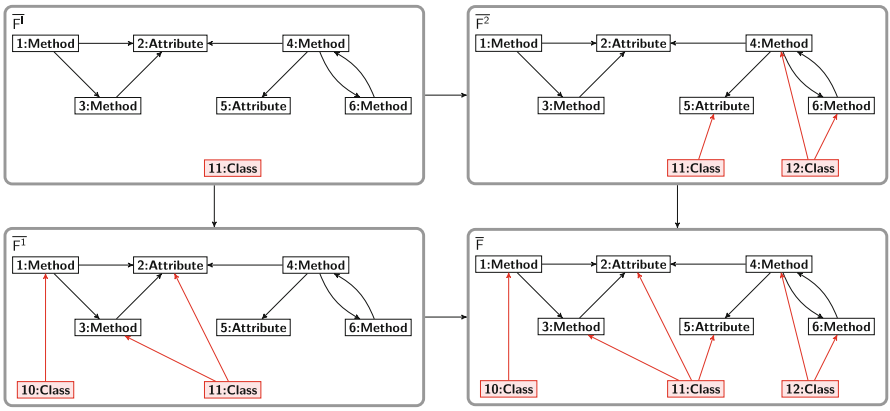


Fig. 9. A split of solution $\overline{\mathsf{F}}$ originally presented in Fig. 2

The next proposition shows that a crossover calculate the offspring correctly, i.e. all offspring calculated represent solutions (for the given problem instance).

Proposition 2 (Correctness of offspring). *Given a problem instance* $\overline{PI}$*, two solutions* $\overline{E}$ *and* $\overline{F}$ *for* $\overline{PI}$*, splits with split objects* $\overline{E^1}, \overline{E^2}, \overline{F^1}, \overline{F^2}$ *and split points* $\overline{E^I}$ *and* $\overline{F^I}$*, respectively, and a crossover point* $\overline{CP}$ *for these splits, then there is always a crossover and the two offspring solutions* $\overline{O_1}$ *and* $\overline{O_2}$ *are solutions for* $\overline{PI}$*.*

Next we characterize the expressiveness of the presented crossover construction: Given two solutions $\overline{E}$ and $\overline{F}$, all solutions that can be understood as

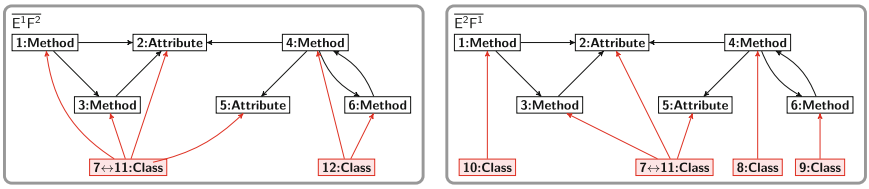


Fig. 10. Two offspring models $\overline{\mathsf{E}^1\mathsf{F}^2}$, $\overline{\mathsf{E}^2\mathsf{F}^1}$, based on the splits of Figs. 6 and 9 and a crossover point containing an additional class

results of splitting $\overline{E}$ and $\overline{F}$ and their recombination can indeed be generated as offspring of the construction in Definition 7 (by different choices of solution splits and crossover points). This is reminiscent of the expressiveness of *uniform crossover* when using arrays of, e.g., bits as genotype [13].

Proposition 3 (Completeness of crossover). *Let the underlying $\mathcal{M}$-adhesive category $\mathcal{C}$ have $\mathcal{M}$-effective unions, and let a problem instance $\overline{PI}$ and solutions $\overline{E}$, $\overline{F}$, and $\overline{O}$ for $\overline{PI}$ be given. The solution $\overline{O}$ can be obtained as offspring from a crossover of $\overline{E}$ and $\overline{F}$ if and only if there are subsolutions $\overline{E^1}$ of $\overline{E}$ and $\overline{F^2}$ of $\overline{F}$ with problem-invariant ce-morphisms $\bar{i}\colon \overline{E^1} \to \overline{O}$ and $\bar{j}\colon \overline{F^2} \to \overline{O}$ such that i and j are jointly epic $\mathcal{M}$-morphisms.*

Discussion. As mentioned earlier, $\mathcal{M}$-adhesive categories include various categories of (typed, labeled, or attributed) graphs that can be used to formalize modeling approaches. In particular, our construction supports crossovers of graphs with inheritance and attribution – concepts that are regularly used in modeling. As for the construction of splits and crossover points, our approach provides several degrees of freedom. In principle, for any implementation of these variation points, the definitions and results in this section are sufficient to complement evolutionary computations in model-based MDO with crossovers. Moreover, our proposed crossover construction is *generic* in the sense that it can be applied to any meta-model; it only needs to be possible to formalize the optimization problem of interest and its search space according to Definition 3. Then, whenever two solution models are chosen for crossover, Proposition 1 ensures that both can be split. Next, Lemma 1 ensures that regardless of which splits are chosen, a crossover point exists for these splits. Finally, Proposition 2 ensures that, for two splits and a crossover point, there is always a crossover that provides solutions of the search space.

Beyond typing, meta-modeling typically employs *integrity constraints* that express further requirements for instances being considered well-formed; *multiplicities* are a typical example. We do not consider such constraints so far. This means that given a meta-model with additional integrity constraints and two of its instance models satisfying these constraints, computing crossover as specified in this work may result in offspring models that violate the constraints. We illustrate this with our running example: In practical applications, the meta-model (type graph) from Fig. 1 would have a constraint requiring each Method and

each Attribute to be associated with at most one Class. A slight adjustment of the split and crossover points in Examples 2 and 3 results in the offspring shown in Fig. 11; both graphs violate the considered constraint. The splits of $\overline{E}$ and $\overline{F}$ were adjusted to additionally include the edge to 5:Attribute in $\overline{E^1}$ as well as in $\overline{F^1}$ (from 7:Class and 11:Class, respectively); the problem part served as the crossover point. Computing offspring that violate such additional constraints is not in itself a problem; several methods have been developed in evolutionary algorithm research to deal with this. For example, such *infeasible solutions* can be eliminated by the selection operator, or they can be tolerated (with a reduced fitness assigned to them); after all, even an infeasible solution can lead to a feasible solution of high quality later during the evolutionary computation. However, producing too many infeasible solutions can waste valuable resources and slow down the evolutionary computation process.

Summarizing, we expect evolutionary search to profit most if domain-specific knowledge is used to direct the choices of splits and crossover points, that is, if these choices are adapted to the problem at hand (possibly including the preservation of additional constraints). Thus, while our construction can principally yield problem-agnostic crossovers, it can also (and maybe better) be understood as a *generic* construction that offers a unifying framework for the implementation of specific crossovers on graph-like structures. In the next section, we substantiate the claim that our construction offers such a unifying framework.

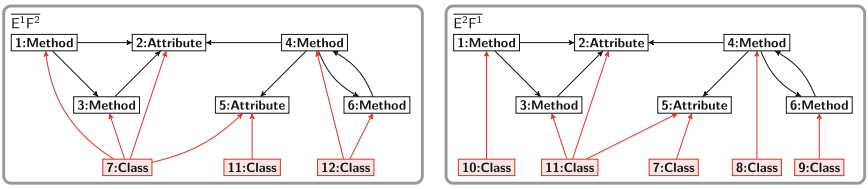


Fig. 11. Offspring violating an integrity constraint

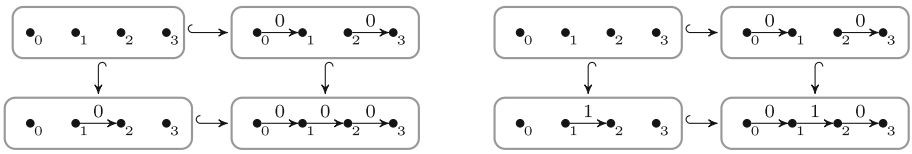
5 Instantiating Existing Approaches to Graph-Based Crossover

In this section, we exemplify how our generic construction includes existing crossover operators that can be applied to graph-like structures. We discuss *uniform*, *k-point* and *subtree crossover*, as these are classic operators that are commonly applied [13,24]. In addition, we consider *horizontal gene transfer* (HGT), which was recently introduced in a setting similar to ours [2].

Uniform and k-Point Crossover are crossover operators commonly used when solutions are encoded as strings (arrays) of bits (or other alphabets) [13]. In k-point crossover, two given parent strings of equal length are split into $k+1$ substrings at k randomly selected crossover points (at equal positions in both

strings). The two offspring solutions are obtained by alternately concatenating a substring from each parent, resulting in solutions of the same length as the given parents. In uniform crossover, a new decision is made at each position (according to a given probability) which offspring gets the entry from which parent. This can be understood as k-point crossover with varying k.

Strings can be represented as graphs by simply considering each character of a string as an edge typed or labeled with that character; see, e.g., [30]. Using this representation, our construction of crossovers can be used to implement uniform and k-point crossover. Here, the problem object (graph) is given by the nodes of the graphs (which encode the length of the given strings). The splits are chosen such that (i) the edges are partitioned (disjointly) into the solution splits and (ii) the same partitions are chosen for both parents (i.e., if the first edge of the first parent is included in its first subsolution, the first edge of the second parent is also included in its first subsolution). This partitioning can be done according to the rules of k-point or uniform crossover. The only available crossover point is the set of nodes (i.e. the problem graph), since the edges are distributed disjointly. The calculation of the crossover, i.e.performing the two pushouts, results in two offspring solutions with the same length as the parents.



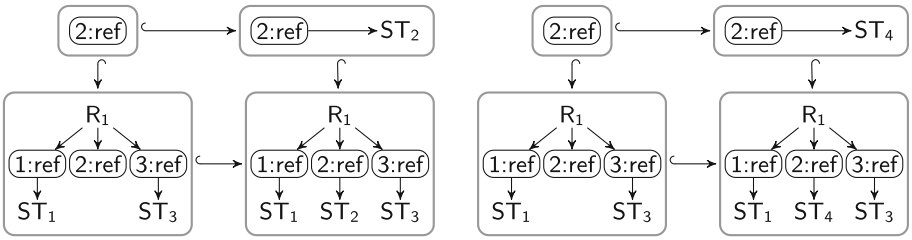
(a) Splitting s_1 (b) Computing o_1

Fig. 12. Implementing classic 2-point crossover

For the k-point crossover, we consider the concrete example of a 2-point crossover of the strings $s_1 : 0|0|0$ and $s_2 : 1|1|1$, where $|$ represents the chosen crossover points. The computed offspring strings are $o_1 : 010$ and $o_2 : 101$. Figure 12 outlines how this calculation is implemented in our approach.

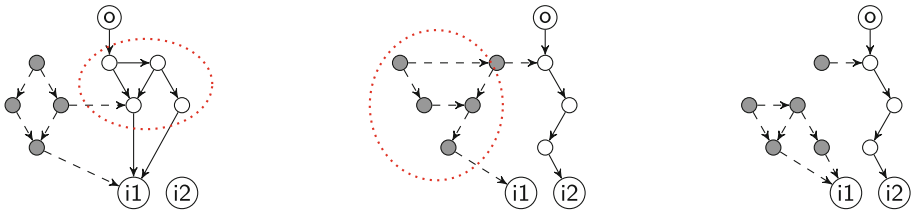
Subtree Crossover is the recombination operator commonly used in genetic programming [24]. In genetic programming, a program is represented by its syntax tree. Such a tree serves as a genotype for an evolutionary computation that aims at finding an (optimal) program for the given task. Given two syntax trees, subtree crossover (randomly) selects and exchanges one subtree from each of them. With our approach, we can implement subtree crossover if we use a little trick in representing the trees: We explicitly encode the edges of the trees as nodes (for a representation of (hyper)edges as special kinds of nodes, see, e.g., their (visual) representation in [31]). The problem tree (graph) is always empty. A split divides a tree into a subtree and the remaining tree, where the node encoding the reference to the subtree is common in both split objects. This

node serves as a crossover point to exchange subtrees crosswise at the correct positions. Figure 13 schematically represents a subtree crossover, where R_1 is the root node of the first tree, all ST_i represent subtrees, and nodes of type ref represent edges. Note that representing edges as nodes allows us to split an edge into two parts and distribute it between the two split parts. In this way, we can redirect edges.



(a) Splitting a tree (b) Performing subtree crossover

Fig. 13. Implementing subtree crossover



(a) Donor (b) Receiver (c) Result graph

Fig. 14. Example of the horizontal gene transfer (HGT) proposed in [2]. o is the fixed output node. Active nodes are depicted in white, passive nodes are gray. i1 and i2 are input nodes. The marked nodes of the receiver (including outgoing edges) are substituted by the marked parts of the donor.

Horizontal Gene Transfer (HGT) was proposed by Atkinson et al. in [2] as a non-recombinative method for transferring genetic information between individuals. In their work, graphs are used to represent functions (or, with small adaptations, neural networks); the reachability of fixed output nodes determines the *active component* of a graph. As indicated in Fig. 14, HGT takes the active component of one graph (the donor) and copies it to the passive component of another graph (the receiver); to maintain a fixed number of nodes, an appropriate number of passive nodes is deleted from the receiver beforehand. Input nodes representing parameters are identified during that process. In our construction the output and input nodes would be considered the problem part. Choosing the active component as the solution split for the donor, the subgraph that remains after deleting the passive nodes as the solution split of the receiver, and the problem part as crossover point, our approach can compute HGT as a crossover.

6 Related Work

In addition to the approaches presented in detail above in Sect. 5, we now relate our crossover construction to other variants of crossover on graph-like structures. For each approach, we clarify whether it can be simulated by our approach and how expressive it is. We then discuss the crossover variants used so far in MDO.

6.1 Further Approaches for Graph-Based Crossover

The two most general crossover variants on graph-like structures that we are aware of are those proposed by Niehaus [27] and Machado et al. [26]. Niehaus introduces *random crossover* on directed graphs, where a subgraph of one graph is removed and replaced by a subgraph of another graph; in particular, only one offspring is computed. To avoid dangling edges, the exchanged subgraphs must have the same in- and out-degrees with respect to the edges that connect them to the rest of the graph. By using the trick of representing edges as a special kind of node, we can realize this crossover with our approach.

Machado et al. [26] also exchange subgraphs between graphs. The subgraphs are constructed as radii around randomly chosen nodes. To connect the exchanged subgraphs to their new host graphs, a correspondence is established between the nodes that were adjacent to them in their former host graphs. If this correspondence is one-to-one, we can implement this operator in our approach by again representing edges by a special type of node. However, Machado et al. also allow for correspondences that are not one-to-one. To implement this feature, we would need to allow non-injective mappings from the crossover point to the splits in our approach. Unlike this approach, our approach is not limited to choosing subgraphs as radii around randomly chosen nodes.

Other approaches are less general since they depend to a greater extent on the chosen representation or semantics of the graphs used [9,10,19,21,22,28]. In these cases, it does not seem straightforward to apply the proposed crossovers in other contexts. The kind of computations that can be performed using crossover may also be less expressive than those in the approaches already discussed [19,21,22,28,29]. We can implement the crossovers proposed in [9,10,19,28,29] in our approach, often by representing edges as a special type of node. The approach by Kantschik and Banzhaf [22] cannot be implemented for reasons similar to those discussed for [26]. Furthermore, we cannot implement the *subgraph crossover* proposed in [21], because this approach allows random insertion of new edges into an offspring and these edges do not come from any parent.

In summary, our generic approach to crossover on graph-like structures encompasses most of the approaches proposed for more specific situations. Our approach allows more general exchanges of subgraphs than most of the approaches discussed. Moreover, our Theorem 3 is the first result (that we know of) that formally clarifies the expressiveness of the proposed crossover. We have identified two reasons why our approach is not able to encompass an existing approach: First, crossover could cause two (or more) edges that targeted different nodes in their original graph to target the same node in their new context.

Second, elements that do not originate from either parent are reintroduced in the offspring. However, both kinds of changes can be realized in our approach by the subsequent application of mutation operators. We could also solve the first problem by allowing non-injective mappings from crossover points to the splits when performing crossover. However, this would complicate the theory we can provide for our construction: Pushouts along any two morphisms need not exist in $\mathcal{M}$-adhesive categories, and even if the necessary pushouts did exist, ensuring that the computed results come from the search space under consideration (i.e., represent an $\mathcal{M}$-morphism) would only be possible for certain morphisms.

6.2 Crossover in MDO

In the rule-based approach to MDO, the solutions are represented as sequences of model transformations [1,4]. This allows traditional crossovers (e.g., k-point crossover, uniform crossover) to be applied seamlessly. However, they have been shown to be disruptive because the transformations can depend on each other [20] and repair strategies must be used to mitigate this problem. As for the effects of crossover in the rule-based approach, no theoretical results are available. To date, neither a formal basis nor alternatives to traditional crossover have been developed in this context.

Burton et al. were the first to perform optimization directly on models as search space elements [8]. Their specific use case allows for the adaptation of single-point crossover through model transformations. However, their crossover implementation is not described in detail. Recent applications of the model-based approach neglect crossover and stick to mutation as their only change operator, such as [7]. In [32], Zschaler and Mandow present a generalized view on the model-based approach to MDO and point out the challenge of specifying crossover in such a setting. They briefly discuss model differencing and model merging as related concepts, but do not elaborate on this idea. To our knowledge, this paper presents the first approach to address this issue.

7 Conclusion

There is theoretical and practical evidence that evolutionary algorithms in general benefit from the use of crossover [2,9,19] in the sense that the search for optimal solutions can be more effective and efficient. However, in the absence of suitable crossover approaches for (the model-based approach to) MDO, the effect of crossover in this context has not yet been studied. Our proposed generic crossover construction can serve as a basis to start with.

How existing solutions are split and the selection of common crossover points for such splits are critical design decisions. Which of these decisions are beneficial to the effectiveness and efficiency of an optimization remains to be explored. Apart from the typing of objects, our approach neglects additional constraints of an optimization problem, i.e., crossover may lead to violations of constraints. Whether our approach needs to be refined to guarantee constraint-preserving

offspring remains for future work. In addition to theoretical exploration of our approach, an implementation is needed to enable empirical analysis. Additionally, specification concepts need to be elaborated to allow users to conveniently specify different split strategies and crossover points that fit their domain.

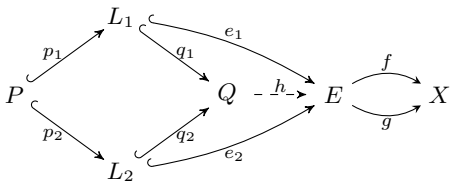
Acknowledgements. This work has been partially supported by the German Research Foundation (DFG), grant no. TA 294/19-1. We thank the anonymous reviewers for their insightful comments.

A Proofs

The following lemma is the central ingredient for the proof of Proposition 1 and also used in the one of Theorem 3. For adhesive categories, it has already been stated in the extended version of [14]. Here, we present it in the more general context of $\mathcal{M}$-adhesive categories. Because of that, we need to additionally assume the existence of $\mathcal{M}$-effective unions.

Lemma 2 (Pullbacks as pushouts). *In an $\mathcal{M}$-adhesive category $(\mathcal{C}, \mathcal{M})$ with $\mathcal{M}$-effective unions, let $(e_1, e_2) : L_1, L_2 \hookrightarrow E$ be a pair of jointly epimorphic $\mathcal{M}$-morphisms. Then the pullback of (e_1, e_2) is also a pushout.*

Proof. Given the diagram below, where P arises as pullback of (e_1, e_2), Q as pushout of (p_1, p_2), and the morphism h from the universal property of Q, we show that h is an isomorphism.



First, since e_1, e_2 are $\mathcal{M}$-morphisms, the morphism h is an $\mathcal{M}$-morphism, assuming $\mathcal{M}$-effective unions. This means that h is a regular monomorphism (compare [25, Lemma 4.8], which is easily seen to also hold in $\mathcal{M}$-adhesive categories).

Secondly, given two morphisms $f, g : E \to X$ with $f \circ h = g \circ h$, it follows that $f \circ h \circ q_1 = g \circ h \circ q_1$ which implies $f \circ e_1 = g \circ e_1$; analogously, $f \circ e_2 = g \circ e_2$ holds. Since e_1, e_2 are jointly epimorphic, it follows that $f = g$, and h is an epimorphism. Thus, h is epi and regular mono and therefore an isomorphism. □

Proof (of Proposition 1). Given a solution split as depicted in Fig. 5, it is straightforward to realize this split via the split construction. One just chooses the already given morphisms s^1 and s^2. As the bottom square in Fig. 5 is a pushout, s^1 and s^2 are jointly epimorphic. Moreover, in an $\mathcal{M}$-adhesive category that square is also a pullback because $E^I \hookrightarrow E^1$ (or, equally, $E^I \hookrightarrow E^2$) $\in \mathcal{M}$.

To show that the construction always computes a solution split, we have to show that it produces a commuting cube of $\mathcal{M}$-morphisms (with isomorphisms at the top) such that the bottom square is a pushout and the four vertical squares constitute ce-morphisms (i.e., are also pullbacks and are compatible with typing). It is well-known that, in every category, in a cube that is computed via pullbacks as stipulated by our construction, all squares are pullbacks; see, e.g., [3, 5.7 Exercises, 2. (b)]. By closedness of $\mathcal{M}$-morphism under pullbacks, this in turn implies that all morphisms are $\mathcal{M}$-morphisms (because e, s^1, and s^2 are). The two morphisms at the front of the top square are isomorphisms by assumption; the other two become isomorphisms by closedness of isomorphisms under pullback. Finally, in an $\mathcal{M}$-adhesive category with $\mathcal{M}$-effective unions, the pullback of jointly epimorphic $\mathcal{M}$-morphisms is always a pushout (see Lemma 2 above). Therefore, the bottom square (computed as pullback of the jointly epic $\mathcal{M}$-morphisms s^1 and s^2) is a pushout as desired. The typing of $\overline{E^1}$ and $\overline{E^2}$ is compatible with the typing of $\overline{E}$ by definition; moreover, the squares obtained from the typing morphisms are pullbacks by pullback composition.

For the last statement, it suffices to observe that E^2 can always be chosen as E, embedded via the identity morphism (which then leads to $E^I \cong E^1$). □

Proof (of Lemma 1). To prove the statement, we have to show that there exists a ce-morphism (a_P, a) from $\overline{CP} := (id : PI_P \hookrightarrow PI_P, t_{PI_P}, tp \circ t_{PI_P})$ to $\overline{E^I}$ such that a_P is an isomorphism and $a \in \mathcal{M}$; the analogous statement for $\overline{F^I}$ is proved in exactly the same way.

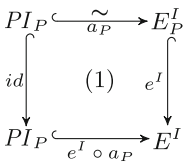


Fig. 15. Showing $\overline{CP}$ to constitute a crossover point

We define such a ce-morphism using the isomorphism a_P with $t_{E^I_P} \circ a_P = t_{PI_P}$ that exists since $\overline{E^I}$ is an element of the search space of $\overline{PI}$. Figure 15 depicts this. The square commutes and $a, e^I \circ a \in \mathcal{M}$ by closedness of $\mathcal{M}$ under isomorphisms and composition. Moreover, using the fact that e^I is a monomorphism, it is also easy to check that the square constitutes a pullback. Finally, using $t_{E^I_P} \circ a_P = t_{PI_P}$ we compute

$$\begin{aligned} t_{E^I} \circ e^I \circ a_P &= tp \circ t_{E^I_P} \circ a_P \\ &= tp \circ t_{PI_P} \end{aligned}$$

which shows $(a_P, e^I \circ a_P)$ to be type-compatible. □

Proof (of Proposition 2). First, in an $\mathcal{M}$-adhesive category, pushouts along $\mathcal{M}$-morphisms exist. This means that, given two solution splits and a crossover point, crossover is always applicable. Since isomorphisms are closed under pushout, the top squares in the construction consist of isomorphisms only. In particular, $(E^1F^2)_P \cong PI_P \cong (E^2F^1)_P$ (because $E^1_P \cong PI_P \cong E^2_P$ by assumption).

By definition, o_1 is the unique morphism such that

$$o_1 \circ a_P = a \circ e^1 \text{ and } o_1 \circ b_P = b \circ f^2,$$

where (a_P, a) and (b_P, b) denote the ce-morphisms from e^1 and f^2 to o_1 (see Fig. 8). A standard diagram chase (using the facts that the top squares in Fig. 8 consist of isomorphisms only and that diagrams remain commutative if one replaces isomorphisms by their inverses) then shows that $a \circ e^1 \circ a_P^{-1}$ (or, equally, $b \circ f^2 \circ b_P^{-1}$) exhibits this universal property. Therefore, $o_1 = a \circ e^1 \circ a_P^{-1} \in \mathcal{M}$ as composition of $\mathcal{M}$-morphisms. Again, this uses the fact that $\mathcal{M}$ contains all isomorphisms.

Finally, that the typing morphisms of $\overline{O_1}$ induce even a pullback square over tp (and not merely a commuting one) follows exactly as in the proof of Lemma 2.2 in [15], using the facts that the ambient category $\mathcal{C}$ is $\mathcal{M}$-adhesive and $tp \in \mathcal{M}$. □

Proof (of Proposition 3). Let solution $\overline{O}$ be computed via a crossover from $\overline{E}$ and $\overline{F}$. It is immediately clear from the construction that there exist the two required ce-morphisms $\bar{i}$ and $\bar{j}$ such that i, j are jointly epic $\mathcal{M}$-morphisms because the projections of a pushout are jointly epi and $\mathcal{M}$-morphisms are closed under pushout.

For the converse direction, O is jointly covered by E^1 and F^2, which stem from subsolutions $\overline{E^1}$ and $\overline{F^1}$ of $\overline{E}$ and $\overline{F}$ by assumption. If the underlying category has $\mathcal{M}$-effective unions, pulling these morphisms back results in a pushout. Let $\overline{CP}$ be the object resulting from that pullback (exactly as in the proof of Proposition 1). We merely have to show that there exist solution splits of $\overline{E}$ and $\overline{F}$ that split up $\overline{E}$ into $\overline{E^1}$ and some suitable subsolution $\overline{E^2}$ of $\overline{E}$ and $\overline{F}$ into $\overline{F^2}$ and some suitable subsolution $\overline{F^1}$ of $\overline{F}$ for which $\overline{CP}$ can serve as a crossover point. As in (the proof of) the second part of Proposition 1, we can use $\overline{E}$ as $\overline{E^2}$ and, because of the symmetric nature of a solution split, $\overline{F}$ as $\overline{F^1}$ and obtain splits of $\overline{E}$ and $\overline{F}$ with $\overline{E^I} = \overline{E^1}$ and $\overline{F^I} = \overline{F^2}$. Hence, $\overline{CP}$, together with the morphisms that stem from its computation as a pullback, can serve as a crossover point for these splits, and applying the crossover construction computes the given solution $\overline{O}$. □

References

1. Abdeen, H., et al.: Multi-objective optimization in rule-based design space exploration. In: Crnkovic, I., Chechik, M., Grünbacher, P. (eds.) ACM/IEEE International Conference on Automated Software Engineering, ASE 2014, Vasteras, Sweden - 15–19 September 2014, pp. 289–300. ACM (2014). https://doi.org/10.1145/2642937.2643005
2. Atkinson, T., Plump, D., Stepney, S.: Horizontal gene transfer for recombining graphs. Genet. Program. Evolvable Mach. **21**(3), 321–347 (2020). https://doi.org/10.1007/s10710-020-09378-1

3. Awodey, S.: Category Theory, Oxford Logic Guides, 2nd edn. vol. 52. Oxford University Press, Oxford (2010)
4. Bill, R., Fleck, M., Troya, J., Mayerhofer, T., Wimmer, M.: A local and global tour on MOMoT. Softw. Syst. Model. **18**(2), 1017–1046 (2019). https://doi.org/10.1007/s10270-017-0644-3
5. Boussaïd, I., Siarry, P., Ahmed-Nacer, M.: A survey on search-based model-driven engineering. Autom. Softw. Eng. **24**(2), 233–294 (2017). https://doi.org/10.1007/s10515-017-0215-4
6. Bowman, M., Briand, L.C., Labiche, Y.: Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. IEEE Trans. Softw. Eng. **36**(6), 817–837 (2010). https://doi.org/10.1109/TSE.2010.70
7. Burdusel, A., Zschaler, S., John, S.: Automatic generation of atomic multiplicity-preserving search operators for search-based model engineering. Softw. Syst. Model. **20**(6), 1857–1887 (2021). https://doi.org/10.1007/s10270-021-00914-w
8. Burton, F.R., Paige, R.F., Rose, L.M., Kolovos, D.S., Poulding, S., Smith, S.: Solving acquisition problems using model-driven engineering. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 428–443. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_32
9. Doerr, B., Happ, E., Klein, C.: Crossover can provably be useful in evolutionary computation. Theor. Comput. Sci. **425**, 17–33 (2012). https://doi.org/10.1016/j.tcs.2010.10.035
10. Downey, C., Zhang, M., Browne, W.N.: New crossover operators in linear genetic programming for multiclass object classification. In: Pelikan, M., Branke, J. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2010, Portland, Oregon, USA, 7–11 July 2010. pp. 885–892. ACM (2010). https://doi.org/10.1145/1830483.1830644
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer, Heidelberg (2006).https://doi.org/10.1007/3-540-31188-2
12. Ehrig, H., Ermel, C., Golas, U., Hermann, F.: Graph and model transformation - general framework and applications. In: Monographs. in Theoretical Computer Science. An EATCS Series, Springer (2015). https://doi.org/10.1007/978-3-662-47980-3
13. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. NCS, Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-44874-8
14. Fritsche, L., Kosiol, J., Schürr, A., Taentzer, G.: Short-cut rules. In: Mazzara, M., Ober, I., Salaün, G. (eds.) STAF 2018. LNCS, vol. 11176, pp. 415–430. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_30
15. Garner, R., Lack, S.: On the axioms for adhesive and quasiadhesive categories. Theory Appl. Categor. **27**(3), 27–46 (2012), https://www.emis.de/journals/TAC/volumes/27/3/27-03abs.html
16. Harman, M., Jones, B.F.: Search-based software engineering. Inf. Softw. Technol. **43**(14), 833–839 (2001). https://doi.org/10.1016/S0950-5849(01)00189-6
17. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: trends, techniques and applications. ACM Comput. Surv. **45**(1), 11:1–11:61 (2012). https://doi.org/10.1145/2379776.2379787
18. Heindel, T.: Adhesivity with partial maps instead of spans. Fundam. Informaticae **118**(1-2), 1–33 (2012). https://doi.org/10.3233/FI-2012-704

19. Husa, J., Kalkreuth, R.: A comparative study on crossover in cartesian genetic programming. In: Castelli, M., Sekanina, L., Zhang, M., Cagnoni, S., García-Sánchez, P. (eds.) EuroGP 2018. LNCS, vol. 10781, pp. 203–219. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77553-1_13
20. John, S., Alexandru Burdusel, R.B., Strüber, D., Taentzer, G., Zschaler, S., Wimmer, M.: Searching for optimal models: comparing two encoding approaches. J. Obj. Technol. **18**(3), 6:1–22 (2019). https://doi.org/10.5381/jot.2019.18.3.a6
21. Kalkreuth, R., Rudolph, G., Droschinsky, A.: A new subgraph crossover for Cartesian genetic programming. In: McDermott, J., Castelli, M., Sekanina, L., Haasdijk, E., García-Sánchez, P. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 294–310. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55696-3_19
22. Kantschik, W., Banzhaf, W.: Linear-graph GP - a new GP structure. In: Foster, J.A., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A. (eds.) EuroGP 2002. LNCS, vol. 2278, pp. 83–92. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45984-7_8
23. Kosiol, J., Fritsche, L., Schürr, A., Taentzer, G.: Double-pushout-rewriting in S-cartesian functor categories: rewriting theory and application to partial triple graphs. J. Log. Algebraic Methods Program. **115**, 100565 (2020). https://doi.org/10.1016/j.jlamp.2020.100565
24. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Complex Adaptive Systems (1992)
25. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. RAIRO Theor. Informat. Appl. **39**(3), 511–545 (2005). https://doi.org/10.1051/ita:2005028
26. Machado, P., Nunes, H., Romero, J.: Graph-based evolution of visual languages. In: Di Chio, C., Brabazon, A., Di Caro, G.A., Ebner, M., Farooq, M., Fink, A., Grahl, J., Greenfield, G., Machado, P., O'Neill, M., Tarantino, E., Urquhart, N. (eds.) EvoApplications 2010. LNCS, vol. 6025, pp. 271–280. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12242-2_28
27. Niehaus, J.: Graphbasierte Genetische Programmierung. Ph.D. thesis, Technical University of Dortmund (2003). http://hdl.handle.net/2003/2744
28. Nobile, M.S., Besozzi, D., Cazzaniga, P., Mauri, G.: The foundation of evolutionary Petri Nets. In: Balbo, G., Heiner, M. (eds.) Proceedings of the International Workshop on Biological Processes & Petri Nets, Milano, Italy, CEUR 24 June 2013. http://ceur-ws.org/Vol-988/paper6.pdf
29. Pereira, F.B., Machado, P., Costa, E., Cardoso, A.: Graph based crossover - a case study with the busy beaver problem. In: Banzhaf, W., Daida, J.M., Eiben, A.E., Garzon, M.H., Honavar, V. (eds.) Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation, GECCO 1999, Vol. 2, pp. 1149–1155. Morgan Kaufmann Publishers Inc., San Francisco (1999)
30. Plump, D.: Termination of graph rewriting is undecidable. Fundam. Informaticae **33**(2), 201–209 (1998). https://doi.org/10.3233/FI-1998-33204
31. Plump, D.: Term graph rewriting. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2, pp. 3–61. World Scientific (1999). https://doi.org/10.1142/9789812815149_0001
32. Zschaler, S., Mandow, L.: Towards model-based Optimisation: using domain knowledge explicitly. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 317–329. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_24