Artificial Intelligence 256 (2018) 105–129

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

A computational framework for conceptual blending

Manfred Eppe [a,b,*], Ewen Maclean [d], Roberto Confalonieri [e,c], Oliver Kutz [e], Marco Schorlemmer [c], Enric Plaza [c], Kai-Uwe Kühnberger [f]

[a] University of Hamburg, Germany
[b] International Computer Science Institute, Berkeley, USA
[c] IIIA-CSIC, Barcelona, Catalonia, Spain
[d] University of Edinburgh, UK
[e] Free University of Bozen-Bolzano, Italy
[f] University of Osnabrück, Germany

ARTICLE INFO

Article history:
Received 1 September 2016
Received in revised form 3 November 2017
Accepted 23 November 2017
Available online 2 December 2017

Keywords:
Computational creativity
Conceptual blending
Cognitive science
Answer set programming

ABSTRACT

We present a computational framework for *conceptual blending*, a concept invention method that is advocated in cognitive science as a fundamental and uniquely human engine for creative thinking. Our framework treats a crucial part of the blending process, namely the generalisation of input concepts, as a search problem that is solved by means of modern answer set programming methods to find commonalities among input concepts. We also address the problem of pruning the space of possible blends by introducing metrics that capture most of the so-called *optimality principles*, described in the cognitive science literature as guidelines to produce meaningful and serendipitous blends. As a proof of concept, we demonstrate how our system invents novel concepts and theories in domains where creativity is crucial, namely mathematics and music.

1. Introduction

Creativity is an inherent human capability that is crucial for the development and invention of new ideas and concepts [3]. This paper addresses a kind of creativity which Boden [3] calls *combinational*, and which has been studied by Fauconnier and Turner [27] in their framework of *conceptual blending*. In brief, conceptual blending is a process where one invents a novel concept, called the *blend*, by combining two familiar input concepts. For illustration, consider the classical example of blending the concepts *house* and *boat* (e.g. [34,27]). A possible result is the invention of a *house-boat* concept, where the medium on which a house is situated (land) becomes the medium on which boat is situated (water), and the inhabitant of the house becomes the passenger of the boat. Another possible blend is the *boat-house*, where the boat 'inhabits' the house.

An inherent computational problem of conceptual blending is to find a common ground, called *generic space*, between the two input concepts [27]. For example, the *house-boat* blend has the generic space of a person being inside an object that is not situated on any medium (or that is situated on a more general medium). Once the generic space has been identified, one can develop possible blends by specialising the generic space with elements from the input concepts in a meaningful way. However, this is not trivial because the naive union of input spaces can lead to inconsistencies. For example, the medium on which an object is situated can not be land and water at the same time. Hence, before combining the input

* Corresponding author at: University of Hamburg, Germany.
E-mail addresses: eppe@informatik.uni-hamburg.de (M. Eppe), emaclea2@inf.ed.ac.uk (E. Maclean), roberto.confalonieri@unibz.it (R. Confalonieri).

https://doi.org/10.1016/j.artint.2017.11.005

concepts, it is necessary to generalise at least one medium assignment. Another problem is the huge number of possible blends, which are often not meaningful. For example, blending *house* and *boat* such that the house becomes the passenger of the boat – imagine a house-transporting cargo vessel – is not very convincing. Consequently, one has to prune the search space by ruling out such low-quality blends.

Conceptual blending is perceived as a milestone in human cultural development [27]. The main motivation behind blending from an AI perspective is to find a computational interpretation of the human blending process, which could be an equally important milestone in the development of intelligent agents and autonomous systems. The value of conceptual blending for the development of creative systems has been witnessed by several works in the field of Artificial Intelligence and cognitive science, where particular implementations of this cognitive theory have been proposed [83,63,64,33,36].

As we show in our survey in Sec. 5, existing approaches propose computational characterisations of conceptual blending by using different formal representations for the input spaces and different techniques for performing the blending operation, and for the evaluation of the blends. For instance, Goguen and Harrell [33] logically formalise conceptual blending in terms of algebraic theories, Pereira [64] uses concept maps and frames, and rules and constraints to implement blend evaluation, while Veale and Donoghue [83] focus on the use semantic networks. The survey shows that providing a full computational account for conceptual blending is very challenging, in particular for the following two reasons:

- When combining two input spaces, the generic space is of particular importance to steer possible variations of blends. However, computing the generic space is a challenging issue (e.g., [76]), especially for expressive representation languages. Most existing blending frameworks are therefore not capable of computing a generic space automatically.
- Having identified a generic space, there typically remains a huge number of possible combinations to generate blends. To prune this result space, blends need to be evaluated. One way to do this is to check their consistency and to apply certain quality metrics.

In this paper, we address these issues and ask the following question:

How can we orchestrate the blending of input concepts in a computationally efficient and feasible way, that is faithful to the cognitive theory of conceptual blending and can therefore be considered as computationally creative?

To answer this question, we build a general creative computational framework for conceptual blending that allows the creation and evaluation of new blended concepts. The main contributions of this paper are as follows:

- We provide a blending framework that accepts input concepts in form of semiotic systems (see Sec. 2.2). Herein, we use algebraic specifications similar to those proposed by Goguen [34, Def. 1], with the difference that we assign *priorities* not only to constructors, i.e., operators, but also to sorts, predicates, and axioms. This extra level of knowledge allows us to guide the generalisation search process and create meaningful generalisations of the input spaces more efficiently, and we also use it for the evaluation of blends.
- We automate the discovery of generic spaces by applying *amalgams*, a notion known from case-based reasoning [61] (see Sec. 2.3). This process coordinates the interleaved generalisation and combination of input concepts as a non-monotonic search problem. We solve this search problem by using the declarative framework of Answer Set Programming (ASP) (e.g., [2]), as described in Sec. 3.
- We evaluate blends by re-interpreting the optimality principles of Fauconnier and Turner [27] and give them a full computational account (see Sec. 2.1 and 3.6). This helps to prune the search space.
- As a proof of concept, we implement our framework as an exploratory creative tool that can create interesting blends in the domain of mathematics and music. We also reproduce several blends that can be found in the literature. Finally, we show how our framework finds blends that belong to different domains (see Sec. 4).
- We provide a survey to characterise existing computational blending systems (see Sec. 5) and position our own work within the state of the art.

2. Preliminaries

Our framework is inspired by the cognitive theory of *conceptual blending* as presented in Fauconnier and Turner [27], whose underlying principles are described in detail in Sec. 2.1. To realise these principles computationally, we follow the work by Goguen [34], who provides a category theoretical account of blending (see Sec. 2.2). Moreover, to make Goguen's work computationally feasible, we implement blending as an *amalgam*-based workflow [61], a notion that was developed in case-based reasoning (see Sec. 2.3). The implementation framework for the amalgam-based workflow is *Answer Set Programming* (ASP), for which we provide a brief background in Sec. 2.4.

2.1. Cognitive principles of blending

Creativity, understood as an unfamiliar combinations of familiar ideas, goes back to the notion of *bisociation*, the idea that creativity is often a result of an intersection and selective combination of rather distinct frames of reference, presented

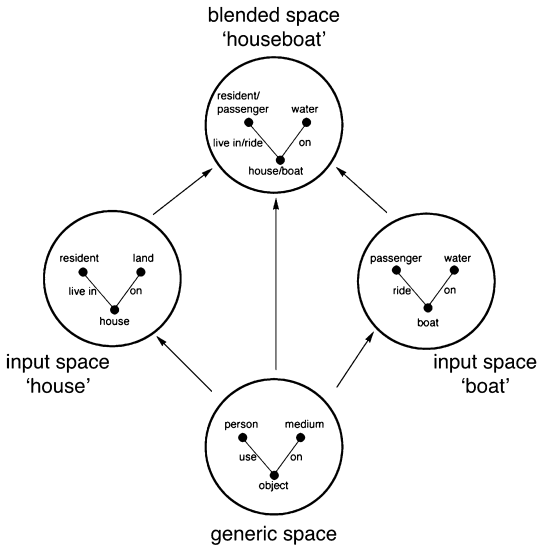

Fig. 1. The 'houseboat' blend, adapted from [33].

by Arthur Koestler in his book *The Act of Creation* in 1964 [46]. Based on these basic intuitions, within the cognitive sciences these ideas have been further developed into more concrete approaches of how to produce novel ideas (which may be concepts, theories, solutions to problems, works of art, etc.). One particular such approach, known as the theory of *conceptual blending* or conceptual integration has been proposed by Fauconnier and Turner [26] as a kind of primitive or fundamental cognitive operation underlying much of everyday thought and language. The process by which two concepts are *blended* into a novel idea is seen as a complex event in which particular elements and their relations pertaining to the initial two concepts are combined selectively into a new whole, which is understood to be structurally richer, in a sense we will make precise below, than the mere commonalities of the two concepts.

Fauconnier's view of concepts is prior to the notion of blending, and his *Mental Spaces Theory* is a highly influential cognitive theory of *meaning construction*, developed in [24] and [25]. According to Fauconnier, meaning construction involves two processes: (1) the building of *mental spaces*; and (2) the establishment of *mappings* between those mental spaces. Moreover, the mapping relations are guided by the local discourse context, which means that meaning construction is always situated or context-dependent.

Fauconnier and Turner [27] describe several constitutive elements of conceptual blending. These are (i) the *input spaces* that are to be blended, (ii) a partial *cross-space mapping* that connects counterparts in the input mental spaces, (iii) a *generic space* that is an abstraction from what the input spaces have in common, (iv) a *blending* operation that produces a blend of the input spaces and into which the structure of the input spaces is selectively projected, and (v) an *emergent structure*, i.e., structure that is a synergistic gain to the naive sum of the structure of input spaces.

These constitutive elements can be organised in a *conceptual integration network*, i.e., the network of all input spaces, generic spaces and blend spaces together with the selective projections that model a particular blending process. Finally, Fauconnier and Turner propose certain *optimality principles* that govern the blending process, and that can be taken as a way to assess the quality of a blend. Let us briefly review these constitutive elements and optimality principles as put forth in Fauconnier and Turner's model, using the *house-boat* blend depicted in Fig. 1 as an illustrative example.

2.1.1. Constitutive elements of conceptual blending

According to cognitive theory, the conceptual blending process involves the following elements:

- *Input Spaces:* Fauconnier and Turner [27] consider the input spaces of a blend to be *mental spaces* – small conceptual packets constructed as we think and talk, for purposes of local understanding and action. According to Fauconnier [24], mental spaces are connected to long-term schematic knowledge, by means of *frames*, which model how certain elements and relations are organised as a package that we already know about. Often these frames are so-called *image schemas* [52], described by Johnson [44] as 'a recurring, dynamic pattern of our perceptual interactions and motor programs that gives coherence and structure to our experience'. Image schemas are also critical for the semantic compositionality of language, and they are, therefore, often used by cognitive linguists to implement embodied construction grammars [22,23,81].

  Moreover, mental spaces are built on-demand, for a particular linguistic situation, resulting in concrete and temporary packets of conceptual structure. For example, contrary to a naive view of meaning, there is no fixed understanding of the concepts of *house* and *boat*: there are mental spaces for *house* and *boat* that are constructed in particular linguistic situations. Hence, one cannot assume that input mental spaces are the same in all possible output blends, since the

content of a mental space is neither constant nor preexisting to the context in which they are used. For example, a house in a poor developing country has different properties than a house in a rich urban area. However, many computational blending systems (e.g., [37,83]) that emanate from the work by Fauconnier and Turner [27] often simplify this aspect of their theory and consider blending to be an operation that takes two input spaces as given and generates one or more output spaces, i.e., blends. The approach taken in this paper is to formalise the notion of mental spaces being built on-demand, but allowing heuristic generalisation and adaptation of the input spaces, as well as the computation of a blend. This is demonstrated in the examples shown in Sec. 4.2 and Sec. 4.3.

- *Cross-space mapping:* According to [27, p. 41], *"[...] a partial cross-space mapping connects counterparts in the input mental spaces."* As an example consider the *house-boat* blend, where the passenger of the boat is mapped to the resident of the house, and the *ride* relation between *passenger* and *boat* is mapped to the *live-in* relation between *resident* and *house*.
- *Generic Space:* The cross-space mapping points to what might constitute the generic space, which essentially contains what is common to or shared among the two input mental spaces. For example, the mapping between *passenger* and *resident* in the house-boat blend suggests a generic space with a concept *person* that generalises the concepts of *passenger* and *resident*.
- *Blend:* The most precise statement about the actual blending operation is possibly the following, from Fauconnier and Turner [27, p. 47]: *"In blending, structure from two input mental spaces is projected to a new space, the blend. Generic spaces and blended spaces are related: Blends contain generic structure captured in the generic space but also contain more specific structure, and they can contain structure that is impossible for the inputs [...]"*

  The blend is neither the intersection nor the union of the input spaces. Parts of the input spaces are selectively projected into the blend, other parts do not become part of the blend. For example, blending *house* and *boat* to *house-boat* requires to select the concept *water* from the *boat* input space, but not the concept *land* from the *house* input space.

  The close relationship between the generic and the blend spaces, makes it important for any formal model of blending to come to grips how the generic space arises from the blending process as well as what role it plays.
- *Emergent Structure – Composition, Completion and Elaboration:* The emergent structure refers to additional structure in the blend that is not directly copied from the inputs. Emergent structure is built through three processes: composition, completion, and elaboration. First, *composition* simply brings the elements of the input spaces together, without any further effect; second, *completion* is the inference of additional information, obtained from accessing related frames and scenarios; and third, *elaboration* intuitively means simulating the behaviour of the elements in the blend interacting among them. As emphasised in Fauconnier and Turner [27, p. 49], *"[t]he creative possibilities of blending stem from the open-ended nature of completion and elaboration."*

  From an Artificial Intelligence viewpoint, composition combines a selective projection of inputs. Completion adds background knowledge to the blend, until it is completed, i.e., has all that is needed for the purpose at hand. Elaboration involves reasoning or inference, in the sense that background knowledge is used to reason about actions that some elements can perform, or actions that can be performed on some elements, in order to elucidate the eventual consequences of the blended mental space.
- *Conceptual Integration Network:* As stated in Fauconnier and Turner [27, p. 44], *"Building an integration network involves setting up mental spaces, matching across spaces, projecting selectively to a blend, locating shared structures, projecting backward to inputs, recruiting new structure to the inputs or the blend, and running various operations in the blend itself."*

2.1.2. Optimality principles

Optimality principles play the role of an assessment measure for blends. As we mention earlier, two input spaces can be blended together in many different ways, and finding a particular blend that is creative and useful is not trivial.

The optimality principles that Fauconnier and Turner [27] mention compete with each other, and satisfying one principle may dissatisfy another one. In the following, we summarize how Fauconnier and Turner [26,27] define the particular principles that we consider in this paper, and we provide examples by explaining their role in the *house-boat* blend.

- *Topology Principle:* *"For any input space and any element in that space projected into the blend, it is optimal for the relations of the element in the blend to match the relations of its counterpart."* [26]

  In terms of the *house-boat* blend, if the elements *passenger* and *resident* are blended to a *passenger-resident* of a *house-boat*, then the *lives-in* relation between the *resident* and the *house* remains the *lives-in* relation between the *passenger-resident* and the *house-boat*.
- *Pattern Completion Principle:* *"Other things being equal, complete elements in the blend by using existing integrated patterns as additional inputs. Other things being equal, use a completing frame that has relations that can be the compressed versions of the important outer-space vital relations between the inputs."* [27, p. 328]

  This principle is related to the work on image schemas by Lakoff [51]. Image schemas are completing frames that are abstract versions of the input spaces. For example, in the *house-boat* blend, the CONTAINER image schema is used as an abstraction for a house, which is a container for a resident and a boat, which is a container for a passenger. The pattern completion principle is related to the *emergent structure* of a blend, because emergent structure only arises through completion and elaboration.
- *Integration Principle:* *"The blend must constitute a tightly integrated scene that can be manipulated as a unit. More generally, every space in the blend structure should have integration."* [26]

For example, one could blend a scene of a house and a scene of a boat as the simple union of both scenes, so that there is a house with a resident and a boat with a passenger in the blend. However, this is not perceived well as a unit. A better blend in terms of integration is the new concept of *house-boat*, which can be treated much better as a unit.

- *Maximisation of Vital Relations Principle:* *"Other things being equal, maximise vital relations in the network. In particular, maximise the vital relations in the blended space and reflect them in outer-space vital relations."*
  Intensification of Vital Relations Principle: *"Other things being equal, intensify vital relations."* [27, p. 330][1]

  According to Fauconnier and Turner [27], blending does not only happen to creatively invent new concepts, but it also serves as a means to compress the information in the input spaces using so-called *vital relations*, which are fundamental in the particular network of interest. As examples, Fauconnier and Turner [27] mention cause-effect, time, space, identity, change, and uniqueness relations. If such relations exist between the input spaces, then blending causes them to reappear in compressed form in the blend. Maximising and/or intensifying vital relations in a blend also increases the degree of compression that a blend produces. This underpins not only the importance of blending as a means for creativity, but also for efficient cognitive operation.
- *Web Principle:* *"Other things being equal, manipulating the blend as a unit must maintain the web of appropriate connections to the input spaces easily and without additional surveillance or computation."* [27, p. 331]

  This refers to relations between spaces in the network. For example, placing a houseboat into a new environment such as a river scene maintains the connections to the input spaces of house and boat: the relation between the *resident* of a *house* and the *passenger-resident* of a *house-boat* is still there. As another example, consider the case where some form of residential fee has to be paid for a *house-boat*. If the fee is raised for a *house-boat*, then one should be able to infer that it is also raised for a house.
- *The Unpacking Principle:* *"Other things being equal, the blend all by itself should prompt for the reconstruction of the entire network."* [27, p. 332]

  For example, by focusing on the concept of a *house-boat* we still can access the concept of *house* with its properties and its relationship to other concepts. Similarly, we can also access the notion of *boat* with its properties and its relationship to other concepts.
- *Relevance Principle:* This is sometimes also called the *Good Reason Principle*. Fauconnier and Turner [27, p. 333] describe it as follows: *"Other things being equal, an element in the blend should have relevance, including relevance for establishing links to other spaces and for running the blend. Conversely, an outer-space relation between the inputs that is important for the purpose of the network should have a corresponding compression in the blend."*

  What eventually constitutes the *house-boat* concept will very much depend on what we are pursuing when blending. So, if the relation *live-in* is relevant, it should be included in the blend. But maybe another relation (e.g., *number-of-rooms*) might not be relevant, and should, therefore, not be considered in the blend.

In general, the optimality principles are defined in a vague cognitive way, and not meant to be perceived as rigid rules. However, if one is to computationalize conceptual blending, then it is necessary to find some way of formalizing the optimality principles. Our attempt to do so results in an encoding of some of the principles as a quantitative evaluation metric, as presented in Sec. 3.6.

2.2. A categorical view on blending

Goguen [34] proposes to model the input concepts of blending as *semiotic systems*, which are essentially *algebraic specifications* described in a logical representation language. The main advantage of this approach is being able to provide a general enough, but computational feasible representation, while being able to resolve inconsistencies. We represent semiotic systems using the Common Algebraic Specification Language (CASL) [60], and extend it by considering priority information for operators, sorts, predicates and axioms as follows:

Definition 1 *(Prioritised CASL specification).* A prioritised CASL specification (PCS) is a tuple $\mathfrak{s} = \langle \mathcal{ST}, \lesssim, \mathcal{O}, \mathcal{P}, \mathcal{A}, prio \rangle$ with:

- a set $\mathcal{ST}$ of sorts, along with a preorder $\lesssim$ that defines a sub-sort relationship;
- a set $\mathcal{O}$ of operators $o : s_1 \times \cdots \times s_n \mapsto s_r$ that map zero or more objects of argument sorts $s_1, \cdots, s_n$ to a range sort s_r;
- a set $\mathcal{P}$ of predicates $p : s_1 \times \cdots \times s_n$ that map zero or more objects of argument sorts $s_1, \cdots, s_n$ to Boolean values;
- a set $\mathcal{A}$ of axioms;
- a function $prio : \mathcal{ST} \cup \mathcal{O} \cup \mathcal{P} \cup \mathcal{A} \mapsto \mathbb{N}^0$ that assigns a priority to all elements in a specification. The lower the number, the lower the priority.

We refer to the listed constituents of a PCS simply as the *elements* of a PCS, denoted by e, and we say that two PCS are *compatible* if all of their elements, except the priority function, are equal. Some elements of a PCS are marked as

[1] We here merge 'maximise/intensify' of vital relations into one principle as they are not distinguished sharply in the original text.

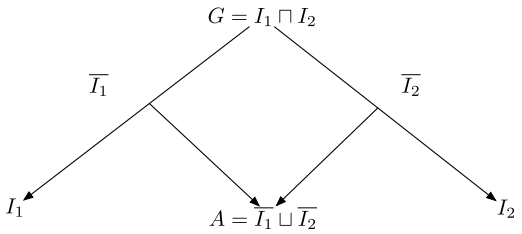

Fig. 2. A diagram of an amalgam $A = \bar{I}_1 \sqcup \bar{I}_2$ from inputs I_1 and I_2. The arrows indicate the subsumption relation $\sqsubseteq$.

data-elements to represent a fixed background vocabulary (see e.g. [34, Def.1]). The priority for data elements is always 0, where 0 is the lowest priority value.

A good motivation for using priorities can be found in prior work [20]. Intuitively, it is an instrument to express the salience level of elements in concept, and it also helps algorithmically to prune the search space, as described in Sec. 3. Goguen [34]'s algebraic view on blending suggests to compute the blend of input specifications as their categorical *colimit* (see e.g., [69]). The colimit is a general unification operation for categories, similar to the *union* operation for sets. In this case the categories are algebraic signatures, so that the colimit can be used to unify algebraic specifications, as described by Mossakowski [57]. The colimit requires *morphisms* to be defined between the signatures of algebraic specifications, in particular between the generic space and the input concepts. Our blending algorithm uses colimit of algebraic specifications as the basis for composing input spaces, as described in Sec. 3.5. We do consider priorities when computing the colimit, but use them instead as a means to reward the compression of structure in a blend. This supports [27]'s optimality principles, as described in Sec. 3.5.

2.3. The notion of amalgams

As shown in [4], the process of conceptual blending can be characterised using *amalgams* [61], a notion that was developed in case-based reasoning as an approach to combine solutions coming from multiple cases. According to this approach, input concepts are generalised until a generic space is found, and pairs of generalised versions of the input concepts are *combined* to create blends.

Formally, the notion of amalgams can be defined in any representation language $\mathcal{L}$ for which a subsumption relation ($\sqsubseteq$) between formulas (or descriptions) of $\mathcal{L}$ can be defined. We say that a description I_1 subsumes another description I_2 ($I_1 \sqsubseteq I_2$) when I_1 is more general (or equal) than I_2. Next, for any two descriptions I_1 and I_2 in $\mathcal{L}$ we can define their *unification*, ($I_1 \sqcup I_2$), which is the *most general specialisation* of two given descriptions, and their *anti-unification*, ($I_1 \sqcap I_2$), defined as the *least general generalisation* of two descriptions, representing the most specific description that subsumes both. Intuitively, a unifier is a description that has all the information in both the original descriptions. The anti-unification $I_1 \sqcap I_2$ contains all that is common to both I_1 and I_2.

An *amalgam* of two descriptions is a new description that contains *parts from these original descriptions*. For instance, an amalgam of 'a red French sedan' and 'a blue German minivan' could be 'a red German sedan'; clearly, there are always multiple possibilities for amalgams, like 'a blue French minivan'. An amalgam of two descriptions is defined as follows.

Definition 2. A description $A \in \mathcal{L}$ is an amalgam of two inputs I_1 and I_2 (with anti-unification $G = I_1 \sqcap I_2$) if there exist two generalisations $\bar{I}_1$ and $\bar{I}_2$ such that (1) $G \sqsubseteq \bar{I}_1 \sqsubseteq I_1$, (2) $G \sqsubseteq \bar{I}_2 \sqsubseteq I_2$, and (3) $A = \bar{I}_1 \sqcup \bar{I}_2$ and A is consistent.

This definition is illustrated in Fig. 2, where the anti-unification of the inputs is indicated as G, and the amalgam A is the unification of two concrete generalizations $\bar{I}_1$ and $\bar{I}_2$ of the inputs.

The generalisation of a description of a language $\mathcal{L}$ is usually done by means of generalisation operators [61]. Formally, a generalisation operator is defined as:

$$\gamma(I) = \{\bar{I} \in \mathcal{L} \mid \bar{I} \sqsubseteq I \text{ and } \nexists \bar{I}' \text{ s.t. } \bar{I} \sqsubset \bar{I}' \sqsubset I\} \tag{1}$$

Intuitively, a generalisation operator 'transforms' a description into a more general (or equal) one. For instance, the description of a 'red French sedan' can be generalised to a 'red European sedan' but also to a 'coloured French sedan' or to a 'red French car', since, typically, more than one generalisation is possible. The anti-unification of our input descriptions – a 'coloured European car' – is obtained by keeping on generalising the descriptions until they are equal. The anti-unification

serves as an upper bound of the generalisation space to be explored and plays the role of the *generic space* in conceptual blending, which states the shared structure of both concepts.[2]

2.4. Answer Set Programming

Answer Set Programming (ASP) is a declarative approach to solve NP-hard search problems (see e.g. [31,2]). ASP is often used for realizing the deliberation and problem solving skills of autonomous systems (e.g., [16,19]) and other forms of logical reasoning, such as epistemic or abductive inference [18,17], decision making under uncertainty [8], and preference reasoning [7]. In this work, we explore its use for Computational Creativity and logical theory generalisation.

An ASP program is similar to a PROLOG program in that it follows a non-monotonic semantics, takes logic programming style Horn clauses as input, and uses negation-as-failure (NaF). However, instead of using Kowalski [47]'s SLDNF resolution semantics as in PROLOG, it employs Gelfond and Lifschitz [32]'s *Stable Model semantics*, which makes it truly declarative. This means that the order in which ASP rules appear in a Logic Program does not affect the solution. Furthermore, the Stable Model Semantics has the advantage that Answer Set Programs always terminate, while PROLOG programs do not. For example, given a program $not\ p \leftarrow q.$ and $not\ q \leftarrow p.$, asking whether p holds results in an infinite loop for PROLOG, while ASP returns two stable models as solution, namely the sets $\{p\}$ and $\{q\}$.

An ASP program consists of a set of rules, facts and constraints. Its solutions are called *Stable Models* (SM). In this paper we only consider so-called *normal* rules [2], which are written as:

$$a_0 \leftarrow a_1, \ldots, a_j, not\ a_{j+1}, \ldots, not\ a_n \tag{2}$$

in which $a_1, ..., a_n$ are atoms and *not* is negation-as-failure. When $n = 0$ the rule $a_0 \leftarrow$ is called a *fact* and the $\leftarrow$ is omitted. A constraint is a rule of the form $\leftarrow a_1, \ldots, a_j, not\ a_{j+1}, \ldots, not\ a_n$. Constraints are rules that are used to discard some models of a logic program.

The stable models of an ASP program are defined in terms of the so-called *Gelfond–Lifschitz reduction* [32]. Let $\mathcal{L}_P$ be the set of atoms in the language of a normal logic program P, then for any set $M \subseteq \mathcal{L}_P$, the Gelfond–Lifschitz reduction P^M is the definite logic program obtained from P by deleting:

(i) each rule that has a formula *not a* in its body with $a \in M$, and
(ii) all formulæ of the form *not a* in the bodies of the remaining rules.

M is called a *stable model* of P if and only if M is the minimal model of P^M. A stable model M of an ASP program P contains those atoms that satisfy all the rules in the program and, consequently, represents a solution of the problem that the program P represents.

ASP is interesting because it can capture complex knowledge representation problems, and also because efficient ASP implementations, such as *clingo* [29], exist. The *clingo* solver offers a step-oriented, incremental approach that allows to control and modify an ASP program at run-time, without the need of restarting the grounding of the solving process from scratch. To this end, a program is partitioned into a base part, describing the static knowledge independent of a step parameter t, a cumulative part, capturing knowledge accumulating with increasing t, and a volatile part specific for each value of t. The grounding and integration of these subprograms into the solving process is completely modular and controllable from a scripting language such as Python.

The ASP implementation in this paper follows this methodology of specifying and solving a problem incrementally. For further details about incremental solving, we refer to [30], where several examples can be found.

3. Blending with amalgams

We employ an interleaved declarative-imperative amalgam-based workflow to search for generalisations of input spaces that produce and evaluate consistent blends. The blending theory in principle allows for any finite number $S \geq 2$ input specifications, but in the implementation we use $S = 2$ for simplicity. The notion of consistency in our work refers to logical consistency of algebraic specifications.

3.1. System description

The workflow of our system is depicted in Fig. 3. Most of the reasoning is done using Answer Set Programming (ASP) by generating a sequence of theory transitions. The ASP implementation is combined with Python scripts that perform external information processing if necessary. The main purpose of the implementation is to realize the theory transitions that occur during the generalisation process. A theory transition can either be the removal of an element (generalisation)

[2] From a cognitive point of view one could argue if the generic space plays either a central or marginal role in blending. From a mathematical point of view, any relationship between input space specifications will determine a generic space. By modelling blending as using amalgams and colimits it makes sense to make this generic space explicit and a constituent part of our computational realisation of blending.

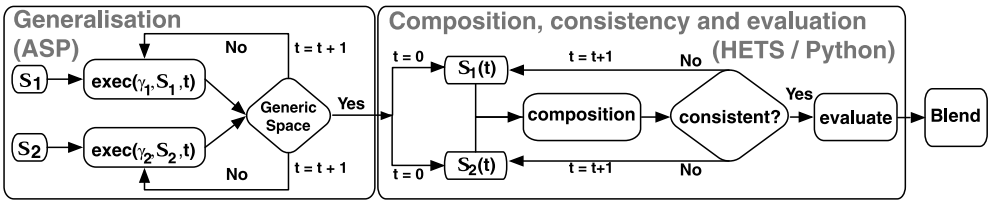

Fig. 3. Amalgam-based workflow.

or the renaming of an element in a specification. Towards this, the input PCS $\mathfrak{s}_1$, $\mathfrak{s}_2$ are first translated into ASP facts, as described in Sec. 3.2. Second, $\mathfrak{s}_1$, $\mathfrak{s}_2$ are undergoing sequences of theory transitions that are guided by an ASP solver until a generic space is found. Each transition is represented by a fact $exec(\gamma, \mathfrak{s}, t)$, where t is an iterator and γ is a transition operator that removes or renames elements in the input specifications (see Sec. 3.3). The execution of transition operators is repeated until the generalised versions of the input specifications are compatible in the sense of Definition 1. We write $\mathfrak{s}(t)$ to denote the specification that results form the t-th transition of $\mathfrak{s}$. For example, after the first theory transition, the house concept might be the concept of a house that is not situated on any medium. Note that in iterative ASP-solving, an iterator t causes atoms to be instantiated for a predefined number of steps (see e.g., [28,29]).

In order to find consistent blends, we apply the category-theoretical *colimit* [57] to compose transitioned input specifications. To this end, we use the HEterogeneous Tool Set (HETS) [58]. The colimit is applied on different combinations of transitions, and for each result we query theorem provers such as *eProver* and *darwin*, which are embedded in HETS, in order to check for consistency. An example for a consistent house-boat blend is the combination of the generalised *boat* on the medium water, but without a passenger, and a generalised *house* with a resident, but without a medium, where 'passenger' in 'boat' is renamed to 'resident' in house.

To eliminate uninteresting blends from our search process, we consider that more promising blends are those that require less generalisations. Consequently, we go from less general generalisations to more general generalisations and stop when a consistent colimit is achieved. Thereafter, the result is evaluated using certain metrics that are inspired by Fauconnier and Turner [27]'s *optimality principles* of blending to assess the quality of the blend (see Sec. 3.5). Note that different stable models, and therefore different generalisations and renamings, can be found by the ASP solver. Each combination leads to a different set of blends.

3.2. Modelling algebraic specifications in ASP

In order to find the generic space and to avoid inconsistencies that arise from the naive combination of input specifications, we weaken prioritised CASL specifications using transition operators in a step-wise search process. Firstly, renaming operators modify specifications by renaming their elements. Secondly, removal operators generalise algebraic specifications by removing operators, sorts, predicates or axioms. In the following, we use t to denote a step-counter that represents the number of transitions that a specification has undergone. We assume that sorts, operators and axioms for data (in the sense of Definition 1) are common among input specifications, so that we do not need to consider them in the ASP-based reasoning process. With this, we represent prioritised CASL specifications in ASP as follows:

- For each *sort* s in a specification $\mathfrak{s}$ with a parent sort s_p we state the facts:

$$sort(\mathfrak{s}, s, t) \tag{3a}$$

$$hasParent(\mathfrak{s}, s, s_p, t) \tag{3b}$$

A fact (3a) assigns a sort s to a specification $\mathfrak{s}$ at a step t, and (3b) assigns a parent sort.

- For each *operator* $o : s_1 \times \cdots \times s_n \mapsto s_r$ in a specification $\mathfrak{s}$ we have:

$$op(\mathfrak{s}, o, t) \tag{4a}$$

$$opHasSort(\mathfrak{s}, o, s_1, 1, t) \quad \cdots \quad opHasSort(\mathfrak{s}, o, s_n, n, t) \tag{4b}$$

$$opHasSort(\mathfrak{s}, o, s_r, rng, t) \tag{4c}$$

Facts (4b), (4c) state the argument sorts and the range sorts of an operator. $1 \ldots n$ determine the position of the argument sort and *rng* is simply a constant to denote that the sort is the range sort of the operator.

- Similarly, for each *predicate* $p : s_1 \times \cdots \times s_n$ in $\mathfrak{s}$ we generate the LP facts:

$$pred(\mathfrak{s}, p, t) \tag{5a}$$

$$predHasSort(\mathfrak{s}, p, s_1, 1, t) \cdots predHasSort(\mathfrak{s}, p, s_n, n, t) \tag{5b}$$

► For each *axiom* a we determine an equivalence class of that axiom, denoted eq^a, by passing the axiom to a respective Python function.[3] All logically equivalent axioms have the same equivalence class, e.g., $\neg a \vee b$ has the same equivalence class as $a \rightarrow b$. We also determine the elements, i.e, sorts, operators and predicates, that are involved in an axiom. This information is used in the preconditions of removal operators. For example, operator removal has the precondition that there exists no atom that involves the operator. Having computed the equivalence class eq^a and determined n_e elements that are involved in an axiom, we generate the following facts for each axiom a in a specification $\mathfrak{s}$.

$$ax(\mathfrak{s}, a, t) \tag{6a}$$

$$axInvolvesElem(\mathfrak{s}, a, e_1, t) \quad \dots \quad axInvolvesElem(\mathfrak{s}, a, e_{n_e}, t) \tag{6b}$$

$$axHasEqClass(\mathfrak{s}, a, eq^a, t) \tag{6c}$$

We represent the priority function *prio* of a PCS as facts $priority(\mathfrak{s}, e, v_p)$ for each element e in a specification $\mathfrak{s}$. Compatibility among two input specifications, as defined in Definition 1, is represented by atoms $incompatible(\mathfrak{s}_1, \mathfrak{s}_2, t)$, which are triggered by additional LP rules if, for $\mathfrak{s}_1$ and $\mathfrak{s}_2$, at step t, (i) sorts or subsort relationships are not equal, or (ii) operator or predicate names are not equal, or (iii) argument and range sorts of operators and predicates are not equal, or (iv) axioms are not equivalent.

3.3. Formalising transition operators in ASP

We consider two kinds of transition operators for PCS. The first kind involves the renaming of an element to the name of an element in another input specification. Since we consider syntactically equal elements to be conceptually equal in our implementation, this can be seen as identifying the commonalities among input spaces. The second kind is generalisation and involves the removal of an element in a specification. Generalisation operators are executed after all renaming actions have happened. Considering Definition 1, it is trivial to see that removal of elements in a PCS are generalisation operators in the sense of Eq. (1).

Each generalisation operator is defined via a precondition rule, an inertia rule, and, in case of renaming operations, an effect rule. Preconditions are modelled with a predicate *poss* that states when it is possible to execute a transition, and inertia is modelled with a predicate *noninertial* that states when an element of a specification stays as it is after a transition. Effect rules model how a transition operator changes an input specification. We represent the execution of a transition operator with atoms $exec(\gamma, \mathfrak{s}, t)$, to denote that a generalisation operator γ was applied to $\mathfrak{s}$ at a step t.

Removal operators. A fact $exec(rm(e), \mathfrak{s}, t)$ denotes the removal of an element e from a specification $\mathfrak{s}$ at a step t. It has different precondition rules for removing axioms (7a), operators (7b), predicates (7c) and sorts (7d):

$$poss(rm(e), \mathfrak{s}, t) \leftarrow ax(\mathfrak{s}, e, t), exOtherSpecWithoutEqivAx(\mathfrak{s}, e, t) \tag{7a}$$

$$poss(rm(e), \mathfrak{s}, t) \leftarrow op(\mathfrak{s}, e, t), exOtherSpecWithoutElem(\mathfrak{s}, e, t), \; 0\{ax(\mathfrak{s}, A, t) : axInvolvesElem(\mathfrak{s}, A, e, t)\}0 \tag{7b}$$

$$poss(rm(e), \mathfrak{s}, t) \leftarrow pred(\mathfrak{s}, e, t), exOtherSpecWithoutElem(\mathfrak{s}, e, t), \; 0\{ax(\mathfrak{s}, A, t) : axInvolvesElem(\mathfrak{s}, A, e, t)\}0, \tag{7c}$$

$$poss(rm(e), \mathfrak{s}, t) \leftarrow sort(\mathfrak{s}, e, t), exOtherSpecWithoutElem(\mathfrak{s}, e, t), \; 0\{ax(\mathfrak{s}, A, t) : axInvolvesElem(\mathfrak{s}, A, e, t)\}0 \; noOpUsesSort(\mathfrak{s}, e, t), noPredUsesSort(\mathfrak{s}, e, t), \; isNotParentSort(\mathfrak{s}, e, t) \tag{7d}$$

The precondition (7a) for removing an axiom from a specification is that an atom $exOtherSpecWithoutEqivAx(\mathfrak{s}, e, t)$ holds. Such atoms are produced, if there exists at least one other specification[4] that does not have an axiom of the same logical equivalence class. For the removal of other elements we have a similar precondition, i.e., $exOtherSpecWithoutElem(\mathfrak{s}, e, t)$, which denotes that an element can only be removed if it is not involved in another specification. Such preconditions are required to allow only generic spaces that are *least general* for all input specifications, in the sense that elements can not be removed if they are contained in all specifications. We also require operators, predicates and sorts not to be involved in any axiom before they can be removed (denoted by $0\{ax(\mathfrak{s}, A, t) : axInvolvesElem(\mathfrak{s}, A, e, t)\}0$). Precondition (7d) for removing sorts has the additional requirement that no operator or predicate with an argument or range of the sort

[3] Ideally, one would check for logical equivalence of axioms. However, since FOL is generally undecidable we check for syntactic equivalence of normalized versions of axioms in the current version of our framework. Logical equivalence would be more difficult to solve due to the undecidability of FOL.

[4] We focus on only two specifications in this paper, but the approach can in general also be applied to more than two specifications.

to be removed exists in the specification, which is implemented with *noOpUsesSort* and *noPredUsesSort* facts respectively. These are triggered by additional simple LP rules. Another condition for sort removal is that the sort is not the parent sort of another sort. Consequently, for sort removal, all axioms, operators and predicate that involve the sort must be removed first, and child sorts must also be removed first. The inertia rules for removing elements from a specification are quite simple:

$$noninertial(\mathfrak{s}, e, t) \leftarrow exec(rm(e), \mathfrak{s}, t) \tag{8}$$

All *noninertial* atoms will cause an element e to remain in a specification (see rule (12)).

Renaming operators. A fact $exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t)$ denotes the renaming of an element e of a specification $\mathfrak{s}$ to an element e' in a specification $\mathfrak{s}'$. In contrast to removal, renaming can only be applied to predicates, operators and sorts. Axioms are automatically rewritten according to the renamings of the involved elements. Again, we have different preconditions for renaming operators (9a), predicates (9b) and sorts (9c):

$$\begin{aligned} poss(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \leftarrow & op(\mathfrak{s}, e, t), op(\mathfrak{s}', e', t), \\ & not\ opSortsNotEquivalent(\mathfrak{s}, e, \mathfrak{s}', e', t), \\ & not\ op(\mathfrak{s}, e', t), not\ op(\mathfrak{s}', e, t), \mathfrak{s} \neq \mathfrak{s}' \end{aligned} \tag{9a}$$

$$\begin{aligned} poss(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \leftarrow & pred(\mathfrak{s}, e, t), pred(\mathfrak{s}', e', t), \\ & not\ predSortsNotEquivalent(\mathfrak{s}, e, \mathfrak{s}', e', t), \\ & not\ pred(\mathfrak{s}, e', t), not\ pred(\mathfrak{s}', e, t), \mathfrak{s} \neq \mathfrak{s}' \end{aligned} \tag{9b}$$

$$\begin{aligned} poss(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \leftarrow & sort(\mathfrak{s}, e, t), sort(\mathfrak{s}', e', t), \\ & not\ sort(\mathfrak{s}, e', t), not\ sort(\mathfrak{s}', e, t) \end{aligned} \tag{9c}$$

A common precondition for all three renaming operations is that the element e must exist in the specification $\mathfrak{s}$, and that e' must exist in $\mathfrak{s}'$. Furthermore, it must not be the case that e' is already part of $\mathfrak{s}$, and that e is part of $\mathfrak{s}'$. In case of renaming operators and predicates, the argument and range sorts of e and e' must also be equivalent for the renaming to become possible. For example, an operator $situatedOn : Object \mapsto Medium$ can not be mapped to an operator $usedBy : Object \mapsto Person$, which has a different range sort.

The inertia rules for renaming elements e in a specification are analogous to the inertial rule for removing elements:

$$noninertial(\mathfrak{s}, e, t) \leftarrow exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \tag{10}$$

For renaming, we have the following set of effect rules that assign the new name for the respective element:

$$sort(\mathfrak{s}, e', t+1) \leftarrow exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t), sort(\mathfrak{s}, e, t) \tag{11a}$$

$$hasParent(\mathfrak{s}, s', s_p, t+1) \leftarrow hasParent(\mathfrak{s}, s, s_p, t), exec(rename(s, s', \mathfrak{s}'), \mathfrak{s}, t) \tag{11b}$$

$$hasParent(\mathfrak{s}, s, s_p', t+1) \leftarrow hasParent(\mathfrak{s}, s, s_p, t), exec(rename(s_p, s_p', \mathfrak{s}'), \mathfrak{s}, t) \tag{11c}$$

$$opHasSort(\mathfrak{s}, o, s', n, t+1) \leftarrow opHasSort(\mathfrak{s}, o, s, n, t), exec(rename(s, s', \mathfrak{s}'), \mathfrak{s}, t) \tag{11d}$$

$$predHasSort(\mathfrak{s}, p, s', n, t+1) \leftarrow predHasSort(\mathfrak{s}, p, s, n, t), exec(rename(s, s', \mathfrak{s}'), \mathfrak{s}, t) \tag{11e}$$

$$op(\mathfrak{s}, o', t+1) \leftarrow exec(rename(o, o', \mathfrak{s}'), \mathfrak{s}, t), op(\mathfrak{s}, o, t) \tag{11f}$$

$$opHasSort(\mathfrak{s}, o', s, n, t+1) \leftarrow opHasSort(\mathfrak{s}, o, s, n, t), exec(rename(o, o', \mathfrak{s}'), \mathfrak{s}, t) \tag{11g}$$

$$pred(\mathfrak{s}, p', t+1) \leftarrow exec(rename(p, p', \mathfrak{s}'), \mathfrak{s}, t), pred(\mathfrak{s}, p, t) \tag{11h}$$

$$predHasSort(\mathfrak{s}, p', s, n, t+1) \leftarrow predHasSort(\mathfrak{s}, p, s, n, t), exec(rename(p, p', \mathfrak{s}'), \mathfrak{s}, t) \tag{11i}$$

$$axInvolvesElem(\mathfrak{s}, a, e', t+1) \leftarrow axInvolvesElem(\mathfrak{s}, a, e, t), exec(rename(e, e', \mathfrak{s}'), \mathfrak{s}, t) \tag{11j}$$

In general, the rules state that a specification will contain an element e' at a step $t+1$ if an element e has been renamed to e' at step t. Rules (11a)–(11e) state how renaming sorts affects the generalization. It also considers the effects on parent-child relations as well as predicate and operator arguments and range. Rules (11f), (11g) and (11h), (11i) describe the effects of renaming operators and predicates respectively. Rule (11j) states how the auxiliary predicate *axInvolvesElem* is affected by renaming.

Inertia. In order to use the inertia rules (8, 10), we need the following rules to state that elements e remain in a specification $\mathfrak{s}$ if they are inertial:

$$sort(\mathfrak{s}, e, t+1) \leftarrow not\ noninertial(\mathfrak{s}, e, t), sort(\mathfrak{s}, e, t) \tag{12a}$$

$$op(\mathfrak{s}, e, t+1) \leftarrow not\ noninertial(\mathfrak{s}, e, t), op(\mathfrak{s}, e, t) \tag{12b}$$

$$pred(\mathfrak{s}, e, t+1) \leftarrow not\ noninertial(\mathfrak{s}, e, t), pred(\mathfrak{s}, e, t) \tag{12c}$$

$$ax(\mathfrak{s}, e, t+1) \leftarrow not\ noninertial(\mathfrak{s}, e, t), ax(\mathfrak{s}, e, t) \tag{12d}$$

Updating axiom equivalence. When operators, predicates or sorts that are involved in an axiom are renamed, then the axiom's equivalent class changes. Determining logical equivalence of FOL axioms is a well understood research domain on its own, and we make use of existing theorem proving tools here. Towards this, we use an external Python function *renameEleAndGetNewEqClass* in rule (13) during the ASP solving process, which updates the equivalence class by querying theorem proving tools that determine a new equivalence class for an axiom if elements are renamed. This happens by accessing an internal dictionary of axioms within Python, that is built dynamically during the ASP grounding process.

$$\begin{aligned} axHasEqClass(\mathfrak{s}, a, eq^a_{new}, t+1) \leftarrow & axHasEqClass(\mathfrak{s}, a, eq^a, t), \\ & exec(rename(\mathfrak{s}, e_1, e_2, t), axInvolvesElem(\mathfrak{s}, a, e_1, t), ax(\mathfrak{s}, a, t), \\ & eq^a_{new} = @renameEleAndGetNewEqClass(eq^a, e_1, e_2) \end{aligned} \tag{13}$$

Additional rules that update the *axInvolvesElem* atoms if elements are renamed are also part of our implementation.

3.4. Transition search process

The search process that we use ASP for, is to find a generic space and transitioned versions of the input specifications which lead to a consistent blend. This is done by successively applying transition operators to the input specifications. Herein, we first apply only renaming operators to find the commonalities among input specifications. We then 'cut off' elements that the input specifications do not have in common by applying removal operators. Note that a simple intersection operation (as in set theory) is not possible because of the preconditions that the transition operators have. These impose a strong restriction on the allowed order in which transition operators may occur and make the problem inherently non-monotonic. A sequence of transition operators defines a *transition path*, which itself consists of a *commonalisation path* followed by a *removal path*.

Definition 3 *(Commonalisation, removal, and transition paths).* Let $\mathcal{S} = \{\mathfrak{s}_1, \ldots, \mathfrak{s}_{|\mathcal{S}|}\}$ be input specifications to be blended. Let $\{\gamma_1, \ldots, \gamma_n\}$ be renaming operators and $t_1 < \cdots < t_n$ be steps. Then we call a set of atoms $C = \{exec(\gamma_1, \mathfrak{s}_x, t_1), \cdots, exec(\gamma_n, \mathfrak{s}_y, t_n)\}$ a commonalisation path of $\mathcal{S}$ (with $\mathfrak{s}_x, \mathfrak{s}_y \in \mathcal{S}$). Let $\{\gamma_{n+1}, \ldots, \gamma_m\}$ be removal operators and $t_{n+1} < \cdots < t_m$ be steps. Then we call a set of atoms $R = \{exec(\gamma_{n+1}, \mathfrak{s}_u, t_{n+1}), \cdots, exec(\gamma_m, \mathfrak{s}_v, t_m)\}$ a removal path of $\mathcal{S}$ (with $\mathfrak{s}_u, \mathfrak{s}_v \in \mathcal{S}$). We call $C \cup R$ a transition path of $\mathcal{S}$.

We refer to transition paths in Section 3.5, when we describe how we combine different transitioned versions of the input specifications, which are generated by prefixes of removal paths. Formally, the *prefix* of a removal path is the subsequence of its first l removal operators.

Definition 4 *(Removal path prefix).* Given a removal path $R = \{exec(\gamma_{n+1}, \mathfrak{s}_u, t_{n+1}), \cdots, exec(\gamma_m, \mathfrak{s}_v, t_m)\}$, we call $R^{pre} = \{exec(\gamma_{n+1}, \mathfrak{s}_u, t_{n+1}), \cdots, exec(\gamma_l, \mathfrak{s}_w, t_l)\}$ a prefix of R if $l \leq m$ (with $\mathfrak{s}_u, \mathfrak{s}_v, \mathfrak{s}_w \in \mathcal{S}$).

A general problem of amalgams is that the space of possible transition paths and hence also the space of possible blends is usually very huge. To prune the search space, we only consider those transition paths where elements with a low priority are removed before elements with a higher priority, and where elements with a high priority are commonalised before elements with a low priority.[5]

Since we have separated the commonalisation and removal process we need to decide when we switch from the one to the other. The simplest solution would be to switch when all possible commonalities have been found, but this would prevent the system from being flexible in that it would not generate blends where not all possible commonalisations are considered. As an example, consider blending house and boat to obtain the boat house concept, described in Sec. 4.1. In this blend, not all commonalities are considered but the concept of a *boat-house* is definitely interesting. Hence, we define a predicate *findCrossSpaceMapPhaseFinished*(t) which determines the step t when the search for the cross space mapping, i.e., the commonalities is finished. This is used in the following choice rule for the commonalisation search process:

[5] To also allow for the consideration of semi-optimal generalisation paths, one could relax the restriction by allowing a limited number of generalisation operators within a path that do not respect the cost order.

$$1\{exec(a, \mathfrak{s}, t) : renameAct(a, \mathfrak{s}), poss(a, \mathfrak{s}, t)\}1 \leftarrow \text{not } findCrossSpaceMapPhaseFinished(t) \tag{14}$$

The *renameAct* predicate denotes that a is a renaming operator. The removal and generic space search process is done using another choice rule as follows:

$$1\{exec(a, \mathfrak{s}, t) : removeAct(a, \mathfrak{s}), poss(a, \mathfrak{s}, t)\}1 \leftarrow notGenericReached(t), findCrossSpaceMapPhaseFinished(t). \tag{15}$$

The predicate *notGenericReached* is triggered when the generic space has been reached, i.e., when the search terminates because the generalised versions of all input specifications are equal. This is the case when all axioms are logically equivalent, and all sorts, predicates and operators are syntactically equal. The *removeAct* predicate denotes that a is a removal operator.

3.5. Composition of transitioned input spaces

The transition part of our framework generates one stable model for each combination of transition paths that leads to a generic space. The next step in the amalgam-based workflow is to compose transitioned versions of input specifications to generate a candidate blend (see Fig. 3). The key component of this composition process is the categorical colimit [57] of the generalised specifications and the generic space. This requires also the morphisms from the generic space to the input specifications, which are inherently given with the commonalisation path (see Definition 3). Since the colimit of algebraic specification signatures does not consider consistency and priority information, we need to define a composition operation for *prioritised* CASL specification, that is based on the colimit but that also considers priorities and consistency.

Definition 5 *(Composition of PCS).* The composition $\mathfrak{c}$ of n PCS $\mathcal{S}$, a generic space $\mathfrak{g}$ and one total morphisms $m : \mathfrak{g} \mapsto \mathfrak{s}$ for each $\mathfrak{s} \in \mathcal{S}$ is defined as follows: Let $\mathfrak{s}_{colimit}$ be the colimit of the PCS without the priorities, i.e., the colimit of the algebraic CASL specifications that underly the PCS. Let the morphisms $m_1^c : \mathfrak{s}_1 \mapsto \mathfrak{s}_{colimit}, \ldots m_n^c : \mathfrak{s}_n \mapsto \mathfrak{s}_{colimit}$ be defined with respect to these specifications. Then the composition $\mathfrak{c}$ of $\mathcal{S}$ is a PCS that is constituted by the colimit $\mathfrak{s}_{colimit}$, enriched with the following priority function for elements e in the composed PCS:

$$prio(e) = \sum_{i=1}^{n} \sum_{(e^s, e) \in m_i} prio^i(e^s) \tag{16}$$

Hence, to assign the priorities for the elements e in the composition, Equation (16) simply adds up the priorities of the respective source elements e^s in the morphisms. The intuition behind this is to give those elements in a blend a higher priority that represent compressed structure. For example, if two operators in two input spaces point to the same operator in the blend, then the blend compresses the structure imposed by the operators. Similar for axioms, sorts and predicates. This supports Fauconnier and Turner [27]'s optimality principles and helps to evaluate the blends, as we explain in Sec. 3.6.

One may argue that also the product or the maximum of the priorities of the input elements could be used instead of the sum. However, during our experiments we found that just using the maximum does not reward compression enough, and the product rewards compression too much. The intuition of using the sum to add up the priorities is that a single compressed target element in a composition should be exactly as important as its individual input elements together.

For example, consider the predicate $liveIn : Person \times House$ of the *House* specification and the predicate $ride : Person \times Boat$ of the *Boat* specification, as shown in Fig. 4. Both are mapped to the same element in the composition. Note that, due to our categorical approach, the syntactic label of that element in the composition is not important. We use either the label of one of the input specifications, or we concatenate the symbols of the input specifications with an _ character in a postprocessing step, as, e.g., $liveIn\_ride : Person\_Passenger \times House\_Boat$. Since *liveIn_ride* is commonalised, it carries the information of both input spaces and it should therefore also have the joint priority of both input elements, as implemented by summing up the priorities of the corresponding operators *liveIn* and *ride* in the input spaces (see Definition 5).

3.6. Evaluating blends

The next step in the blending process is to evaluate the composition as a whole, according to several factors that reflect the rather informal optimality principles proposed by Fauconnier and Turner [27]. To this end, we take the following three evaluation metrics into account:

1. We support blends that keep as much as possible and the most important parts from their input concepts by using the priority information of elements in the input concepts. This supports those optimality principles in [27], which imply that as much as possible from the input spaces should be projected to the blend, namely at least the so-called *unpacking*, *web* and *integration* principles.
2. We support blends that maximise common relations among input concepts as a means to compress the structure of the input spaces. Relations are made common by appropriate renamings of elements in the input specification. This supports the *vital relations* principles in [27].

3. We support blends where the amount of information from the input specifications is balanced. This supports the *multi-scope* property of blends, which is described by Fauconnier and Turner [27] as "*... what we typically find in scientific, artistic, and literary discoveries and inventions.*"

The rationale behind choosing the above metrics is to have a simple but effective way to evaluate the blends from two perspectives: the informational content and the content structure. On the one hand, we want to favour those blended concepts that maintain the information from the input concepts as much as possible. This is done by a first metric defining the *amount of information* of a blend as the sum of the priorities of all its elements. On the other hand, we also want to favour those blends that maintain the structure of the input concepts as much as possible. This is done by other two metrics, blend *compression*, which essentially measures how many morphisms were applied to the different elements, and blend *imbalance*, which penalizes those blends created mainly using elements of one of the input spaces. We now define these metrics formally.

The *amount of information* in a PCS is given as:

$$infoValue(\mathfrak{s}) = \sum_{e \in \mathfrak{s}} prio(e) \tag{17}$$

Equation (17) defines the amount of information in a PCS as the sum of the priorities of all of its elements. A measure for the *compression of structure* in a composition $\mathfrak{c}$ with n morphisms $M = \{m_1 : \mathfrak{s}_1 \mapsto \mathfrak{c}, \ldots, m_n : \mathfrak{s}_n \mapsto \mathfrak{c}\}$ is defined as:

$$compression(\mathfrak{c}) = \sum_{e \in \mathfrak{c}} eleComp(e) \qquad \text{where} \tag{18}$$

$$eleComp(e) = prio(e) \cdot \frac{|\{m \in M | \exists e^s \,.\, (e^s, e) \in m\}|}{n} \tag{19}$$

The compression value of the composition $\mathfrak{c}$ is the sum of the compression values of its individual elements (denoted by *eleComp*). The compression value of an individual element is the priority of that element, multiplied by the number of morphisms to it, and normalised by the total number of input morphisms.

We also account for the *balance of information* from both input specifications. That is, we consider blends to be better where the amount of information from the input specifications is similar. Towards this we define an imbalance penalty as the half of the difference of the amount of information from the input specification as follows:

$$imbalance(\mathfrak{c}) = \frac{abs(infoValue(\mathfrak{s}_1) - infoValue(\mathfrak{s}_2))}{2} \tag{20}$$

Taking only the half of the difference as imbalance penalty turned out to be more useful than taking the full difference, because this still encourages blends which have more information in total, even if they are imbalanced. The final evaluation of a blend is done by summing up the three evaluation metrics and by considering logical consistency as follows:

$$value(\mathfrak{c}) = \begin{cases} infoValue(\mathfrak{c}) + compression(\mathfrak{c}) - imbalance(\mathfrak{c}) & \text{if } \mathfrak{c} \text{ is consistent} \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

Note that the imbalance penalty can never be bigger than the information value, so that the blend value is always positive. An interesting extension of computing the value is to parameterise the different metrics. However, we have so far not experimented with this.

The transition path pruning described in Sec. 3.4 supports the information value and compression value metrics because by removing low-priority elements before high-priority elements (information value), and by commonalising high-priority elements before low-priority elements (compression value), low-value blends are not generated. We have not yet implemented a feasible solution to also consider the imbalance penalty in the pruning mechanism.

The issue of how blends are to be evaluated is an important one in the theory of conceptual blending, and in addition to these evaluation metrics, which are generic for any kind of blends, more domain specific criteria can (and eventually should) be added to the blending framework. For this reason we have also explored how different audiences and their respective values can be brought into the blend evaluation process [12] and we have proposed evaluation techniques that draw from argumentation theory [9] and coherence theory [75]. Beyond the evaluation of blends, the Computational Creativity community has extensively studied how to evaluate the creativity of computational systems in general, and several models have been proposed [72,6,45]; we have applied some of these models to the evaluation of our framework [62], as we describe in the following Section 4.

4. Proof of concept

To demonstrate our system we first present the use case of the house-boat, which primarily serves illustrative purposes. Then we consider several examples from two real-world domains where creativity is important, namely mathematics and music. The examples for harmony invention (Sec. 4.2), lemma invention (Sec. 4.3) and cross-domain blending (Sec. 4.4) are also illustrated in the supplementary video.

```
spec HOUSE =
    sorts  Medium
           House < Object
           Person < Object
    ops    house : House
           resident : Person
           land : Medium
    preds  liveIn : Person × House
           on : Object × Medium
    . liveIn(resident,house)
    . on(house,land)
end
```

```
spec BOAT =
    sorts  Medium
           Boat < Object
           Person < Object
    ops    boat : Boat
           passenger : Person
           water : Medium
    preds  ride : Person × Boat
           on : Object × Medium
    . ride(passenger,boat)
    . on(boat,water)
end
```

Fig. 4. The house and boat specifications in CASL.

```
spec GENERICSPACE =
    sorts  Medium
           House < Object
           Person < Object
    ops    house : House
           resident : Person
    preds  liveIn : Person × House
           on : Object × Medium
    . liveIn(resident,house)
end
```

HOUSE	⇐	GENERICSPACE	⇒	BOAT
House	⇐	sort of the containing object	⇒	*Boat*
house	⇐	containing object	⇒	*boat*
resident	⇐	contained object	⇒	*passenger*
liveIn	⇐	relation between contained and the containing object	⇒	*ride*

Fig. 5. The generic space for the *house-boat* blend.

```
spec HOUSE_BOAT =
    sorts  Medium
           Boat < Object
           Person < Object
    ops    boat : Boat
           resident : Person
           land,water : Medium
    preds  liveIn : Person × Boat
           on : Object × Medium
    . liveIn(resident,house)
    . on(boat,water)
end
```

```
spec BOAT_HOUSE =
    sorts  Medium
           House < Object
           Boat < Object
    ops    house : House
           boat : Boat
           land,water : Medium
    preds  liveIn : Boat × House
           on : Object × Medium
    . liveIn(boat,house)
    . on(house,land)
    . on(boat,water)
end
```

Fig. 6. The *house-boat* and *boat-house* blends.

4.1. The house-boat blend

A classical concept blending example is the blend between the concepts house and boat [34,27]. In Fig. 4 we depict the axiomatisation of these concepts which is similar to how they are proposed in [35]. Priorities are equal for all elements in the specifications, so we omit them here for brevity. The CASL theories for HOUSE and BOAT introduce the sorts, operators and predicates that form the mental spaces *house* and *boat* by focusing on particular properties of these concepts. The precise formalisation is not critical at this point, different ones exist [35,63], but all provide similar distinctions. In principle, the HOUSE and BOAT specifications could be directly blended (they do not generate any inconsistency), but the blended specification is not considered a good blend since it does not maximise the common relations among the two input specifications. Instead, more interesting blends are created depending on how the HOUSE and BOAT are transitioned. A *house-boat* blend is created when the medium on which a house is situated (land) becomes the medium on which boat is situated (water), and the resident of the house becomes the passenger of the boat. In our system, this is achieved using the generic space depicted in Fig. 5. This generic space is obtained by removing the $on(house, land)$ axiom from the HOUSE specification, the *water* operator and the $on(boat, water)$ axiom from the Boat specification, and by renaming the operators *ride* to *liveIn*, *passenger* to *resident*, *boat* to *house* and the sort *Boat* to *House* in the BOAT specification. The morphisms between the generic space and the input spaces are shown in the right part of Fig. 5.

The colimit that results from the generic space and the above morphisms is a theory in which the boat is a house and the passenger is a resident of the *house-boat* (see Fig. 6, left). Our system is also able to generate other classic blends. For instance, a *boat-house* is created which turns the boat into a person by renaming *resident* to *boat* and *Person* to *Boat*, along with some other minor generalisations (see Fig. 6, right). Here, the colimit expresses a blend in which the boat lives in the house.

spec PERFECTCAD = CHORDPROG **then**

op *c1Perf* : *Chord*	p:10
op *c2Perf* : *Chord*	p:10
. *succ*(*c1Perf*, *c2Perf*)	p:5
. *absNote*(*c1Perf*, 7)	p:2
. *absNote*(*c1Perf*, *11*)	p:3
. *absNote*(*c1Perf*, 2)	p:1
. *absNote*(*c1Perf*, 5)	p:2
. *relNote*(*c1Perf*, 0)	p:3
. *relNote*(*c1Perf*, 4)	p:3
. *relNote*(*c1Perf*, 7)	p:2
. *relNote*(*c1Perf*, *10*)	p:3
. *root*(*c2Perf*) = 0	p:1
...	

end

spec PHRYGCAD = CHORDPROG **then**

op *c1Phryg* : *Chord*	p:10
op *c2Phryg* : *Chord*	p:10
. *succ*(*c1Phryg*, *c2Phryg*)	p:5
. *absNote*(*c1Phryg*, *10*)	p:2
. *absNote*(*c1Phryg*, 1)	p:1
. *absNote*(*c1Phryg*, 5)	p:2
. *relNote*(*c1Phryg*, 0)	p:3
. *relNote*(*c1Phryg*, 3)	p:3
. *relNote*(*c1Phryg*, 7)	p:2
. *root*(*c2Phryg*) = 0	p:1
...	

end

Fig. 7. The relevant excerpts of the perfect cadence and the Phrygian cadence specifications. Only the first chord is fully specified here for brevity. The second chord is a plain *Cmaj* in both cadences.

4.2. Harmony invention for music composition

In the music domain we invent novel chord progressions by blending existing ones, and demonstrate how our approach extends the musicological framework proposed in our previous work [20]. Specifically, we show how we blend chords to invent novel *cadences* – short chord progressions that can be understood as 'punctuation' within a music piece. While the system depicted in [20] is limited to blending single chords, we are now able to blend whole chord progressions, as for example the *perfect cadence* and *Phrygian cadence*. We present the relevant parts, i.e., the first chord of each cadence, as algebraic specifications in Fig. 7.

Both specifications are built on a background theory CHORDPROG about chord progressions, which defines the predicate *succ* to denote the successor relation among chords, the predicate *absNote* to determine the absolute notes of a chord, and the predicate *relNote* to determine the notes of a chord relative to the root note (indicated by the *root* operator). CHORDPROG also contains an axiom $hasRelNote(c, n) \Leftrightarrow hasAbsNote(c, plus(n, root(c)))$ that defines the relation between absolute and relative notes in the background theory, and that states when a chord is dissonant. Dissonance is captured via axioms that forbid certain relative note combinations. For example, they express that a chord cannot have a major third (relative note 4) and a minor third (relative note 3) at the same time, i.e., $\forall c : Chord \ . \ \neg(relNote(c, 3) \wedge relNote(c, 4))$. Given a *C* major key, the Phrygian cadence involves a *B*♭*min* chord followed by a *C* chord, and the perfect cadence is a *G*7 chord followed by a *C*. The priorities of the axioms that assign notes to the chords are musicologically justified as described in [20], i.e., the relatives are given a higher priority, and those absolute notes which are salient within the key are also given a higher priority. In addition to these axioms, our system also considers the priority of individual chords which are represented as operators *c1Perf* (*G*7 chord), *c2Perf* (*C* chord), *c1Phryg* (*B*♭*min* chord) and *c2Phryg* (*C* chord).

Blending the two cadences produces a *Tritone substitution* cadence as the result with the highest value. The tritone substitution was invented in jazz music decades after the Phrygian and perfect cadence. It takes the *D*♭ note from the first chord of the Phrygian cadence, specified by $absNote(c1Phryg, 1)$, as root of the first chord of the novel Tritone cadence. The blending also adds the relative seventh of the *G*7 chord of the perfect cadence ($relNote(c1Perf, 10)$), as well as the major third ($relNote(c1Perf, 4)$) and the fifth ($relNote(c1Perf, 7)$) which are present in both chords. The result is a *D*♭7 chord as first chord of the Tritone substitution. The system also allows to blend the second chord of one cadence with the first chord of another, so that a novel chord progression of three notes is produced. Note that, due to the renaming operators, this can be done on the level of cadences and chord progressions as a whole, and not only on the level of single chords.

4.3. Lemma invention for theorem proving

In the mathematics domain, we present a general approach for using blending to exploit existing creative lemmas in a well understood theory, to automatically generate creative lemmas in another less understood theory. For illustration, consider the prioritised theories of natural numbers and lists shown in Fig. 8. An illustration and demo of this blending application is provided in the supplementary video (after 7:00).

Important elements of these specifications are the constructor operators *s*, i.e., the successor function, and *cons*, i.e., the constructor for lists. Due to their importance, we give them a high priority. Of particular interest here are also the theorems **(NT)** and **(LT)**, which are also given a high priority because they provide important insights about the relation between the tail-recursive functions *qrev* and *qsum*, and their primitively recursive counterparts *rev* and *sum*. Proving such theorems by induction is very hard due to the absence of a universally quantified variable in the second argument of the tail-recursive version [42]. An expert's solution here is to use a lemma that generalises the theorem. An example of such a generalisation is the eureka lemma **(NL)** in the naturals, which we assume to be known in this scenario. Discovering such lemmas is in general a very challenging and well-known problem – see [56,43] for example. Our goal is to use blending to discover an analogous lemma which facilitates the inductive proof of **(LT)** in LIST.

spec NAT =

	sort *Nat*	p:3
	ops *zero* : *Nat*;	p:2
	s : *Nat* → *Nat*	p:3
	sum : *Nat* → *Nat*	p:2
	qsum : *Nat* × *Nat* → *Nat*	p:2
	plus : *Nat* × *Nat* → *Nat*	p:1
	∀ *x*, *y* : *Nat*	
(0)	. *sum*(*zero*) = *zero*	p:2
(1)	. *sum*(*s*(*x*)) = *plus*(*s*(*x*), *sum*(*x*))	p:2
(2)	. *qsum*(*s*(*x*), *y*) = *qsum*(*x*, *plus*(*s*(*x*), *y*))	p:2
(3)	. *qsum*(*zero*, *x*) = *x*	p:2
(4)	. *plus*(*zero*, *x*) = *x*	p:1
(5)	. *plus*(*s*(*x*), *y*) = *s*(*plus*(*x*, *y*))	p:1
(NT)	. *sum*(*x*) = *qsum*(*x*, *zero*)	p:3
(NL)	. *plus*(*sum*(*x*), *y*) = *qsum*(*x*, *y*)	p:3

end

spec LIST =

	sorts *El*	p:3
	L	p:3
	ops *nil* : *L*;	p:2
	cons : *El* × *L* → *L*;	p:3
	app : *L* × *L* → *L*;	p:2
	rev : *L* → *L*;	p:2
	qrev : *L* × *L* → *L*	p:2
	∀ *x*, *y* : *L*; *h* : *El*	
(6)	. *rev*(*nil*) = *nil*	p:2
(7)	. *rev*(*cons*(*h*, *x*)) = *app*(*rev*(*x*), *cons*(*h*, *nil*))	p:2
(8)	. *qrev*(*nil*, *x*) = *x*	p:2
(9)	. *qrev*(*cons*(*h*, *x*), *y*) = *qrev*(*x*, *cons*(*h*, *y*))	p:2
(10)	. *app*(*nil*, *x*) = *x*	p:1
(11)	. *app*(*cons*(*h*, *x*), *y*) = *cons*(*h*, *app*(*x*, *y*))	p:1
(LT)	. *rev*(*x*) = *qrev*(*x*, *nil*)	p:3

end

Fig. 8. Specifications of the natural numbers and lists.

The first step of blending is to find a generic space. However, this is problematic because the constructor $s(n)$ in the naturals is unary, whereas the constructor $cons(h, l)$ in lists is binary. In order to resolve this problem we take inspiration from a classical set theoretic construction of the naturals as the cardinality of a set (see [1] for example). That is, we describe the theory of the naturals as a theory of lists of exactly the same element.[6] As a result, we generalise the theory of naturals by adding an extra argument to the successor function, i.e., $s(n)$ becomes $s(c, n)$ where c is some canonical element of a canonical sort C. The definition of plus, for example, then simply becomes $plus(s(c, n), m) = s(c, plus(n, m))$. This way we obtain a correctly typed generic space by interpreting the renaming and removal operators from the ASP solver, which allows us to associate the constructors of NAT and LIST with the following generalisation paths[7]:

$$P_{\text{NAT}} = \{$$
$$exec(rename(Nat, L, \text{LIST}), \text{NAT}, 0), exec(rename(zero, nil, \text{LIST}), \text{NAT}, 1),$$
$$exec(rename(C, El, \text{LIST}), \text{NAT}, 2), exec(rename(s, cons, \text{LIST}), \text{NAT}, 3),$$
$$exec(rename(sum, rev, \text{LIST}), \text{NAT}, 4), exec(rename(qsum, qrev, \text{LIST}), \text{NAT}, 5),$$
$$exec(rename(plus, app, \text{LIST}), \text{NAT}, 6),$$
$$exec(rm(4), \text{NAT}, 7), exec(rm(5), \text{NAT}, 8), exec(rm(1), \text{NAT}, 9),$$
$$exec(rm(2), \text{NAT}, 10), exec(rm(c), \text{NAT}, 11), exec(rm(NL), \text{NAT}, 12)\}$$

$$P_{\text{LIST}} = \{$$
$$exec(rm(10), \text{LIST}, 0), exec(rm(11), \text{LIST}, 1), exec(rm(9), \text{LIST}, 2), exec(rm(7), \text{LIST}, 3)\}$$

After applying the respective renamings and removals, a generic space is reached, using the symbols from the *List* theory. Note, that even though the symbols of the lists theory are used in the generic space, their meaning is now much more general because they map to both, the *List* and the *Nat* theory, and represent now analogies between both theories as depicted in Table 1. That is, the generic space is a general theory with sorts for the constructed data types and their elements, a binary constructor, a terminal element, a primitively recursive function, and a tail-recursive function which is defined in terms of the auxiliary function. After finding the generic space, our framework iterates over different combinations of transitioned input specifications and computes the colimit. It then checks the colimit's consistency and computes the blend value. In this example, the highest composition value for a consistent colimit is 90, where the specifications that result from the 4th transition of LIST and the 8th transition of NAT are used as input. The result is a theory of lists with the newly invented lemma $app(rev(x), y) = qrev(x, y)$ which can be used successfully as a generalisation lemma to prove **(LT)**.

4.4. Cross-domain blending

So far we only considered input concepts from the same domains, i.e., we blend chord progressions with chord progressions and mathematical theories with mathematical theories. However, one of the biggest challenges in creative systems

[6] The future work part in Sec. 6 elaborates on this way to recursively construct data structures.

[7] In the expressions $rm(4)$, $rm(5)$, $rm(1)$, etc., natural numbers correspond to the labels of the axioms in the input specifications in Fig. 8.

Table 1
The generic space and its mappings to the theories List and Nat.

Nat	Generic Space	List
Nat	Constructed datatype	*L*
C	Constructed datatype element	*El*
zero	terminal element	*nil*
s	constructor	*cons*
sum	recursive function	*rev*
qsum	tail-recursive function	*qrev*
plus	auxiliary function	*app*

spec CyclicGroup =
sorts *Nat*
Ele
Group
ops *zero* : *Nat*
s : *Nat* → *Nat*
ele : *Ele*
group : *Group*
binop : *Ele* × *Ele* → *Ele*
binops : *Ele* × *Nat* → *Ele*
ident : *Ele*
inv : *Ele* → *Ele*
pred *hasElement* : *Group* × *Ele*
(1) . *hasElement*(*group*,*ele*)
∀ *g*: *Ele*, *n*: *Nat*
(2) . *binops*(*g*, *zero*) = *ident*
(3) . *binops*(*g*,*s*(*n*)) = *binop*(*g*,*binops*(*g*,*n*))
(4) . *hasElement*(*group*,*binops*(*ele*,*n*))
∀ *g*: *Ele*. ∃ *n*: *Nat*
(5) . *binops*(*ele*, *n*) = *g*
⋮
end

spec GeneratorChord =
sorts *Chord*
Note
ops *chord* : *Chord*
gennote : *Note*
root : *Chord* → *Note*
plus : *Note* × *Note* → *Note*
nextNote : *Note* × *NoteNote*
preds *hasRelNote* : *Chord* × *Note*
hasAbsNote : *Chord* × *Note*
(6) . *hasRelNote*(*chrod*,*gennote*)
∀ *c* : *Chord*. ∃ *n* : *Note*
(7) . *root*(*c*) = *n* ∧ *hasAbsNote*(*c*,*n*)
∀ *c* : *Chord*, *n* : *Note*
(8) . *hasRelNote*(*c*,*n*) ⇔*hasAbsNote*(*c*,*plus*(*root*(c),*n*))
.
⋮
end

Fig. 9. Specifications of the cyclic groups and generator chord theories.

is being able to deal also with concepts coming from different domains. Generally, in cross-domain blending, one aims to transfer the knowledge of the two domains, to create something new, possibly by some adaptation. Blending multi-domain concepts is challenging, since different domains contain different knowledge and different symbols and representations that need to be cross-mapped in order to find new meaningful blended concepts. Our framework is general enough to blend concepts from different domains, and in this example, we show how we can blend chords with mathematical theories. We refer to our supplementary video for a detailed illustration (after 8:30) and for audio samples (after 18:00).

In what follows we construct theories via blending which specify a *logical hull*. This means that the definitions in the theories need to be evaluated in order to determine the contents of the theory. In particular we make use of recursive functions. Evaluating these on all inputs gives facts about the theories about which we can reason.

4.4.1. Group theory and chord construction

Consider the case of blending the specification of cyclic groups of elements with a specification for chords defined via intervals between their notes, as shown in Fig. 9.

The general theory of a group in mathematics is the combination of a set of elements $\mathbb{G}$ and a binary operation $\sim$. A group $(\mathbb{G}, \sim)$ has the following axioms:

$$\forall x, y \in \mathbb{G}.\, x \sim y \in \mathbb{G} \tag{22}$$

$$\exists e \in \mathbb{G}.\, \forall x \in \mathbb{G}.\, x \sim e = x \tag{23}$$

$$\forall x \in \mathbb{G}.\, \exists x^{-1} \in \mathbb{G}.\, x \sim x^{-1} = e \tag{24}$$

$$\forall x, y, z \in \mathbb{G}.\, x \sim (y \sim z) = (x \sim y) \sim z \tag{25}$$

Extending the notion of group to a cyclic group, involves restricting the set of elements to be finite, and to introduce the notion of a *generator element*. The size of the set is then given by an *order o*. Elements of the group can be constructed with successive applications of the binary operation on the generator element and we introduce the shorthand $\lambda x, y.x^y$ to denote successive application; for example $g^3 = g \sim (g \sim g)$. We then add the following axioms about a cyclic group,

$$\exists g \in \mathbb{G}.\forall x \in \mathbb{G}.\exists n \in \mathbb{N}.(0 \leq n < o) \wedge x = g^n \tag{26}$$

spec CyclicGroupChordBlend =
sorts *Note*
Chord
Nat
ops $plus : Note \times Note \mapsto Note$
$binops : Note \times Nat \mapsto Note$
$chord : Chord$
$gennote : Note$
$ident : Note$
$root : Chord \mapsto Note$
$s : Nat \mapsto Nat$
$zero : Nat$
preds $hasAbsNote : Chord \times Note$
$hasAbsNote : Chord \times Note$
$. hasRelNote(chord,gennote)$
$\forall\ gc : Chord, en : Note, n : Nat$
$. hasRelNote(chord,binops(gennote,n))$
$. binops(en,s(n)) = plus(en, binops(en,n))$
$. hasRelNote(gc,en) \Leftrightarrow hasAbsNote(gc,plus(root(gc),en))$
$. binops(en,zero) = ident$
$\forall\ gc : Chord, \exists\ en : Note$
$. root(gc) = en \wedge hasAbsNote(gc,en)$
$\forall\ en : Note, \exists\ n : Nat$
$. binops(gennote,n) = en$

Fig. 10. The blend of cyclic groups and generator chords.

Table 2
The generic space and its mappings to the theories CyclicGroup and GeneratorChord.

CyclicGroup	Generic Space	GeneratorChord
Group	Constructed datatype	*Chord*
Ele	Constructed datatype element	*Note*
binop	Binary operation between elements	*plus*
group	Instance of constructed datatype	*chord*
ele	Generator element	*gennote*
hasElement	Predicate denoting membership	*hasRelNote*

This axiom means that the inverse (x^{-1}) of an element x can be constructed also from the generator element, as can the identity element e.

In the CASL theories shown in Fig. 9, we model these axioms in the specification CyclicGroup and can compute the elements of the cyclic group using the function *binop* for $\sim$ and introduce the recursive function *binops* for the successive application operator ($\lambda x, y.\ x^y$). We also introduce a predicate *hasElement* which defines the elements of the cyclic group.

In the case of the generator chord specification shown also in Fig. 9 we extend the notion of chord previously defined in Sec. 4.2 to include a "generator note" which is included as a relative note in the chord. Thus a generator chord with generator note 3 will always contains a minor third. We also introduce the notion of adding an interval with the operator *plus*.

In our case, the generic space was computed as depicted in Table 2. During blending we used a priority of 1 for all elements. This leads to the blend specification show in Fig. 10.

Conceptually this blend corresponds to a specification of a chord, whose notes are generated using the axioms of a cyclic group of order 12 – the number of notes in a scale. The specification alone does not define the notes in the chord – it defines a way of calculating the specific notes given a generator note or interval. Calculating the logical hull of the specification given a particular instantiation of *gennote* gives us the notes in the chord. Specifically evaluating the function *binops* over the natural numbers generates the notes which exist relatively in the logical hull by virtue of the axiom $hasRelNote(chord, binops(gennote, n))$.

A theorem of cyclic group theory is that if the prime factors of the generator do not divide the cardinality of the set, then it will generate every member of the set. In this case, the numbers which do not divide 12 – the number of notes in a scale – is the set $\{1, 5, 7, 11\}$. Let us first discuss the significance of the chord specifications generated using generated notes which do divide 12 and hence form *subgroups*, These are itemised here by the axiom which is added to the specification to generate chord by calculating the logical hull.

$. gennote = 2$
: This corresponds to adding a relative major second to the specification. The logical hull of adding this axiom to the CyclicGroupChordBlend gives us the relative notes $\{0, 2, 4, 6, 8, 10\}$ which is a chord constructed from the *whole tone* scale. This can sometimes be used as an *altered* chord. For example in the key of C this gives us $\{C, D, E, F\#, A\flat, B\flat\}$. Generalising this chord by removing some elements allows us to create various potential chords including a C7♭13, a C *augmented* or even a C9.

. *gennote* = *3*
: This corresponds to adding a relative minor third to the specification. The logical hull of adding this axiom to the CyclicGroupChordBlend gives us the relative notes $\{0, 3, 6, 9\}$ which specifies a diminished chord. For example, in the key of C, this is $\{C, Eb, F\#, A\}$, which is a *diminished* chord. This is interesting as it forms the top 4 notes of an A♭7♭9 chord, which is a modified dominant chord used very often in jazz to resolve to the key of D♭. Because the chord is naturally cyclic (on account of the blend with cyclic group theory), it can equally be used to transition to the keys of E, G or B♭ meaning that its function is very versatile. Bach used this versatility of the diminished chord to perform seemless but unexpected key changes in his pieces very often.

. *gennote* = *4*
: This corresponds to adding a relative major third to the specification. The logical hull of adding this axiom to the CyclicGroupChordBlend gives us the relative notes $\{0, 4, 8\}$ which specify an *augmented* chord. In the key of C this is $\{C, E, A\flat\}$. This is often used as a transition chord.

. *gennote* = *6*
: This corresponds to adding a diminished fifth to the specification. This only produces the two notes $\{0, 6\}$ and as such is not particularly interesting except to provide a means for dissonance within a piece.

. *gennote* = *8*
: This is equivalent to the situation where we add . *gennote* = *4*

. *gennote* = *9*
: This is equivalent to the situation where we add . *gennote* = *3*

. *gennote* = *10*
: This is equivalent to the situation where we add . *gennote* = *2*

Sequences generated from the remaining numbers involve every note in the scale. This would simply generate chords consisting of all 12 notes which does not seem very interesting. However, if we slightly extend the theory and consider the order in which the notes are produced, we notice the reproduction of a very creative technique in modern jazz, namely *voicing*. With this, the specifications of the chords become indeed quite feasible. As an example, consider the chord specified by the note sequence $\{0, 3, 7, 10, 2\}$ – this is a minor ninth. As we have no notion of voicing this represents playing the notes $\{C, D, E\flat, G, B\flat\}$. In reality one would often place the D above the $B\flat$ – an octave and a second above the root. If we now consider the construction of these sequences as being monotonically increasing, then we can introduce a notion of voicing by allowing the notes always to be higher in pitch than the previous in the list. Let us consider the construction of the sequence created by adding these numbers.

. *gennote* = *1*
: This is conceptually uninteresting as the chord is formed sequentially by playing the chromatic scale, and hence almost immediately too dissonant to have function.

. *gennote* = *5*
: This corresponds to adding a perfect fourth, thus generating the sequence of relative notes $\{0, 5, 10, 3, 8, 2, 6, 11, 4, 9, 2, 7\}$. This corresponds to the way in which *tetrachords* are formed. For example in C one could play $\{C, F, B\flat, E\flat\}$ – as specified by the initial 4 notes in this sequence. This is the sort of chord that McCoy Tyner uses often to create a modern accompanying sound.

. *gennote* = *7*
: This corresponds to adding a perfect fourth, thus generating the sequence of relative notes $\{0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10, 5\}$. This corresponds to playing chords constructed by incrementing intervals of 5ths. For example in C one could play $\{C, G, D, A\}$ – as specified by the initial 4 notes. This constitutes a *suspended* type of chord.

. *gennote* = *11*
: This corresponds to adding a major 7th incrementally, thus generating the sequence of notes $\{0, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1\}$. Enforcing the constraint that the constructed chord must consist of increasing intervals makes it unplayable and very dissonant. We do not know of any piece where this type of chord is employed.

4.4.2. Progression theory and constructing Coltrane changes

In a similar way to chord construction, one can construct chord progressions using blends which involve cyclical groups. The only difference is that each number is associated either with a chord or the tonal centre of a cadence. Fig. 11 shows how the cyclic group can be defined using the specification of list given in Fig. 8. Polymorphism is modelled here using the CASL **with** operator where the elements of the list in the List spec can be defined as a particular sort.

spec ListNote =
List **with** $El \mapsto Note$, $L \mapsto LN$
end

spec ListChord =
List **with** $El \mapsto Chord$, $L \mapsto LC$
end

In order to model cyclic groups in a more computational way, we introduce a function *calc_subseq* which calculates a list of elements in a group of a certain order. The binary operation associated with the group is constrained by introducing a partial *minus* operation into the natural numbers. The idea is then that a list of elements of a cyclic group can be associated

Table 3
The generic space and its mappings to the theories Cyc12 and Progression.

Cyc12	Generic Space	Progression
GroupElement	Constructed Datatype	*Chord*
subseq	Input List	*lin*

```
SPEC Progression =
      PerfCadence
then ListNote
then ListChord
then ops   c1, c2 : Chord
      op   p : Perf
      op   lin : LN
      op   lout : LC
      op   calc_prog : LN → LC
      . calc_prog(nil) = nil
      ∀ h : Note; t : LN
      . calc_prog(cons(h, t))
        = cons(pre(p, h), cons(post(p, h), calc_prog(t)))
      . lout = calc_prog(lin)
end

spec Cyc12 =
      CyclicGroup
then op    gen : G
      . gen = 4 as G
      . order = 12 as G
      . subseq = calc_subseq(gen)
end
```

```
spec CyclicGroup =
      NatSuc
then sort  G < Nat
then List with El ↦ G, L ↦ ListG
then op    subseq : ListG
      op   binop : G × G → G
      op   ident : G
      op   inv : G → G
      pred gen : G
      op   order : Nat
      op   calc_subseq : G → ListG
      op   calc_subseq_r : G × G → ListG
      ∀ x, y, z : G
      . binop(x, binop(y, z)) = binop(binop(x, y), z)
      . binop(x, ident) = x
      . binop(x, inv(x)) = ident
      . order < plus(x, y) ⇒ binop(x, y) = minus(plus(x, y), order)
      . x = y ⇒ calc_subseq_r(x, y) = nil
      . ¬ x = y
        ⇒ calc_subseq_r(x, y)
            = cons(binop(x, y), calc_subseq_r(x, binop(x, y)))
      . calc_subseq(x) = calc_subseq_r(x, x)
end
```

Fig. 11. Specifications of the progression and cyclic groups as lists theories.

with tonal centres to form a progression. The tonal centre can be simply a chord, or as is shown in Fig. 11, a perfect cadence. The progression has a function *calc_prog* which shows how to compute a list of chords defining the progression. The operator *lin* defines a list of input tonal centres. This way, the list of tonal centres defined by computing different cyclic groups of order 12 can generate interesting progressions. For example, Fig. 11 shows a refined specification Cyc12, which defines an order of 12, and a generator element of 4. The blend then produces a theory which has successive perfect cadences at tonal centres determined by the list of elements of the group. The Generic space and morphisms is shown in Table 3. Depending on the choice of value for *gen*, and the choice of cadence, different progressions are generated. In the case of the Cyc12 specification we generate a progression known in jazz as *Coltrane Changes*. With a generator of 4, *subseq* is calculated as 0,8,4. If these are interpreted as the tonal centres of progressive perfect cadences as is specified by the blend, then in the key of B major this becomes | F#7 B | D7 G | B♭7 E♭ | which is (excepting the initial dominant F#7) exactly how the iconic John Coltrane tune *Giant Steps* is started.

Other examples of progressions depend on the choice of cadence and the choice of *gen* and include for example:

. gen = 3; Cadence = Perfect
An example of such a progression (with F as root) is | C7 F | A7 D | F#7 B | Eb7 Ab | – a progression exploited in the piece *maths* by jazz group Algorhythmical.

. gen = 2; Cadence = Phrygian
An example of such a progression (with C as root) is | Bb- C | A♭- Bb | F#- A*flat* | E- F# | D- E |. As far as the authors know this has not been used in any existing composition.

. gen = 5; Cadence = Tritone
An example of such a progression (with C as root) is | D♭7 C| F#7 F | B7 B♭ | E7 E♭ | A7 A♭ | D7 D♭ | G7 F# | C7 B | F7 E | B♭7 A | E♭7 D | A♭7 G |. As far as the authors know this has not been used in any existing composition.

The above examples show that using this approach both recreates existing creative leaps in the theory and composition of jazz progressions, and also creates some previously unused progressions, which cannot be discovered in existing compositions. The technique of employing conceptual blending allows elements of music theory to exist in the blend, together with elements from the mathematical theory of cyclic groups. Had the system simply picked notes from integer sequence generation, the constraints from music theory that exist within the ListChord theory would not have been expressed, and notions of cadences (as discussed in Sec 4.2) and their associated constraints would have rendered an entirely unconstrained chord and progression theory. The restriction of the resulting blends to musically surprising, yet viable progressions can be attributed directly to the approach taken in this paper.

5. Related work

Several approaches of formal and computational models for conceptual blending have been proposed [11,13,12,20,21,49, 35,83,63,64,33,36]. Many of these models are inspired by the work of Fauconnier and Turner [27], but there are also other approaches emanating from analogical reasoning [76] and neuroscience [79].

Amalgam-based conceptual blending have been developed to blend $\mathcal{EL}^{++}$ concepts in [10,11,13]. In these works, the generalisation of an $\mathcal{EL}^{++}$ concept is achieved by means of a generalisation refinement operator. This line of work has also been extended to the general debugging problem for ontologies combining ideas from social choice theory and generalisation operators [70]. The refinement operator is implemented in ASP as a step-wise transition process – similar to the one presented in this paper – that looks for a generic space between two (or more) concepts. The operator generalises a concept by taking the TBox knowledge (terminology) into account. Good blends are selected by re-interpreting some optimality principles. Blending ontologies rather than concepts has been explored in the ontological blending framework of Kutz et al. [49], where blends are computed as *colimits* of blending diagrams specified according to the Distributed Ontology Language (DOL) [59], a recent OMG international ontology interoperability standard. In that framework, the blending process is not characterised in terms of amalgams, nor are input concepts generalised syntactically. Rather, the generic space is assumed to be given and mapped to the input ontologies via theory interpretations.

Confalonieri et al. [12] propose a general process model for concept invention that extends the conceptual blending theory of Fauconnier and Turner [27]. Apart from the blending mechanism modelling the creation of new concepts, the authors focus on two extra dimensions that are typically not addressed in computational approaches of concept blending. On the one hand, they describe how a Rich Background supports the discovery of input concepts to be blended. On the other hand, they show how arguments, promoting or demoting the values of an audience, to which the invention is headed, can be used to evaluate candidate blends. Besides, in Schorlemmer et al. [75], the evaluation of new blended concepts is achieved by taking the computational theory of conceptual coherence by Thagard [78] into account. In this setting, newly invented concepts are evaluated with respect to a background ontology of conceptual knowledge so as to decide which of them are to be accepted into a system of familiar concepts. A similar approach has been taken more recently in [14] for determining the joint coherence of $\mathcal{ALC}$ concepts w.r.t. a background ontology.

The Alloy algorithm [35] for conceptual blending incorporates many ideas of the algebraic semiotics approach by Goguen [34] and the conceptual blending theory by Fauconnier and Turner [27]. Alloy has been integrated in the Griot system for automated narrative generation [35,39,38]. Apart from the primary *conceptual* blending approach realised with Alloy, Griot also uses a secondary *structural* blending mechanism that blends the dynamic elements of natural language narratives to generate poetry. The input spaces of the Alloy algorithm are theories defined in the algebraic specification language BOBJ (see e.g., [53]). This allows one to represent sorts, operators, constants and axioms. The blending algorithm generates two binary trees that are based on the input graph. The two trees represent (i) the space of possible mappings of relations and (ii) the space of possible mappings of constants respectively. The trees are then combined in the sense that the leaves of the constants-tree are applied to the relations-tree. The resulting combined tree has leaves that represent all possible sort-preserving mappings of relations and constants, i.e., all possible blends. During the tree generation, certain optimality principles are applied to prune the space. The authors do not explicitly account for a cross-space mapping. However, the edges between relations and constants from different theories can be understood as a weak form of a cross-space mapping. Alloy uses these edges (and hence the potential cross-space mapping) to determine a generic space.

Sapper was originally developed by Veale and Keane [84] as a computational model of metaphor and analogy. It computes a mapping between two separate domains – understood as graphs of concepts – that respects the relational structure between the concepts in each domain. Strictly speaking, Sapper does not produce blends, but structural mappings between input concepts. It can, however, be seen as a computational model for conceptual blending, because the mappings between the pairs of concepts that constitute its output can be manipulated as atomic units, as blended concepts [83]. Furthermore, Sapper does not work with *a priori* given input spaces. It is the structure mapping algorithm itself which, given two domains to be mapped, determines the set of concepts and relations between these concepts that constitute the spaces that are blended. It does so by searching within its semantic memory for the largest substructures (bounded by a previously fixed size) at the root concepts of these domains that are isomorphic as with respect to their relational structure, and whose concepts are metaphorically related. Semantic memory is represented as a semantic network [71], a graph whose nodes represent concepts and whose edges represent binary relations between concepts. The actual correspondence or blend between concepts is computed by applying a spreading activation algorithm according to which Sapper locates pairs of paths that are structurally isomorphic (of equal length and constituted by the same sequence of semantic relations) and that terminate at concepts that Sapper considers semantically bridgeable. In Sapper, most of the optimality principles are captured and serve to rank and filter the correspondences that comprise the mappings computed by the algorithm. Veale [82] also proposes a web-based approach to generate more loosely defined blends that he calls *conceptual mash-ups*. Inconsistencies and evaluation metrics are not addressed in this work; instead, the author focuses on the Internet as a provider for the massive amount of background information that is required to generate meaningful blends.

The research in [65–67] led to the development of Divago [63,68,64], probably the first complete implementation of conceptual blending. Pereira draws the terminology and definitions for his formal and computational model from Wiggin's formalisation of creative systems [86]. The implementation of Divago is realised in Prolog. The Divago's architecture includes different modules. A knowledge base contains different micro-theories and their instantiations. Of these, two are selected

for the blending by the user or randomly. A mapper then generates the generic space between the inputs, and passes it to a blender module which generates the 'blendoid', i.e., a projection that defines the space of possible blends. Blending in Divago is done on a pair of concepts, taken from the multi-domain[8] knowledge base of the system. A factory component is used to select the best blends among the blendoid by means of a genetic algorithm. A dedicated module implements the optimality principles. Given a blend, this module computes a measure for each principle. These measures yield a preference value of the blend that is taken as the fitness value of the genetic algorithm. Recent works [87,55] describe the use of Divago as a blending component within a computational creativity infrastructure, and discuss the relevance of the optimality principles in the Divago's architecture.

Guhe et al. [36] present an approach to use Heuristic-Driven Theory Projection (HDTP) [76] for blending and concept invention. HDTP is originally a framework for analogical reasoning, using a many-sorted first order language to represent conceptual spaces. In HDTP-based analogical reasoning, knowledge is mapped and transferred from a usually well-known source domain S to a target domain T. This happens in two phases. In the mapping phase, source and target are compared to find commonalities. In particular, HDTP uses second-order anti-unification, which is restricted in way that renders the process decidable. In the transfer phase, unmatched knowledge in source is mapped to the target to establish new hypotheses. The blending of two theories happens in three steps. First, core blend laws are applied, second, preferred conjectures are added, and third, extra conjectures are added. The authors do not directly account for the optimality principles by Fauconnier and Turner. However, the preferred and extra conjectures can be seen as alternative optimality principles, similar to the structural optimality principles proposed by Goguen and Harrell [35], that guide the algorithm to select useful blends among the huge space of possible blends.

A complementary approach to syntactic-driven approaches, such as HDTP, is to focus on the cognitively inspired notion of image schemas, as mentioned in Sec. 2.1.1. The theory of image schemas was developed within cognitive linguistics, and has been extremely influential since Lakoff & Johnson's work in the late 1980s. In cognitive science, image schemas are identified as fundamental pattern of cognition, that are perceived, conceptualised and manipulated spatially and temporally [54]. Examples of image schemas, proposed in the literature, are CONTAINMENT, SUPPORT, and SOURCE_PATH_GOAL. The core idea is that after an image schema has been formed, it can be generalised and the structure can be transferred through analogical reasoning to other domains with similar characteristics [54]. From a formal perspective, previous research on image schemas (e.g., [48,85,77]) has provided a valuable portfolio of approaches that can be build on further. The relation of image schemas and conceptual blending – and, particularly, their role in computational concept invention – have been explored in [40,41,74]. In this setting, image schemas are conceived as a set of theories ordered by logical entailment that can be used to guide the search for a generic space and as heuristics for the creation of new concepts.

The combinatorial kind of creativity [3] that we are interested in has been investigated from a neurological perspective by Thagard and Stewart [80]. The major motivation of their approach is to explain and to model the *Aha!* or *Eureka!* effect that occurs when humans make serendipitous discoveries by means of creative thinking. The authors build their work on findings from neuroscience and approaches to realise human thinking with neural networks [79]. The key idea is to represent mental concepts as activity patterns of vectors of neurons and to perform a convolution operation to combine these patterns. Activity patterns of neurons are mathematically represented as vectors of numbers that represent the firing rate of neurons. According to Thagard and Stewart [80], a mental concept can then be represented as a huge but finite vector of such numbers. The blend is generated by mathematical convolution of vectors. The underlying mathematical model is based on the so-called LIF model of neuronal activity (see e.g., [79]). It accounts for various details on the neuronal level, such as neuron voltage, input current, membrane time, direction vector of neuron patterns, and synaptic connection weights. Thagard and Stewart [80] do not use Fauconnier and Turner's optimality principles to distinguish reasonable blends within the huge space of possible blends. Instead, they combine the blend of two input spaces with another space representing emotional reaction to assess blends. However, the authors do not provide a detailed description how to model the emotional input spaces computationally.

6. Conclusion

We present a computational approach for conceptual blending and implement the generalisation of algebraic specifications using a transition system semantics of preconditions and postconditions within ASP. This allows us to access generalised and commonalised versions of the input specifications, which let us find useful blends. By using theorem solvers, we are able to generate only logically consistent blends. To the best of our knowledge, there exists currently no other blending framework that can resolve logical inconsistencies. Except for [13], we have not seen another blending framework that automatically finds a generic space, while using a similarly expressive representation language. On top of the ASP-based implementation, we propose metrics to evaluate the quality of blends, based on the cognitive optimality principles by Fauconnier and Turner [27].

As overviewed in our related work section, a number of researchers in the field of computational creativity have recognised the value of conceptual blending, and particular implementations of this cognitive theory have been pro-

[8] A domain for Pereira is a set of concepts such that all of them relate to a unique, underlying concept.

posed [83,63,64,33,36,20]. The existing approaches that we investigated lack a sophisticated evaluation to determine formally how 'good' a blend is, with the exception of [63,64], which also use optimality criteria based on [27]'s theory.

The described amalgam-based workflow for conceptual blending is part of a bigger computational framework, the Cobble system,[9] implementated within the COINVENT project [73]. A fairly comprehensive overview of the results of the COINVENT project can be found in [15].[10] A prototypical implementation of the system can be accessed online.[11]

While the proof-of-concept in Section 4 demonstrates some interesting examples, it is by no means close to a solid empirical evaluation. Running the system on a multitude of concept definitions, e.g. from ontology databases such as *Ontohub* [50,5], and developing an evaluation method to quantify the quality of generated blends is subject to future work.

Another more theoretical branch of future work is to generalise our approach to discover creative 'eureka lemmas' in mathematics for more elaborated data structures. For example, a general form of describing a data structure is to define a constructor as $c : list(\tau) \times list(\sigma) \rightarrow \sigma$. This is to say that a constructor can take any number of non-recursive and recursive arguments to form another version of itself. In the example of naturals, the constructor is $s([], [x]) \equiv s(x)$ and for lists $cons([h], [l]) \equiv cons(h, l)$. For binary trees with data at the nodes where the constructor is $t_{::}([h], [l1, l2])$ since there are two recursive arguments. This allows us to find a mapping in the generic space between constructors, and hence to use the techniques expressed in this paper to discover eureka lemmas in new theories.

7. Previous work

This work is based on and extends the publications [20,21] and the COINVENT project deliverable [4]:

- In Eppe et al. [20], published at the International Joint Conference on Artificial Intelligence (IJCAI), we started exploring blending in the music domain and performed first experiments to blend single chords. This paper shows how whole chord progressions are blended.
- In Eppe et al. [21], published at the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), we presented a first version of the amalgam-based workflow. The system was extended substantially in the present paper. Besides performance improvements, the evaluation metrics were improved. Furthermore, we now separate the generalisation path in the amalgam-based workflow into a removal path and a commonalisation path, which eliminates duplicate generic spaces.
- In Bou et al. [4], we provide the theoretical background to demonstrate in how far an amalgam-based workflow can be seen as blending with colimits using Goguen [34]'s approach. In this paper, we go a step further and implement the theory.

In addition to the above considerations, we extend our previous work by presenting cross-domain blending examples.

Acknowledgements

We thank the reviewers for their valuable and constructive comments and feedback. We also thank Mihai Codescu, Free University of Bozen-Bolzano, Italy, for his support with HETS, as well as Maximos Kaliakatsos and Emilios Cambouropoulos, Aristotle University of Thessaloniki, Greece, for their valuable ideas and help with the music examples. The research presented in this article was partially supported by the COINVENT project (FET-Open grant number: 611553). Manfred Eppe received support by the German Academic Exchange Service (DAAD) as participant in the FITweltweit programme. Oliver Kutz and Roberto Confalonieri were supported by the unibz CRC project COCO "Computational Technologies for Concept Invention". The authors thank the Department of Innovation, Research and University of the Autonomous Province of Bozen/Bolzano for covering the Open Access publication costs.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.artint.2017.11.005.

References

[1] D. Anderson, E. Zalta, Frege, boolos and logical objects, J. Philos. Log. 33 (2004) 1–26.
[2] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University Press, 2003.
[3] M.A. Boden, Creativity, in: M.A. Boden (Ed.), Artificial Intelligence (Handbook of Perception and Cognition), Academic Press, 1996, pp. 267–291.
[4] F. Bou, M. Eppe, E. Plaza, M. Schorlemmer, D2.1: reasoning with Amalgams, Technical report, COINVENT Project, available at http://www.coinvent-project.eu/fileadmin/publications/D2.1.pdf, October 2014.
[5] M. Codescu, E. Kuksa, O. Kutz, T. Mossakowski, F. Neuhaus, Ontohub: a semantic repository engine for heterogeneous ontologies, Appl. Ontol. (2017), https://doi.org/10.3233/AO-170190.

[9] http://server.coinvent-project.eu, accessed June 2016.
[10] http://www.coinvent-project.eu, accessed June 2016.
[11] https://github.com/meppe/Amalgamation, accessed August 2016.

[6] S. Colton, J.W. Charnley, A. Pease, Computational creativity theory: the FACE and IDEA descriptive models, in: D. Ventura, P. Gervás, D.F. Harrell, M.L. Maher, A. Pease, G.A. Wiggins (Eds.), Proceedings of the Second International Conference on Computational Creativity, Mexico City, Mexico, April 27–29, 2011, 2011, pp. 90–95.
[7] R. Confalonieri, J.C. Nieves, Nested preferences in answer set programming, Fundam. Inform. 113 (1) (2011) 19–39.
[8] R. Confalonieri, H. Prade, Using possibilistic logic for modeling qualitative decision: answer set programming algorithms, Int. J. Approx. Reason. 55 (2) (2014) 711–738.
[9] R. Confalonieri, J. Corneli, A. Pease, E. Plaza, M. Schorlemmer, Using argumentation to evaluate concept blends in combinatorial creativity, in: H. Toivonen, S. Colton, M. Cook, D. Ventura (Eds.), Proceedings of the Sixth International Conference on Computational Creativity, Park City, Utah, USA, June 29–July 2, 2015, 2015, pp. 174–181.
[10] R. Confalonieri, M. Schorlemmer, E. Plaza, M. Eppe, O. Kutz, R. Peñaloza, Upward refinement for conceptual blending in description logic: an asp-based approach and case study in EL++, in: Proceedings of the Joint Ontology Workshops 2015 Episode 1: The Argentine Winter of Ontology co-located with the 24th International Joint Conference on Artificial Intelligence, 2015.
[11] R. Confalonieri, M. Eppe, M. Schorlemmer, O. Kutz, R. Peñaloza, E. Plaza, Upward refinement operators for conceptual blending in the description logic EL++, Ann. Math. Artif. Intell. (2016) 1–31.
[12] R. Confalonieri, E. Plaza, M. Schorlemmer, A process model for concept invention, in: International Conference on Computational Creativity, ICCC, 2016.
[13] R. Confalonieri, M. Schorlemmer, O. Kutz, R. Peñaloza, E. Plaza, M. Eppe, Conceptual blending in EL++, in: M. Lenzerini, R. Peñaloza (Eds.), Proceedings of the 29th International Workshop on Description Logics, Cape Town, South Africa, April 22–25, in: CEUR Workshop Proc., vol. 1577, CEUR-WS.org, 2016.
[14] R. Confalonieri, O. Kutz, P. Galliani, R. Peñaloza, D. Porello, M. Schorlemmer, N. Troquard, Coherence, similarity, and concept generalisation, in: A. Artale, B. Glimm, R. Kontchakov (Eds.), Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18–21, 2017, in: CEUR Workshop Proc., vol. 1879, CEUR-WS.org, 2017.
[15] R. Confalonieri, A. Pease, M. Schorlemmer, T. Besold, O. Kutz, E. Maclean, M. Kaliakatsos-Papakostas (Eds.), Concept Invention: Foundations, Implementation, Social Aspects and Applications, Comput. Synth. Creat. Syst., Springer, 2018.
[16] M. Eppe, M. Bhatt, Narrative based postdictive reasoning for cognitive robotics, in: International Symposium on Logical Formalizations of Commonsense Reasoning, CR, 2013.
[17] M. Eppe, M. Bhatt, Approximate postdictive reasoning with answer set programming, J. Appl. Log. 13 (4) (2015) 676–719.
[18] M. Eppe, M. Bhatt, F. Dylla, Approximate epistemic planning with postdiction as answer-set programming, in: Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning, 2013.
[19] M. Eppe, M. Bhatt, J. Suchan, B. Tietzen, ExpCog: experiments in commonsense cognitive robotics, in: International Workshop on Cognitive Robotics, CogRob, 2014.
[20] M. Eppe, R. Confalonieri, E. Maclean, M.A. Kaliakatsos-Papakostas, E. Cambouropoulos, W.M. Schorlemmer, M. Codescu, K. Kühnberger, Computational invention of cadences and chord progressions by conceptual chord-blending, in: Q. Yang, M. Wooldridge (Eds.), Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25–31, 2015, AAAI Press, 2015, pp. 2445–2451.
[21] M. Eppe, E. Maclean, R. Confalonieri, O. Kutz, W.M. Schorlemmer, E. Plaza, ASP, amalgamation, and the conceptual blending workflow, in: F. Calimeri, G. Ianni, M. Truszczynski (Eds.), Logic Programming and Nonmonotonic Reasoning – 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27–30, 2015. Proceedings, in: Lect. Notes Comput. Sci., vol. 9345, Springer, 2015, pp. 309–316.
[22] M. Eppe, S. Trott, J. Feldman, Exploiting deep semantics and compositionality of natural language for human–robot-interaction, in: International Conference on Intelligent Robots and Systems, IROS, 2016, pp. 731–738, arXiv:1604.06721.
[23] M. Eppe, S. Trott, V. Raghuram, J. Feldman, A. Janin, Application-independent and integration-friendly natural language understanding, in: Global Conference on Artificial Intelligence, GCAI, 2016, pp. 340–352.
[24] G. Fauconnier, Mental Spaces: Aspects of Meaning Construction in Natural Language, MIT Press, 1985.
[25] G. Fauconnier, Mappings in Thought and Language, Cambridge University Press, Cambridge, England, 1997.
[26] G. Fauconnier, M. Turner, Conceptual integration networks, Cogn. Sci. 22 (2) (1998) 133–187, reprinted in: D. Geeraerts (Ed.), Cognitive Linguistics: Basic Readings, pp. 303–371.
[27] G. Fauconnier, M. Turner, The Way We Think: Conceptual Blending And The Mind's Hidden Complexities, Basic Books, 2002.
[28] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, S. Thiele, Engineering an incremental ASP solver, in: International Conference on Logic Programming, LPNMR, 2008.
[29] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Clingo = ASP + control: preliminary report, CoRR, arXiv:1405.3694, 2014.
[30] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, Potassco User Guide 2.0, Technical report, University of Potsdam, May 2015.
[31] M. Gelfond, Y. Kahl, Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach, Cambridge University Press, New York, NY, USA, 2014.
[32] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of the Fifth International Conference on Logic Programming, ICLP, The MIT Press, 1988, pp. 1070–1080.
[33] J. Goguen, D.F. Harrell, Style: a computational and conceptual blending-based approach, in: S. Argamon, K. Burns, S. Dubnov (Eds.), The Structure of Style: Algorithmic Approaches to Understanding Manner and Meaning, Springer, 2010, pp. 291–316.
[34] J.A. Goguen, An introduction to algebraic semiotics, with application to user interface design, in: Computation for Metaphors, Analogy, and Agents, 1999, pp. 1–39.
[35] J.A. Goguen, D.F. Harrell, Foundations for active multimedia narrative: semiotic spaces and structural blending, https://cseweb.ucsd.edu/goguen/pps/narr.pdf, 2005 (Accessed 15 September 2017).
[36] M. Guhe, A. Pease, A. Smaill, M. Martínez, M. Schmidt, H. Gust, K.-U. Kühnberger, U. Krumnack, A computational account of conceptual blending in basic mathematics, Cogn. Syst. Res. 12 (3–4) (2011) 249–265.
[37] B. Hampe, Image schemas in cognitive linguistics: an introduction, in: B. Hampe, J.E. Grady (Eds.), From Perception to Meaning: Image Schemas in Cognitive Linguistics, in: Cogn. Linguist. Res., vol. 29, Mouton de Gruyter, 2005, pp. 1–12.
[38] D.F. Harrell, Shades of computational evocation and meaning: the GRIOT system and improvisational poetry generation, in: 6th Digital Arts and Culture Conference, 2005.
[39] D.F. Harrell, Theory and Technology for Computational Narrative: An Approach to Generative and Interactive Narrative With Bases in Algebraic Semiotics and Cognitive Linguistics, Ph.D. thesis, University of California, San Diego, 2007.
[40] M.M. Hedblom, O. Kutz, F. Neuhaus, Choosing the right path: image schema theory as a foundation for concept invention, J. Artif. Gen. Intell. 6 (1) (2015) 21–54.
[41] M.M. Hedblom, O. Kutz, F. Neuhaus, Image schemas in computational conceptual blending, Cogn. Syst. Res. 39 (2016) 42–57.
[42] A. Ireland, A. Bundy, Productive use of failure in inductive proof, J. Autom. Reason. 16 (1–2) (1996) 79–111.
[43] M. Johansson, L. Dixon, A. Bundy, Conjecture synthesis for inductive theories, J. Autom. Reason. 47 (2011) 251–289.
[44] M. Johnson, The Body in the Mind. The Bodily Basis of Meaning, Imagination, and Reasoning, The University of Chicago Press, 1987.

[45] A. Jordanous, Stepping back to progress forwards: setting standards for meta-evaluation of computational creativity, in: S. Colton, D. Ventura, N. Lavrac, M. Cook (Eds.), Proceedings of the Fifth International Conference on Computational Creativity, Ljubljana, Slovenia, June 10–13, 2014, 2014, pp. 129–136, computationalcreativity.net.
[46] A. Koestler, The Act of Creation, Hutchinson & Co., 1964.
[47] R. Kowalski, Predicate logic as programming language, in: Proceedings of International Federation for Information Processing, 1974, pp. 569–574.
[48] W. Kuhn, An image-schematic account of spatial categories, in: S. Winter, M. Duckham, L. Kulik, B. Kuipers (Eds.), Spatial Information Theory, in: Lect. Notes Comput. Sci., vol. 4736, Springer, 2007, pp. 152–168.
[49] O. Kutz, J. Bateman, F. Neuhaus, T. Mossakowski, M. Bhatt, E pluribus unum: formalisation, use-cases, and computational support for conceptual blending, in: T.R. Besold, M. Schorlemmer, A. Smaill (Eds.), Computational Creativity Research: Towards Creative Machines, Thinking Machines, Atlantis/Springer, 2014.
[50] O. Kutz, T. Mossakowski, F. Neuhaus, M. Codescu, Blending in the Hub: towards a computational concept invention platform, in: International Conference on Computational Creativity, ICCC, 2014.
[51] G. Lakoff, Women, Fire, and Dangerous Things, University of Chicago Press, 1987.
[52] G. Lakoff, M. Johnson, Philosophy in the Flesh: The Embodied Mind and Its Challenge to Western Thought, Basic Books, 1999.
[53] G. Malcolm, Software Engineering with OBJ: Algebraic Specification in Action, Kluwer, 2000.
[54] J.M. Mandler, C. Pagán Cánovas, On defining image schemas, Lang. Cogn. 6 (2014) 1–23.
[55] P. Martins, S. Pollak, T. Urbančič, A. Cardoso, Optimality principles in computational approaches to conceptual blending: do we need them (at) all? in: International Conference on Computational Creativity, ICCC, 2016.
[56] O. Montano-Rivas, R. McCasland, L. Dixon, A. Bundy, Scheme-based synthesis of inductive theories, in: MICAI, in: Lect. Notes Comput. Sci., vol. 6437, 2010, pp. 348–361.
[57] T. Mossakowski, Colimits of order-sorted specifications, in: Recent Trends in Algebraic Development Techniques, in: Lect. Notes Comput. Sci., vol. 1376, Springer, Berlin, 1998, pp. 316–332.
[58] T. Mossakowski, C. Maeder, K. Lüttich, The heterogeneous tool set, Hets, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS, Springer Berlin Heidelberg, 2007, pp. 519–522.
[59] T. Mossakowski, M. Codescu, F. Neuhaus, O. Kutz, The distributed ontology, modelling and specification language – DOL, in: The Road to Universal Logic – Festschrift for 50th birthday of Jean-Yves Beziau, vol. II, in: Stud. Univers. Log., Birkhäuser, 2015.
[60] P.D. Mosses, CASL Reference Manual – The Complete Documentation of the Common Algebraic Specification Language, Springer, 2004.
[61] S. Ontañón, E. Plaza, Amalgams: a formal approach for combining multiple case solutions, in: I. Bichindaritz, S. Montani (Eds.), Case-Based Reasoning. Research and Development, ICCBR, Springer, 2010, pp. 257–271.
[62] A. Pease, J. Corneli, Evaluating creativity, in: Concept Invention: Foundations, Implementation, Social Aspects and Applications, Springer, ISBN 978-3-319-65602-1, http://www.springer.com/de/book/9783319656014, in press.
[63] F.C. Pereira, A Computational Model of Creativity, Ph.D. thesis, Universidade de Coimbra, 2005.
[64] F.C. Pereira, Creativity and Artificial Intelligence: A Conceptual Blending Approach, Mouton de Gruyter, 2007.
[65] F.C. Pereira, A. Cardoso, The boat-house visual blending experiment, in: Proceedings of the 2nd Workshop on Creative Systems: Approaches to Creativity in AI and Cognitive Science, ECAI 2002, Lyon, France, 2002.
[66] F.C. Pereira, A. Cardoso, Optimality principles for conceptual blending: a first computational approach, AISB J. 1 (4) (2003) 351–370.
[67] F.C. Pereira, A. Cardoso, The horse-birdcreature generationexperiment, AISB J. 1 (3) (2003) 257–280.
[68] F.C. Pereira, A. Cardoso, Experiments with free concept generation in Divago, Knowl.-Based Syst. 19 (7) (2006) 459–470.
[69] B. Pierce, Basic Category Theory for Computer Scientists, MIT Press, 1991.
[70] D. Porello, N. Troquard, R. Confalonieri, P. Galliani, O. Kutz, R. Peñaloza, Repairing socially aggregated ontologies using axiom weakening, in: Proc. of the 20th International Conference on Principles and Practice of Multi-Agent Systems, PRIMA 2017, October 30–November 3, Nice, France, Springer, 2017.
[71] M. Quillian, Semantic Memory. Semantic Information Processing, 1968.
[72] G. Ritchie, Assessing creativity, in: G. Wiggins (Ed.), Proceedings of the AISB'01 Symposium on AI and Creativity in Arts and Science, 2001, pp. 3–11.
[73] M. Schorlemmer, A. Smaill, K.-U. Kühnberger, O. Kutz, S. Colton, E. Cambouropoulos, A. Pease, COINVENT: towards a computational concept invention theory, in: Fifth International Conference on Computational Creativity, ICCC, 2014.
[74] M. Schorlemmer, R. Confalonieri, E. Plaza, The Yoneda Path to the Buddhist Monk Blend, in: First International Workshop on Cognition and Ontologies, CAOS 2016, Annecy, France, 6th July, 2016.
[75] M. Schorlemmer, R. Confalonieri, E. Plaza, Coherent concept invention, in: T.R. Besold, O. Kutz, C. León (Eds.), Proceedings of the Workshop on Computational Creativity, Concept Invention, and General Intelligence, C3GI 2016, co-located with the 28th European Summer School in Logic, Language and Information, ESSLLI 2016, Bozen-Bolzano, Italy, August 20–22, 2016, in: CEUR Workshop Proc., vol. 1767, CEUR-WS.org, 2016.
[76] A. Schwering, U. Krumnack, K.-U. Kühnberger, H. Gust, Syntactic principles of heuristic-driven theory projection, Cogn. Syst. Res. 10 (3) (2009) 251–269.
[77] R.St. Amant, C.T. Morrison, Y.-H. Chang, P.R. Cohen, C. Beal, An image schema language, in: International Conference on Cognitive Modeling, ICCM, 2006, pp. 292–297.
[78] P. Thagard, Coherence in Thought and Action, The MIT Press, 2000.
[79] P. Thagard, The Brain and the Meaning of Life, Princeton University Press, 2010.
[80] P. Thagard, T.C. Stewart, The AHA! experience: creativity through emergent binding in neural networks, Cogn. Sci. 35 (1) (2011) 1–33.
[81] S. Trott, M. Eppe, J. Feldman, Recognizing intention from natural language: clarification dialog and construction grammar, in: Workshop on Communicating Intentions in Human–Robot Interaction, 2016.
[82] T. Veale, From conceptual "mash-ups" to "bad-ass" blends: a robust computational model of conceptual blending, in: International Conference on Computational Creativity, ICCC, 2012.
[83] T. Veale, D.O. Donoghue, Computation and blending, Cogn. Linguist. 11 (3–4) (2000) 253–282.
[84] T. Veale, M. Keane, The competence of sub-optimal theories of structure mapping on hard analogies, in: IJCAI, 1997, pp. 232–237.
[85] L. Walton, M. Worboys, An algebraic approach to image schemas for geographic space, in: Proceedings of the 9th International Conference on Spatial Information Theory, COSIT, France, 2009, pp. 357–370.
[86] G.A. Wiggins, A preliminary framework for description, analysis and comparison of creative systems, Knowl.-Based Syst. 19 (7) (2006) 449–458.
[87] M. Žnidaršič, A. Cardoso, P. Gervas, P. Martins, R. Hervas, A. Alves, H. Oliveira, P. Xiao, S. Linkola, H. Toivonen, J. Kranjc, N. Lavraĉ, Computational creativity infrastructure for online software composition: a conceptual blending use case, in: International Conference on Computational Creativity, ICCC, Paris, France, 1st July, 2016.