Matija Kučko

LotTraveler: Workflow Management in Failure Analysis

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Studium: Masterstudium Angewandte Informatik

Alpen-Adria-Universität Klagenfurt

Gutachter
Assoc. Prof. Dipl.-Ing. Dr. Konstantin Schekotihin
Alpen-Adria-Universität Klagenfurt
Institut für Artificial Intelligence und Cybersecurity

Klagenfurt, Juni 2022

Affidavit

I hereby declare in lieu of an oath that

- the submitted academic paper is entirely my own work and that no auxiliary materials have been used other than those indicated,
- I have fully disclosed all assistance received from third parties during the process of writing the thesis, including any significant advice from supervisors,
- any contents taken from the works of third parties or my own works that have been included either literally or in spirit have been appropriately marked and the respective source of the information has been clearly identified with precise bibliographical references (e.g. in footnotes),
- to date, I have not submitted this paper to an examining authority either in Austria or abroad and that
- when passing on copies of the academic thesis (e.g. in bound, printed or digital form), I will ensure that each copy is fully consistent with the submitted digital version.

I am aware that a declaration contrary to the facts will have legal consequences.

Matija Kučko m.p. Klagenfurt, June 2022

Acknowledgements

First and foremost, I want to thank my supervisor, Mr. Schekotihin. My supervisor was continuously patient with me, gave me essential feedback throughout the whole project and ultimately reviewed this thesis by providing crucial editorial suggestions. The guidance and assistance from my supervisor was invaluable in establishing the problem to be solved through multiple meetings with experts and stakeholders from Infineon AG, as well as in guiding and nudging me towards relevant literature works, industry tools and overall strategy for addressing the problem statement. In the same vein, I want to thank all stakeholders from Infineon AG for their valuable time, feedback and insights, without whom this work would not exist. Furthermore, I wish to thank all faculty members who equipped me with the necessary skills, fundamental knowledge and tool expertise, so that I may continually learn and overcome any challenges that may come in my professional future. Their patience with me is likewise commendable. I would not be where I am today without the measureless support of my family, friends and colleagues, who stood by me through the more challenging phases of my school career. I want to thank dear God, Who I could always confide in, Who provided me with solace in dire times of need, and Who helped me overcome seemingly insurmountable problems in my life.

Thank you all from the bottom of my heart.

Disclaimer: Throughout this paper the pronoun “we” is used to refer to both me, the author, and my esteemed supervisor.

Abstract

In typical production environments, a client requests a service from a provider or a product from a vendor. These requests may contain a specific sequence of activities to be performed, stating not *what* the desired outcome is, but *how* it is to be achieved. Experts then perform these activities by hand using appropriate tools and machines. Consider, for example, a chemistry lab that offers services for the chemical analysis of substances provided by the client. A client may request a specific sequence of analysis activities for a sample, ranging from optical microscopy to biodegradation, for example. While the former activity performs a non-invasive optical inspection of the provided substance, the latter involves a non-reversible breakdown of the sample by biological means. Another example is the production of substances by chemical synthesis, which allows the produced substance to be used for further analysis tasks. Therefore, the order of activities is crucial and directly affects the final analysis outcome. Regardless of the particular application area, the sequence of activities needs to make sense and be feasible. On the one hand, the client must ensure that the activities form a meaningful workflow that leads to the desired result; on the other hand, domain experts must review incoming requests and reject any invalid ones. As mentioned earlier, some activities may irreversibly alter samples, requiring all initial non-invasive analyses to be performed in advance as opposed to later. In addition, some tasks may require that other prerequisite tasks be completed first. Clients and experts typically need to validate such requests by hand. This manual process of checking whether incoming requests comply with common ground rules is inherently repetitive and error-prone.

To address the repetitive and error-prone nature of manual workflow checks, an automated process needs to be put in place that allows experts to (1) describe these business rules and domain constraints in a formal way and (2) check incoming client requests against these constraints. The system should not only issue appropriate error messages indicating which business rules a workflow violates but also provide an option to repair such workflows. To address the first criterion (1), we use a process modeling language for which all (implicitly) possible execution sequences represent the allowed behavior. For the second criterion (2), we check whether the given sequence of tasks fits the reference model perfectly. If it does not fit, we repair the workflow by choosing the most similar valid alternative instead.

We answer all resulting research questions by implementing an approach that leverages the modeling concepts of *tasks* and *connections* for both representing and drawing conclusions on allowable workflows. We evaluated the proposed approach by applying it to the area of failure analysis (FA) in Infineon AG Villach and found our implementation successful in modeling and reasoning about concepts and workflows in this domain. In particular, we were able to properly encode ground rules common to all 14 elicited template workflows.

Zusammenfassung

Kunden können Dienstleistung von einem Anbieter oder ein Produkt von einem Verkäufer anfordern. Diese Anfragen können eine bestimmte Abfolge von auszuführenden Tätigkeiten enthalten. Dabei wird nicht angegeben, *was* das gewünschte Ergebnis ist, sondern *wie* es erreicht werden soll. Experten führen diese Tätigkeiten dann von Hand mit den entsprechenden Werkzeugen und Maschinen aus. Nehmen wir als Beispiel ein Chemielabor, das Dienstleistungen für die chemische Analyse von Substanzen anbietet, die der Kunde zur Verfügung stellt. Ein Kunde kann eine bestimmte Abfolge von Analysetätigkeiten für eine Probe anfordern, die z. B. von der optischen Mikroskopie bis zum biologischen Abbau reichen kann. Während die erste Aktivität eine nicht-invasive optische Inspektion der bereitgestellten Substanz durchführt, beinhaltet die zweite einen nicht-reversiblen Abbau der Proben durch biologische Mittel. Ein weiteres Beispiel ist der Aufbau von Substanzen durch chemische Synthese, sodass die aufbereitete Substanz für weitere Analyseaufgaben verwendet werden kann. Die Reihenfolge der Tätigkeiten ist daher entscheidend und wirkt sich direkt auf das endgültige Analyseergebnis aus. Unabhängig vom jeweiligen Anwendungsbereich muss die Abfolge der Tätigkeiten sinnvoll und durchführbar sein. Einerseits muss der Kunde sicherstellen, dass die Aktivitäten zu einem sinnvollen Arbeitsablauf führen, der das gewünschte Ergebnis liefert, andererseits müssen die Fachexperten die eingehenden Anfragen prüfen und ungültige Anfragen zurückweisen. Wie bereits erwähnt, können einige Tätigkeiten die Proben unumkehrbar verändern, sodass nicht-invasive Analysen eher im Voraus als später durchgeführt werden müssen. Darüber hinaus können einige Aufgaben andere Vorbereitungsaufgaben erfordern, die vor diesen durchgeführt werden müssen. Kunden und Experten müssen solche Anfragen in der Regel von Hand validieren. Dieses manuelle Verfahren, bei dem geprüft wird, ob die eingehenden Anfragen den gemeinsamen Grundregeln entsprechen, ist von Natur aus repetitiv und fehleranfällig.

Um die repetitive und fehleranfällige Natur der manuellen Workflow-Prüfungen zu beheben, muss ein automatisiertes Verfahren eingeführt werden, welches den Experten ermöglicht, erstens, diese Geschäftsregeln und Domäneneinschränkungen auf formale Weise zu beschreiben und, zweitens, eingehende Kundenanfragen anhand dieser Einschränkungen zu überprüfen. Das System sollte nicht nur entsprechende Fehlermeldungen ausgeben, die aufzeigen, gegen welche Geschäftsregeln ein Workflow verstößt, sondern auch eine Möglichkeit bieten, solche Workflows zu reparieren. Um das erste Kriterium zu erfüllen, verwenden wir eine Prozessmodellierungssprache, für die alle (implizit) möglichen Ausführungssequenzen das zulässige Verhalten darstellen. Für das zweite Kriterium prüfen wir, ob die gegebene Aufgabenabfolge perfekt zum Referenzmodell passt. Wenn sie nicht passt, reparieren wir den Arbeitsablauf, indem wir stattdessen die ähnlichste gültige Alternative wählen.

Wir beantworten alle sich daraus ergebenden Forschungsfragen, indem wir einen solchen Ansatz implementieren, der die Modellierungskonzepte von *Aufgaben* und *Verbindungen* sowohl für die Darstellung als auch für das Schlussfolgerungsziehen über zulässige Arbeitsabläufe nutzt. Wir haben den vorgeschlagenen Ansatz für den Anwendungsbereich der Fehleranalyse (FA) in der Infineon AG Villach evaluiert und festgestellt, dass unsere Implementierung für die Modellierung und Schlussfolgerung über Konzepte & Workflows, die in diesem Bereich vorkommen, geeignet ist. Insbesondere waren wir in der Lage, Grundregeln zu kodieren, die allen 14 ermittelten Musterworkflows gemeinsam sind.

Contents

1 Introduction

In *typical* production environments, a client requests a service from a provider or a product from a vendor. In many cases, the ordered deliverable is ready-made and provided according to a static, set-in-stone process description. Occasionally, however, this requested service or product may further be configured to suit a client's particular needs, be either by letting the client choose from a predefined set of product variants or allowing the client to customize the product by selecting from an extensive set of configuration options. Another alternative revolves around the client requesting a specific sequence of activities to be performed instead, stating not *what* the desired outcome is, but *how* it is to be achieved. Experts then *typically* perform such activities by hand using appropriate tools and machines. Figure 1.1 depicts this scenario, where a client supplies a sequence of tasks to be enacted by the workflow management system (WFMS) of the service provider or product vendor. The WFMS takes care of coordinating and orchestrating tasks between available resources in production environments, such as workers and tools.

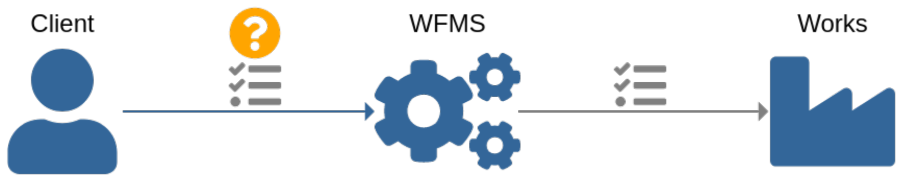


Figure 1.1: Custom-made deliverable obtained by completing a sequence of activities supplied by the client

Consider, for example, a chemistry lab that offers services for the chemical analysis of substances provided by the client. A client may request a specific sequence of analysis activities for a sample, ranging from optical microscopy to biodegradation, for example. While the former activity performs a non-invasive optical inspection of the provided substance, the latter involves a non-reversible breakdown of the sample by biological means. Another example is the production of substances by chemical synthesis, which allows the produced substance to be used for further analysis tasks. Therefore, the order of activities is crucial and directly affects the final analysis outcome.

Regardless of the particular application area, the sequence of activities needs to make sense and be feasible. On the one hand, the client must ensure that the activities form a meaningful workflow that leads to the desired result; on the other hand, domain experts must review incoming requests and reject any invalid ones. As mentioned earlier, some activities may irreversibly alter samples, requiring all initial non-invasive analyses to be performed in advance as opposed to later. In addition, some tasks may require that other prerequisite tasks be completed first. Clients and experts typically need to validate such requests by hand. This manual process of checking whether incoming requests comply with common ground rules is inherently repetitive and error-prone.

Use case: Failure Analysis in Infineon

Through elicitation of requirements with the help of experts from Infineon's failure analysis lab, we have identified such a real-world application area for improving the day-to-day operations in a *typical* FA lab. This problem is encountered in the work of Infineon AG Villach on automation of the job management in the failure analysis (FA) department. The department is responsible for diagnosis of faults in integrated circuits delivered by both the manufacturing department and clients.

A job is a box comprising possibly faulty integrated circuits (five or so) associated with a partially ordered set (poset) of possibly time-coupled analysis tasks to be accomplished, such as sonography, chemical dissolving, visual inspections, etc. The partial order defines a number of admissible sequences in which tasks must be executed and depends on the requirements of internal or external clients. For instance, the order of visual inspection and sonography is not important, whereas microscope inspections can only be done after dissolving. Also, some of the tasks are destructive and lead to loss of samples and, thus, must be avoided if possible. Moreover, some tasks following such destructive actions have to be performed within a short time-frame afterwards. For example, some visual inspections have to be performed just after chemical dissolving, since the sample may deteriorate further over time.

These time-based and order-based constraints on analysis tasks are defined based on the underlying workflow, which is a sequence of failure analysis tasks that are (implicitly) requested by the client according to their requirements. In some cases, these requirements can be fulfilled by following a specific workflow template. One such template is the Highly Accelerated Stress Test (HAST), in which components under test are subjected to high temperature and high humidity in order to accelerate the build-up of corrosion inside [GuMM81]. The underlying workflow is static and consists of the same set of partially ordered tasks each time a client requests the analysis of possibly faulty integrated circuits using the aforementioned template. On the other hand, clients can have more specific needs such as a custom-tailored workflow to efficiently test for specific flaws or to make sure no irrelevant analysis tasks taint these samples and thus the analysis result. However, even custom-tailored workflows must follow some ground rules, e.g. an internal visual inspection task cannot be performed unless the integrated circuit has been decapsulated and its "internals" have been exposed.

To tie everything together with proper terminology, we refer to the amalgamation of these ground rules as the meta-model of FA jobs, the actual workflow following these rules (be it a workflow template or custom-tailored workflow) as the model of FA jobs and each FA job as a specific instance of the respective (workflow) model.

In addition to workflows, jobs and samples, there are other factors at play in the FA department. Each worker (employee) can usually accomplish a number of different tasks using lab's tools. In many cases, however, a worker has preferences with respect to tasks. Some reasons for these preferences are: the worker is still learning how to use the tool (trainee) or lacks expertise for a tool (machine) on which the task is to be accomplished, simply prefers to do a specific task, or is currently unavailable (vacation or sick leave). Some tools (machines) allow tasks to be stacked onto them, allowing multiple tasks to be completed simultaneously.

A more in-depth look into the importance of failure analysis techniques for increasing yield of integrated circuit is given in Oberai and Yuan's work [ObYu18].

1.1 Problem statement

To address the repetitive and error-prone nature of manual workflow checks, there needs to be an automated procedure in place to validate and adjust possibly invalid workflows, which ensures that the resulting workflow adheres to all business rules and domain restrictions. Figure 1.2 depicts this scenario, where a client supplies a sequence of tasks to be enacted by the workflow management system (WFMS) of the service provider or product vendor. However, in contrast to the previous scenario represented in Figure 1.1, each client request is first checked, and thus approved or rejected by an automatic procedure, before being supplied to the WFMS.

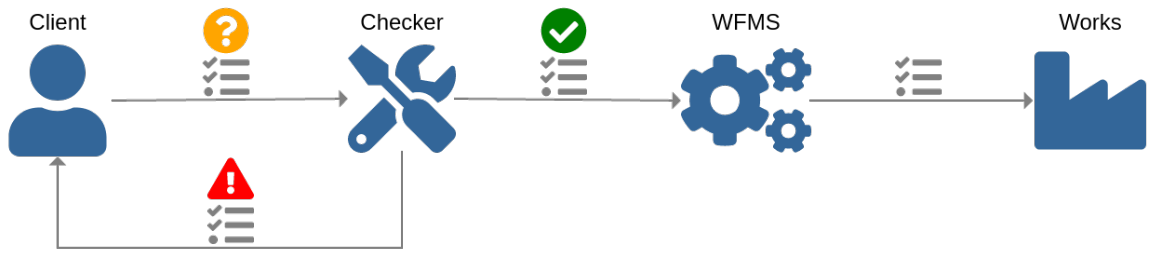


Figure 1.2: Custom-made deliverable obtained by completing a sequence of activities supplied by the client, checked by an automatic procedure

To this end, production environments need to have such a system in place that allows experts to, firstly, describe these business rules and domain constraints in a formal way and, secondly, check incoming client requests against these constraints. The system should not only issue appropriate error messages indicating which business rules a workflow violates but also provide an option to repair such workflows. Since clients usually know what specific tasks they want performed but do not necessarily give importance to the order of tasks or the required dependency tasks, the repaired workflow must be an extension of the requested sequence of tasks.

Therefore, the system needs to provide the following functionalities:

F1 Design-time representation of valid workflows through task ordering and additional restrictions

F2 Run-time detection of potentially invalid workflows, as well as automated repair of such inconsistent workflows

A workflow (model) is a bounded linear sequence of tasks[1]. In the former part, domain experts create a workflow meta-model which captures all ground rules for any such custom-tailored

[1]In failure analysis, these linear sequences *typically* consists of no more than 15 tasks.

or template workflow (model). Experts list all possible tasks, which tasks may follow other tasks, as well as mandatory tasks that must appear alongside other ones. In the latter part, an automated procedure checks a custom-tailored (or even template) workflow against those ground rules, whether it contains unknown tasks, whether the tasks adhere to the specified order and whether mandatory dependencies are included in the workflow. If the workflow is invalid, the "best" valid alternative is recommended instead. Otherwise, the checked workflow is indeed a valid model of the workflow meta-model. This workflow (model) may then be executed multiple times, with each execution being referred to as a workflow (model) instance.

1.2 Research questions

In order to solve the postulated problem statement, we need to break it down and find answers to the following more specific questions:

RQ1 What do these workflows *typically* look like? How can we model all potential workflows that are requested by clients and that are subsequently checked by experts in order to determine whether they follow business rules and domain restrictions? Which notation, with formally defined semantics, can we use to represent such valid workflows? Can we identify common modeling elements to capture ground rules which all custom-tailored workflows need to adhere to?

RQ2 How can we define and verify these validness constraints of workflows? In which way can these previously identified modeling elements be used to encode such constraints? How can we check workflow consistency automatically?

RQ3 How can an automated repair procedure construct the "best" valid workflow for an inconsistent one? How can we find such alternative efficiently?

By finding answers to this first set of questions (*RQ1*) we address the first part of the problem statement (*F1*). Analogously, answers to the latter sets of questions (*RQ2* & *RQ3*) guide us in the right direction for handling the latter part of the problem statement (*F2*). In the remainder of this work, we aim to answer these questions.

1.3 Background

In order to find solutions to the problem statement and research questions formulated before, we have to view the subject matter from another perspective. In the same vein that models need to adhere to the syntax and semantics facilitated by their meta-models, (model) instances need to adhere to their respective model framework. Therefore, by reducing the subject matter by one abstraction level, solutions that allow us to model workflows and check if specific workflow executions adhere to such model can, in turn, be also applied on the abstraction level above, for creating workflow meta-models and determining whether workflow models are valid instances of those meta-models.

1.3.1 Process modeling

For addressing *RQ1*, we thus leverage (business) process modeling languages for presenting ground rules (meta-model) for all workflow (models). The area of workflow modeling is well-researched, an overview of the state-of-the-art in process modeling languages is provided in

Abbad Andaloussi et al.'s work [ABSK$^+$20]. These languages can be categorized into imperative process modeling languages, "where all execution alternatives must be explicitly specified" [Pesi08][ABSK$^+$20], declarative process modeling languages, which provide an "implicit specification of [all] execution alternatives" [Pesi08] that satisfy the given constraints [ABSK$^+$20], as well as hybrid process modeling languages, which combine the advantages of both imperative and declarative paradigms —understandability and flexibility, respectively [ABSK$^+$20].

1.3.2 Process mining

For addressing research questions ***RQ2*** and ***RQ3*** we turn to the family of techniques encountered in process mining. This area is like-wise well-researched and is an umbrella term that encompasses various disciplines that deal with the analysis of, and relationships between, process models and process instances.

While typical process modeling languages are used to denote process models, so-called *event logs* are used to represent corresponding process instances [AaAD12]. Events can be thought of as log statements that are produced sequentially during the execution of a workflow [AaAD12]. An event includes a reference to the *activity* (task) that was executed and a reference to the *case* (process instance) in which this task was executed [AaAD12]. It can contain additional information, such as a *timestamp* when it occurred, a *resource* link that identifies the actor (user or service) that executed the task, as well as any *data* properties entered during the execution of such task [AaAD12].

These event logs enable different types of process mining. In *process discovery*, one "take[s] an event log and produce[s] a model without using any other a priori information" [AaAD12]. *Conformance checking* "identifies deviations between the process instances' actual behavio[]r (*as-is*) and its model[]ed behavio[]r (*to-be*)" [DSMB19] and vice versa [AaAD12]. Lastly, *process enhancement* deals with *repairing* or *extending* "an existing process model using information about the actual process recorded in some event log" [Aals16] or vice versa [DSMB19].

A general overview of process mining is given in Van Der Aalst's book [Aals16]. A state-of-the-art review of conformance checking literature is provided in Dunzer et al.'s work [DSMB19]. Yasmin, Bukhsh and De Alencar Silva investigate the state of the art in process enhancement in their work [YaBD18].

Conformance checking

In the context of conformance checking, we are interested in how well a process model reflects the actual process found in event logs. Such appropriateness of a model is measured along multiple quality dimensions [BDWVB14][Aals16]:

- *fitness*: describes to what extent the model allows the behavior displayed by event logs
- *precision*: describes to what extent the model forbids behavior not encountered in event logs
- *generalization*: describes to what extent the model is able to generalize on current and future event logs, by not specializing too much on example event logs
- *simplicity*: describes the complexity of the model relative to all other *fitting* models

There exist three general approaches for conformance checking, primarily aimed at computing the fitness value: *log replay*, *trace alignment* and *footprint comparison* algorithms

[BDWVB14][Aals16]. An event log L, which is common to all approaches, contains all process executions, which are for the purpose of these algorithms viewed as sequences of executed tasks $L = \{\langle a, b\rangle, \langle a, b, a, c, d\rangle, ...\}$.

In a common variant of the first approach, known as *token-based* log replay, the process model is converted into a corresponding Petri net if possible, on which a token simulation is performed [BDWVB14][Aals16]. Namely, by firing the appropriate transition that matches the next event in a case's execution sequence, the state of the Petri net is advanced. If a subsequently expected transition is not enabled (due to a mismatch between the model and the log), artificial tokens are produced in the appropriate places to fire such transition and keep the simulation rolling. When the Petri net reaches its end state, the process conformance is calculated based on the "sum of all superfluous and generated tokens" [BDWVB14]. This approach is repeated for all cases in the event log to compute the final conformance value.

In the second approach, an algorithm tries to find an *optimal alignment* [BDWVB14][Aals16]: To this end, all possible execution sequences of a model are compared with a single execution sequence from the event log. While comparing one such model sequence with the event sequence, appropriate skip / noop moves (») are inserted in the model or log sequence, such that these sequences align. For each two sequences, multiple such alignments can be found. Skip moves performed on the model sequence are referred to as *model moves*, whereas those performed on a case's execution trace are known as *log moves. Synchronous moves* are used to tag activities that match in both execution sequences. In addition, both log and model moves incur a (potentially different) cost. An optimal alignment is then the one that has the lowest total cost of moves associated with aligning the most favorable model sequence with the given execution sequence. Multiple such cost-minimal alignments can exist. The amount of non-synchronous moves in an optimal alignment is divided by the amount of non-synchronous moves in the *worst-case* alignment in order to determine the conformance, whereas the latter amount is just the sum of both execution sequence's lengths. This approach is repeated for all cases in the event log to compute the final conformance value.

In the third approach, an algorithm compares *footprints* of the process model and the event log [Aals16][SuDL17]: A footprint consists of pair-wise ordering and casual relations between every two tasks. In particular, $a > b$ denotes that b is a direct successor of a in at least a single execution sequence facilitated by the model $a >_M b$ or contained in the event log $a >_L b$. $a >> b$ states the indirect successor relation, with one task following the other after a number of intermediary tasks $a > k > ... > b$ in at least a single execution sequence. $a \rightarrow b$ represents a direct casual relation between tasks a and b, such that in all execution sequences a is a direct successor $a > b$ of b with the inverse never occurring $a \not> b$. $a \Rightarrow b$ denotes an indirect casual relation, where at least a single execution sequence exists where those two tasks are causally related via intermediary tasks $a \rightarrow k \rightarrow ... \rightarrow b$. Mutual exclusion $a \# b$ is inferred when both tasks are not allowed to occur together, so that $a \not> b \land b \not> a$ or $a \not\gg b \land b \not\gg a$. Parallelism between tasks $a||b$ follows if both tasks can occur alongside each other in different variations, such that $a > b \land b > a$ or $a >> b \land b >> a$. A difference matrix can then be constructed for entries where the inferred pair-wise relations differ between model and log. A simple conformance value amounts to the quotient of mismatched entries divided by the total number of pairs.

Furthermore, special cases of conformance checking can be applied to declarative process models, e.g. by checking the event log passes all rules defined in a rule-based process modeling language [Aals16].

Process enhancement

In the context of process enhancement, we are interested in how we can repair or extend an existing process model so that it better reflects the actual process found in event logs. Multiple approaches have been proposed to repair process models based on viewing the event log as the reference of actual process behavior [YaBD18]. However, it is also possible to repair event traces that are incomplete by treating the process model as the *as-is* behavior instead. All these approaches expand the notion of conformance checking in order to determine to what extent different execution sequences conform to each other.

In model repair, appropriate procedures transform a given non-conforming process model to the most similar model that conforms to the event log as best as possible [FaAa15]. Therefore, "[t]here are two forces guiding such repair. First of all, there is the need to improve conformance. Second, there is the desire to clearly relate the repaired model to the original model, i.e., repaired model and original model should be similar" [FaAa15]. In event log repair, an appropriate procedure does the reverse. It finds a model execution sequence that best conforms to the given non-conforming event sequence. The given event sequence is then repaired by transforming it to perfectly align with this model execution sequence. The repaired event sequence, however, should be as close as possible to the original one.

The looked-at approaches for repairing event logs [RSMAW13][WSZL13] augment an incomplete event sequence by inserting missing tasks at appropriate sequence positions. These approaches share in common the focus on the recovery of missing events via insertion operations only, since the original sequence s must be a subsequence of the repaired one s', with $|s| \leq |s'|$ and $\forall k \in [1, |s|] : \quad |\{i \in [1, |s|] \mid i \geq k \wedge s[i] = s[k]\}| \;\leq\; |\{j \in [1, |s'|] \mid j \geq k \wedge s'[j] = s[k]\}|$.

1.4 Approach

As modeling languages facilitate "the concretization of a business process" and the "descri[ption of] the way a business process operates in the real world in a formal or informal way" [ABSK+20], we compare such modeling languages on their suitability for (***RQ1***) formally stating all ground rules that custom-tailored workflows need to adhere to, according to a subjective evaluation of relevant quality criteria. With the reducibility of the subject matter to a lower abstraction level in mind, we then adapt one such language candidate from this non-exhaustive collection for workflow meta-modeling.

For addressing (***RQ2***) the validity of workflows with respect to the given process meta-model with formally defined semantics from the previous step, we can view the sequence of tasks in the workflow from the perspective of an event sequence instead. Using conformance checking, we can determine the fitness[2] of how well this sequence of tasks matches the behavior allowed by the process model. Therefore, valid workflow (models) are only those that fit the process (meta-)model perfectly with a fitness value of 1.

[2]The other quality criteria are not relevant for the purposes of this work, since we are only interested whether the observed behavior matches the allowed behavior.

For addressing (*RQ3*) the recommendation of a “best” valid workflow alternative with respect to a given invalid workflow and the given process model, we can again view the workflow’s sequence of tasks as an event execution sequence instead. Using process enhancement techniques, we can repair this sequence of tasks by finding the most similar sequence of tasks that best conforms to a favorable process model execution sequence. However, in this case, the repaired task sequence must be valid and thus completely align with a model sequence. Therefore, we focus our attention on finding perfectly fitting task sequences only and choose the most similar one to the given workflow’s sequence of tasks.

Upon a closer inspection of possible approaches for repairing such workflows, additional intricacies are revealed which need to be considered. The proposed approaches for the repair of event logs [RSMAW13][WSZL13] consider only the addition of missing tasks. A workflow’s sequence of tasks may already contain all necessary tasks but in an incorrect order. The usefulness of these approaches regarding task transposition needs to be evaluated. If these approaches turn out to be inadequate, we need to extend them or devise new ones. Yet, “if a case does not fit, the [token-based] approach does not create a corresponding path through the model” [Aals16] that we can use for repair. Moreover, footprint comparison algorithms cannot be used, because the sample size of event traces is too small to infer any causal dependencies. Namely, the sample size is 1, since each workflow needs to be treated as an individual case that is independent of other cases. However, with alignment-based approaches “detailed diagnostics can be given per case” [Aals16], based on which we can attempt to build an imperative algorithm that accounts for task transposition properly.

To address the remaining question of workflow repair, we tackle the problem from another, more convenient direction. Van Der Aalst proposes the use of a “genetic algorithm that minimizes the edit distance while ensuring a minimal fitness level” for the repair of process models [Aals16]. Furthermore, we can check the extent to which an event log passes all rules defined in a rule-based process modeling language in order to evaluate its fitness [Aals16]. By combining these ideas, we examine the use of a declarative process model for workflow (meta-) modeling, with which we can validate whether a workflow (model)’s sequence of tasks passes all the rules in such (meta-)model. Furthermore, we can generate such valid workflow (models) and then use an edit distance similarity metric to find the best possible alternative to a given invalid workflow (model). Although “generating the optimal recovery of missing events is [] `NP`-hard” [WSZL13], research into *typical* lab processes [MLBZ16] shows that the state space is small enough not to run into performance concerns. Upon further inspection of the problem domain, we gain additional insights from Grambow, Oberhauser and Reichert’s work for the construction of our declarative process model and modeling elements used therein [GrOR11].

1.5 Glimpse of results & contributions

We answer all the posed research questions by implementing a customized approach, which combines different technologies that suit the needs of service users by being easy to use, as well as being easy to implement and verify for service providers. In particular, we introduce the modeling concepts of *tasks* and *connections* as a domain-specific alternative for both modeling and reasoning on allowed process models.

We evaluated the proposed approach by applying it to the area of failure analysis (FA) in Infineon AG Villach and found our implementation successful in representing and drawing conclusions about concepts and workflows in that domain. To this end, we constructed a workflow meta-model encountered in a *typical* failure analysis department. Using this meta-model, we were able to properly encode ground rules common to 14 elicited template workflows. All these workflows successfully passed workflow validation according to those previously defined constraints.

Despite the base ingredients being well known and researched, we believe the proposed approach is a novel combination of procedures for event log repair in small process state spaces, not only for incomplete but also for incorrect event traces.

During the process of finding such answers, we make the following contributions, in order of significance:

- A customized approach that addresses the particular requirements of workflow management in *typical* production environments, alongside a formal specification thereof.
- A working and tested system implementing the proposed approach[3], with auxiliary materials, such as setup and run instructions, as well as a comprehensive user guide.
- Usage examples of various technologies for addressing not only the problem statement but also for further application areas that may be of interest in the pursuit of Industry 4.0.
- An overview of theoretical foundations of the underlying concepts behind this work so that interested readers may gain a deeper understanding of such concepts.
- Bug reports for two defects in the reference implementation of the W3C Shapes Constraint Language, which were fixed by the authors in the meantime[4].

1.6 Overview

The rest of this work is structured as follows: A basic understanding of the underlying concepts henceforth used are established in Chapter 2. This includes the used terms workflow and tasks, by also exploring available technologies for modeling and reasoning on such processes. Chapter 3 summarizes the (non-exhaustive) set of different approaches, which can be used to solve both parts of the problem statement. In particular, Section 3.1 examines methodologies for (meta-)modeling workflows, whereas the latter Section 3.2 explores approaches for validating and repairing invalid workflows. Afterwards, an appropriate approach that addresses both parts of the problem statement is selected for the complete system implementation. The implementation details of this devised system are presented in Chapter 4. An overview of the system and its modules is followed by more detailed descriptions of each module's implementation. In conclusion, the system's deployment possibilities are examined. Chapter 5 presents the results of this work by giving the rationale for choosing the proposed approach. Next, this work's findings are used to answer the posed research questions. Lessons learned are presented along the way before discussing threats to validity of the proposed approach and limitations of the implementation. Chapter 6 lists the state of the art of alternative process modeling approaches

[3]The source code repository containing the system implementation is available at `https://github.com/mucaho/lottraveler`.

[4]See `https://github.com/TopQuadrant/shacl/issues/111` and `https://github.com/TopQuadrant/shacl/issues/112` for the respective issue details.

not explored in this thesis, and it examines related scientific articles about workflow validation and repair. Chapter 7 concludes this work and states potential future work related to the problem statement, opportunities for refining the presented approach and possible additional evaluations. Furthermore, another potential application area for improving the day-to-day operations in a *typical* plant is presented. Appendix A, B & C provide the system specification, source code listings and a user guide to the implemented system, respectively.

Disclaimer

This work is motivated and inspired by the work of Infineon AG Villach and their failure analysis (FA) department. We had the pleasure to visit the lab and talk with industry specialists about a *typical* layout of a FA lab, the *typical* day-to-day operations and about possible, *typical* automation-based improvements that could be applied in such an environment. We emphasize the word *typical*, because publicly available work has to take special care not to divulge any sensitive information or trade secrets of the company. To combat this, we use *generic* examples and problem statements that are *likely to be encountered* in other labs over the world, or we refer to publicly available works where applicable.

2 Background

Before identifying the problem in a formal way, we establish a basic understanding of the underlying concepts. In the next Section, the previously used terms workflow and task are examined, by also exploring available technologies for modeling such processes. Afterwards, tasks and relationships between tasks are viewed from another perspective in order to describe those concepts with a formally defined meaning. Furthermore, answer set programming is examined in detail, which deals with constraints and preferences at its core. The theoretical background is concluded by looking at domain-specific modeling, which explains the relationship between process models and process instances.

2.1 Business process management

Although the theory behind business process management, workflows and related technologies is vast, we focus on highlighting important topics that are used in artifacts of this work.

2.1.1 Definitions

The Workflow Management Coalition[1] defines a workflow as a "computerized facilitation or automation of a business process, in whole or part" [Holl95]. A business process is composed of several tasks, the "logical unit[s] of work" [AaHe04]. These units of work are atomic and thus either fully completed or rolled back on partial completion [AaHe04].

Each business process has (possibly multiple) start and end states. The end states are reachable by a subset of all the different task paths that originate from the start states. The particular routing and ordering of these tasks depends mostly[2] on the gateways present in such flow-based process model [Wesk12]. The layout of tasks, gateways and joining connections thus induces the control-flow of the business process, the aspect of workflows we deal with in this work.

Besides these execution constraints on tasks [Wesk12], a business process can be viewed from other perspectives, e.g. in terms of the data dependencies between tasks or e.g. in terms of the usage of possibly limited resources. Furthermore, the business process can be viewed from the outside as a black-box and how it interacts with other business processes in a process choreography, in contrast to viewing the internals of a single business process in a white-box-based process orchestration [Wesk12].

[1]The WfMC is a "global organization of adopters, developers, consultants, analysts, as well as university and research groups engaged in workflow and BPM Workflow", see `https://www.wfmc.org/` for more details.

[2]For simplicity's sake, we disregard exception handling, external & intermediate events and other constructs that impact the regular control flow inside a business process.

2.1.2 Business Process Model and Notation

The Business Process Model and Notation (hereinafter BPMN) serves a similar purpose for business process modeling, as the Unified Modeling Language serves for object-oriented design [Wesk12]. It's a notation intended to be used and understood by the whole spectrum of stakeholders involved in a business process, be it business managers or technical developers [GRot14]. BPMN allows the representation of both process orchestration and choreography. Its development is steered by The Object Management Group[3].

As BPMN "has a familiar and easy basic graphical notation" [WaSi06], the particularities of BPMN are not described further. If needed, the reader is advised to familiarize themselves with the basics of this notation by, e.g., taking a look at the specification [GRot14].

2.1.3 Case Management Model and Notation

The Case Management Model and Notation (hereinafter CMMN) serves a similar purpose for case management, as BPMN serves for business process management [KSFL15]. Case management offers additional flexibility compared to standard business process management [MaHM16], where the process is well-structured and statically defined in advance [KSFL15]. A case worker can, based on the particular case information they are working with, decide during run-time which tasks to perform and can make ad hoc adaptations by e.g. adding additional tasks in exceptional circumstances [KSFL15]. CMMN's development is steered by The Object Management Group[3].

As only "24% of all elements in CMMN are in 1:1 correspondence between [perceived] semantic constructs and graphical symbols" [KPHJ19], the most important concepts relevant to our work are presented in the following paragraphs. For more details, refer to the official specification [GRot16].

A CMMN case plan model consists of multiple elements called plan items, which describe the business process. The two main plan items are tasks and stages. Task are differentiated by their type, e.g. human tasks are handled by human actors, or e.g. process tasks call other business processes. Tasks can be nested inside one or more containers called stages. Plan items may be repeated by marking them as repeatable (#).

Plan items go through different lifecycle phases / states. Unless a task or stage is marked with manual activation (▷), it becomes automatically `active` if it has no entry criteria (◊). Otherwise, these elements become `activated` by their entry criteria when the required lifecycle transition events are received from connected elements. Once a plan item becomes `active` it must be `completed` or `terminated`[4]. On the other hand, tasks with manual activation do not automatically become `active`. They can either be `activated` or `disabled` by the user[4] once their entry criteria are fulfilled. `Activated` tasks or stages may be manually `terminated` by the user, or by satisfying the exit criteria (⧫) when the required lifecycle transition events are received from connected elements. Elements that are in a `disabled`, `completed` or `terminated` states are considered to be in a (semi-)terminal state. Furthermore, a stage is automatically `completed` if all contained elements are in a (semi-)terminal state[5], which necessarily means

[3]See `https://www.omg.org/` for more details.
[4]The lifecycle of plan items is more nuanced than that involving multiple other states and state transition. However, for clarity, we describe only a relevant part of the greater picture.

that no `active` children are left. In turn, the whole plan model instance is `completed` once all contained elements are in a (semi-)terminal state.

2.1.4 Workflow nets

We introduce workflow nets for the application area for verification, as well as for construction of workflows.

Workflow verification

A business process model can be analyzed for its conformance or performance properties [HeSW13]. The former verifies whether the process conforms to the desired business rules (qualitative analysis), while the latter produces indicators by simulating the given model (quantitative analysis) [HeSW13]. In the following paragraphs, we introduce relevant qualitative aspects of a business process.

In particular, by checking the structure of a Petri net (PN), we can verify its correctness as well as other characteristics of a process definition [AaHe04]. A Petri net is a graph consisting of a finite set of (circular) nodes called *places* P, a finite set of (rectangular) nodes named *transitions* T with $P \cap T = \emptyset$, as well as directed edges connecting those nodes called *arcs* $F \subseteq (P \times T) \cup (T \times P)$. Petri nets can also contain (black) tokens inside places; the marking M describes the current state of the PN, by mapping each place to the number of tokens it contains $M \equiv P \to \mathbb{N}$.

Petri nets have an active component to them —they can change their state by firing enabled transitions. A transition $t \in T$ is said to be enabled iff all input places of t (connected by a directed edge towards this transition) contain at least one token, that is $\forall p \in P, (p,t) \in F, \exists\ p \mapsto m \in M : m > 0$. This enabled transition $t \in T$ may then fire, consuming one token from each input place and producing one token in each output place (connected by a directed edge from this transition), that is $\forall p \in P, (p,t) \in F, p \mapsto m \in M : m' = m - 1$ and $\forall p \in P, (t,p) \in F, p \mapsto m \in M : m' = m + 1$. For brevity, we denote the firing of the transition $t \in T$ by $M \xrightarrow{t} M'$ and the firing sequence of $0..n$ subsequent transitions as $M_i \xrightarrow{*} M_n$. If the latter firing sequence is possible, M_n is said to be reachable from M_i. The reachability graph is constructed by using these markings M as nodes and connecting them via directed edges to markings M' obtained by firing the appropriate transitions $M \xrightarrow{t} M'$. The set of all markings reachable from the start marking M_i is denoted as $R(M_i) \equiv \{M_n \mid M_i \xrightarrow{*} M_n\}$, and the set of all possible firing sequences from the start marking as $L(M_i) \equiv \{t \in T \mid M_i \xrightarrow{*} M \xrightarrow{t} M'\}$ [ZuZh94].

A PN is said to be k-bounded iff for every reachable state from the initial state the maximum amount of tokens in each place p is k, that is $\forall_n\ M_i \xrightarrow{*} M_n, p \mapsto m \in M_n : m \leq k$. A PN is safe iff it is 1-bounded. A PN is live iff for every reachable state M from the initial state any transition t can be enabled again. That is, $\forall M \in L(M_0), \forall t \in T, \exists M' \in L(M_0) : M \xrightarrow{t} M'$. A PN is dead if there is at least a single transition $\exists t \in T$ such that t is never fired in the set of all possible firing sequences from the start marking $L(M_i)$. A PN consists of potentially fireable transitions if for all transitions $\forall t \in T$ the transition t is fired at least once in $L(M_i)$. In conclusion, for a non-dead PN with potentially fireable transitions, the following holds: $\forall t \in T, \exists M, M' : M_i \xrightarrow{*} M \xrightarrow{t} M'$.

[5]This is true for stages that are not marked as auto complete (■). Stages marked as auto complete are `completed` automatically once just their required tasks (marked with **!**) are in a (semi-)terminal state

A workflow net (WFN) is a Petri net (P, T, F)[6] with additional constraints [AaHe04]:

- There is a single token source place $i \in P$ with no incoming arcs $\nexists t \in T : (t, i) \in F$, which marks the start of the workflow.
- There is a single token sink place $o \in P$ with no outgoing arcs $\nexists t \in T : (o, t) \in F$, which marks the end of the workflow.
- Every node $\forall n \in P \cup T,\ n \neq i,\ n \neq o$ is on a path from the source place i to the sink place o, namely $(i, n) \in F^*$ and $(n, o) \in F^*$ with F^* denoting the reflexive transitive closure of F [AHHS+11].

We conclude the quality aspects of PNs / WFNs used to model business processes by introducing the notion of soundness. "Soundness is a general purpose sanity check for workflows" [HeSW13]. A sound workflow is guaranteed to be able to come to its conclusion eventually, while also not having any dangling activity control-flow paths that lead nowhere. Soundness is thus arguably the most important property to keep in mind when designing business processes. Formally, a WFN is sound under the following conditions [AaHe04]:

- Eventual end: From any possible state (marking) in a WFN M there exists a firing sequence that ends in the sink state o, that is $\forall M \in L(M_i) : M \xrightarrow{*} M_o$, with M_o being the end state.
- Clean finish: The end state M_o consists of a single token only in the sink place o. Moreover, no other states may contain any tokens in the sink place o. That is, $\forall M \in L(M_i), M \neq M_o, o \mapsto m \in M : m = 0$.
- Not dead: The WFN is non-dead and contains potentially fireable transitions.

Alternately, a finished WFN must be repeatable infinitely without accumulating (leaking) any tokens or improperly disabling any transitions after their repeated use. This can be verified by connecting the WFN's sink place o back to they initial source place i via a transition $t*$ —$T' = T \cup \{t*\}$, $F' = F \cup \{(o, t*), (t*, i)\}$. Iff the thus obtained Petri net is properly repeatable by being live and bounded, then the original workflow net is sound [AHHS+11].

Workflow construction

A business process can be modeled as a WFN using two different paradigms [HeSW13]. The difference becomes clearer when comparing these modeling paradigms using an example `order` process. In the first approach, business process actions, like `check stock`, are modeled as transitions of the workflow net. Places then represent the passive states of the business process. For example after `checking stock` the workflow would either be in an `in stock` or `out of stock` state. Conversely, using the latter paradigm, business process actions are depicted as places of the workflow net, whereas the enclosing transitions indicate the start or end of these activities. By firing the `begin delivery` transition the workflow is put in a `delivering goods` intermediary state, which is resolved after firing the `finish delivery` transition. Other places and transitions that do not portray these ongoing activities still represent regular states and state transitions respectively.

[6] For the purposes of this work, we do not use nor introduce the notion of an extended Petri net. Strictly speaking, workflow nets are an extension of extended Petri nets [AHHS+11].

In addition to the aforementioned paradigms for constructing workflow nets manually from scratch by a domain expert, we can apply automatic methods to derive similar workflows. The process of expanding an existing WFN is called workflow refinement; a variety of refinement rules are given in van Hee et al's work [HeSW13]. On the other hand, a more detailed workflow can be converted to a more abstract one using workflow reduction, with some of the reduction rules being presented in Van Der Aalst et al's work [AHHS$^+$11]. In both approaches, various workflow properties (like soundness) are preserved by these workflow transformations.

2.2 Mathematical logic

This Section introduces basic concepts related to mathematical logic. It sets up the proper terminology in order to understand the fundamentals of how ontologies (see Section 2.3 and answer set programming (see Section 2.4) work.

2.2.1 Model-theoretic view

In this formally correct viewpoint on logic, a so-called *knowledge base* KB is composed of fundamental logical building blocks referred to as *propositions* p [HiKR09]. All identifiers and (literal) values from these propositions make up the *vocabulary* V. An *interpretation* I maps the elements of such vocabulary to different sets (referred to as interpretation *domain* $\triangle^I$ [HoKS06]) and imposes certain conditions on and between those sets. This interpretation function (also known as *valuation* $\cdot^I$ [HoKS06]) is then applied on every proposition p, and the result p^I is evaluated against the aforementioned conditions. If these conditions are *satisfied* for a proposition under such interpretation p^I, the interpretation is said to be a *model* of the proposition $I \models p$ [HiKR09]. Moreover, if these conditions hold for every proposition of the knowledge base under such interpretation KB^I, the interpretation is said to be *correct* and a *model* of the knowledge base $I \models KB$ [HiKR09]. However, as an empty, nonsensical interpretation could be considered a model, we are only interested in interpretations that are “truthful representation[s]” of the knowledge base [HiKR09]. A knowledge base KB then *entails* another knowledge base $KB \models KB'$, iff every model of the original $M \models KB$ is also a model of the entailed one $M \models KB'$ [HiKR09].

2.2.2 Inference rules

Instead of having to reason in terms of all possible interpretations and their correctness when looked at via the model-theoretic view, *inference* rules can be used to *syntactically* obtain such additional knowledge [HiKR09]. In particular, these rules deduce additional propositions (*consequents*) in the presence of given propositions (*antecedents*) $p_1 \wedge p_2 \wedge ... \wedge p_n \rightarrow p'$, with $P \vdash P'$ denoting the set of all inferred propositions obtained via applying a combination of such deduction rules on the set of original propositions [HiKR09]. These deductions are *sound* if the *syntactic* consequence also implies the *semantic* consequence $P \vdash P' \Rightarrow KB \vDash KB'$, that is, the set of original and deduced propositions are also models of the original and entailed knowledge base, respectively. On the other hand, they are *complete* if every *entailment* is also found as a *deduction* $KB \vDash KB' \Rightarrow P \vdash P'$ [HiKR09].

2.2.3 Propositional logic

Before diving into more advanced logic, some fundamental logic concepts are introduced.

Propositional logic, as well as other logics introduced in the following subsections, are all part of *classical logic*, in which we model the world along two truth values —something is or something is not, with no in-between [Fitt12]. There are other logics, like *fuzzy logic*, where we for example model the degree of truthfulness along a $[0, 1]$ dimension.

Syntax

In propositional logic, the basic building blocks, as introduced in Section 2.2.1, are *propositional letters* $P, Q, ...$ [Fitt12]. Propositions combined using *propositional connectives* then build up propositional *formulae* F [Fitt12]. These connectives are either nullary (also referred to as *constants*), unary, binary or so-on-arity connectives, with $\bot$ / $\top$, $\neg$, $\vee$ / $\wedge$ / $\rightarrow$ / $\leftarrow$ / $\leftrightarrow$ being the most commonly used representatives, respectively [Fitt12]. In this case, *bottom* $\bot$ refers to falsehood $false$, whereas *top* $\top$ denotes the truth *true* [Fitt12]. In the most trivial case, a single propositional letter constructs an *atomic* propositional formula $F \equiv P$. More advanced formulae can include any valid combination of letters and connectives, like $F \equiv (P \vee Q) \wedge (R \vee S) \rightarrow \neg T$. *Literals* are a special form of extended, atomic formulas which contain a constant or a (possibly negated) propositional letter $F \equiv \neg P$ only [Fitt12]. A set of formulas $\{F_1, ..., F_n\}$ is also known as a *knowledge base* KB [HiKR09].

Semantics

We can regard the set of possible connectives and letters (the set of formulae) as the vocabulary V_p of a propositional logic, as introduced in the model-theoretic view on formal logic in Subsection 2.2.1. Now an interpretation I maps those elements into the interpretation domain $\triangle^I$ using the interpretation valuation $\cdot^I$ by mapping every formula F into its value under that interpretation F^I. In propositional logic, logicians have agreed to consider interpretations only that make sense, thus *Boolean interpretations* B use the *Boolean domain* $\triangle^B = \{true, false\}$ and the *Boolean valuation* $\cdot^B$, defined by $\cdot^B(\top) = true$, $\cdot^B(\bot) = false$, $\cdot^B(\neg P) = \neg \ \cdot^B(P)$ and $\cdot^B(P \circ Q) = \cdot^B(P) \circ \cdot^B(Q)$, with $\neg$ inverting the truth value and the binary connectives $\circ$ mapping the truth value according to *truth tables* commonly found in literature [Fitt12]. As the Boolean valuation is strictly defined for these connectives, the only dynamic part of such an interpretation is the mapping of letters to their truth value $\cdot^B(P) = true$ / $\cdot^B(P) = false$. With this final stone in place, each formula F can then be evaluated under such Boolean interpretation F^B by "recursively" applying the valuation for the various connectives encountered in such formula.

If a Boolean interpretation B evaluates to $F^B \rightsquigarrow true$ for a given formula F, the formula is said to be *satisfiable* [Fitt12] and the Boolean interpretation B is thus a *model* $B \models F$ of this formula F. Moreover, if a formula F can be satisfied by every Boolean interpretation B, the formula is said to be a *tautology* $\models F$ [Fitt12] —always *true*. Conversely, if a formula F can never be satisfied by any Boolean interpretation B, the formula is said to be a *contradiction* or also *unsatisfiable* $\models \neg F$ [HiKR09] —always $false$. Similar to singular formulas, the whole knowledge base is *satisfiable* $B \models KB$ if there exists a Boolean interpretation B such that the knowledge base under this interpretation evaluates to $KB^B \rightsquigarrow true$, which in turn means every formula of the knowledge base evaluates to $F_i^B \rightsquigarrow true$ under such interpretation B [Fitt12].

Reasoning

Analog to the motivation of using inference rules in Subsection 2.2.2, we would like to find an interpretation that satisfies our knowledge base without having to "brute-force" search for such an interpretation by going through all possible interpretations and checking their satisfiability.

Thankfully, several such *proof procedures* exist that are both sound and complete, with the most applicable for machine processing being the propositional *tableux* and propositional *resolution* method [Fitt12]. The particularities of these *refutation* systems are out-of-scope of this work and thus not further explored in detail, hence the interested reader is referred to appropriate literature, e.g. Fitting's book [Fitt12]. Propositional logic is decidable [HiKR09], that is we can find a "truthy" interpretation (or also a tautology [Fitt12]) in finite amount of operation steps.

2.2.4 First-order logic

Although propositional logic is simple and easy to understand, it lacks the expressive power to state certain things about the world. One prominent example is the lack of *quantifiers*; we cannot say that a logical relation holds for some or all individuals of the world that meet some criteria. *First-order logic* allows one to express such relations, in spirit similar to working with existentially and universally qualified restrictions encountered in OWL (see Subsection 2.3.2).

Syntax

Hence, first-order logic augments the syntax of propositional logic with *quantifiers* $\forall$ / $\exists$, *variables* (e.g. X, Y, ...), *function symbols* of various arity (e.g. f_0, $f_1(t_a)$, $f_2(t_a, t_b)$, ...), as well as *predicate symbols* of various arity (e.g. p_0, $p_1(t_a)$, $p_2(t_a, t_b)$, ...). Nullary function symbols f_0 are also referred to as first-order logic *constants* and are more commonly written as a, b, ... [Fitt12]. Furthermore, zero-arity *predicates* resemble propositional letters [HiKR09], thus promoting first-order logic as a true super-set of propositional logic. A first-order logic *term* consists of either variables or function symbols that can in turn only contain terms themselves in such a recursive fashion, e.g. a, X, $f(a, X), f(g(a, X), h(b, y))$ [Fitt12]. A *closed* term contains no variables [Fitt12]. Similar to the definition in propositional logic, an *atomic* first-order logic *formula* consists of a single predicate, which must contain only terms itself, e.g. $p(a, X, f(g(b), y))$ [Fitt12]. All other *formulae* can then be obtained by applying all valid combinations of boolean connectives to atomic formulae, as well as *binding* a universal quantifier $(\forall X)F$ or an existential quantifier $(\exists X)F$ to a formula [Fitt12]. One such formula is e.g. $F \equiv \bot \vee \{\forall X[\forall Y(p(X, Y) \wedge \ \neq (X, Y))]\} \rightarrow \{\exists Z[p(Y, Z) \wedge \ \neq (Y, Z) \ \wedge \ p(X, Z)]\}$ with $\neq$ being a binary function symbol. Building upon the notion closed terms, a *closed* formula (also called *sentence*) is a formula with all variables being bound by quantifiers that encompass them [Fitt12]. If a variable is not bound by a quantifier, it is said to be a *free* variable [Fitt12]. In conclusion, a first-order *knowledge base* is made up of a set of sentences [HiKR09].

Semantics

As the vocabulary and thus the set of possible formula expressions of first-order logic V_f is more expressive than the propositional case, an interpretation I for first-order logic is more complex. In particular, the interpretation valuation $\cdot^I$ maps [Fitt12]:

- every n-ary function symbol f_n to a n-ary function over the domain $\cdot^I(f_n) : \triangle^{I^n} \rightarrow \triangle^I$
- as a special case of the former, every constant c to an element of the domain $\cdot^I(c) = c^I \in \triangle^I$

- every n-ary predicate symbol P_n to a n-ary predicate over the domain $\cdot^I(P_n) \subseteq \triangle^{I^n}$

Additionally, as formulae can contain variables, we need an *assignment* Z of variables X to elements of the domain $\cdot^Z(X) = X^Z \in \triangle^Z$ [HiKR09].

As in the propositional case, we are interested in the Boolean interpretation B of first-order formulae with truth values in the Boolean domain $\triangle^B = \{true, false\}$. However, without further modification of this domain, a variable assignment Z cannot simultaneously be used for constructing proper formulae through the *substitution* of variables with elements of this limited domain [Fitt12]. Therefore, we consider the domain of variable assignment $\triangle^Z$ to be the set of closed terms. The combined valuation is predefined in terms of the Boolean valuation for such closed terms $\cdot^{B,Z}(t) = \cdot^B(t)$, and can be obtained by applying the Boolean valuation on variables substituted with closed terms in the other case $\cdot^{B,Z}(t) = \cdot^B(tZ)$. Such an interpretation involving closed terms is called a *Herbrand interpretation* [Fitt12] and will be further explored in Section 2.4 on answer set programming.

Using the Boolean valuation $\cdot^B$ and a variable assignment $\cdot^Z$ the truth values can thus be determined "recursively" by [Fitt12][HiKR09]:

- $X^{B,Z} = X^Z$ for variable terms
- $f_n(t_1, ..., t_n)^{B,Z} = {f_n}^B({t_1}^{B,Z}, ..., {t_n}^{B,Z})$ for function symbol terms
- therefore, trivially, $c^{B,Z} = c^B$ for constant terms
- $P_n(t_1, ..., t_n)^{B,Z} = true \Leftrightarrow ({t_1}^{B,Z}, ..., {t_n}^{B,Z}) \in {P_n}^B$ for atomic formulae consisting of a predicate symbol
- $\bot^{B,Z} = false$ and $\top^{B,Z} = true$ for formulae with nullary connectives
- $(\neg F)^{B,Z} = \neg F^{B,Z}$ for formulae with unary connectives
- $(F_i \circ F_j)^{B,Z} = {F_i}^{B,Z} \circ {F_j}^{B,Z}$ for formulae with binary connectives
- $[(\forall X)F]^{B,Z} = true \Leftrightarrow F^{B,(Z \cup X \mapsto d)} = true$, for all $d \in \triangle^Z$
- $[(\exists X)F]^{B,Z} = true \Leftrightarrow F^{B,(Z \cup X \mapsto d)} = true$, for at least one $d \in \triangle^Z$

The prominent dynamic part of different Boolean interpretations is the mapping of predicates to truth values, like e.g. $\cdot^B(person(john)) = true \vee false$, $\cdot^B(person(mary)) = true \vee false$, $\cdot^B(married(mary, john)) = true \vee false$. Constants, like e.g. *mary* and *john* are assumed to be always *true* and function symbols, like e.g. $\neq$, have predefined truth values depending on their input. In the case of Herbrand interpretations, variable assignments map back to the set of closed terms, and therefore the truth values of more complex formulae can be obtained by "recursively" following the above definitions.

Similar to propositional logic, an interpretation B with variable assignment Z is a *model* $B, Z \models F$ of formula F if the formula evaluates to $F^{B,Z} \rightsquigarrow true$ under this interpretation B for these variable assignments Z [HiKR09]. If a formula is closed, the variable assignment has no impact on the correctness of the formula, and is thus denoted as $B \models F$ [HiKR09]. Just like in propositional logic, a formula can be a *tautology* $\models F$ or a *contradiction* $\models \neg F$ if the formula evaluates to $F^I \rightsquigarrow true$ or $F^I \rightsquigarrow false$, respectively, regardless of the interpretation used. A knowledge base KB consisting of closed formulas is *satisfiable* $B, Z \models KB$ if there exists a satisfying interpretation B and satisfying assignment Z for which all contained sentences evaluate to ${F_i}^{B,Z} \rightsquigarrow \mathit{true}$ [Fitt12].

Reasoning

The *refutation* proof systems by *tableux* or *resolution*, introduced in the reasoning paragraph on propositional logic, have to be extended to handle universal and existential quantification. It suffices to prove only closed formulas (sentences), as "the validity of arbitrary formulas can be reduced to that of sentences" [Fitt12]. Implementations of tableux and resolution systems work with free variables which are introduced instead of proper substitutions during the processing of universally- / existentially-bound formulae. *Unification algorithms* then take care of finding proper substitutions between those variables that would make them identical [Fitt12]. However, even with the use of free variables and unification algorithms, first-order logic is semi-decidable [HiKR09], that is, these algorithms are guaranteed to terminate only if there exists such *unifier* that makes those variables identical. The interested reader is again referred to appropriate literature, e.g. Fitting's book [Fitt12], for additional details on these procedures in the context of first-order logic.

2.2.5 Decidability

What does it mean for a logic to be *decidable*? It is the ability for any possible (already devised or not yet known) automated procedure to find a correct interpretation that holds for all propositions in the given logical expression $M \models KB$ within a finite amount of operational steps [BoGG01].

However, a logic can be *semi-decidable*, that means that given that one logical expression semantically entails another one $KB \models KB'$, the set of respective syntactic deduction rules $P \vdash P'$ will eventually terminate. If the entailment does not hold $KB \not\models KB'$ the respective inference rules $P \vdash P'$ may or may not terminate [HiKR09].

These properties are "the central problem of mathematical logic" [BoGG01], alongside computational time and space complexities.

2.2.6 Complexity classes

What do these time complexities mean, that typically appear alongside an analysis of the expressive power of a language? They denote complexity classes, in particular, in the case of time complexity, how many computational steps it takes at most for an abstract mathematical model (*Turing machine*), which can simulate all possible computational procedures "with little loss of efficiency" [ArBa09], to find a correct interpretation that holds for all propositions. Conversely, in the case of space complexity, we are interested in the upper bounds of the memory consumption of such a *Turing machine* [ArBa09]. Usually, the time complexity matters more, since a Turing machine can only access a single memory location at a time [ArBa09]. However, in the case of less space required than the input length, it is still noteworthy to mention such *sub-linear* complexity [ArBa09].

When looking at the different complexities we further distinguish between *deterministic* and *non-deterministic* Turing machines. In the latter case, we observe a complexity given that the machine's operation procedure can arbitrarily branch using the right choice amidst multiple choices [ArBa09].

A computational problem can also be considered *hard* or *complete* for a complexity class [ArBa09]. A problem is said to be *hard* for a complexity class if every other problem in the given complexity class can be reduced (converted) to it with "roughly" the same amount of time / space as problems from the given complexity. A problem is said to be *complete* for a complexity class if it is a *hard* problem for that complexity class and is itself member of that complexity class. Examples are given in the following paragraphs.

A prominent complexity class is `PTIME` (often abbreviated to just `P`) and represents the class of problems that is "efficiently *solvable*" for a deterministic Turing machine in at most polynomial number of steps to the input length $O(n^c)$ [ArBa09]. With *solving* we mean finding a or no e.g. correct interpretation in the given time frame. For clarification, in the previous notation O represents the upper-bound of operational steps as a function on the amount of e.g. propositions n, while disregarding any constant, non-input dependent factors in the exact calculation of such amount [ArBa09]. The constant c denotes e.g. the degree of the polynomial function used in the inner-most "loop" for finding such interpretation. Such `P` problems L are then `P`-complete if every other problem solved in "roughly" polynomial time L_i can be reduced to them in polynomial time, denoted as $L_i \leq_p L$.

Another prominent complexity class is `NPTIME` (often abbreviated to just `NP`) and represents the class of problems that is "efficiently *verifiable*" for a deterministic Turing machine in polynomial time $O(n^c)$[ArBa09]. In this case, *verification* refers to the ability to determine the correctness of e.g. a given interpretation in the given time frame. In addition, `NPTIME` can equivalently be understood to be "efficiently solvable" by a non-deterministic Turing machine in polynomial time [ArBa09]. Do note the distinction on determinism of the Turing machine used in previous statements.

Similarly, the complexity class `N2EXPTIME`[7] can be solved by a non-deterministic Turing machine within at most $O(2^{2^{n^c}})$ computational steps.

As a seemingly complementary problem class, `co-NPTIME`, or succinctly written as `co-NP`, represents the class of problems that is efficiently *disqualifiable* for a deterministic Turing machine in polynomial time [Papa94]. In this case, *disqualification* refers to the ability to determine the unsuitability of e.g. a given interpretation in the given time frame [Papa94]. In addition, a problem in `co-NPTIME` can equivalently be understood as being efficiently provable to not have a "positive" solution by non-deterministic Turing machine in polynomial time [ArBa09].

As the name suggests, `NLSPACE` problems, commonly referred to as `NL`, require only logarithmically as much memory as the problem's input size $O(log_c\ n)$, given that the Turing machine is non-deterministic. Intuitively, this can be imagined as the amount of "bits" required to save an index for the currently read memory location of the input sequence [ArBa09]. Note that, analogous to time complexities, such `NL` problems L are then `NL`-complete if every other problem solved with "roughly" logarithmic space L_i can be reduced to them using logarithmic space, denoted as $L_i \leq_l L$.

2.2.7 Types of computational problems

Reasoners, among other algorithm implementations, can be used to solve different types of computational problems:

[7]In `N2EXPTIME` the number 2 before `EXP` can be thought of as capturing the degree of "exponentiality".

- *Decision problems* determine whether a boolean property $\mathcal{Q}$ holds for a given input x [DuKo11].

  $$DP(x) = \begin{cases} \text{"Yes"} & \text{if } \mathcal{Q}(x) \\ \text{"No"} & \text{otherwise} \end{cases}$$

- *Search problems* find a suitable output solution y that satisfies a boolean relation $\mathcal{R}$ for given input x [DuKo11].

  $$SP(x) = \begin{cases} y & \text{if } \exists y : \mathcal{R}(x,y) \\ \text{"No"} & \text{otherwise} \end{cases}$$

- *Optimization problems* search for the optimal output solution y that satisfies a boolean relation $\mathcal{R}$ for given input x. The optimal solution y has the smallest (or highest) *value* $v(y)$ among all other solutions [DuKo11].

  $$OP(x) = \begin{cases} y & \text{if } \exists y : \mathcal{R}(x,y) \land y = \arg\min v \\ \text{"No"} & \text{otherwise} \end{cases}$$

- *Counting problems* count the number of output solutions y that satisfy a boolean relation $\mathcal{R}$ for the given input x [DuKo11].

  $$CP(x) = |\{y \mid \mathcal{R}(x,y)\}|$$

2.3 Ontologies and related concepts

This Section gives an introduction to ontologies, as well as related concepts and technologies surrounding ontologies, which are used throughout artifacts of this work.

2.3.1 Definitions

"An ontology is a description of knowledge about a domain of interest, the core of which is a machine-processable specification with a formally defined meaning" [HiKR09]. It is a type of taxonomy depicting a hierarchy of classes, individuals and their relationships between them [ArSS15], for the observed universe of discourse. Ontologies do not only describe a domain of interest, but can also be used to infer additional knowledge from the already available knowledge [HiKR09], or on the other hand, to detect logical inconsistencies in their specification [Usch18].

Although in this work we use an ontology to represent a small business domain, the application area of ontologies is broader in scope. Semantic Web[8], as an extension of the World Wide Web, deals with distributed and linked data [HeGA20]. The goal of the semantic web is to systematically connect data with metadata across different resource locations, such that this data and the (possibly inferred) relationships can be shared and reused "across application, enterprise and community boundaries" [ZhDP09]. Similar to the World Wide Web, "anyone

[8]The Semantic Web and related technologies are being developed by the Wide Web Consortium (W3C), "an international community where Member organizations, a full-time staff, and the public work together to develop Web standards". See `https://www.w3.org/2013/data/` for more details.

can say anything about any topic" [HeGA20], thus the underlying technologies are challenged to support conflicting or incomplete information about entities (*open world assumption*), which may additionally be named differently (*non-unique naming*).

Concepts represented in an ontology can be modeled using the following building blocks and their defined meaning [Usch18]:

- Kinds of entities
- Generalization and specialization hierarchies among those kinds
- Entity individuals
- The instance-of relation, which connects individuals to (possibly multiple) kinds
- Relationships between individuals
- Relationships between individuals and literals

2.3.2 Ontology representation languages

The Web Ontology Language (OWL) builds upon the Resource Description Framework (RDF) and Resource Description Framework Schema (RDFS) as the basic representation language for modeling ontologies [HeGA20].

Resource Description Framework

Using RDF, instead of a rigid, relational database-like structure, data is represented as a directed graph, with nodes being connected by directed edges [HiKR09]. Both nodes and edges are labeled with a unique identifier.

As the graph is typically sparse when it comes to the amount of relationships that exist between entities [HiKR09], graphs are viewed as a set of edges called *triples*. Each triple consists of the *subject* node, the *object* node and the directed *predicate* edge connecting them [HiKR09]. The set of these triples is thus called an RDF *graph* and is also known as an RDF *document*, given that these triples reside in the same file. Subjects and objects can refer to proper, named "objects of interest" called *resources*, in the case of objects to *literal* values (such as a string), as well as to "helper resources with merely a structural function" called *blank nodes* [HiKR09]. These structural helpers are needed when trying to model e.g. n-ary relationships between resources [HiKR09].

To uniquely identify a resource on a global web with distributed, linked RDF documents, a Unique Resource Identifier (URI) is used. Moreover, it is considered good practice for the URI to be a Uniform Resource Locator (URL) that can be opened in a web browser [HeGA20]. On the other hand, blank nodes must be unique inside the current RDF document only, and are thus given a random identifier.

RDF also provides a standard vocabulary, that facilitates a way of giving meaning to (some) data. This vocabulary is small and offers the possibility to say that a resource is an *instance-of* another resource, to declare that a resource should be treated as a *property* (predicate relating type instances), as well as offering a few collection types[9] [HeGA20].

[9]See the respective W3C recommendation [Coot14c] for a comprehensive list of available `rdf:` identifiers.

There exist multiple serialization formats for storing these triples, like RDF/XML, N-Triples, JSON-LD and Turtle [HeGA20]. For the purposes of this work we will primarily use the Turtle format for displaying triple statements. Whereas the N-Triples format stores the triples directly, with each line consisting of fully-qualified identifiers representing the subject, predicate and object, the Turtle format abbreviates these identifiers through the use of URI prefixes [HeGA20]. Moreover, Turtle allows concisely defining multiple statements that share parts of previous statements. This syntactic conciseness is topped off with a special syntax for blank nodes. An example showcasing the intuitiveness of these constructs is given in source Listing 3.1. For further explanation and examples, the reader is advised to take a look at the W3C recommendation [Coot14b].

Resource Description Framework Schema

The RDF schema is an extension to RDF and offers “mechanisms for describing groups of related resources [...] and of the relationships between them” [ArSS15].

The RDF schema is similar to a database schema and how it describes columns with their allowed datatypes and constraints [HeGA20], as well as expressing specialization relationships over (multiple) tables. In contrast to RDF’s loose, graph-based representation of data, RDFS brings structure to this graph. It treats sets of subjects, predicates or objects as rows in imaginary database columns and defines constraints that must thus hold for all of those elements. These constraints are called type assertions or schema assertions [ArSS15].

RDFS offers an extended standard vocabulary, which can be used to declare such assertions. In particular, these assertions[10] state that [HiKR09]:

- *class*: a resource should be treated as a type
- *instance*: a resource should be treated as a representative of a class
- *property*: a resource should be treated as a predicate relating instances, or instances to literals
- *subClass*: all instances of a class are *also* instances of another super-type class
- *subProperty*: all instances related via a property are *also* related via another super-type property
- *domain* & *range*: instances related via a property are *also* instances of the specific domain or range class
- *literal* & *datatype*: a resource should be treated as a literal value of a datatype

The word *also* is emphasized on purpose, because the RDFS vocabulary enables the inference of additional triples based on the existing ones. This inference is highlighted in Subsection 2.3.3.

Web Ontology Language

OWL is in turn an extension of RDFS and offers more expressive means to make more refined statements and “represent more complex knowledge” about a domain of interest [HiKR09].

Extending the previous comparison between RDF schema and a database schema, we now describe additional properties that can be asserted on data using OWL constructs. As ontologies

[10]See the respective W3C recommendation [Coot14c] for a comprehensive list of available `rdfs:` identifiers.

use *non-unique naming* and thus two entities with different names may refer to the same thing, specifying the uniqueness of these imaginary primary keys requires additional, formally-defined semantic assertions [ArSS15]. Moreover, as in conventional database design using the third normal form [LiTK81], ontologies may want to assert a functional dependency of non-prime columns on primary column(s). Furthermore, OWL can be used explicitly list all known entities in an otherwise *open-ended world* of possible entities, which has further implications on the thus resulting reasoning [HiKR09]. Going back to the limitation of n-ary relationship modeling as discussed in Subsection 2.3.2, OWL now facilitates the means to represent n-ary relationships in a structured way using *relation reification* [Usch18]. Similar to how in traditional database design we represent n-ary relations in separate tables that point to all related entities [Camp02], in OWL we can declare an (anonymous) individual to be a representative of a class whose *class expression* is composed of qualified *restrictions* on related entities. The emphasized concepts are explained in ensuing paragraphs, whereas multiple implementation variants are shown in the respective W3C working group note [Coot06a].

OWL furthermore supports meta-modeling, "the practice of using a model to describe another model as an instance" [HeGA20]. In RDF, one may create instantiation (*instance-of*) hierarchies, whereas with RDFS one can create subsumption (*is-a*) taxonomies [HiKR09]. However, OWL uses *punning* to semantically distinguish between these two different views of a single resource —the same resource occurring in a subject position of an *instance-of* or *is-a* triple is viewed from an instance perspective, whereas a resource occurring in the object position of such triple is viewed from a class perspective [HiKR09]. Thus, the same resource identifier is overloaded depending on where it is used [HeGA20]. However, there exists no semantic connection between these two different views on such an identifier, unless the identifier is explicitly related to itself, via e.g. an *instance-of* relationship *ex*:*Animal a ex*:*Animal*.

OWL offers an extended vocabulary that, compared to RDFS, provides the means to express such and other assertions. In particular, these constructs declare [ArSS15]:

- *object property*: a specialized property that relates instances only with each other
- *data property*: a specialized property that relates instances only with literal values
- *inverse property*: an inverse property that relates objects with subjects of the original property
- *property characteristics*: different characteristics on properties, that are used to mark a property as transitive, (inverse) functional, (ir)reflexive or (a)symmetric (these characteristics impact the outcome of inference)
- *instance enumeration*: a class via enumeration of all its possible representatives
- *class expression*: a class via a combination of multiple class expressions using boolean operators
- *equivalence & disjointedness*: the equivalence or disjointedness of classes or instances (important for reasoning)
- *restriction*: a specialized class that is declared in terms of the instances it contains, defined through the use of conditions on a data / object property

- *universal quantification*: a specialized restriction that contains an individual if a property holds for all values in the object position
- *existential quantification*: a specialized restriction that contains an individual if a property holds for some values in the object position
- *cardinality*: a specialized restriction that contains an individual if a property holds for at least, at most or for exactly a number of values in the object position
- *explicit value*: a specialized restriction that contains an individual if a data property points to an exact value in the object position
- *local reflexivity*: a specialized restriction that contains an individual if an object property holds between that exact same individual

Moreover, there exist different OWL variants, also called *profiles*, which expose restricted OWL vocabularies. Such limited vocabularies guarantee decidability and worst-case computational complexities for reasoners operating on these data sets. Subsection 2.3.3 provides a more in-depth look at these reasoning capabilities.

Simple Protocol and RDF Query Language

SPARQL is the standard query language for accessing and modifying data on RDF triple stores [HeGA20].

The base structure of a SPARQL query looks similar to a Structured Query Language (SQL) query used for accessing data from relational databases [ArSS15], with e.g. the commonly used `SELECT` and `WHERE` statement blocks. However, instead of querying relational database row projections obtained via table joins, it queries RDF graphs using graph patterns (set of triples) encoded in the Turtle serialization format [HiKR09]. SPARQL supports, just like SQL, variables, delete / insert queries, sub-queries, aggregates, groups, group ordering, built-in functions, and more. SPARQL allows query creation via all available vocabularies introduced thus far (e.g. OWL).

An example SPARQL query is given as part of source Listing 3.9. Although we believe the presented SPARQL source listings are intuitive on its own, the particularities of SPARQL syntax and semantics are beyond the scope of this work, thus, if needed, the reader is advised to take a look at the appropriate W3C recommendation [Coot13].

2.3.3 Built-in reasoning in ontologies

As briefly mentioned in the previous Subsection, some ontology constructs have a strong impact on logical *reasoning* —drawing conclusions based on premises. Ontologies are given a formally defined meaning (or *semantics*), which is a mathematically "precise way to interpret each expression of the language" [Usch18]. These precise interpretations are the foundations of logical reasoning, whether used to *entail* new knowledge based on already present knowledge, or to model and enforce *expectations* on future data [HeGA20]. In the former case, we are interested in additional knowledge that is semantically *entailed* from previous knowledge [HiKR09], and this can be achieved using built-in formal semantics which are discussed in detail in this Subsection.

Model-theoretic view

From a theoretical standpoint, ontologies are dominantly analyzed from a *model-theoretic* view [HiKR09], as introduced in Subsection 2.2.1. In this view, the given ontology's RDF graph is considered the *knowledge base* and its triples $s\ p\ o$ represent its basic building blocks known as *propositions*, in contrast to e.g. propositional letters $P, Q, ...$ used in propositional logic. For a given ontology this proposition set is made up of triples from any used standard vocabularies (as explained in Subsection 2.3.2), as well as of triples directly contained or otherwise indirectly imported in the ontology's RDF graph.

Entailment relations

As previously mentioned in Section 2.2.1, interpretations operate on a particular vocabulary and impose certain conditions. Moreover, interpretations that share the same conditions on a particular vocabulary are commonly named after that vocabulary. Hence, the W3C recommendations on RDF(S) semantics, OWL-2 RDF-based semantics and OWL-2 direct semantics aim to standardize these conditions on RDF, RDFS and OWL vocabularies[11]. Although the particularities of these conditions are out of scope of this work, they now come into effect in facilitating different entailment relations [HiKR09], which infer additional triples based on existing ones.

The *simple entailment* relation is the default one used behind the scenes when running a SPARQL query and ensures that variables in place of subjects or objects are properly matched against resources from the original RDF graph. In particular, the original RDF graph is iteratively extended with inferred triples containing blank nodes until those blank nodes "line up" with variables from the SPARQL query, so that the thus entailed RDF graph matches the graph pattern from the query.

The *RDF entailment* relation extends the set of simple entailment inference rules. It is narrow in scope, as it is governed by the limited expressiveness of the RDF vocabulary. However, it sets up the inference foundations used in the following entailment relations.

RDFS entailment builds on RDF entailment and provides a few simple but helpful inferences, e.g. via the type propagation rules. Moreover, "simple inferences are often more useful than elaborate ones" [HeGA20].

OWL entailment roughly augments the previous form of entailment. As OWL entailment is more powerful than what can merely be expressed by virtue of inference rules, such rules can only be applied under some restrictions of the full OWL syntax[12]. These rules are elaborate and infer much more implicit knowledge than in the previous entailment relations. Furthermore, OWL entailment can detect flaws introduced while modeling the ontology, thus making it easier to design and "maintain the ontology" [Usch18]. In particular, we distinguish between *incoherent ontologies*, which contain one or more *inconsistent classes* for which no individual can be a representative of these class in any model of the ontology, and between *inconsistent ontologies*, for which there exists not a single model [HBPS08]. However, the full scope of OWL expressiveness comes at a price.

[11]These conditions are presented in detail in the respective W3C recommendations [Coot14a][Coot12c][Coot12a].

[12]The full list of restrictions for this so-called OWL2-RL profile is described in the respective W3C recommendation [Coot12b].

OWL Profiles

As mentioned in the previous paragraphs, some restrictions apply that govern the necessary preconditions for the previous OWL inference rules to be used. The OWL language specifies multiple semantic subsets with certain restrictions applied to the usage of the OWL vocabulary, called *profiles*. As the full spectrum of RDFS and OWL vocabulary without further restrictions leads to *undecidable* reasoning in general (see explanation further below), multiple such profiles have been adopted to facilitate "favorable computational properties" [HiKR09] for the respective, specialized application areas they are designed for:

- *OWL Full* The full scope of the OWL language is a proper superset of the RDFS vocabulary and contains the complete OWL vocabulary. Except some basic sanity checks, no further restrictions apply to ontologies to be considered OWL 2 Full conform. This profile offers the greatest expressive power in terms of reasoning and also encapsulates all possible inferences that can be made using other profiles [HiKR09]. However, in this profile reasoning is undecidable, which means it is unknown whether a reasoner will terminate for a given ontology or not. More explanations are given further below, but this is in many cases undesirable, since we cannot realistically reason about whether the given ontology is coherent or consistent [HeGA20], let alone derive further implicit knowledge. However, it is an option in application areas "for conceptual modeling in cases where automated reasoning is not required" [HiKR09].
- *OWL DL* OWL DL forbids the usage of some identifiers from the RDFS vocabulary, but otherwise also includes the complete OWL vocabulary. In particular, these are mostly identifiers which are replaced with OWL-equivalent ones, like $rdfs : Class$ and $rdfs : Property$. Alongside some additional extended sanity checks, further restrictions encountered in practice apply to the usage of properties. Firstly, the property hierarchy closure must adhere to a strict order. Secondly, the (inverse-)functional, irreflexive and asymmetric property characteristics, as well as property disjointedness assertions, may only be assigned to *simple* properties. Moreover, any class expressions using cardinality or local-reflexivity restrictions must refer to *simple* properties solely. Noteworthily, *simple* properties are super-properties of non-transitive sub-properties from the property hierarchy closure. For a complete list of restrictions the reader is advised to refer to the respective W3C recommendation [Coot12d]. Being based on *description logic*, a decidable fragment of otherwise *semi-decidable first-order logic*, OWL DL is itself decidable [HiKR09]. Moreover, DL reasoning is `2NEXPTIME`-complete [Glim11]. Refer to Subsection 2.3.4 for more details. As seen in later paragraphs, even though the DL profile guarantees termination, it does not scale very well with larger amounts of data [Usch18]. For such application areas *tractable* solutions are preferred, which are *efficiently solvable* [Glim11]. Note that the DL profile is a semantic super-set of the following profiles [HeGA20], and thus inference in DL can draw at least the same conclusions as a reasoner for these other profiles.
- *OWL RL* We have assumed an OWL ontology restricted to the specification of this profile before for discussing the majority of deduction rules in OWL entailment. Intuitively, it thus defines the "subset of OWL that can be faithfully processed by such rule systems" [HeGA20]. It "offers some amount of interoperability [with external rule-based languages]" [HiKR09], as seen in later Subsection 2.3.4, and "has become the preferred approach for reasoning with Web data" [Usch18]. RL reasoning falls in the `PTIME`-complete complexity

class [Glim11]. A detailed list of restrictions to the RL profile are given in the respective W3C recommendation [Coot12b].

- *OWL EL* Designed for ontologies with large amounts of classes and properties (*TBOX*), this profile specializes in inferring additional such relationships [Usch18] and is `PTIME`-complete [Glim11].
- *OWL QL* Designed for ontologies with large amounts of instances (*ABOX*) [Usch18], where queries can be expressed in SQL and run on common relational database systems [HiKR09]. It is `PTIME`-complete and `NLSPACE`-complete [Coot12b].

2.3.4 User-defined reasoning in ontologies

Since OWL was carefully designed to model only a specific form of (implicit) knowledge [Usch18], user-defined constructs have to be used for modeling other forms of knowledge that can be reasoned about. In particular, such user-defined reasoning can be facilitated via custom rules that are similar to inference rules (with antecedents and consequents) described before [HiKR09]. For example, deriving all relations between different family members is straightforward to do by hand with SPARQL queries, but difficult to express implicitly through available OWL constructs. On the other hand, custom rules can be used to state such implications directly [HeGA20].

As seen before, OWL allows for more expressive constructs than what can be captured purely by entailment via inference rules. Thus, it is vital to look at how and under which restrictions OWL entailment and custom entailment can be combined to achieve an expressive way of modeling knowledge with desirable performance characteristics [HiKR09]. Recent efforts have been made to thus "get best of both worlds" by seamlessly integrating web-oriented rule languages with the Web Ontology Language in frameworks like the Rule Interchange Format (RIF) and the aforementioned OWL2-RL profile [HeGA20].

Description logic

As first-order logic is semi-decidable, a decidable fragment of first-order logic named *description logic* (DL) was introduced which thus offers "favorable trade-offs between expressivity and scalability" [HiKR09]. This variant of description logics used in OWL is named $\mathcal{SROIQ}(\mathcal{D})$ [HiKR09]. Each letter represents a feature of expressiveness of such description logic, with different combinations of letters allowing different syntactic constructs with known computational guarantees [HiKR09].

How these constructs are exactly connected to the vocabulary of OWL (as showcased in Subsection 2.3.2) is out-of-scope of this work and thus not further explored in detail, hence the interested reader is referred to appropriate literature, e.g. Hitzler et al's book [HiKR09].

Syntax and semantics

The common syntax used throughout literature for description logics is related, but distinct to what has been used thus far in describing first-order logic. It resembles a short-hand form of first-order logic syntax with variables being omitted. Moreover, using the *first-order semantics* of description logics, description logic statements can be directly translated into first-order logic formulas [TsHo03]. More specifically, this translation requires additional predicates resembling (in)equality $=$ / $\neq$ [HiKR09].

Indeed, a concept subsumption statement of the form $C \sqsubseteq D$ translates to a universally-qualified implication with unary predicates of the form $\forall X[P_C(X) \rightarrow P_D(X)]$. Moreover, a role subsumption statement of the form $R \sqsubseteq S$ corresponds to an implication with universally-bound variables over binary predicates of the form $\forall X\{\forall Y[P_R(X,Y) \rightarrow P_S(X,Y)]\}$ [TsHo03]. Description logic statements about concepts as in the former form are commonly referred to as the *TBox* (the terminological knowledge), whereas statements about roles in the latter form are referred to as the *RBox* (knowledge about roles). Combined with the *ABox* (assertional knowledge) that concerns itself with statements about individuals, these form a description logic *knowledge base* [HiKR09].

However, more advanced description logic constructs, like transitive roles or complex role axioms, require an additional first-order logic variable [TsHo03]. For example the statement $Trans(T)$ matches the following implication using three universally-qualified variables $\forall X\{\forall Y[\forall Z(\ P_T(X,Y) \land P_T(Y,Z) \land X \neq Y \land Y \neq Z \rightarrow P_T(X,Z)\ )]\}$ [TsHo03].

Using these and additional translations, which are in detail described in Grosof et al's work [GHVD03], an e.g. $\mathcal{SROIQ}$ knowledge base can be translated into a first-order logic knowledge base [HiKR09], for which the previously introduced first-order logic semantics and reasoning apply. Beside the first-order semantics, description logics can be interpreted using *direct model-theoretic semantics* [HiKR09], which has been briefly covered while talking about OWL entailment in Subsection 2.3.3.

Reasoning

As shown before, in the translation process from description logics to first-order logic some universally-quantified variables need to be introduced [GHVD03]. In many cases, this translation results in first-order logic formulae with one or two bound variables, which are known to be decidable in `NEXPTIME` [TsHo03][BaHS08]. However, some translations result in three or more bound variables, making the resulting knowledge base undecidable [TsHo03][BaHS08].

Therefore, decidable, "dedicated reasoning procedures for DLs" [BaHS08] where devised, by adapting the refutation systems from first-order logic to the framework of description logics [HiKR09]. In particular, the previously mentioned tableux method adapted to description logics can prove whether such a $\mathcal{SROIQ}(\mathcal{D})$ knowledge base is satisfiable [HiKR09] in `N2EXPTIME`. The complete tableux algorithm is given in Horrocks et al's work [HoKS06].

Combining OWL with custom rules

With the fundamental background explained, we now explore how the strengths of OWL-DL / $\mathcal{SROIQ}(\mathcal{D})$ entailment can be combined with custom entailment, in particular so that such combination is also practical and still decidable [HiKR09].

In general, custom entailment suitable for interoperability with OWL is achieved through the use of *rule*-based languages [HiKR09]. In this context, rules facilitate the syntax and modified semantics of a defined subset of all possible universally-qualified first-order logic implications [HiKR09], which look similar to the formula translations explored in paragraph 2.3.4 about description logic semantics. As any first-order logic formula can be converted with more or less effort into such form, this limitation on a specific subset is required for providing favorable computational guarantees [HiKR09]. Although the specifics are discussed in detail in Section 2.4, it intuitively looks like e.g. the following formula $\forall X\{\forall Y[\,person(X) \land person(Y) \land married(X,Y) \rightarrow$

$spouse(X)$]}. However, instead of a general first-order logic prover, more specific and more performant dedicated reasoners are used [HiKR09].

These rule-based languages are *monotonic* so that adding rules to an existing knowledge base $KB \subseteq KB'$ can only expand the set of logical consequences drawn from it $\{F : KB \vDash F\} \subseteq \{F' : KB' \vDash F'\}$ [HiKR09]. This is in direct contrast to *non-monotonicity* introduced in Section 2.4.

As both description logics and custom rules are based on first-order logic semantics, a combination of these concepts is indeed possible and conclusions can be drawn by finding interpretations that satisfy such a compound knowledge base [HiKR09]. However, although both description logics and custom rule languages are decidable on their own, the fusion of these approaches is not decidable without further restrictions [HiKR09]. One possible restriction is to express custom rules in the form of description logic statements known as *description logic rules*. However, in this case, the expressiveness is noticeably limited due to the syntactic restrictions and does not allow for some simple statements such as e.g. $\forall X\{\forall Y[\ married(X,Y) \wedge divorced(X,Y) \rightarrow inComplicatedRelationship(X,Y)\ ]\}$ [HiKR09]. On the other hand, researchers have proposed *DL-safe* rules with precisely defined restrictions such that a combined knowledge base of those concepts still leads to decidable reasoning [HiKR09].

For a rule to be considered *DL-safe* all variables used in predicates part of the description logic knowledge base must also be bound in the rule antecedent by predicates not part of the description logic knowledge base [HiKR09]. For example, with $Knows$ being an OWL property, the rule $\forall X\{\forall Y[\ Knows(X,Y) \rightarrow heardOf(X,Y)\ ]\}$ would not be DL-safe, whereas $\forall X\{\forall Y[\ married(X,Y) \rightarrow Knows(X,Y)\ ]\}$ would. However, any rule can be made DL-safe by adapting the rule semantics and introducing artificial predicates that hold for all constant symbols of the knowledge base [HiKR09]. Therefore, by adding $\rightarrow temp(c)$ for all constants c of the knowledge base, the previous rule can be made DL-safe $\forall X\{\forall Y[\ Knows(X,Y) \wedge temp(X) \wedge temp(Y) \rightarrow heardOf(X,Y)\ ]\}$. Although this seems semantically equivalent to the original rule definition on first look, it is not [HiKR09]. In particular, such rules cannot hold for any unnamed individuals X in OWL given by e.g. $\exists Y(Knows(X,Y) \wedge Person(Y))$, where $Person$ is another OWL concept. Furthermore, a reasoner supporting DL-safe rules may alter such semantics implicitly in the background, without the need to explicitly introduce these utility predicates. These reasoners operate either by adding such utility predicates themselves, or, equivalently, limit variable assignments to constant symbols of the knowledge base (OWL explicitly named individuals) [HiKR09].

Apache Jena rules

The "Jena framework is a free and open source Java framework for building Semantic Web applications" [AlDC15]. Alongside other features for managing knowledge bases, this framework includes Apache Jena rules, one such rule-based reasoner that can be used to drive custom entailment in ontologies [AlDC15][Foot19].

The Apache Jena rule reasoner operates in a forward-chaining, backward-chaining or hybrid mode [AlDC15] [Foot19] [HFBPL09]. In forward-chaining mode, all possible rules are examined and satisfied ones are executed[13], whenever new rules or new triples are added to the ontology [HFBPL09][Foot19]. Such rules that *fire* produce new triples or add entire new rules to

[13]Satisfied rules are found more efficiently through the use of the *RETE* algorithm [Forg89].

the knowledge base. This process is repeated iteratively until no new rules can be satisfied [HFBPL09]. One example forward rule looks like

[$\mathit{transitiveRuleF}$:
$(?x\ foaf{:}knows\ ?y),\ notEqual(?x, ?y), (?y\ foaf{:}knows\ ?z),\ notEqual(?y, ?z) \rightarrow (?x\ foaf{:}knows\ ?z)$
].

This rule fires and thus produces a new triple consequent once all conjunctively connected triples and functions are satisfied in the antecedent. On the other hand, the backward-chaining mode operates by trying to find antecedents that would satisfy a consequent, whenever a query is posed against the ontology [HFBPL09][Foot19]. In particular, the rule may be satisfied directly by existing triples or (recursively) through triples obtained by other rules via resolution [Foot19][14]. One example backward rule looks like

$\rightarrow\ table(transitiveRuleB)$.
[$\mathit{transitiveRuleB}$:
$(?x\ foaf{:}knows\ ?z) \leftarrow (?x\ foaf{:}knows\ ?y),\ notEqual(?x, ?y),\ (?y\ foaf{:}knows\ ?z),\ notEqual(?y, ?z)$
].

Finally, the hybrid mode runs the forward-chaining engine first, before running the backward-chaining one on top of the previous deductions [Foot19]. The forward-chaining engine may even produce new backward-chaining rules constrained to the satisfying variable assignments of the respective forward rule [Foot19]. This allows the ontology engineer to get best of both worlds by leveraging the machine processing performance of logic resolutions [Fitt12] “that are relevant to the dataset at hand” [Foot19]. One example hybrid rule looks like

[$\mathit{transitiveRuleH}$:
$(?x\ foaf{:}knows\ ?y),\ notEqual(?x, ?y) \rightarrow [(?x\ foaf{:}knows\ ?z) \leftarrow (?y\ foaf{:}knows\ ?z),\ notEqual(?y, ?z)]$
].

Example Jena rules are given in source Listing 3.8.

Shapes Constraint Language

The Shapes Constraint Language (*SHACL*) is a W3C recommendation that presents a schema language for “describing and validating RDF graphs” [Coot17b]. In particular, it provides constraints for enforcing a specific structure and content of an RDF graph [Usch18][PKNŞ19].

SHACL evolves around the declaration of *shapes*, descriptions of what part of the RDF graph to validate and how to do it. These shapes specify *constraints* on a given collection of RDF subjects, properties or objects referred to as shape *targets*. The W3C recommendation lists various built-in generic constraints, like e.g. limits for the maximum length of a property value, that can be further parameterized, with e.g. the specific maximum length. Furthermore, users can specify their own custom constraints via SPARQL queries that return all violations in a specific format.

[14] The exact semantics are that of Datalog [CGT^{+}89]. Infinite loops are avoided through the use of tabling by memoizing the result of previously satisfied rules [Foot19].

Given an input RDF graph, SHACL-supporting tools then generate a *validation report* of all constraint violations, whereby the report itself is encoded as an RDF graph. An example SHACL shape is given in source Listing 3.9. For more details on SHACL validation, the reader is advised to take a look at the respective recommendation [Coot17b].

In addition to modeling expectations on data, SHACL facilitates the deduction of new *inferenced* RDF triples based on existing *asserted* ones [HeGA20][Usch18]. SHACL *rules* are, like constraints, associated with shapes and are thus triggered for each RDF node of a shape target declaration. Rules can be given in the form of *triple rules* or *SPARQL rules.* Triple rules pertain to the production of subject, predicate and object triples for target nodes that pass certain *conditions* and are declared as a collection of triples on the respective shape. SPARQL rules are, on the other hand, given as a SPARQL construct queries on the associated shape [HeGA20]. The details of this and other SHACL extension features are described in the respective W3C working group note about SHACL advanced features [Coot17a].

2.4 Answer set programming

As mentioned in Subsection 2.3.4 about using rule languages in ontologies, rules are of a similar syntactic form as a subset of all possible universally-qualified first-order logic implications [HiKR09]. However, answer set programming (hereinafter ASP) semantics are different to that of mathematical logic. As "ASP has become very attractive for the representation and solving of search-problems both for academia and industry" [FFST$^+$18], this Section explores this *logic programming language* and its relevant concepts.

The first Subsection describes the syntax of such logic rules and programs, as used in answer set programming. The next Subsection explains the semantics behind logic programs and how such programs are different to traditional propositional or first-order logic knowledge bases. Finally, we explore what different kinds of reasoning one can perform with answer set programming, touch on how a machine performs reasoning more efficiently, and what sorts of computational complexities one may expect, before concluding with available types of reasoning.

2.4.1 Syntax: Basic building blocks

The basics building blocks of answer set programming are a subset of the ones encountered in first-order logic, as shown in Subsection 2.2.4.

Constants refer to "individuals from the domain of discourse" (e.g. $a, mary, john, ...$), whereas *variables* Var are placeholders for individuals from this universe of discourse (e.g. $X, Person, \_, \_TEMP, ...$) [EiIK09]. *Function* symbols are a generalization of constants with greater arity [GKKS12]. Function symbols form *terms* T when combined with other constants and variables (e.g. $f_3(a, b, X), \neq (mary, john), ...$) [EiIK09]. Terms can also be *simple* if they consist of a single variable or constant (e.g. $Z, john, ...$) [GKKL$^+$15]. A term without any variables is referred to as *ground*, whereas a term with variables can be converted to a ground one by substituting all variables with a constant [GKKS12]. *Predicate* symbols P of 0-arity are called *propositions* (e.g. $p, goalReached, ...$), and are otherwise made up of multiple terms (e.g. $p(a), married(mary, john), ...$) [EiIK09][GKKS12].

Although predicates and functions look similar to each other on the surface, they serve different purposes. Functions either represent built-in Boolean constants and arithmetic / aggregate operations (e.g. $\#true \mathrel{\widehat{=}} \top$, $\#false \mathrel{\widehat{=}} \bot$ and $+(ValueX, ValueY) \mathrel{\widehat{=}} ValueX + ValueY$) which are evaluated during reasoning, or are otherwise uninterpreted syntactic sugar that "often make[s] the modelling easier and the resulting encodings more readable" [BrET11]. On the other hand, predicates relate individuals to each other for which the reasoner determines whether this relation holds or not.

Atomic formulae A, or short *atoms*, consist of a single predicate (e.g. $goalReached$), whereas *literals* L can also include the (default) negated version of that predicate (e.g. $\sim goalReached$) [BrET11]. The set of positive literals of a logic *program* is thus defined as $L^+ = \{a \in A \mid a \in L\}$ and the set of negative literals as $L^- = \{a \in A \mid \sim a \in L\}$ [GKKS12]. Atoms and literals are *ground*, like terms, if they contain no variables. A logic *program LP* consists of a set of *rules* $\{R_1, R_2, ..., R_n\}$ [EiIK09], whose different types are defined in the following subsections. Common to all rules are their *head* (consequent) and *body* (antecedent) sets of atoms $a \in A$, denoted as $R \mathrel{\widehat{=}} head(R) \leftsquigarrow body(R)$ [GKKS12]. Furthermore, we distinguish positive body atoms for a rule $body(R)^+ = \{a \in L^+\}$ from negative ones $body(R)^- = \{a \in L^-\}$ [GKKS12]. A program and its rules are *ground* if they are variable-free.

2.4.2 Syntax: Normal Logic Program rules

With the basic building blocks covered, we now explain what specific forms of rules are used in answer set programming.

The following overview is limited to different variants of *normal logic program* (NLP) rules [EiIK09]. As more expressive rules, such as e.g. *disjunctive logic program* (DLP) rules, increase computational complexity [GKKS12][GKKL$^+$15][EiIK09], this work's contribution does not make use of them and these rules are thus not further explored.

The different NLP rule forms can all be converted mechanically to the default normal rule form [GKKS12], making all these NLP rules thus inhibit a common set of semantic properties. We intuitively touch on the meaning of these presented constructs in the following subsections, by having X denote a set comprised of truthy atoms $X \subseteq A$ and representing a possible *solution* to a normal logic program NLP. A more in-depth and precise description of solution semantics is intentionally left to the following Section 2.4.3. However, even in this abstract form, this notation helps to formalize the presented rule variants.

Facts

Facts are the simplest form of rules, denoted by:

$$a \tag{R_f}$$

A fact consists of a head atom $a \in A$ and no body atoms $body(R) = \emptyset$ and is equivalent to a *positive* rule of the form $a \leftarrow \top$ [GKKS12].

Intuitively, the head atom a always holds with $head(R) \subseteq X$ [GKKS12].

Positive rules

Starting out with more interesting rules, a *positive* rule R_p has the following structure:

$$a_0 \leftarrow b_1, b_2, ..., b_m \tag{R_p}$$

A positive rule is thus made up of the singular head atom $a_0 \in A$ and body atoms $b_1, b_2, ..., b_m \in A$ connected with each other via conjunctions, with the restriction that the head atom of the rule is the only positive occurring literal $\mathit{body}(R)^- = \emptyset$ [EiIK09][GKKS12]. This restriction is also evident when viewing the rule R_p in its seemingly equivalent first-order logic form with logic implication $a \leftarrow b_1 \wedge b_2 \wedge ... \wedge b_m$, or more specifically, $a \vee \neg b_1 \vee \neg b_2 \vee ... \vee \neg b_m$.

Intuitively, we can derive the head of a rule $\mathit{head}(R) \subseteq X$ whenever all its body atoms hold $\mathit{body}(R) \subseteq X$ [GKKS12]. However, the semantics of a simple rule $a \leftarrow b$ differ from the semantics of a seemingly equivalent first-order logic implication $a \Leftarrow b$ ($\equiv a \vee \neg b$) [EiIK09], the latter of which are described in Subsection 2.2.4. Namely, given the Boolean valuation $\cdot^B$ of proposition $a^B = false$, this partial interpretation can be extended in exactly one possible way with $b^B = false$ to form a model of that simple first order logic formula. On the other hand, knowing that $\neg a$ holds does not automatically derive $\neg b$ in an answer set programming solution. In that sense, rules are intuitively more akin to imperative programming if $\mathit{body}(R)$ then $\mathit{head}(R)$ statements [EiIK09]. As an outlook, we have to additionally state the implication's contraposition explicitly by adding the rule $\neg b \leftarrow \neg a$ for this answer set program to have equivalent semantics to the first-order logic implication, as motivated by the logical equivalence $a \Leftarrow b \equiv \neg b \Leftarrow \neg a$.

Normal rules

Normal rules extend the notion of positive rules with the concept of default negation [EiIK09]:

$$a_0 \leftarrow b_1, b_2, ..., b_m, \sim c_1, \sim c_2, ..., \sim c_n \qquad (R_n)$$

In contrast to a positive rules, a normal rule may contain (default) negated literals $\sim c_1, \sim c_2, ..., \sim c_n$ constructed from atoms $c_i \in A$ in its body [EiIK09], with the default negation $\sim$ often being denoted with the symbol *not* instead.

Extending the intuitive meaning from positive rules, the head of the rule $\mathit{head}(R) \subseteq X$ may be true, if all positive body atoms are "(provably) true" $\mathit{body}(R)^+ \subseteq X$ and none of the negative body atoms are "(possibly) false" $\mathit{body}(R)^- \cap X = \emptyset$ [GKKS12]. The default negation can also be understood as that there is no reason to believe a default negated literal $\sim c_i$ to be satisfied [BrET11], unless the truthfulness of its corresponding positive literal c_i is stated or can be (finitely) derived otherwise [EiIK09]. Therefore, default negation is alternatively also referred to as *negation as failure* [EiIK09]. In order to emphasize the difference to *strong negation*, which is introduced later on, it is also known as *weak negation* [EiIK09].

Integrity constraints

Integrity constraints are syntactic sugar for a special form of normal rules:

$$\leftarrow b_1, b_2, ..., b_m, \sim c_1, \sim c_2, ..., \sim c_n \qquad (R_i)$$

These are equivalent to normal rules of the form $false^{R_i} \leftarrow \sim false^{R_i}, b_1, b_2, ..., b_m, \sim c_1, \sim c_2, ..., \sim c_n$, with atoms $b_i \in A$ and $c_i \in A$, and $false^{R_i} \notin A$ being a fresh atom [EiIK09].

Intuitively, integrity constraints filter out possible solutions in which the rules' body is satisfied with $\mathit{body}(R)^+ \subseteq X$ and $\mathit{body}(R)^- \cap X = \emptyset$ [GKKS12]. Namely, if the given literals b_i, $\sim c_i$ are satisfied, the freshly introduced atom $false^{R_i}$ cannot be both true and false at the same time, thus filtering out such solutions [EiIK09]. Furthermore, it follows that the synthesized head of such integrity rule may never be derived $head(R) \cap X = \emptyset$ [GKKS12].

Rules with default negation in the head

Another syntactic extension to ordinary NLP rules is the usage of default negation in the head of rules:

$$\sim a_0 \leftarrow b_1, b_2, ..., b_m, \sim c_1, \sim c_2, ..., \sim c_n \tag{R_d}$$

Given that rule's body is satisfied with $body(R)^+ \subseteq X$ and $body(R)^- \cap X = \emptyset$, that is, $b_i \in A$ hold and atoms $c_i \in A$ are possibly false, the default negated atom $a_0 \in A$ occurring in the head of the rule $head(R) = \{a_0\}$ must not known to be satisfied $head(R) \cap X = \emptyset$.

Conditional literals

Another utility construct is a *condition literal* [GKKS12], with $a_i \in A$ being the *head* literal and $b_{i_j}, \sim c_{i_j} \in A$ the *body* conditions [GKKL+15]:

$$a_i : b_{i_1}, ..., b_{i_m}, \sim c_{i_1}, ..., \sim c_{i_n} \tag{∂R_l}$$

A condition literal can be thought of as a nested implication $a_i \leftarrow b_{i_1}, ..., b_{i_m}, \sim c_{i_1}, ..., \sim c_{i_n}$ [GKKL+15], or alternatively, as a mathematical set notation $\{a_i \mid b_{i_1}, ..., b_{i_m}, \sim c_{i_1}, ..., \sim c_{i_n}\}$ [GKKS12]. In order to resolve such constructs to ordinary NLP rules, this inner implication needs to be evaluated first.

As an example, the rule $a \leftarrow b : c$ holds $head(R) \subseteq X$ and derives a whenever c does not, in which case the inner implication resolves to $a \leftarrow \emptyset$, or when both c and b hold, in which case the nested "expression" amounts to $a \leftarrow b$ [GKKS12]. The power of condition literals comes from combining them with variables, which allows the user to concisely specify a variety of literal expressions inside rules, as also shown in source code Listing 3.14.

Choice rules

Choice rules seemingly extend normal rules with the notion of choice for deriving head atoms:

$$\{a_1, a_2, ..., a_p\} \leftarrow b_1, b_2, ..., b_m, \sim c_1, \sim c_2, ..., \sim c_n \tag{R_c}$$

Given that rule's body is satisfied with $body(R)^+ \subseteq X$ and $body(R)^- \cap X = \emptyset$, that is, $b_i \in A$ hold and atoms $c_i \in A$ are possibly false, then any subset of atoms $a_i \in A$ occurring in the head of the rule $head'(R) \in \mathcal{P}(head(R))$ may hold as well $head'(R) \subseteq X$ [GKKS12]. Note that default negated literals in the choice rule's head have no effect on the obtained solution [GKKS12], and therefore we are interested in positive literals $head'(R) \cap X^- = \emptyset$ only.

Cardinality rules

Cardinality rules are an extension to the previously introduced choice rules. In contrast to choice rules, cardinality rules control the amount of head literals that are derived if a rule's body is fulfilled. Furthermore, they can restrict the amount of body literals that must hold for the head of a rule to be derived [GKKS12].

A basic cardinality rule with a lower bound restriction on the body literals looks like:

$$a_0 \leftarrow lb\ \{b_1, b_2, ..., b_m, \sim c_1, \sim c_2, ..., \sim c_n\} \tag{$R_{cb_{lb}}$}$$

The rule's body holds if at least (non-negative) lower bound lb many literals b_i, $\sim c_i$ are satisfied so that the rule's head atom $head(R) = \{a_0\}$ may be derived $head(R) \subseteq X$ [GKKS12].

A more advanced cardinality rule includes an upper bound restriction on the amount of satisfied body literals. It is of the following form:

$$a_0 \leftarrow lb \; \{b_1, b_2, ..., b_m, \sim c_1, \sim c_2, ..., \sim c_n\} \; ub \qquad (R_{cb_{ub}})$$

Here, the previous definition is tightened by dismissing the derivation of a rule's head $head(R) \subseteq X$ if the cardinality of fulfilled literals b_i, $\sim c_i$ is greater than the (non-negative) upper bound ub, given that $0 \leq lb \leq ub$ [GKKS12].

After covering cardinality restrictions on rule bodies, we now explore cardinality limits in the rule head:

$$lb \; \{a_1, a_2, ..., a_p, \sim d_1, \sim d_2, ..., \sim d_q\} \; ub \leftarrow b_1, b_2, ..., b_m, \sim c_1, \sim c_2, ..., \sim c_n \qquad (R_{ch})$$

If the body atoms b_i, $\sim c_i$ are satisfied with $body(R)^+ \subseteq X^+$ and $body(R)^- \cap X^- = \emptyset$, an answer set solver may derive $0 \leq lb \leq r \leq ub$ head literals a_i, $\sim d_i$, with $r \leq p + q$ [GKKS12].

Finally, the most generic form of cardinality rules permits *cardinality restrictions* in place of rule literals [GKKS12]. Given such cardinality restrictions $lb_i \; \partial R_{c_i} \; ub_i$ consisting of atoms $a_{i_j} \in A$ and lower & upper bounds lb_i, ub_i:

$$lb_i \; \{a_{i_1}, a_{i_2}, ..., a_{i_m}, \sim a_{i_{m+1}}, \sim a_{i_{m+2}}, ..., \sim a_{i_n}\} \; ub_i \qquad (lb_i \; \partial R_{c_i} \; ub_i)$$

The general form of cardinality rules can then be expressed as [GKKS12]:

$$lb_0 \; \partial R_{c_0} \; ub_0 \leftarrow lb_1 \; \partial R_{c_1} \; ub_1, \; lb_2 \; \partial R_{c_2} \; ub_2, \; ..., \; lb_k \; \partial R_{c_k} \; ub_k \qquad (R_{cg})$$

Intuitively, all body cardinality restrictions $lb_i \; \partial R_{c_i} \; ub_i$ (with $i \geq 1$) must be satisfied $lb_i \leq |\; (\{a_{i_1}, a_{i_2}, ..., a_{i_m}\} \cap X) \; \cup \; (\{a_{i_{m+1}}, a_{i_{m+2}}, ..., a_{i_{m+n}}\} \setminus X) \;| \leq ub_i$ [GKKS12], so that the head cardinality restriction $lb_0 \; \partial R_{c_0} \; ub_0$ may be derived.

Weight rules

In contrast to cardinality restrictions that model limits on the amount of satisfied literals, *weight* rules limit the sum of *weighted* satisfied literals $l_i \mapsto w_i$, where a non-negative[15] integer weight w_i is associated with a literal l_i [GKKS12]. A basic weight rule introduces a lower bound lb on the sum of weights w_i which belong to their respective literals [GKKS12]:

$$a_0 \leftarrow lb \; \{b_1 \mapsto w_1, b_2 \mapsto w_2, ..., b_m \mapsto w_m, \sim c_1 \mapsto w_{m+1}, \sim c_2 \mapsto w_{m+2}, ..., \sim c_n \mapsto w_{m+n}\} \qquad (R_{wb_{lb}})$$

This rule holds $head(R) \subseteq X$ if the sum of weights w_i belonging to satisfied literals b_i, $\sim c_i$ is greater or equal to the lower bound $lb \geq 0$.

[15] Although the inclusion of negative weights is possible in theory, it would "lift computational complexity by one level in the polynomial time hierarchy" [GKKS12]. A detailed explanation as to how this added complexity comes into existence in a solver implementation is given in Gebser et al's work [GKKS12].

Notably, this similarity of weight rules to cardinality rules carries over to the various weight rule forms and their respective NLP translations [GKKS12]. This includes the notion of upper bounds $ub \geq 0$ and weights w_i in the rule head:

$$lb\ \{a_1 \mapsto w_1, ..., a_p \mapsto w_p, \sim d_1 \mapsto w_{p+1}, ..., \sim d_r \mapsto w_{p+r}\}\ ub \leftarrow b_1, ..., b_m, \sim c_1, ..., \sim c_n \qquad (R_{wh})$$

And, furthermore, carries over to *weight restrictions* $lb_i\ \partial R_{w_i}\ ub_i$ in place of rule literals, with atoms $a_{i_j} \in A$ and respective bounds lb_i, ub_i:

$$lb_i\ \{a_{i_1} \mapsto w_{i_1}, a_{i_2} \mapsto w_{i_2}, ..., a_{i_m} \mapsto w_{i_m}, \sim a_{i_{m+1}} \mapsto w_{i_{m+1}}, \sim a_{i_{m+2}} \mapsto w_{i_{m+2}}, ..., \sim a_{i_n} \mapsto w_{i_{m+n}}\}\ ub_i \qquad (lb_i\ \partial R_{w_i}\ ub_i)$$

Similar to the general form of cardinality rules, the general form of weight rules thus looks like:

$$lb_0\ \partial R_{w_0}\ ub_0 \leftarrow lb_1\ \partial R_{w_1}\ ub_1,\ lb_2\ \partial R_{w_2}\ ub_2,\ ...,\ lb_k\ \partial R_{w_k}\ ub_k \qquad (R_{wg})$$

Intuitively, all body weight restrictions $lb_i\ \partial R_{w_i}\ ub_i$ (with $i \geq 1$) must be satisfied $lb_i \leq \sum_{j=1}^{m} w_{i_j}[a_{i_j} \in X] + \sum_{j=m+1}^{m+n} w_{i_j}[a_{i_j} \notin X] \leq ub_i$[16] [GKKS12], so that the head weight restriction $lb_0\ \partial R_{w_0}\ ub_0$ may be derived.

Aggregate rules

Aggregates generalize the notion of "functions on multisets of weights" [GKKS12].

Similar to previous restrictions, *aggregate atoms* $lb_i\ \partial R_{a_i}\ ub_i$ may be used in place of rule literals [GKKL+15]:

$$lb_i\ \prec_{lb_i} \#func\ [a_{i_1} \mapsto w_{i_1}, ..., a_{i_m} \mapsto w_{i_m}, \sim a_{i_{m+1}} \mapsto w_{i_{m+1}}, ..., \sim a_{i_n} \mapsto w_{i_{m+n}}]\ \prec_{ub_i}\ ub_i \qquad (lb_i\ \partial R_{a_i}\ ub_i)$$

Aggregate atoms thus contain a function symbol $\#func$, which represents the aggregation operation to be evaluated against the multiset of contained weights w_{i_j} of respective atoms $a_{i_j} \in A$, further restricted by lower & upper bounds lb_i, ub_i [GKKS12]. Aggregate functions operate on multisets[17], since they have to properly handle and account for multiple weights w_{i_j} of the same value [GKKS12]. The comparison symbols $\prec_{lb_i}$ and $\prec_{ub_i}$ may be any of the comparison operators $<, \leq, =$ and are assumed to be $\leq$ if omitted [GKKL+15]. Moreover, the lower or upper bound restriction may be removed altogether from either side, as in $lb_i\ \partial R_{a_i}$ or $\partial R_{a_i}\ ub_i$, in which case the missing restriction is assumed to be trivially satisfied [GKKS12]. Similar to previous definitions, the general form of aggregate rules corresponds to:

$$lb_0\ \partial R_{a_0}\ ub_0 \leftarrow lb_1\ \partial R_{a_1}\ ub_1,\ lb_2\ \partial R_{a_2}\ ub_2,\ ...,\ lb_k\ \partial R_{a_k}\ ub_k \qquad (R_{ag})$$

An aggregate atom is then satisfied, if the evaluation of the aggregate function $\#func$ lies between the lower and upper bound [GKKS12] $lb_i \prec_{lb_i} \#func([w_{i_j} \mid (a_{i_j} \mapsto w_{i_j} \wedge a_{i_j} \in X) \vee$

[16] The sums make use of the Iverson bracket notation, as explained in Graham et al's book [GrKP94]. If the condition inside the square brackets [ ] is not fulfilled it evaluates to 0, which is then multiplied with the summation element, effectively ignoring the summation element in the summation.

[17] Multisets are denoted using square brackets [ ] in contrast to the braces notation { } used for ordinary sets.

$(\sim a_{i_j} \mapsto w_{i_j} \wedge a_{i_j} \notin X)]) \prec_{ub_i} ub_i$. Moreover, all body aggregate atoms $lb_i\ \partial R_{a_i}\ ub_i$ (with $i \geq 1$) must be satisfied [GKKS12] so that the head aggregate atom $lb_0\ \partial R_{a_0}\ ub_0$ may be derived.

The previously introduced cardinality and weight constraints are a syntactic short-form for *#count* and *#sum* aggregate atoms, respectively [GKKS12]. A cardinality constraint $lb_i\ \{a_{i_1}, ..., a_{i_m}, \sim a_{i_{m+1}}, ..., \sim a_{i_n}\}\ ub_i$ is thus syntactic sugar for the aggregate atom $lb_i \leq \#count\{a_{i_1} \mapsto 1, ..., a_{i_m} \mapsto 1, \sim a_{i_{m+1}} \mapsto 1, ..., \sim a_{i_{m+n}} \mapsto 1\} \leq ub_i$. In this instance, the *#count* aggregate function is treated as a special case and is evaluated against a set of trivially weighted atoms, so that multiple occurrences of an atom a_{i_j} are counted only once [GKKS12]. On the other hand, the weight constraint $lb_i\ \{a_{i_1} \mapsto w_{i_1}, ..., a_{i_m} \mapsto w_{i_m}, \sim a_{i_{m+1}} \mapsto w_{i_{m+1}}, ..., \sim a_{i_n} \mapsto w_{i_{m+n}}\}\ ub_i$ is modeled by the equivalent aggregate atom $lb_i \leq \#sum\ [a_{i_1} \mapsto w_{i_1}, ..., a_{i_m} \mapsto w_{i_m}, \sim a_{i_{m+1}} \mapsto w_{i_{m+1}}, ..., \sim a_{i_n} \mapsto w_{i_{m+n}}] \leq ub_i$ In addition to these aggregation functions, two other common aggregate functions are *#min* and *#max* [GKKL+15].

Weak constraints

Weak constraints combine the syntax of integrity constraints and weight rules in order to model preferences among multiple solutions, to distinguish between "better and poorer solutions" [BrET11].

A weak constraint assigns the entire conjunction of body literals b_i, $\sim c_i$ a weight w and a non-negative integer *priority level* p [GKKL+15]:

$$\leftsquigarrow b_1, b_2, ..., b_m, \sim c_1, \sim c_2, ..., \sim c_n \mapsto w\ @\ p \qquad (R_{wc})$$

When the body literals are satisfied with $body(R)^+ \subseteq X$ and $body(R)^- \cap X = \emptyset$, this weak constraint adds its weight w to an accumulating cost function at priority level p [GKKL+15]. The priority level $@p$ can be omitted, in which case it is assumed to be 0 [GKKL+15].

Intuitively, multiple weak constraints contribute their weights to the same cost function at that priority level. The program *NLP* solutions are then ranked by the sums of these cost functions at each level. The total cost of a solution is a priority-weighted aggregation of all these sums, with higher priority amounts contributing more to the total cost than any other lower priority amount [GKKL+15]. A solution is then *optimal* in regard to weak constraints, if no other solution has a lower total cost associated with it [GKKL+15].

Optimization statements

Optimization statements are a syntactic short-form for expressing multiple basic weak constraints compactly [GKKL+15] in order to solve "(multi-criteria) optimization problems" [GKKS12]. They are made up of *priority-weighted literals* $l_i \mapsto w_i@p_i$, for which the previously introduced weight literals $l_i \mapsto w_i$ are augmented with a priority level p_i:

$$\#minimize\ [b_1 \mapsto w_1@p_1, ..., b_m \mapsto w_m@p_m, \sim c_1 \mapsto w_{m+1}@p_{m+1}, ..., \sim c_n \mapsto w_{m+n}@p_{m+n}] \qquad (R_{o_{min}})$$

Each of the satisfied body literals $b_i \in X$, $c_i \notin X$ contributes its weight w_i to a cost function, which accumulates a multiset of weights [GKKS12], at priority level p_i [GKKL+15]. As in the case of weak constraints, a program's *NLP optimal* solution minimizes the sum of weights at each priority level, with lower priority minimization objectives being always less important than higher priority ones [GKKS12]. The priority level @ p_i can again be omitted, in which it defaults to 0 [GKKL+15]. More formally, let $\sum_p^X$ denote the sum $\sum$ of weights w_i belonging to priority-weighted literals $l_i \mapsto w_i$ @ p that are satisfied in $X \models l_i$ at priority level p [GKKS12] such that $\sum_p^X = \sum_{i=1}^m w_i[b_i \mapsto w_i@p][b_i \in X] + \sum_{i=m+1}^{m+n} w_i[c_{i-m} \mapsto w_i@p][c_{i-m} \notin X]$. A solution X_{sub} is *dominated* by another solution X_{opt} if it has a higher sum at a lower priority level and equal sums at higher priority levels $\forall p_{hi} > p_{lo} : \sum_{p_{lo}}^{X_{sub}} > \sum_{p_{lo}}^{X_{opt}} \wedge \sum_{p_{hi}}^{X_{sub}} = \sum_{p_{hi}}^{X_{opt}}$ [GKKS12]. If a solution is not dominated by any other solution, it is *optimal* [GKKS12].

Alternatively to *#minimize* statements, *#maximize* statements maximize the sum of weights. Such a statement of a trivial form *#maximize*$[b \mapsto w]$ is equivalent to the opposite statement with negative weights *#minimize*$[b \mapsto -w]$ [GKKS12].

Optimization objectives are a key part of addressing this work's problem statement. An example is given in source code Listing 3.15.

Strongly negated literals

In contrast to the so far used default negation $\sim a$ that assumes an atom a to be false by default unless explicitly known or proven otherwise, *classical* (or also *strong*) *negation* $\neg a$ holds only if an atom a is known or proven to be false [EiIK09]. This is further exemplified by weak negation $\sim a$ being fulfilled whenever the respective atom is not part of the solution $a \notin X$, whereas the strong negation $\neg a$ holds whenever the complement of the atom is explicitly part of the solution $\neg a \in X$ [GKKS12]. Strong negation $\neg$ is often denoted with the symbol $-$ instead [GKKL+15].

The difference between the meaning of these two kinds of negation is highlighted by the following example: Given the rule $go\_on\_trip \leftarrow \sim car\_has\_defect$, a person would go on a trip unless they knew the car had a defect, e.g. by noticing it during a previous trip. On the other hand, with $go\_on\_trip \leftarrow \neg car\_has\_defect$ a person would make sure to have the car checked thoroughly before each trip. However, checking for every possible defect before a trip would be cumbersome in practice.

Formally, the set of atoms A of NLP programs is augmented by the complementary set of atoms $\overline{A} = \{\neg a \mid a \in A\}$, with $A \cap \overline{A} = \emptyset$ [GKKS12]. A rule of such an *extended logic program* (ELP for short)[EiIK09] looks like:

$$a_0 \leftarrow b_1, b_2, ..., b_m, \sim c_1, \sim c_2, ..., \sim c_n \qquad (R_e)$$

All atoms in such an ELP rule R_e are either ordinary ones or strongly negated ones $a_i, b_i, c_i \in A \cup \overline{A}$ [EiIK09].

As with previous syntax extensions, an ELP rule can be reduced into ordinary NLP rules by treating the newly introduced atoms independently of their complements $A \cap \overline{A} = \emptyset$ and by adding an implicit integrity constraint that prevents a pair of such complementary atoms of being satisfied jointly [EiIK09][GKKL+15][GKKS12]:

$$\leftarrow a, \neg a \qquad \forall a \in A \qquad (R_{e\triangle n})$$

Although the semantics follow that of ordinary NLP rules, the particularities of resulting solutions are more nuanced. Namely, since these complementary, yet independent, atoms $a, \neg a$ are not linked to each other besides via this integrity constraint, an interpretation of an atom

a may be either true, false or unknown, with the latter having neither a nor $\neg a$ as part of the interpretation [GKKL+15]. This "three-valued view" is in contrast to the "two-valued view" of the so far viewed solutions to ordinary NLP rules, where an atom a may either be true or assumed to be false otherwise [EiIK09].

2.4.3 Semantics

As touched upon in the previous Subsection on NLP syntax, the semantics of logic programming rules $a \leftarrow b$ differ from that of logic implications $a \Leftarrow b$ [EiIK09]. To further distinguish between the previously used model-theoretic view and the semantics of answer set programming, we introduce a similar, yet distinct, formalization.

Interpretation

The basic concepts of an interpretation follow that of first-order logic. Therefore, a boolean valuation $\cdot^B$ maps elements from the normal logic program's NLP possible set of expressions V_{NLP} to the boolean domain $\triangle^B = \{true, false\}$. Additionally, a variable assignment $\cdot^Z$ maps variables to a variable domain $\triangle^Z$. The combined valuation of less complex expressions from the vocabulary V_{NLP}, like e.g. $c^{B,Z} = c^B$ for constant terms, follows inductively as in the case of first-order logic. Refer to Subsection 2.2.4 for more details.

However, answer set semantics adapt this notion of an interpretation. In particular, the set of all possible ground (variable-free) terms T is called the *Herbrand Universe* $HU(NLP)$ of a normal logic program NLP [EiIK09]. Furthermore, the set of all possible ground (variable-free) atoms A is known as the *Herbrand Base* $HB(NLP)$, and is thus comprised of those ground terms applied to predicates of NLP [EiIK09]. A *Herbrand Interpretation* $HI(NLP)$ is in turn any possible subset of all these possible ground atoms $HI(NLP) \subseteq HB(NLP)$ [EiIK09]. In contrast to the previous type of interpretation I and its boolean valuation to $\cdot^B : V_{NLP} \rightarrow \{true, false\}$, a Herbrand interpretation HI thus explicitly expresses only its $true$ atoms, or, stated alternatively, atoms that are *entailed* by it $HI \models a$ [EiIK09]. As a purely "cosmetic" side effect, atoms $a \in A$ that can only be assumed as $false$ are not explicitly listed in HI.

As a more constrained form of general variable assignment $\cdot^Z$, a *substitution* θ maps variables Var back to the set of closed terms $\theta : Var(NLP) \rightarrow HU(NLP)$ [EiIK09]. Furthermore, let Θ denote the set of all possible variable substitutions $\Theta = \{\theta : Var(NLP) \rightarrow HU(NLP)\}$. When a substitution is *applied* to a term $t\theta$ or formula $F\theta$, respective variables in the term $Var(t)$ or formula $Var(F)$ are replaced by the substitution result $\theta(var_i)$. Therefore, one can find all possible ground instantiations of a rule with variables $ground(R) = \{R' \mid \exists\theta \in \Theta, \exists R' \in V_{NLP} : R' = R\theta\}$, as well as the grounding of a program $ground(NLP) = \bigcup_{R \in NLP} ground(R)$ [EiIK09]. These ground instantiations trivially include variable-free rules and programs.

Model

As in the first-order logic case, of interest are *correct* interpretations that satisfy the logic program, out of otherwise all possible sets of ground atoms. Given a Herbrand interpretation HI, it *entails*, or equivalently, is a *model* HM for [EiIK09][GKKS12]:

- $HM \models a$: a ground atom $a \in A$, whenever it is part of the interpretation $a \in HM$
- $HM \models {\sim}a$: the default negation of a ground atom $a \in A$, whenever it is not contained in the interpretation $a \notin HM$

- $HM \models \mathit{head}(R)$: the head $\mathit{head}(R) = \{a\}$ of a ground rule $R \in NLP$, whenever it is part of the interpretation $\mathit{head}(R) \subseteq HM$
- $HM \models \mathit{body}(R)^+$: the positive body $\mathit{body}(R)^+ = \{b_1, ..., b_m\}$ of a ground rule $R \in NLP$, whenever all the positive body atoms are included in the interpretation $\mathit{body}(R)^+ \subseteq HM$
- $HM \models {\sim}\mathit{body}(R)^-$: the default negation of the negative body $\mathit{body}(R)^- = \{c_1, ..., c_n\}$ of a ground rule $R \in NLP$, whenever none of the negative body atoms are entailed in the interpretation $\mathit{body}(R)^- \cap HM = \emptyset$
- $HM \models \mathit{body}(R)$: the body $\mathit{body}(R)$ of a ground rule $R \in NLP$, whenever both its positive $HM \models \mathit{body}(R)^+$ and negated negative $HM \models {\sim}\mathit{body}(R)^-$ parts are entailed
- $HM \models R$: a ground rule $R \in NLP$, with $R \mathrel{\widehat{=}} \mathit{head}(R) \leftsquigarrow \mathit{body}(R)$, whenever either the head is entailed $HM \models \mathit{head}(R)$ or the body is not entailed $HM \not\models \mathit{body}(R)$
- $HM \models R$: a non-ground rule $R \in NLP$, whenever the interpretation entails every ground instantiation of this rule $\forall R' \in \mathit{ground}(R) : HM \models R'$
- $HM \models NLP$: the entire ground program NLP, whenever all of its rules are entailed by it $\forall R \in NLP : HM \models R$
- $HM \models NLP$: the entire non-ground program NLP, whenever its grounding is entailed by the interpretation $\forall R \in \mathit{ground}(NLP) : HM \models R$

Minimal model

Even by examining a smaller set of correct interpretations HM only, there are still models left that encode superfluous information not relevant to the program at hand [EiIK09]. To restrict this undesired behavior, we look for "*canonical model[s] of a program which contain[] only the atoms which are necessarily true according to* [NLP]" [EiIK09]. Such necessarily true atoms are also known as *founded* atoms [EiIK09]. A *minimal model* HMM is thus every model HM of logic program NLP, for which there exists no smaller model of that logic program NLP that is also a subset of it $\nexists HM(NLP) : HM(NLP) \subset HMM(NLP)$ [EiIK09].

Founded atoms can be found iteratively using the *consequence operator* $T_{NLP} : HB(NLP) \rightarrow HB(NLP)$ [GKKS12][EiIK09], defined as:

$$T_{NLP}(HI) = \{\mathit{head}(R) \mid \exists R \in \mathit{ground}(NLP) : HI \models \mathit{body}(R)\}$$

The consequence operator states that "the head atom of a rule must be *true*, if the rule's body is *true*" [GKKS12]. Based on this operator T_{NLP} we define the *consequence sequence* ${}^{HI}T^n_{NLP}$ as [GKKS12][EiIK09]:

$${}^{HI}T^n_{NLP} = \begin{cases} HI & \text{if } n = 0 \\ T_{NLP}\left({}^{HI}T^{n-1}_{NLP}\right) & \text{otherwise } \forall n > 0 \end{cases}$$

For brevity, let the sequence T^n_{NLP} be a short-hand for the default one that starts with the empty set initially ${}^{\emptyset}T^n_{NLP}$. Starting with facts as trivially founded atoms, any rule's head atom $\mathit{head}(R)$ must thus also be regarded as founded, given the rule's body is satisfied $HI \models \mathit{body}(R)$ [EiIK09]. These newly discovered, necessarily true atoms are then used for potentially finding even more founded atoms. The sequence T^n_{NLP} thus attempts to build a larger set of founded atoms with each iteration. However, T^n_{NLP} *converges* at some point [EiIK09], which means that the set of necessarily true atoms does not change anymore. Given that T_{NLP} is monotonic, its *least fixpoint* $lfp(T_{NLP})$, where the operator's input set of founded atoms matches its output set of founded atoms, corresponds to the sequence's convergence point [GKKS12][EiIK09]. This smallest set of founded atoms, that cannot be extended anymore, is exactly the minimal model comprised of necessarily true atoms we are looking for $HMM(NLP) = lfp(T_{NLP})$ [EiIK09].

Least model

Before proceeding onto describing solutions for more complex cases, we first consider a subset of normal logic programs called *positive logic programs* (hereinafter PLP). A PLP only contains positive rules $body(R)^- = \emptyset$ and thus lacks any kind of negation in its rules [GKKS12][EiIK09].

A positive logic program has a unique minimal model, known as the *least model* of the program $HLM(PLP) = HMM(PLP)$ [EiIK09]. This follows intuitively from the definition of the consequence sequence T^n_{PLP}, which in the case of positive rules monotonically derives new head atoms $head(R)$ in the presence of satisfied positive body atoms $HI \models body(R)^+$. Therefore, as each iteration $T^n_{NLP} = T_{NLP}(T^{n-1}_{NLP})$ deterministically discovers new founded atoms, there can only be one such minimal model $HLM(PLP)$.

In this simpler case of positive programs, it is the "best" model [EiIK09], and we consider it the only *solution* $X(PLP) = HLM(PLP)$ to PLP.

Perfect model

Stratified logic programs (SLP) are another subset of normal logic programs, that trade expressive power in favor of being easier to reason about [EiIK09]. In particular, they allow a specific class of programs with negation, in which "one can find an ordering for the evaluation of the rules in the program, such that the value of negative literals can be predetermined" [EiIK09].

In order to determine whether a generic NLP is an SLP and to identify one such evaluation order, a *dependency graph* $dep(NLP) = (V, E)$ is constructed [EiIK09], with:

- V: the set of nodes, containing all program predicates $V = P(NLP)$
- E: the set of labeled arcs (directed edges) between nodes, with an arc $p_h \mapsto p_b \in E$ describing a dependency between a predicate in the head of a rule $p_h = P(head(R))$ on a predicate in the body of a rule $p_b \in P(body(R)^+)$, and $p_h \xmapsto{neg} p_b \in E$ labeling the dependency on a negated predicate body literal $p_b \in P(body(R)^-)$

Given this dependency graph $dep(NLP)$, if a path between two predicates $p_{succ} \xrightarrow{+} p_{pred}$ contains a dependency on a negated predicate $p_j \xrightarrow{neg} p_i$ somewhere along this path $p_{succ} \xrightarrow{*} p_j \xrightarrow{neg} p_i \xrightarrow{*} p_{pred}$, then p_{pred} must be evaluated before p_{succ} [EiIK09]. This guarantees that all negative literals in a rule $body(R)^-$ are evaluated before the rule itself $head(R) \leftsquigarrow body(R)$ can be evaluated. Furthermore, for the program at hand, we can find such an evaluation order and it is thus indeed a SLP "iff in its dependency graph there are no cycles containing a negative edge" [ApBW88].

Given such a SLP, to take this evaluation order into account, a so called *stratification* $\Sigma = \{S_i \mid i \in \{1..k\}\}$ is constructed, in which all program predicates $P(SLP)$ are partitioned into k

non-empty $|S_i| \geq 0$ and disjoint $\forall i \neq j : S_i \cap S_j = \emptyset$ sets of predicates S_i called *strata* [EiIK09], such that:

- if $p_h \mapsto p_b \in E$ then $p_h \in S_{idx_h}$ and $p_b \in S_{idx_b}$ with $idx_h \geq idx_b$
- if $p_h \xmapsto{neg} p_b \in E$ then $p_h \in S_{idx_h}$ and $p_b \in S_{idx_b}$ with $idx_h > idx_b$

If such a stratification can be constructed, the "best" model can be found by computing the *iterative least model* $HILM(SLP)$ [EiIK09].

Let SLP_{S_i} denote rules $R_{S_i} \in SLP$ containing a predicate of that respective stratum S_i in their heads $head(R_{S_i})$ [EiIK09], with $SLP_{S_i} = \{R_{S_i} \in SLP \mid P(head(R_{S_i})) \subseteq S_i\}$. Furthermore, let $HB^*(SLP_{S_j})$ represent all ground atoms HB that can be constructed from predicates of the respective stratum S_j and all previous strata $S_i (\forall i < j)$ [EiIK09], formally $HB^*(SLP_{S_j}) = \bigcup_{1 \leq i \leq j} \{a \in HB(SLP) \mid P(a) \in S_i\}$. Given the stratification $\Sigma = \{S_1, ..., S_k\}$ of SLP, the iterative least models $HILM_i(SLP_{S_i}) \subseteq HB(SLP)\ \forall 1 \leq i \leq k$ can then be evaluated as follows [ApBW88]:

$$HILM_i = \begin{cases} {}^{\emptyset}T^n_{SLP_{S_1}} & \text{(I) for } i = 1 \\ {}^{HILM_{i-1}}T^n_{SLP_{S_i}} & \text{(II) } \forall\ 1 < i \leq k \end{cases}$$

At each step the least model $HILM_i$ is constructed using the intermediate consequence sequence $T^n_{SLP_{S_i}}$, such that this sequence is bootstrapped by the least model of the previous step ${}^{HILM_{i-1}}T^n_{SLP_{S_i}}$. Alternatively, the second case (II) can be expressed in a less operational way, by finding the least subset $HILM_i \in HB(SLP)$ such that $HILM_i$ is a model $HILM_i \models SLP_{S_i}$ and shares at least the same ground atoms as previous iterative least models $HILM_i \cap HB^*(SLP_{S_{i-1}}) = HILM_{i-1} \cap HB^*(SLP_{S_{i-1}})$ [EiIK09].

A stratified logic program can have multiple stratifications Σ [EiIK09] and thus a different amount of intermediary models $k_\Sigma \neq k_{\Sigma'}$, and even those intermediary models can be different $HILM_{k_\Sigma - 1} \neq HILM_{k_{\Sigma'} - 1}$. However, the final iterative least model $HILM_k(SLP)$ is the same for every one of these stratifications Σ [EiIK09], denoted simply as $HILM(SLP)$. As the used evaluation order forces negative literals $\sim a$ to be evaluated before the head of the respective rule [EiIK09], the set of founded atoms can only ever increase with each model iteration, because derived head atoms can never be retracted from this set later. Intuitively, due to this monotonic property of T^n_{SLP}, all stratifications Σ eventually lead to the same "conclusion" $HILM$, after considering the whole spectrum of rules in the program SLP.

In the case of stratified programs, $HILM$ is "the canonical model for [SLP], which is referred to as [the] *perfect model*" [EiIK09]. It is the only *solution* $X(SLP) = HILM(SLP)$ to SLP.

Supported models

In case no such stratification for a logic program NLP exists, one cannot find a sequential evaluation order for rules of that program. That means at some point a non-deterministic *choice* has to be made between multiple alternative evaluation possibilities in order to drive further deterministic *propagation* of founded atoms [GKKS12]. Unlike a *unique intended model* for previous classes of programs, due to this non-deterministic choice, there can be *multiple preferred models* that satisfy NLP [EiIK09]. One variant of multiple preferred models are so-called *supported models*, which are considered the "best" models for a class of normal logic programs considered to be *tight* (hereinafter TLP) [GKKS12].

In order to determine whether a generic NLP is a TLP, a *positive dependency graph* $dep^+(NLP) = (V, E^+)$ is constructed. This graph $dep^+(NLP) = (V, E^+)$ is a looser case of the previously introduced general dependency graph $dep(NLP) = (V, E)$. Namely, it contains [GKKS12]:

- V: the set of nodes, containing all program predicates $V = P(NLP)$
- E^+: the set of arcs between nodes, composed only of positive dependencies between predicates $E^+ = \{p_h \mapsto p_b \mid p_h = P(head(R)) \land p_b \in P(body(R)^+)\}$

Alternatively expressed, it is a sub-graph resulting from negative dependency arcs being omitted from the general dependency graph $dep^+(NLP) = (V, E \setminus \{p_h \overset{neg}{\longmapsto} p_b \in E\})$. If the graph contains no cycles, such that there exists no directed path containing the same node more than once $p \xrightarrow{*} p$, the program at hand is indeed considered a TLP [GKKS12]. Alternatively, all cycles can be found as strongly-connected node-induced subgraphs of $dep^+(NLP)$, where each node is reachable by some non-zero path from every other node [GKKS12], formally $cycles(NLP) = \{V' \mid dep^+(NLP) = (V, E) \land \exists V' \in \mathcal{P}(P(NLP)) : V' \subseteq V \land dep^+(NLP)[V'] = (V', E') \land \forall v_i, v_j \in V' : (v_i \neq v_j \rightarrow v_i \xrightarrow{+} v_j)\}$.

In order to limit the amount of non-deterministic choices, another set of atoms may be considered. Namely, that is the set of necessarily false, *unfounded* atoms according to TLP, that can, in contrast to founded atoms discovered by T_{NLP}, never be part of a particular *canonical model* [EiIK09]. Unfounded atoms can be found iteratively using the *contraposition operator* $F_{NLP} : HB(NLP) \rightarrow HB(NLP)$, which operates on a conceptually negated set of atoms NI, with $\overline{NI}$ being equivalent to $HB(NLP) \setminus NI$:

$$F_{NLP}(NI) = \Big\{a \in HB(NLP) \mid \forall R \in ground(NLP) : head(R) = a \;\Rightarrow\; \overline{NI} \not\models body(R)\Big\}$$

The contraposition operator states that "an atom must be $false$, if the body of each rule having it as head is $false$" [GKKS12]. Based on this operator F_{NLP} we define the *contraposition sequence* ${}^{NI}F^n_{NLP}$ as:

$${}^{NI}F^n_{NLP} = \begin{cases} NI & \text{if } n = 0 \\ F_{NLP}\left({}^{NI}F^{n-1}_{NLP}\right) & \text{otherwise } \forall n > 0 \end{cases}$$

For brevity, let the sequence F^n_{NLP} be a short-hand for the default one that starts with the empty set initially ${}^{\emptyset}F^n_{NLP}$. However, on its own F^n_{NLP} cannot find any necessarily false atoms, unless given inconsistent rules that can never be satisfied $\not\models R$, e.g. of the form $a \leftarrow b, {\sim}b$, for which $F^n_{\{a \leftarrow b, \sim b\}} = \{a\}$.

We thus consider a combined *completion operator* Φ_{NLP} that leverages both T_{NLP} and F_{NLP} in order to limit the amount of non-deterministic choices [GKKS12]. It works on a three-valued interpretation $TUF(NLP) = \langle T, F\rangle$ with $T \subseteq HB(NLP)$, $F \subseteq HB(NLP)$ and $T \cap F = \emptyset$ [GKKS12], as introduced in Subsection 2.4.2. The interpretation $TUF(NLP) = \langle T, F\rangle$ of each ground atom $a \in HB(NLP)$ may thus be considered true $a \in T$, false $a \in F$ or unknown $a \notin T \cup F$ [GKKS12]. $\Phi_{NLP} : TUF(NLP) \rightarrow TUF(NLP)$ is defined as follows [GKKS12]:

$$\Phi_{NLP}\langle T, F\rangle = \langle\ \Phi T_{NLP}(T, F),\ \ \Phi F_{NLP}(T, F)\ \rangle \text{with} \Phi T_{NLP}(T, F) = \Big\{head(R) \mid \exists R \in ground(NLP) : body(R)^+$$

Operator Φ_{NLP} derives a TUF interpretation of atoms $a \in A$ based on logical consequence and contraposition of all other currently deemed to be necessarily $true$ and $false$ atoms.

Analogous to previous definitions, the *completion sequence* ${}^{\langle T,F\rangle}\Phi^n_{NLP}$ is based on this operator [GKKS12]:

$$
{}^{\langle T,F\rangle}\Phi^n_{NLP} = \begin{cases} \langle T, F\rangle & \text{if } n = 0 \\ \Phi_{NLP}\left\langle {}^{\langle T,F\rangle}\Phi^{n-1}_{NLP}\right\rangle & \text{otherwise } \forall n > 0 \end{cases}
$$

For brevity, let the sequence Φ^n_{NLP} be a short-hand for the default one that starts with an empty three-valued interpretation initially ${}^{\langle\emptyset,\emptyset\rangle}\Phi^n_{NLP}$. Although the completion operator and sequence can be applied to general NLPs [GKKS12], the following paragraphs focus on their usage for SLPs in order to find "best" models for such class of programs.

Starting with facts as trivially founded atoms $\Phi T^1_{SLP} = \{a \in HB(SLP) \mid \exists R \in \mathit{ground}(SLP) : \mathit{head}(R) = a \wedge \mathit{body}(R) = \emptyset\}$ and heads of inconsistent rules as trivially unfounded atoms $\Phi F^1_{SLP} = \{a \in HB(SLP) \mid \forall R \in \mathit{ground}(SLP) : \mathit{head}(R) = a \Rightarrow \not\models R\}$, this combined sequence Φ^n_{SLP} tries to find more such atoms with each iteration. Moreover, Φ_{SLP} grows monotonically [GKKS12]. Intuitively, this is due to its respective interpretation sets T and F increasing monotonically as well, based on the respective sub-operator ΦT_{SLP}, ΦN_{SLP} definitions. Once an atom is exclusively part of either T or F, both sub-operators can never detract a previously (un)founded atom from it, since the operator conditions always hold for this atom in later iterations. Namely, with $C^{n-1} \subseteq C^n$ representing either set at the respective iteration, it follows that $\mathit{body}(R)^{\pm} \subseteq C^{n-1} \Rightarrow \mathit{body}(R)^{\pm} \subseteq C^n$ and $\mathit{body}(R)^{\pm} \cap C^{n-1} \neq \emptyset \Rightarrow \mathit{body}(R)^{\pm} \cap C^n \neq \emptyset$. Its *least fixpoint* $lfp(\Phi_{SLP})$, where the operator's input interpretation of (un)founded atoms matches its output interpretation, corresponds to the sequence's $\Phi^n_{SLP} = \langle T, F\rangle$ first point where no further deterministic propagation of atoms can take place. It serves as the base interpretation for finding all further fixpoints via non-deterministic choice [GKKS12].

A non-deterministic choice assigns, in the case of three-valued interpretations $TUF = \langle T, F\rangle$, a truth value to an atom a that is not yet part of this *partial* interpretation $\exists a \in HB(SLP) : a \notin T \cup F$ [GKKS12]. Therefore, a new atom $a \notin T \cup F$ is chosen and both resulting interpretations $TUF' = \langle T', F'\rangle$, where that atom a is either assumed true $TUF'^{+} = \langle T \cup \{a\}, F\rangle$ or false $TUF'^{-} = \langle T, F \cup \{a\}\rangle$, are further explored [GKKS12]. Given that a resulting interpretation TUF' is still a model for the program $TUF'\langle T', F'\rangle \models SLP \Leftrightarrow T' \models SLP$, the completion sequence "branches off" for this non-deterministic choice ${}'\Phi^n_{SLP} = \langle T', F'\rangle$ and is continued until the next fixpoint $fp(\Phi_{SLP})$ is reached, when another non-deterministic choice is needed or all atoms have already been assigned [GKKS12]. This process is repeated, until all ground atoms a are assigned to such a *total* interpretation $\forall a \in HB(SLP) : a \in T \cup F$ [GKKS12].

In the case of tight programs, all possible total interpretations from the completion operator sequence "leaves" $TUF_i = \langle T_i, F_i\rangle = {}_i\Phi^n_{SLP}, \forall a \in HB(SLP) : a \in T_i \cup F_i$ are thus obtained. These total interpretations are known as *supported models* $HSM_i(SLP) = T_i$ [GKKS12], and represent all solutions $X_i(SLP) = HSM_i(SLP)$ to the program SLP at hand.

Stable models

In case a program contains both a positive and a negative circular dependency in $\mathit{dep}(NLP)$, it cannot be assigned to any previous specialized class of programs, and is thus treated as a generic NLP. Analogous to the case of supported models, there can be multiple preferred models which are evaluated non-deterministically. However, this variant of multiple preferred models is referred to as *stable models*, which are considered the “best” models for any generic NLP [EiIK09]. The difference to supported models lies in the existence of these positive dependency cycles contained in NLP. Positive circular dependencies lead to “circular derivations” in NLP, where an atom is (partially) derived from itself [GKKS12]. In order to find a proper preferred model for programs containing such *self-supported* atoms [EiIK09], there must be additional *external support* ES for that atom to be considered for inclusion into the founded set and subsequently for further propagation of founded atoms [GKKS12]. Otherwise, if such an external support ES is missing, that atom needs to be part of the unfounded set and thus excluded from every model of the program NLP [GKKS12][EiIK09]. The set of rules facilitating external support for a set of atoms A is thus formalized as $ES_{NLP}(A) = \{R \in ground(NLP) \mid head(R) \in A \land body(R)^+ \cap A = \emptyset\}$ [GKKS12].

However, the operational formalization of this extended unfounded set requires another concept to be explored first. A *Gelfond-Lifschitz reduct* (just *reduct* for short) NLP^{HI} of a program NLP with respect to an interpretation HI is a positive logic program obtained by incorporating (not) entailed atoms by the interpretation $c \in HI$ into the program, thereby “enforc[ing] truth values for negative literals” [EiIK09], and thus “reducing” it into a more compact form. NLP^{HI} is constructed as follows [EiIK09][GKKS12]:

- remove all rules from the program $R' = NLP \setminus R$, for which a negative literal $\sim c$ in the body $c \in body(R)^-$ is not entailed by the interpretation $HI \not\models \sim c \Leftrightarrow c \in HI$
- simplify the remaining rules R' by removing all negative literals $\sim c$ from the body $body'(R')^- = body(R')^- \setminus \{c\}$, because their negations are entailed by the interpretation $HI \models \sim c \Leftrightarrow c \notin HI$

Intuitively, an interpretation HI can be seen as an assumption about the truth values of negated literals, and the respective reduct NLP^{HI} as the (positive) consequences drawn from incorporating that assumption [EiIK09]. As the reduct NLP^{HI} is a positive program, there exists a unique minimal model of its necessarily true atoms $LM(NLP^{HI})$ [EiIK09]. Therefore, HI does not *contradict* NLP^{HI} if the necessarily true consequences match each other $HI = LM(NLP^{HI})$ [EiIK09]. Alternatively expressed, “[HI] can be reconstructed from scratch by applying the rules of [NLP^{HI}]” [EiIK09], such that “negative literals must only be true, while positive ones must also be provable” [GKKS12]. Models $HM_i(NLP)$ that do not contradict its reduct $HM_i = LM(NLP^{HM_i})$ are known as *stable models* $HSTM_i(NLP) = HM_i(NLP)$ [LeRS97][EiIK09][GKKS12] and thus represent the “best” solutions to any generic normal logic program NLP at hand $X_i(NLP) = HSTM_i(NLP)$.

Although, from a semantic standpoint, it suffices to compare all possible models $HM_i(NLP)$ against their reducts NLP^{HM_i} to find all stable models of NLPs $HSTM_i(NLP)$ [EiIK09], there exists an operational-focused approach on how to find such models $HSTM_i(NLP)$ more efficiently, as for previous specialized classes of programs. The *strengthened completion operator* Ω_{NLP}[18] leverages both the operator for finding founded atoms T_{NLP} and a now strengthened operator for finding unfounded atoms ${U^{RI}}_{NLP}$, which additionally discovers self-supported atoms missing external support [GKKS12]. The particularities of this operator Ω_{NLP} are out of scope of this work and are discussed in appropriate literature, e.g. Gebser et al's book [GKKS12].

2.4.4 Reasoning

Although, from a formal standpoint, it suffices to evaluate all fixpoints of Ω^n_{NLP} to find all stable models of a logic program NLP, we touch on real-world operational-focused reasoner approaches on how to find such models $HSTM_i(NLP)$ more efficiently, as described in the introductory subsections. Afterwards, involved computational complexities are explored, before we conclude this Section with various types of reasoning tasks a user can execute via a reasoner.

Grounding

Typical reasoner implementations split their work into two parts —the *grounding* and the *solving* step [EiIK09][GKKS12].

In the first step, a (potentially) non-ground normal logic program with variables NLP is converted into grounded one without any variables $NLP' = \mathit{ground}(NLP)$. The goal is thus to produce "a finite and succinct propositional representation of a first-order program" [GKKS12]. This step is needed due to typical solvers implementations operating on such propositional encodings, as seen in the next Subsection. Importantly, the grounded program NLP' generates the same stable models as its non-ground version NLP [EiIK09][GKKS12]. However, some restrictions do apply on the (non-ground) program syntax so that its corresponding "ground program [NLP'] is small and easy to evaluate" [EiIK09].

In particular, a program NLP containing only *safe* rules is guaranteed to have a finite ground counterpart NLP' in the absence of function symbols of non-zero arity [GKKS12]. A rule $R \in NLP$ is safe, if all of its contained variables $Var(R)$ appear in a positive body atom $b \in \mathit{body}(R)^+$ of that rule $\forall v \in Var(R) : v \in Var(\mathit{body}(R)^+)$, bar a relaxation of this restriction surrounding built-ins (e.g. $=$) [EiIK09][GKKS12].

However, rule safety alone does not guarantee that the corresponding ground program is finite. Further restrictions need to be defined for the usage of function symbols [GKKS12], to reject even safe programs of e.g. the form $p(a).\ p(f(X)) \leftarrow p(X).$, which result in infinite grounding [KLPS16][LiLi09]. There exist multiple variants of these restrictions on the syntax of normal logic program, in order for them to be provably *finitely ground* [KLPS16]. The restrictions of a general finitely ground syntactic subset of NLPs are described in Calimero et al's work [CCIL08]. However, checking whether a generic NLP is finitely ground is semi-decidable [KLPS16]. Therefore, Lierler and Lifschitz propose an *argument restricted* subset of NLPs, which is decidable [LiLi09]. Moreover, any NLP without function symbols in rule heads is trivially argument restricted and thus finitely ground [LiLi09]. *Domain restricted* rules require that their variables also occur in *domain predicates* as part of positive body literals. "The domain predicates of [NLP] are the predicates that are not defined in terms of negative recursion or using choice rules" [SyNi01] and are thus deterministically inferred using a predetermined evaluation order. However, as these domain predicates are stratified, the domain predicates in the body necessarily need to belong to a lower stratum than that of the rule head [SyNi01]. As an aside, domain

[18]Operators Φ_{NLP} and Ω_{NLP} are both referred to as *completion operators* in literature. We slightly adapt the nomenclature of the latter in order to distinguish one from the other in both textual and symbolic notation.

predicates are the recommended way of specifying the antecedent of nested implications, so that these condition literals can be resolved and simplified during grounding [GKKL$^+$15].

In practice, we have not encountered issues with infinite grounding while leveraging function symbols as a knowledge modeling capability [KLPS16]. Source code Listing 3.7 shows an example usage of function symbols for this purpose.

Solving

In the following, step solvers take the ground program $NLP' = \mathit{ground}(NLP)$ as input and output any found stable models $HSTM_i(NLP') = HSTM_i(NLP)$ [EiIK09][GKKS12].

Basic solver implementations conceptually use a *tableux* calculus [GKKS12], which shares the same base tree structure as the tableux method used in propositional and first-order logic (see Subsection 2.2.3 and 2.2.4, respectively). This calculus describes syntactic inference rules for the construction of stable models of NLPs [GKKS12].

More recent solvers use a set of *nogoods* $\Delta_{NLP} = \{\delta_1, \delta_2, ..., \delta_n\}$ where each nogood $\delta_i = \{\sigma_1, \sigma_2, ..., \sigma_m\}$ captures an inadmissible combination of value mappings $\sigma_i = v \mapsto T/F$ in an assignment $\mathcal{X}_{\Delta_{NLP}}$ [GKKS12]. Therefore, an assignment $\mathcal{X}_{\Delta_{NLP}}$ can be a solution only if it contains no such nogood $\forall \delta \in \Delta_{NLP} : \delta \not\subseteq \mathcal{X}_{\Delta_{NLP}}$ [GKKS12]. Given the ground program NLP and the set of nogoods for that program Δ_{NLP}, a modern solver can then compute the program's stable models $HSTM_i(NLP)$. Namely, since the set of nogoods Δ_{NLP} can be understood in terms of propositional logic formulas in conjunctive normal form [KLPS16], modern boolean constraint satisfiability solvers (hereinafter SAT solvers) are, with some adaptations, leveraged to compute solutions $\mathcal{X}_{\Delta_{NLP}}$ satisfying these nogoods [GKKS12].

A more in-depth explanation on modern ASP solvers is given in appropriate literature, e.g. Gebser et al's book [GKKS12].

Computational complexities

The characteristics of the underlying operational procedures of answer set programming now give rise to the involved computational time complexities.

In general, NLPs are efficiently verifiable by a deterministic Turing machine (`PTIME`-complete), that is, deciding whether a (guessed) interpretation $HI(NLP)$ is a stable model $HSTM(NLP) \overset{?}{=} HI(NLP)$ takes a polynomial amount of time steps [EiIK09][GKKS12]. Intuitively, this is because stable model checking boils down to the comparison of an interpretation with the least model of the program's reduct under that interpretation $HLM(NLP^{HI})$, which in itself computable in polynomial time [DEGV01]. Moreover, variable-free, ground NLPs are efficiently solvable by a non-deterministic Turing machine (`NPTIME`-complete), that is, deciding whether a NLP has a stable model $HSTM(NLP)$ (and thus finding such model) takes polynomial time steps, given that the Turing machine can always make the correct non-deterministic choice between multiple alternatives [EiIK09][GKKS12]. Intuitively, this is because a non-deterministically guessed model can be checked in polynomial time [EiIK09], as described before. This time complexity is also evident when looking at the operational characteristics described in previous subsections. Knowing the correct non-deterministic choice before each further deterministic propagation would result in no more than such polynomial amount of steps until an assignment's leaf, unlike the (potentially) exponential amount of steps $2^{HB(NLP)}$ required to "blindly" search the whole assignment tree through all non-deterministic branches.

However, non-ground NLPs display higher time complexities [GKKS12][EiIK09]. As a ground instantiation $NLP' = ground(NLP)$ is usually exponential in size of its corresponding non-ground program NLP [GKKS12][EiIK09], finding a stable model of NLP' is `NEXPTIME`-complete. By allowing the full expressiveness of syntactic constructs involving function symbols, solving such NLP is even undecidable [EiIK09]. However, with appropriate restrictions on the usage of function symbols, as discussed in the Subsection 2.4.4 on grounding, the complexity can still be limited to `N2EXPTIME` for stricter restrictions [EiIK09].

Due to the predetermined evaluation order without any non-deterministic choices in positive and tight (stratified) logic programs, finding a stable model for such ground PLPs and TLPs takes only a polynomial amount of time on a corresponding deterministic Turing machine and is thus `PTIME`-complete [EiIK09][GKKS12]. However, ground instantiations of logic programs with variables and (restricted) function symbols may still lead to exponential `EXPTIME` and `2EXPTIME` complexities, respectively [EiIK09].

Optimization statements additionally raise the complexity bounds. Namely, checking whether an interpretation HI is an optimal stable model is `co-NPTIME`-complete [GKKS12]. This follows from the semantic formalization of optimization statements given in the syntax Subsection 2.4.2, as a reasoner needs to check whether an optimal stable model X_{opt} dominates every other stable model X_{sub}. Finding an optimal stable model, if it exists, is $\texttt{P}^{\texttt{NP}}\texttt{TIME}$-complete [GKKS12], that is, solvable by a deterministic Turing machine in polynomial time with an oracle for solving problems in `NP` time [EiIK09]. Such optimization problem can be alternatively understood as "any polynomial number of queries, and in fact queries computed adaptively, based on the answers of previous queries" [Papa94]. Solving non-ground NLPs with optimization statements may lead to an exponentially larger workload as in the previous cases, such as characterized by e.g. $\texttt{P}^{\texttt{NEXP}}\texttt{TIME}$ [GKKS12] and $\texttt{P}^{\texttt{N2EXP}}\texttt{TIME}$.

Types of reasoning

In conclusion, answer set programming supports different reasoning modes [GKKS12], by solving the corresponding decision, search and optimization problems:

- *satisfiability*: finding a stable model
- *unsatisfiability*: showing that no stable model exists
- *enmuration*: finding all stable models
- *optimization*: finding the most optimal stable model(s)
- *cautious reasoning*: finding atoms that are true in all stable models [EiIK09]
- *brave reasoning*: finding atoms that hold in at least some stable models [EiIK09]

2.5 Domain-specific modeling

This Section gives a short background on domain specific modeling (DSM), particularly in the scope[19] of letting users define their business requirements using a domain-specific modeling language (hereinafter DSML) in a domain-specific modeling environment (hereinafter DSME). The

[19]Domain specific modeling is a term used to refer to the software engineering practice, where domain-specific models are used to drive code generation for applications and services that adhere (preferably) exactly to the

introductory subsections describe general characteristics and concepts about DSM, DSMLs and DSMEs, whereas the final one introduces `WebGME`, a meta-modeling language and accompanying tooling used for modeling such business requirements in this thesis' implementation.

2.5.1 Domain-specific modeling

"DSM raises the level of abstraction beyond current programming languages by specifying the solution directly using problem domain concepts" [KeTo08].

Traditionally, in code-driven development, stakeholders agree on higher-level-of-abstraction models, after which developers implement the specific executable code according to such specification [KeTo08]. This process is ideally iterated upon, until the models and implementation match the requirements. However, this manual process of keeping model and code in sync is a duplication of effort, "tedious and error prone" [KeTo08]. In model-driven development, models depicting requirements can directly influence the final application's implementation by generating code from these models [KeTo08]. Instead of writing code in a general-purpose programming language and then having the compiler generate specific machine instructions, the models themselves serve as the "source" that is translated all the way down into executable artifacts [KeTo08].

2.5.2 Domain-specific modeling language

However, model-driven development requires the implementation of a code generation layer and similar tooling. Due to the narrower focus and semantic scope of DSMLs on particular domain concepts [FrRu05], code generation for models made in domain-specific modeling languages is realistically achievable [KeTo08]. In contrast, having to cover all possible scenarios that are capable of being expressed with general-purpose modeling languages, like the Unified Modeling Language, full code generation is "difficult, if not impossible" [KeTo08]. As an additional benefit, DSMLs represent domain concepts directly in the model itself so that users can understand such models more easily [FrRu05], whereas in a general-purpose modeling language one needs to map domain concepts to (a multitude of) design concepts instead [KeTo08]. "This leads to the claim that domain specific models are better suited for requirements engineering than the [Unified Modeling Language]" [FrRu05].

Formally, the essence of any modeling language L can be described as "a set of usable expressions as well as rules for their composition", such that "well-formed composed expressions define a program that may be executed" [GNTG$^+$07]. In particular, a language $L = \langle C, A, S, M_c, M_s \rangle$ is defined by a concrete syntax C (which may either be in textual or graphical form [GNTG$^+$07]), an abstract syntax A, the semantics S of a program's execution, the syntactic mapping $M_c : C \to A$ from the concrete syntax C to the abstract syntax A, as well as the semantic mapping $M_s : A \to S$ from the abstract syntax A to the semantics S [GNTG$^+$07]. Such language further defines allowed compositions of abstract expressions (M_s), execution errors (S), as well as syntactic constraints (A) [GNTG$^+$07]. Therefore, a DSML introduces domain concepts in either or both of its syntax sets C and A [GNTG$^+$07].

requirements expressed in such models [KeTo08]. In this work, we use domain specific modeling languages to enable domain experts to express their business requirements for parts of the application during the application's runtime, which is thus distinctly different in scope.

2.5.3 Domain-specific modeling environment

In an effort to make the development of DSMLs more systematic, meta-editors and similar graphical editor environments have been developed to assist engineers in the creation of graphical DSMLs and accompanying tooling [FrRu05]. These meta-editors thus allow the creation of domain-specific modeling languages, as well as domain-specific modeling environments [GNTG+07].

DSMEs are to DSMLs what integrated development environments are to general-purpose programming languages. That is, they assist developers in using DSMLs, such as enforcing composition & association rules and other syntactic and semantic constraints of the DSML, with the goal to build domain models using the language's concrete syntax [GNTG+07]. These models are thus "*syntactically correct-by-construction*" [GNTG+07].

A *DSME* typically offers the ability to [GNTG+07]:

- *Metamodeling*: view & change the DSML's metamodel, such as syntactic elements, constraints and semantics; the metamodel dictates how a user can interface with the DSME and what kind of domain-specific models they can create via the DSML
- *Views*: view specific subsets and representations of the model, based on the user's responsibility in order to e.g. let the user focus on parts they are interested in
- *Serialization*: persist models via serialization
- *Versioning*: version and track changes between different model versions, as well as automatic synchronization of changes between views
- *Artifact generation*: generate and produce different artifacts based on models, such as by code generation or other compilers that receive the model as an input and transform it into other artifacts
- *Plugin support*: run third-party tools in the editor, like simulations and customized model validations

2.5.4 Generic Modeling Environment for the Web

The Generic Modeling Environment for the Web (`WebGME` for short) is such a meta-editor that allows the creation of customized DSMEs and corresponding DSMLs, primarily through the specification of the domain-specific metamodel [MKKB+14].

At its core lies `WebGME`'s meta-metamodel, which defines the basic conceptual building blocks that can be used for the creation of such domain-specific metamodel [MKKB+14]:

- *objects* are used to depict domain concepts
- *`Root`* is a special object representing the root of the *composition* hierarchy
- *`FCO`* (first-class object) is another special object representing the root of the *inheritance* hierarchy
- *pointers* model directed, named associations between two objects, and restrict their allowed association target types

- *connections* are syntactic sugar for visualizing directed, named associations that carry additional information; these are ordinary objects containing two pointers, particularly named *src* and *dst*
- *sets* depict named associations between one object and an unordered set of other objects, and have no restriction on the association target type
- *attributes* associate additional information with objects, in the form of properties
- *aspects* capture views on different subsets of an object's children

"*Hierarchical decomposition* is the most widely used technique to handle complexity" [MKKB+14]. That is why objects may contain other objects as children, as part of such composition hierarchy starting from `Root`. On the other hand, "[*prototypical inheritance*] is [another] very powerful way to help the modeler handle the inherent complexity in large models and intricate DSMLs" [MKKB+14]. At the base of such inheritance tree lies the `FCO`, from which every other object inherits from, per default. Otherwise, any other object can serve as the prototype for any instance object inheriting from it, whereas such instances can themselves be prototypes for yet other instance objects inheriting from them [MKKB+14]. In contrast to typical prototypical inheritance in object-oriented programming languages, the instance object becomes a deep copy of the prototype's associations and properties [MKKB+14]. This also includes any children of the prototype down the composition hierarchy, which are cloned and set to inherit from the prototype's respective children, resulting in the instance object containing such deeply-cloned composition tree [MKKB+14]. Furthermore, any changes made to a prototype are immediately propagated down the inheritance tree e.g. for any properties that are not overwritten in the instances themselves [MKKB+14].

This approach facilitates a form of multi-level metamodeling, such that each prototype-instance inheritance corresponds to a metamodel-model relationship [MKKB+14]:

- changes to a metamodel are propagated immediately to its models
- constraints, rules and other metamodeling information can be refined further at any level in the inheritance and composition hierarchies
- partially realized domain objects can become abstract building blocks on a more specialized modeling level
- different (meta)model variations and versions can exist side-by-side in the same specification

`WebGME` also supports *cross-cuts*, where objects from different levels of the composition hierarchy can be arranged on the same view [MKKB+14]. One such out-of-the box cross-cut is the `metamodel` one. All models (objects) and their associations (pointers, sets) from any composition level, which are relevant for the specification of the DSML, are thus added in this view. Meta-information can only be edited in this metamodel overview [MKKB+14]. In addition, meta-information, constraints and rules can be *mixed in* from other objects in this view, thus enabling additional modularity during metamodeling.

`WebGME` supports the typical features of a DSME, such as editor plugin support, as well as code generation and other features. How the metamodeling side influences the structure and semantics of the corresponding DSML, as well as the interface elements of the DSME, is best viewed on a concrete example, such as in Section 3.1.8 on the DSML approach. A more in-depth overview of `WebGME` is furthermore given on the project's webpage[20].

[20]See `https://webgme.org/` for more details.

3 Approach

This Chapter summarizes the (non-exhaustive) set of different approaches, which can be used to solve the problem statement. Afterwards, we settle on an appropriate approach for the complete system implementation.

As described in the introductory Chapter 1, the problem statement can be split into two distinct parts:

F1 Design-time representation of valid workflows through task ordering and additional restrictions

F2 Run-time detection of potentially invalid workflows, as well as automated repair of such inconsistent workflows

A workflow (model) is a bounded linear sequence of tasks[1]. In the former part, domain experts create a workflow meta-model which captures all ground rules for any such custom-tailored or template workflow (model). Experts list all possible tasks, which tasks may follow other tasks, as well as mandatory tasks that must appear alongside other ones. In the latter part, an automated procedure checks a custom-tailored (or even template) workflow against those ground rules, whether it contains unknown tasks, whether the tasks adhere to the specified order and whether mandatory dependencies are included in the workflow. If the workflow is invalid, the "best" valid alternative is recommended instead. Otherwise, the checked workflow is indeed a valid model of the workflow meta-model. This workflow (model) may then be executed multiple times, with each execution being referred to as a workflow (model) instance.

3.1 Workflow meta-modeling

In this Section, we look at different approaches and how they can be used to solve the former part of the problem statement. This non-exhaustive set of approaches is composed of mainstream technologies commonly found in literature on workflow management and business process modeling [AaHe04][Wesk12][HeSW13], as well as of some alternative process modeling techniques. To this end, we start by exploring an example failure analysis process and then proceed onto primarily (control-)flow-based process models for representing such workflows. Afterwards, we identify domain-specific concepts and explore how constraint-based domain-specific models can be used to capture task ordering and similar restrictions, instead. Each approach is accompanied by a brief summary on its qualitative aspects —its pros and cons —discovered during the modeling of the example FA process using that respective approach. Note that these summaries

[1]In failure analysis, these linear sequences *typically* consists of no more than 15 tasks.

correspond to our own experiences and findings, and are not as elaborate or as thorough as proper case studies.

3.1.1 Problem description

In order to be able to compare the different approaches that will be outlined in the following sections, we model and analyze the same example using each approach. For reference, we use a failure analysis process of integrated circuits encountered in the communication industry [MLBZ16]. For our example we use an altered version of the reference analysis process, by tying the process steps to specific tools if applicable, by loosening the restrictions on the ordering and repetition of some process steps, or by otherwise introducing additional artificial constraints for demonstration purposes.

The example failure analysis process is thus structured as follows:

- Before opening up the chip, an external visual inspection (`EVI`) may be performed, to determine whether the sample is soiled outright.
- Afterwards, the chip's internal structure may be inspected using the scanning acoustic microscope (`SAM`) or the X-ray diffraction microscope (`XRAY_MIC`), or both in any order.
- As the final, non-destructive task, the sample may be subjected to an electrical performance test (`EPT`) in order to identify any short circuits or leakages.
- The chip's encapsulation may be removed by mechanical or chemical means (`DECAP`).
- Afterwards, an internal visual inspection (`IVI`) may reveal any abnormal phenomena.
- Following is an optional, iterative sub-process, whereas layers are subsequently stripped (`STRIP`) to be then analyzed by scanning electron microscopy (`SEM`), transmission electron microscopy (`TEM`), or both in any order.
- Immediately after layer stripping in this iterative sub-process, an X-ray energy spectroscopy (`XRAY_SPEC`) may be performed to identify any foreign bodies in the composition.

There are, however, additional constraints that need to be considered when analyzing the example failure analysis process:

- To allow a wide variety of custom-tailored workflows, all process tasks are optional. However, the order of FA tasks commissioned by the client must adhere to the structure above.
- Necessarily, all non-destructive tasks must be carried out before any destructive tasks.
- In addition to the basic partial order imposed by the structure, we introduce hard dependencies between some tasks:
  - An internal visual inspection cannot be performed if the chip has not been decapsulated before. Vice versa, an internal visual inspection is (almost always) mandatory, to reveal any potential problems prohibiting further analysis. These two tasks are encountered in tandem.
  - Likewise, any layer stripping or subsequent analysis must be carried out on an opened chip.

An example workflow adhering to this FA process is the task sequence $\langle EVI, SEM, EPT \rangle$.

3.1.2 Business Process Model and Notation

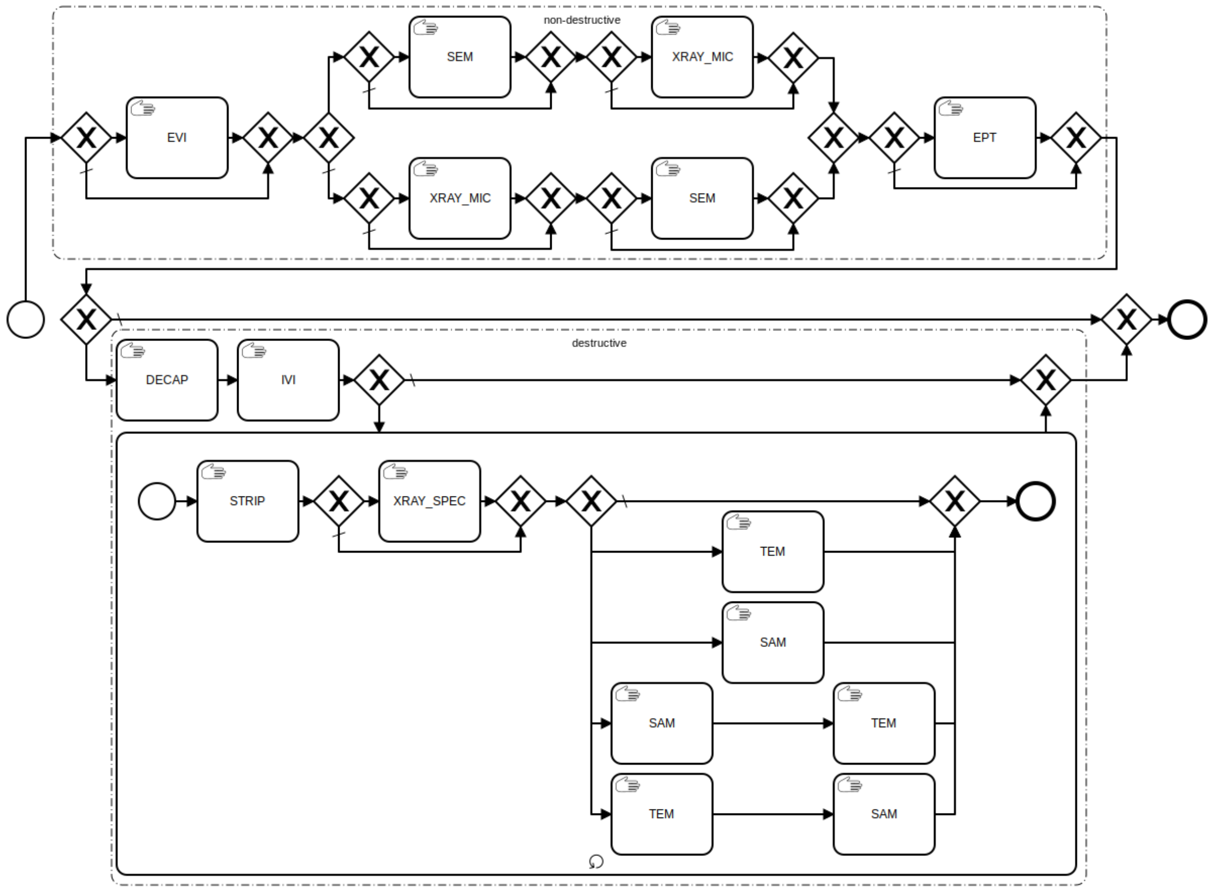


Figure 3.1: Example FA process modeled using BPMN

Figure 3.1 shows a complete and sound BPMN model[2] of the example failure analysis process given in Section 3.1.1. Due to the optional nature of the majority of tasks, the model includes many exclusive gateways which allow skipping over those optional process steps. The manual tasks are aggregated into the respective group depending on whether they irreversible alter the analyzed sample or not. The iterative nature of stripping and analyzing parts of the chip is reflected in the contained sub-process, which can be repeated as many times as needed.

Moreover, following the default gateway connections, the FA business process results in an empty workflow, without any tasks between the start and end node, in that most trivial case. Otherwise, all possible and valid process executions, according to the example definition, are represented in this BPMN model.

Summary:

[2]The decision which path to take must be supplied on starting the workflow by adding appropriate process variables. Each exclusive gateway then performs the routing based on those values. Ad hoc decisions are not possible, unless the BPMN model is extended by prepending user tasks that make those decisions before each exclusive gateway. Furthermore, the loop count of the iterative sub-process must be supplied beforehand. Alternatively, a user task signaling the need for further iterations can be placed before each loop.

+ easy to design, since modeling is simple and evolves around connecting tasks via exclusive gateways solely
+ easy to understand, due to explicit control-flow via gateways
+ good tool support via variety of open-source and proprietary products
− explodes in size when modeling many optional tasks and tasks with loose ordering requirements

3.1.3 Case Management Model and Notation

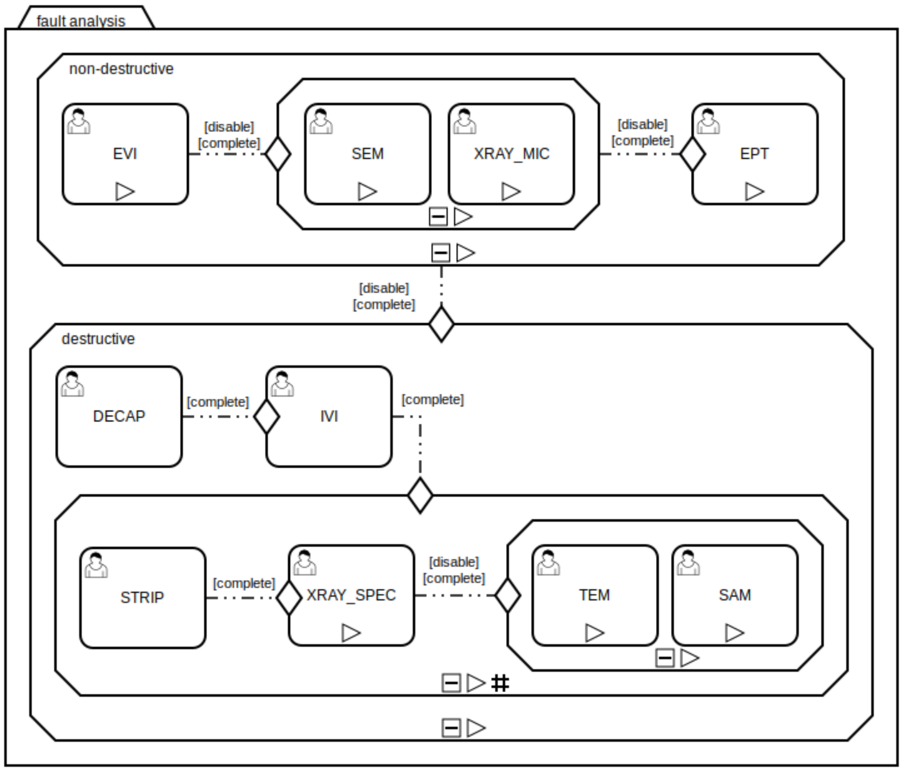


Figure 3.2: Example FA process modeled using CMMN

Figure 3.2 shows a complete and sound CMMN model of the example failure analysis process given in Section 3.1.1. Optional tasks and stages are marked with manual activation (▷), allowing them to be `activated` and `completed` or skipped by being `disabled`. After reaching one of those (semi-) terminal states in the predecessor elements the same decision can (only) then be made for the successor elements. For example, after `activating` the stage containing `SEM` and `XRAY_MIC`, the user may perform any or neither of those tasks to `complete` the stage and continue the FA process. The *destructive* stage, if `activated`, contains the then mandatory task sequence `DECAP` → `IVI`. The repeated layer stripping and respective analyses are modeled using the appropriately marked sub-stage.

Moreover, by `disabling` all relevant tasks and stages, an empty failure analysis process is created, in that most trivial case. Otherwise, all possible and valid process executions, according to the example definition, are represented in this CMMN model.

Summary:

– harder to design, because more modeling elements are being used

– harder to understand, because the user needs to know about all these different elements and the lifecycle phases the tasks go through [ABSK^{+}20]

– limited amount of tools available[3]

+ notation allows for concise representation of many loosely ordered and optional tasks

3.1.4 Workflow nets

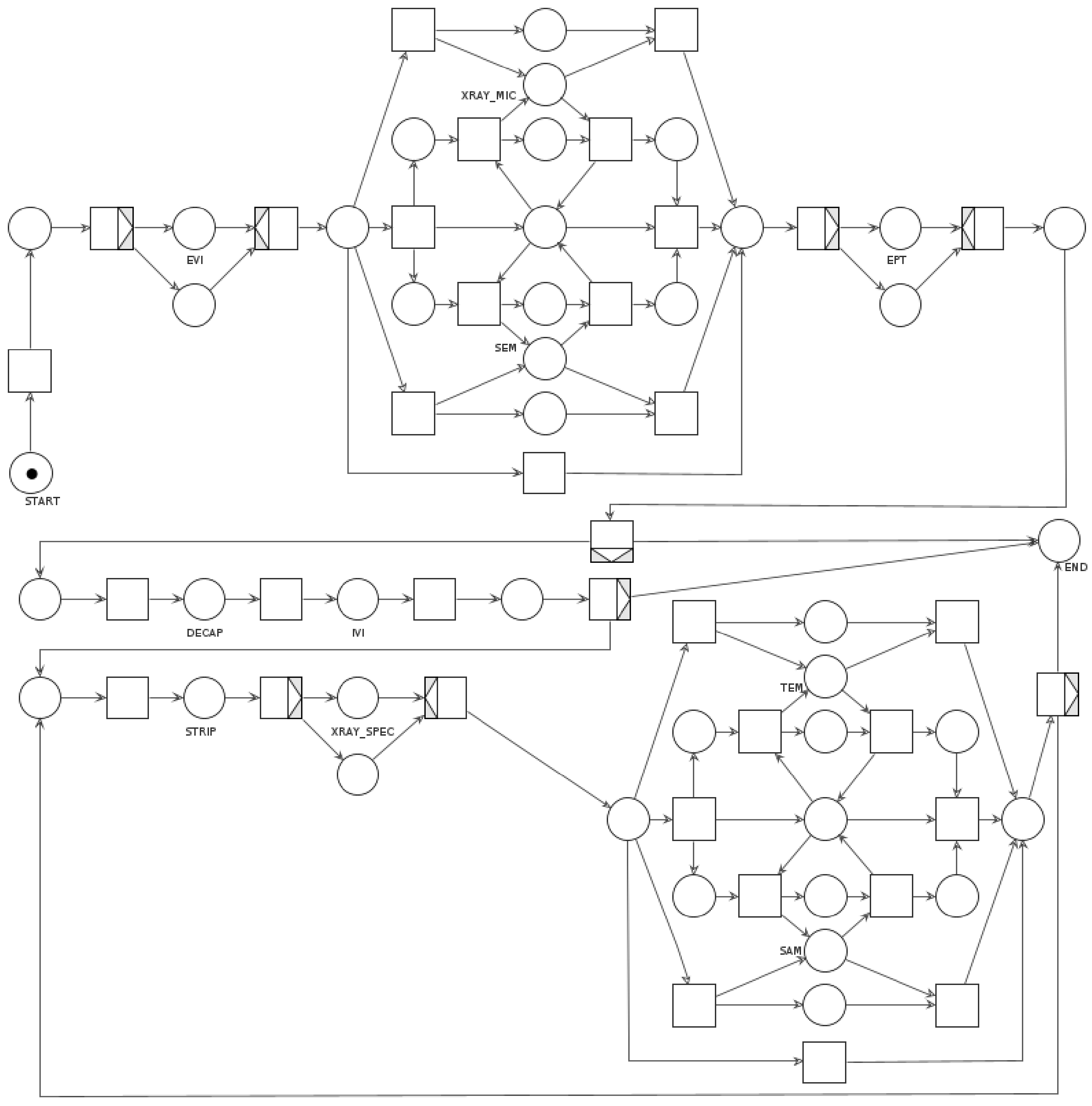


Figure 3.3: Example FA process modeled using Petri Net

[3]To our knowledge, only the workflow management engine Camunda (`https://camunda.com/`) allows the execution of CMMN models.

Figure 3.3 shows a complete, safe, sound and executable[4] workflow net of the example failure analysis process given in Section 3.1.1. The workflow net consists of regular transitions, as well as exclusive-choice splits (transitions containing the $\rhd$ symbol) and joins (transitions with the $\lhd$ symbol) [AaHe04], with the latter ones being used to model optional tasks. Failure analysis activities are modeled as named places in the WFN, which emphasizes their ongoing nature. A token in the place `EVI` represents the respective FA task currently being executed on a tool. On the other hand, unnamed places depict auxiliary states and facilitate the workflow's control flow. Furthermore, named places are unique, which means no other place can govern the execution of that specific FA task. A special, sequential, multiple-choice construct has to be for that reason used to model all possible execution paths for a set of optional tasks, like for `XRAY_MIC` and `SEM`. This is done on purpose for easier reasoning about task ordering, despite loosing well-structuredness and the free-choice characteristic of the workflow net[5]. Namely, the task ordering is induced by all the possible paths between nodes containing a marking on a named place in the workflow net's reachability graph. The iterative nature of the layer stripping is modeled as well, using a simple cycle pattern[6].

Moreover, by firing all transitions that lead to a single, unnamed place, an empty failure analysis process is created, in that most trivial case. Otherwise, all possible and valid process executions, according to the example definition, are represented in this Petri Net Markup Language (PNML) [Frey05] model.

Summary:

– harder to design, because modeling of optional tasks involves complex patterns such as the multiple-choice construct

○ easy to understand basic control-flow, yet hard to immediately gain an overview over all possible execution paths without running simulations by hand or in tools

+ research-level tools available

– the necessarily complex constructs for loosely ordered and optional tasks lead to a blow-up of model size

3.1.5 Problem domain

The previous approaches leverage flow-based process-oriented modeling languages to model the example failure analysis process, its tasks and the (partial) order and hard dependencies between those tasks. As motivated in background Section 2.5.2, we now consider the direct use of domain concepts as building blocks of such process models instead.

Concepts

To this end we analyzed the example failure analysis process, as well as sat down with experts from Infineon's FA lab and identified domain concepts, their properties and associations in

[4]Soundness verified and execution tested via WoPeD (`https://woped.dhbw-karlsruhe.de`), a workflow Petri net designer [Frey05].

[5]There exist other properties of PNs like well-structuredness and free-choice. Although well-structuredness is a good rule-of-thumb for the construction of sound workflows, it is not a necessary condition for the soundness of a PN [DeZi04]. Many properties of free-choice PNs can be decided in polynomial time, while the same does not hold for not free-choice nets [AHHS+11]. Detailed explanations for these and other not mentioned properties are out of scope of this work. For further details see the referenced articles.

[6]For more workflow patterns see `http://www.workflowpatterns.com`.

processes of *typical* production environments. In the following, we formalize parts of the problem context and problem description introduced in Chapter 1 and Section 3.1.1, respectively:

Tasks represent the main building blocks of every workflow. They can belong to the *non-destructive* group of tasks, which must be performed before any *destructive* tasks. Alternatively, they may be classified into *both* groups, such that no such general order is imposed between these tasks. Tasks can optionally be *repeated* as many times as desired. A partial order and hard dependencies between tasks can be established through use of directed *connections*:

- a *has_successor* connection from task A to B indicates that task A may not be performed after task B, given that they are part of the same sub-process iteration
- a *has_predecessor* connection from task B to A states that task B may not be executed before task A, given that they are part of the same sub-process iteration; it is syntactic sugar to the inverse connection A *has_successor* B
- a *has_mandatory_successor* connection from task A to B stipulates that task B must be done after A, in the same sub-process iteration; it is a specialization of *has_successor*
- a *has_mandatory_predecessor* connection from task B to A declares that task A must be carried out before B, in the same sub-process iteration; it is a specialization of *has_predecessor*; note that this connection is **not** the inverse to *has_mandatory_successor*

Finally, a job is thus a *workflow* consisting a sequence of tasks that must adhere to the stipulated problem domain. Each execution of such job in the lab is thus considered an *instance* of this workflow.

The presented modeling concepts of tasks and connections are to a great extent inspired by Grambow, Oberhauser and Reichert's findings and observations [GrOR11] for solving a similar problem in their business domain. The similarities and differences between this and the referenced approach are discussed in detail in related work Section 6.1.2[7].

Semantics

Let us further explore the implicit, not-so-apparent properties and implications of these connections. The partial order between tasks is induced by these connections transitively. If a task is not transitively connected to another task, then those two tasks may be performed in any pair-wise order. Otherwise, they must be executed according to this transitive ordering. With A *has_successor* B *has_successor* C and E *has_predecessor* D, the sequences of tasks $\langle A, C\rangle$ & $\langle E, A\rangle$ are valid, whereas $\langle C, A\rangle$ & $\langle D, E\rangle$ are not valid according to the induced partial order. Additionally, although hard dependencies *has_mandatory_successor* & *has_mandatory_predecessor* are specializations of the former connection types *has_successor* & *has_predecessor* and thus implicitly induce a partial order on their own, these two groups of connection types differ semantically from each other. Namely, with A *has_successor* B, we specify the necessary ordering between those two tasks if they both happen to be in the same workflow. However, with A *has_mandatory_successor* B, we additionally state that task A

[7]Although the referenced approach motivates the usage of the such modeling concepts, we choose to describe their work in a separate Section as to not overwhelm the reader with supplementary information. However, even if the related Section is not essential to the comprehension of herein presented workflow ground rules, the reader is still advised to follow up on the referenced work in order to get a broader understanding of these and similar constraints.

can never occur without task B in a workflow. To express that, additionally, A must occur alongside the (possibly) sole task B in a workflow, we have to explicitly stipulate the inverse connection B *has_mandatory_predecessor* A as well. Finally, we explore how partial ordering between tasks behaves in the presence of (unlimited) task repetitions. For any partial order between tasks, where not both are marked as *repeatable*, the situation is clear. The last repetition of the predecessor task must come before the first repetition of the successor task. On the other hand, no ordering can be enforced between *repeatable* predecessor A and successor B tasks, since these can belong to different sub-process iterations, making any combination of those two tasks (e.g. $\langle\langle B\rangle, \langle A\rangle, \langle A, B\rangle, \langle B\rangle, \langle B\rangle, \langle A\rangle\rangle$[8]) a valid workflow sequence pattern. A stricter ordering can be enforced for repeatable tasks with hard dependencies, because any hard dependency on a repeatable task A *has_mandatory_successor* B must necessarily be repeated before/after each repetition of the dependent task (as in e.g. $\langle A, B, B, B, A, B\rangle$). The full scope of these constraints is given in Section 3.1.1 on workflow validation.

Restrictions

Additional restrictions apply and reductions can be performed during the modeling of such workflows containing a partially ordered set of tasks. A partial order relation $\leq$ on a set of tasks T is both *antisymmetric* $\forall a, b \in T : a \leq b \land b \leq a \Rightarrow a = b$ and *transitive* $\forall a, b, c \in T : a \leq b \land b \leq c \Rightarrow a \leq c$ [Wall11]. Furthermore, in this case the partial order relation is also *irreflexive* $\forall a \in T : a \nleq a$, which also known as a *strong* partial order relation [Wall11]. Any binary relation α on a set S can be converted to a directed graph $G = (V, E)$, with $V \subseteq S$ and $E \subseteq V \times V$, such that all set elements that are connected via the relation $a\ \alpha\ b$ are connected in the graph via edges as well $\forall a^V, b^V \in V, a^S, b^S \in S : (a^V, b^V) \in E \Leftrightarrow a^S\ \alpha\ b^S$. As "[n]o order relation contains a cycle of length greater than 1" [Wall11], any order relation thus induces a directed acyclic graph G that allows self-loops $a\ \alpha\ a$. A graph's transitive closure $G^T = (V, E')$ is induced by $G = (V, E)$, such that its arcs are a superset of the original graph's arcs $E' \supseteq E$, with a new arc being part of the transitive closure $a \mapsto c \in E'$ whenever there exists a directed path between two nodes in the original graph $a \overset{*}{\mapsto} c \in E \Leftrightarrow a \mapsto b \mapsto \ldots \mapsto c \in E$ [AhGU72]. Moreover, there exists a unique, minimal *transitive reduction* G^t that is a sub-graph of any directed acyclic graph G with self-loops, such that it induces the same transitive closure $(G^t)^T = G^T$ [AhGU72]. Such transitive reduction G^t can be constructed for this work's strong partial order relation $\leq$ on tasks T as well. This means, firstly, that the partial ordering must necessarily contain no cycles, and secondly, that one can always find a minimal set of connections that transitively induce the desired partial order.

Specification

Before providing an unambiguous description of the problem domain, we briefly explain the purpose of such *specification*. "A specification is a written description of what a system is supposed to do. Specifying a system helps us understand it. It[is] a good idea to understand a system before building it, so it[is] a good idea to write a specification of a system before implementing it" [Lamp02]. To this end, we use a state-based *specification language* based on mathematical set theory called `TLA+` [MaKE18], so that we can easier identify any fundamental *specification errors* in our design [Wayn18]. As `TLA+` uses a mathematical notation, it can express domain concepts and constraints more elegantly and accurately than any general-purpose

[8]The inner sequences $\langle\ \rangle$ represent tasks contained in each sub-process iteration, for better visual distinction.

programming language [Wayn18]. Furthermore, any such specification is *checked* by the `TLC` model checker, to make sure the system adheres to any stated expectations [Wayn18]. Moreover, any properties of the system can be *proven* to hold by the `TLAPS` proof system [MaKE18]. Although we hope the mathematical notation feels intuitive to the reader, the exact syntax and semantics of this specification language are beyond the scope of this work, thus, if needed, the reader is advised to take a look at e.g. Lamport's [Lamp02] or Wayne's book [Wayn18] on this topic.

In the following, we thus provide an unambiguous `TLA+` specification of the problem domain. It describes the expected structure of the static design-time part of the system (tasks & connections between them), and of the dynamic run-time part of the system (workflow):

MODULE *WorkflowDefinition*

GIVEN the set of tasks:

CONSTANT *Tasks*

```
e.g. Tasks ==
{   [ name |-> "EVI", repeatable |-> FALSE, group |-> "non-destructive" ]
,   [ name |-> "IVI", repeatable |-> FALSE, group |-> "non-destructive" ]
}
```

such that it adheres to the expected structure

ASSUME $IsFiniteSet(Tasks)$

ASSUME $\forall\, task \in Tasks : (\forall\, id \in DOM(task) : id \in \{\text{“name”}, \text{“repeatable”}, \text{“group”}\})$

ASSUME $\forall\, task \in Tasks : task.name \in \text{STRING}$

ASSUME $\forall\, task, otherTask \in Tasks : task.name = otherTask.name \implies task = otherTask$

ASSUME $\forall\, task \in Tasks : task.repeatable \in \text{BOOLEAN}$

ASSUME $\forall\, task \in Tasks : task.group \in \{\text{“destructive”}, \text{“non-destructive”}, \text{“both”}\}$

GIVEN the set of connections between tasks:

CONSTANT *Connections*

```
e.g. Connections ==
{   [ name |-> "has_successor", srcName |-> "EVI", dstName |-> "IVI" ]
}
```

such that it adheres to the expected structure

ASSUME $IsFiniteSet(Connections)$

ASSUME $\forall\, conn \in Connections :$

$\forall\, id \in DOM(conn) : id \in \{\text{“name”}, \text{“srcName”}, \text{“dstName”}\}$

ASSUME $\forall\, conn \in Connections : conn.name \in$

$\{$ “has_successor”

, “has_predecessor”

$,$ "has_mandatory_predecessor"

$,$ "has_mandatory_successor"

$\}$

ASSUME $\forall\, conn \in Connections : \exists\, task \in Tasks : task.name = conn.srcName$

ASSUME $\forall\, conn \in Connections : \exists\, task \in Tasks : task.name = conn.dstName$

ASSUME $\forall\, conn \in Connections : conn.srcName \neq conn.dstName$

WHEN checking for errors in an input workflow

CONSTANT *Workflow*

```
e.g. Workflow == << "EVI", "IVI" >>
```

such that it adheres to the expected structure

ASSUME $DOM(Workflow) = 1 \,..\, Len(Workflow)$ a proper tuple

ASSUME $\forall\, t \in RAN(Workflow) : t \in$ STRING

The problem domain can be instantiated for the example FA process used throughout this Chapter as follows:

MODULE *WorkflowExample*

$WorkflowExample \triangleq$ INSTANCE *WorkflowDefinition* WITH

$Tasks \leftarrow$

$\{$ $[name \mapsto$ "EVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "non-destructive"$]$

$,$ $[name \mapsto$ "SEM", $repeatable \mapsto$ FALSE, $group \mapsto$ "non-destructive"$]$

$,$ $[name \mapsto$ "XRAY_MIC", $repeatable \mapsto$ FALSE, $group \mapsto$ "non-destructive"$]$

$,$ $[name \mapsto$ "EPT", $repeatable \mapsto$ FALSE, $group \mapsto$ "non-destructive"$]$

$,$ $[name \mapsto$ "DECAP", $repeatable \mapsto$ FALSE, $group \mapsto$ "destructive"$]$

$,$ $[name \mapsto$ "IVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "destructive"$]$

$,$ $[name \mapsto$ "STRIP", $repeatable \mapsto$ TRUE, $group \mapsto$ "destructive"$]$

$,$ $[name \mapsto$ "XRAY_SPEC", $repeatable \mapsto$ TRUE, $group \mapsto$ "destructive"$]$

$,$ $[name \mapsto$ "SAM", $repeatable \mapsto$ TRUE, $group \mapsto$ "destructive"$]$

$,$ $[name \mapsto$ "TEM", $repeatable \mapsto$ TRUE, $group \mapsto$ "destructive"$]$

$\},$

$Connections \leftarrow$

$\{$ $[name \mapsto$ "has_predecessor", $srcName \mapsto$ "SEM", $dstName \mapsto$ "EVI"$]$

$, \ [\mathit{name} \mapsto \text{“has\_predecessor”},\ \mathit{srcName} \mapsto \text{“XRAY\_MIC”},\ \mathit{dstName} \mapsto \text{“EVI”}]$

$, \ [\mathit{name} \mapsto \text{“has\_predecessor”},\ \mathit{srcName} \mapsto \text{“EPT”},\ \mathit{dstName} \mapsto \text{“SEM”}]$

$, \ [\mathit{name} \mapsto \text{“has\_predecessor”},\ \mathit{srcName} \mapsto \text{“EPT”},\ \mathit{dstName} \mapsto \text{“XRAY\_MIC”}]$

$, \ [\mathit{name} \mapsto \text{“has\_predecessor”},\ \mathit{srcName} \mapsto \text{“DECAP”},\ \mathit{dstName} \mapsto \text{“EPT”}]$

$, \ [\mathit{name} \mapsto \text{“has\_mandatory\_successor”},\ \mathit{srcName} \mapsto \text{“DECAP”},\ \mathit{dstName} \mapsto \text{“IVI”}]$

$, \ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”},\ \mathit{srcName} \mapsto \text{“IVI”},\ \mathit{dstName} \mapsto \text{“DECAP”}]$

$, \ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”},\ \mathit{srcName} \mapsto \text{“STRIP”},\ \mathit{dstName} \mapsto \text{“IVI”}]$

$, \ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”},\ \mathit{srcName} \mapsto \text{“XRAY\_SPEC”},\ \mathit{dstName} \mapsto \text{“STRIP”}]$

$, \ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”},\ \mathit{srcName} \mapsto \text{“SEM”},\ \mathit{dstName} \mapsto \text{“STRIP”}]$

$, \ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”},\ \mathit{srcName} \mapsto \text{“TEM”},\ \mathit{dstName} \mapsto \text{“STRIP”}]$

$, \ [\mathit{name} \mapsto \text{“has\_predecessor”},\ \mathit{srcName} \mapsto \text{“SEM”},\ \mathit{dstName} \mapsto \text{“XRAY\_SPEC”}]$

$, \ [\mathit{name} \mapsto \text{“has\_predecessor”},\ \mathit{srcName} \mapsto \text{“TEM”},\ \mathit{dstName} \mapsto \text{“XRAY\_SPEC”}]$

$\},$

$\mathit{Workflow} \leftarrow$

$\langle \text{“EVI”}, \text{“EPT”}, \text{“DECAP”}, \text{“IVI”}, \text{“STRIP”}, \text{“SAM”}, \text{“STRIP”}, \text{“XRAY\_SPEC”}, \text{“TEM”} \rangle$

3.1.6 Ontology

Having defined the problem domain, an ontology can now be used for modeling the example FA process as “description of knowledge about [such] domain of interest” [HiKR09].

Figure 3.4 shows a visual representation of the example FA process modeled[9] using OWL individuals / classes with associated data properties, which are related to each other using object properties. In this ontology, OWL individuals / classes represent the respective FA tasks, whereas associated data properties depict the tasks' repeatability and destructiveness. Object properties are used to define the partial order and hard dependencies between tasks.

```
### http://ainf.aau.at/lottraveler/group
:group rdf:type owl:DatatypeProperty ;
      rdfs:subPropertyOf owl:topDataProperty ;
      rdf:type owl:FunctionalProperty ;
      rdfs:domain :Task ;
      rdfs:range [ rdf:type rdfs:Datatype ;
                   owl:oneOf [ rdf:type rdf:List ;
                               rdf:first "both"^^xsd:string ;
                               rdf:rest [ rdf:type rdf:List ;
                                          rdf:first "destructive"^^xsd:string ;
                                          rdf:rest [ rdf:type rdf:List ;
                                                     rdf:first "non-destructive"^^xsd:string ;
```

[9]Ontology constructed via Protégé (https://protege.stanford.edu/), an ontology editor and framework for building intelligent systems. The initial visual representation was generated by Protégé's Ontograf plugin and its "Export to DOT graph" feature, after which the generated graph was further manually modified for better visual clarity.

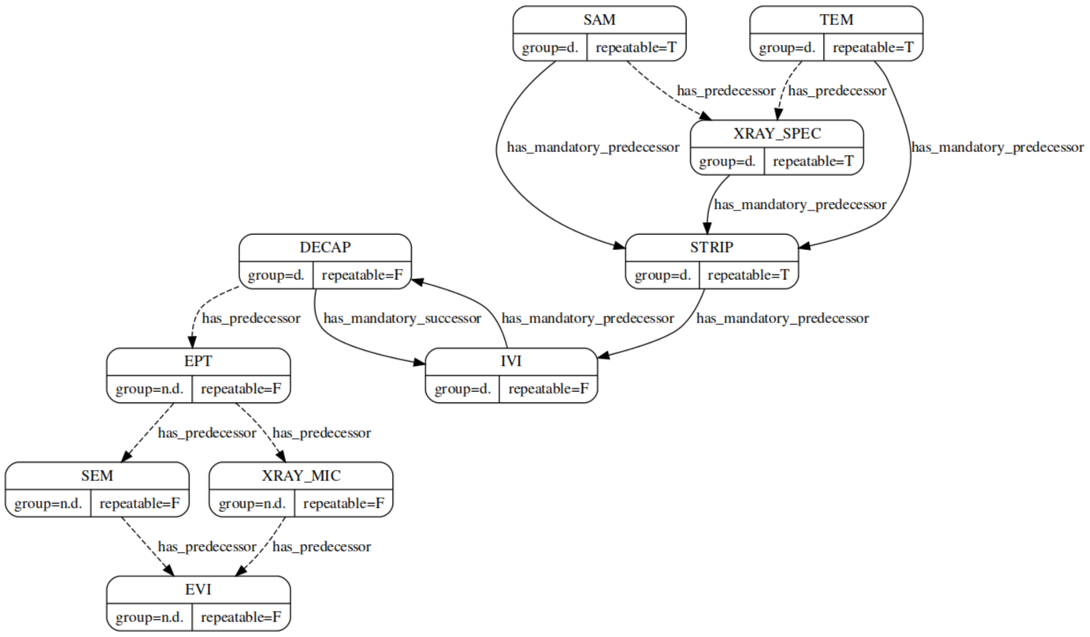


Figure 3.4: Example FA process modeled using OWL individuals / classes with associated data properties, which are related to each other using object properties

```
                                                 rdf:rest rdf:nil
                                               ]
                                     ]
                           ]
                ] .

### http://ainf.aau.at/lottraveler/repeatable
:repeatable rdf:type owl:DatatypeProperty ;
           rdfs:subPropertyOf owl:topDataProperty ;
           rdf:type owl:FunctionalProperty ;
           rdfs:domain :Task ;
           rdfs:range xsd:boolean .
```

Src. 3.1: Data properties of the example FA process modeled as an OWL2-DL ontology

Listing 3.1 shows the definition of these data properties of the ontology. Both data properties are marked as *functional* so that any FA tasks can only be associated to one such value via the respective property. Any subjects connected as the *domain* of these properties are considered to be tasks. On the other hand, the value *range*s of these properties are restricted to the respective datatypes. Here, :*group* defines its custom value datatype, consisting of a choice between the respective strings.

```
### http://ainf.aau.at/lottraveler/is_connected_to
:is_connected_to rdf:type owl:ObjectProperty ;
               rdfs:subPropertyOf owl:topObjectProperty ;
               rdf:type owl:TransitiveProperty ;
               rdfs:domain :Task ;
               rdfs:range :Task .

### http://ainf.aau.at/lottraveler/depends_on
:depends_on rdf:type owl:ObjectProperty ;
           rdfs:subPropertyOf owl:topObjectProperty ;
           # rdf:type owl:AsymmetricProperty ,
           #        owl:IrreflexiveProperty ;
```

```
          rdfs:domain :Task ;
          rdfs:range :Task .

###  http://ainf.aau.at/lottraveler/has_successor
:has_successor rdf:type owl:ObjectProperty ;
             rdfs:subPropertyOf :is_connected_to ;
             owl:inverseOf :has_predecessor ;
             rdf:type # owl:AsymmetricProperty ,
                     # owl:IrreflexiveProperty ,
                     owl:TransitiveProperty ;
             rdfs:domain :Task ;
             rdfs:range :Task .

###  http://ainf.aau.at/lottraveler/has_predecessor
:has_predecessor rdf:type owl:ObjectProperty ;
               rdfs:subPropertyOf :is_connected_to ;
               owl:inverseOf :has_successor ;
               rdf:type # owl:AsymmetricProperty ,
                       # owl:IrreflexiveProperty ,
                       owl:TransitiveProperty ;
               rdfs:domain :Task ;
               rdfs:range :Task .

###  http://ainf.aau.at/lottraveler/has_mandatory_successor
:has_mandatory_successor rdf:type owl:ObjectProperty ;
                       rdfs:subPropertyOf :depends_on ,
                                         :has_successor ;
                      # rdf:type owl:AsymmetricProperty ,
                      #         owl:IrreflexiveProperty ;
                       rdfs:domain :Task ;
                       rdfs:range :Task .

###  http://ainf.aau.at/lottraveler/has_mandatory_predecessor
:has_mandatory_predecessor rdf:type owl:ObjectProperty ;
                         rdfs:subPropertyOf :depends_on ,
                                           :has_predecessor ;
                        # rdf:type owl:AsymmetricProperty ,
                        #         owl:IrreflexiveProperty ;
                         rdfs:domain :Task ;
                         rdfs:range :Task .
```

Src. 3.2: Object properties of the example FA process modeled as an OWL2-DL ontology[10]

Listing 3.2 shows the definition of these object properties of the ontology. The four connection types are modeled as the respective object properties *:has_successor*, *:has_predecessor*, *:has_mandatory_sucessor* and *:has_mandatory_predecessor*. As described in the problem domain in Section 3.1.5, the mandatory dependency connections are specializations (*rdfs:subPropertyOf*) of their respective looser variants. Furthermore, only these looser connection properties have each other defined as their inverse (*owl:inverseOf*). Additional auxiliary properties *:is_connected_to* and *:depends_on* are introduced to further organize these connections into these looser and stricter connection type groups, respectively. As the partial order of tasks is induced transitively, the former connection properties are marked as such *owl:TransitiveProperty*s. Even though marking those object properties as *owl:AsymmetricProperty* and *owl:IrreflexiveProperty* is more precise from a semantic standpoint, the resulting ontology

[10] Adding the commented-out triples breaks the OWL2-DL restriction on simple roles, as explained in `https://www.w3.org/TR/owl2-syntax/#The_Restrictions_on_the_Axiom_Closure`. OWL2-DL conformance checked via OWL API profile checker (`https://github.com/stain/profilechecker`).

would no longer be OWL2-DL conform[10]. All of these object properties have their *domain* and *range* set to the class of :*Task*s.

```
### http://ainf.aau.at/lottraveler/Task
:Task rdf:type owl:Class ;
     rdfs:subClassOf owl:Thing .
```

Src. 3.3: Task class of the example FA process modeled as an OWL2-DL ontology

Listing 3.3 shows the definition of the :*Task* class of the ontology. It serves as a trivial super-class for all specific tasks.

We identified three variants of modeling all specific tasks:

- Listing 3.4 shows the first variant, where specific tasks are modeled as named individuals of the :*Task* class. Each individual is associated by the aforementioned data and object properties to literal values and other task individuals, respectively.

```
### http://ainf.aau.at/lottraveler/DECAP
:DECAP rdf:type owl:NamedIndividual ,
              :Task ;
      :has_mandatory_successor :IVI ;
      :has_predecessor :EPT ;
      :group "destructive"^^xsd:string ;
      :repeatable "false"^^xsd:boolean .
```

Src. 3.4: Excerpt from example FA process modeled using an instance-based OWL2-DL ontology

- Listing 3.5 shows a different variant, which uses *punning* to view a specific task resource from both class and individual view. Each specific task from the individual view is related by data and object properties, as in the first variant, to model its specific properties and connections. Furthermore, each task individual is connected to its respective class via an *instance-of* relationship in order to establish a semantic connection between those two views.

```
### http://ainf.aau.at/lottraveler/DECAP
:DECAP rdf:type owl:Class ;
      rdfs:subClassOf :Task .

### http://ainf.aau.at/lottraveler/DECAP
:DECAP rdf:type owl:NamedIndividual ,
              :DECAP ;
      :has_mandatory_successor :IVI ;
      :has_predecessor :EPT ;
      :group "destructive"^^xsd:string ;
      :repeatable "false"^^xsd:boolean .
```

Src. 3.5: Excerpt from example FA process modeled using a punning-based OWL2-DL ontology

- Listing 3.6 shows another variant, where specific tasks of the example FA process are modeled as classes. In order to associate properties and connections with such classes, each class *is-a* $rdfs{:}subClassOf$ class expressions composed of qualified restrictions on data and object properties, with values fixed on respective targets[11]. Therefore, each specific :*Task* class is declared in terms of specific restrictions on the respective task's properties and connections.

[11]This modeling variant is proposed in use case 1 in the W3C working group note on n-ary relations [Coot06a].

```
### http://ainf.aau.at/lottraveler/DECAP
:DECAP rdf:type owl:Class ;
      rdfs:subClassOf :Task ,
                      [ rdf:type owl:Restriction ;
                        owl:onProperty :has_mandatory_successor ;
                        owl:someValuesFrom :IVI
                      ] ,
                      [ rdf:type owl:Restriction ;
                        owl:onProperty :has_predecessor ;
                        owl:someValuesFrom :EPT
                      ] ,
                      [ rdf:type owl:Restriction ;
                        owl:onProperty :group ;
                        owl:hasValue "destructive"^^xsd:string
                      ] ,
                      [ rdf:type owl:Restriction ;
                        owl:onProperty :repeatable ;
                        owl:hasValue "false"^^xsd:boolean
                      ] .
```

Src. 3.6: Excerpt from example FA process modeled using a class-based OWL2-DL ontology

The punning variant is considered an anti-pattern and "poor engineering practice" [HeGA20]. Although the instance-based variant is the most idiomatic and straightforward way of encoding the problem domain, the class-based variant provides the possibility for further reasoning on specific task executions in the FA lab that can be modeled as individuals of the respective task class.

Summary:

+ easy to design, because domain concepts are modeled (almost) directly via respective triples in such domain ontology
∘ easy to understand, yet hard to gain an overview due to weak visual feedback among multiple visualization tools
+ good tool support via variety of products
+ domain-specific encoding allows for concise representation of many loosely ordered and optional tasks

3.1.7 Answer set programming

Domain concepts can, in a similar vein, be used in answer set programming to model the domain of discourse.

```
task("EPT").
group_non_destructive(task("EPT")).
% atom just omitted for default negation, instead of explicit classical negation
%-repeatable(task("EPT")).
has_predecessor(task("EPT"), task("SEM")).
has_predecessor(task("EPT"), task("XRAY_MIC")).
```

Src. 3.7: Excerpt from answer set program that models example FA process

Listing 3.7 shows an excerpt from an executable answer set program[12] that models the failure analysis process given in Section 3.1.1. This excerpt represents the majority of the *problem*

[12]Execution tested via `clingo` (`https://potassco.org/clingo/`), an answer set grounder and solver.

instance's facts NLP_I as part of such typical *uniform* problem definition [GKKL+15][GKKS12]. On the other hand, the rules for solving any instance in such *problem class* NLP_C [GKKS12], will be explored in later sections. This problem instance's domain is thus encoded as NLP facts, such that domain concepts map directly to corresponding predicates. Tasks are encoded as the respective predicate on each task's name. Task properties, like *repeatable* and *group_non_destructive*, are modeled as unary predicates on these task names. Associations between tasks, like *has_predecessor*, are, on the other hand, designed as binary predicates on respective task names. These predicate arguments are, however, wrapped in function symbols, for added clarity on the meaning of these string identifiers. Additional NLP rules are added to check the validity of an ASP encoded problem domain. These dismiss nonsensical, wrongly structured or otherwise erroneous facts about tasks and their connections, and make sure that the partial order induced by the provided associations contains no cycles.

Summary:

+ easy to design, because domain concepts are modeled (almost) directly via respective facts in such problem instance
– easy to understand, yet hard to gain an overview over all associations from such textual representation
+ variety of grounders and solvers available
+ domain-specific encoding allows for concise representation of many loosely ordered and optional tasks

The full listing of ASP domain model source codes, including all domain validation, utility and helper rules, is given in appendix Chapter B.

3.1.8 Domain-specific modeling language

A DSML is another approach that integrates domain concepts directly into the modeling notation. However, since all connection types induce a partial order between tasks, the DSMLs associations facilitate a similar control-flow feel as previous flow-based process-oriented modeling languages.

Figure 3.5 shows the metamodel of a DSML used to model the example failure analysis process given in Section 3.1.1, designed using the building blocks of `WebGME`'s[13] meta-metamodel. The metamodel, as seen in `WebGME`'s metamodel cross-cut view, consists of two domain objects *Task* and *Connection*, where the latter is further specialized by the respective four connection type objects. A *Task* contains, beside the (inherited) *name* string, the *group* string and the *repeatable* boolean attributes. The base *Connection* object is marked as abstract (non-instantiable) and serves as the prototype for visualizing directed, partial order or hard dependency relations between a *src* and a *dst Task* object.

Figure 3.6 shows the domain-specific model of the example failure analysis process given in Section 3.1.1, designed using the buildings blocks of the DSML defined by the previous metamodel. The *Task* object is instantiated for each task *name* found in the example FA process, with correspondingly set *group* and *repeatable* attributes. These attributes are visualized as part of task

[13]Metamodel constructed via `WebGME` (`https://webgme.org/`), a generic modeling environment for the web [MKKB+14].

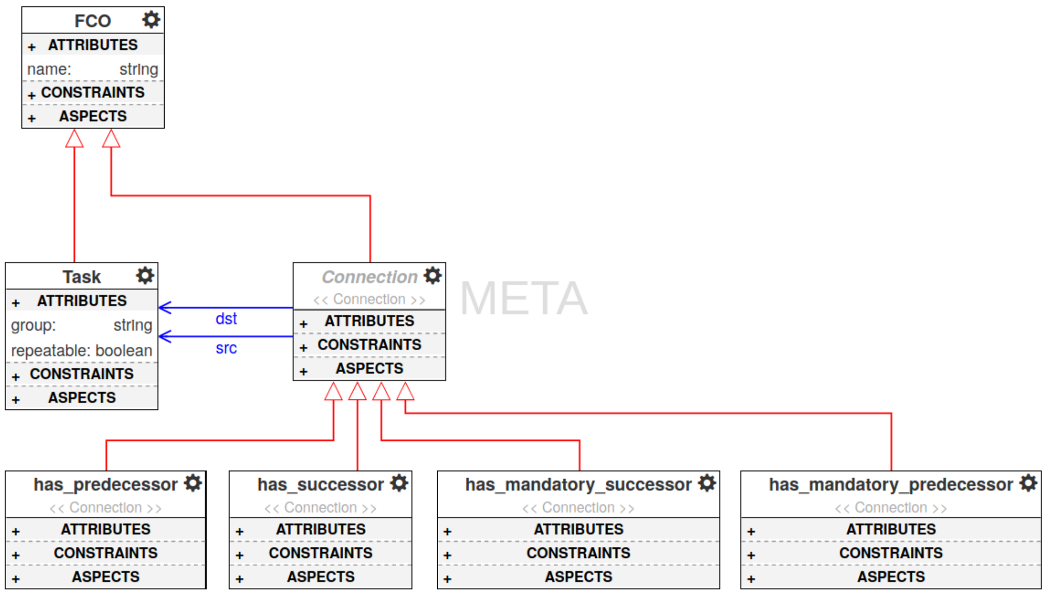


Figure 3.5: Metamodel of the DSML used to model the example FA process, designed using the building blocks of `WebGME`'s meta-metamodel

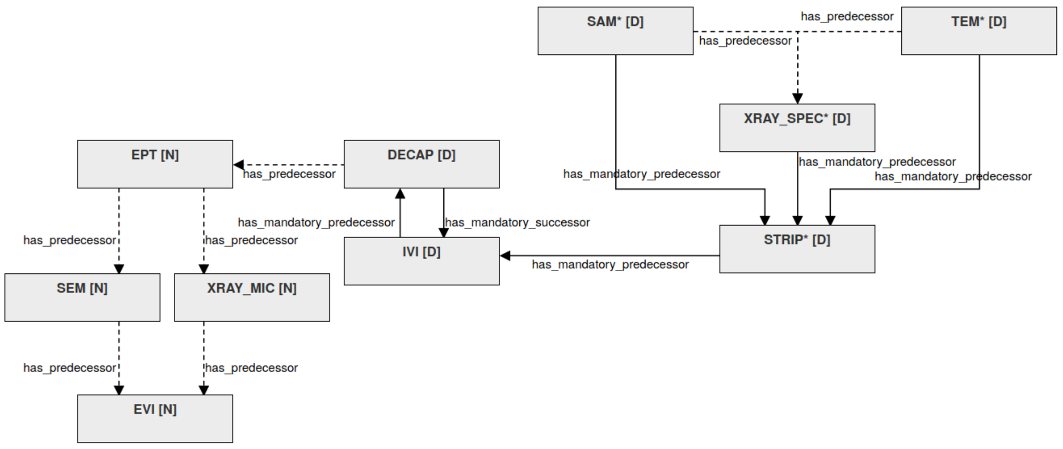


Figure 3.6: Domain-specific model of the example FA process, designed using the buildings blocks of the DSML defined by the previous metamodel

names in the form of a suffix to distinguish between [N]on-destructive and [D]estructive tasks, as well as in the form of an asterisk suffix * for repeatable tasks[14]. These *Task* object instances are connected via appropriate instances of *Connection* specializations in order to depict the respective partial order and hard dependencies between tasks.

Moreover, since no tasks are mandatory on their own, an empty failure analysis workflow can be created, in that most trivial case. Otherwise, all possible and valid workflows, according to the example definition, are represented in this domain-specific workflow model.

Summary:

[14]This visualization is achieved through a graphical decorator to the DSME provided by `WebGME`, as explained in the implementation Chapter 4.

+ easy to design, because modeling elements directly represent domain concepts
+ easy to understand, because long as the semantics involving the different connection types are discerned
+ tools for the design, use and execution of domain-specific models are available
+ domain-specific notation allows for concise representation of many loosely ordered and optional tasks

3.2 Workflow validation & repair

In this Section, we look at different approaches and how they can be used for solving the latter part of our problem statement. We start by analyzing the requirements of this part of the problem statement, what it means for workflows to be validated and repaired. Afterwards, we explore how previously introduced approaches for workflow modeling can be used to achieve this task. Each approach is accompanied by a brief summary of its viability to perform workflow validation & repair.

3.2.1 Problem reasoning: Workflow validation

At first, we want to validate a sequence of tasks and thus check whether such workflow passes a specific set of constraints modeled beforehand. These constraints operate on tasks, their partial order and hard dependencies between them.

For example, in the context of the example FA process given in Subsection 3.1.1, the workflow $\langle EVI, EPT \rangle$ is a valid workflow respecting the given ground rules. On the other hand, the sequence of tasks $\langle EPT, EVI \rangle$ is not valid, because the task `EPT` must always come after the task `EVI` according to the given rules.

Constraints

We identified the following constraints by analyzing the problem domain from Section 3.1.5, as well as by writing tests to confirm whether the result of validating various test workflows matches our expectations:

- all tasks occurring in a workflow must be known; misspelled or unrecognized task names are to be dismissed
- all tasks marked as non-repeatable must never be repeated in a workflow
- all non-destructive tasks must come before all destructive tasks in the given sequence of tasks
- all tasks must adhere to the transitively induced partial order, inside each sub-process iteration
- all dependency tasks must be performed before / after the respective dependent task
  - once before / after the dependent task
  - and each time before / after another occurrence of a dependent task, in the same sub-process iteration

The former constraints follow intuitively from the description of the problem domain in Section 3.1.5. The penultimate constraint checks for proper partial ordering, and needs to accept any combination of task repetitions from different sub-process iterations, as mentioned in this previous Section. The last constraint deals with hard dependencies, and thus enforces stricter requirements on the order of repeatable dependency tasks.

Specification

As before, we formalize these constraints using an unambiguous `TLA+` specification, starting with some preliminary definitions:

MODULE *WorkflowValidation*

T ... The set of all tasks T

$T \triangleq TaskNames$ e.g. $T = \{"EVI", "XRAY", "IVI"\}$

t^* ... The predicate indicating the repeatability of a task t

$t^* \triangleq Task(t).repeatable$ e.g. $"XRAY"^* = true$

$destr(t)$ ... The predicate indicating the destructiveness of a task t

$destr(t) \triangleq Task(t).group = \text{“destructive”}$ e.g. $destr("IVI") = true$

$non\_destr(t)$ ... The predicate indicating the non-destructiveness of a task t

$non\_destr(t) \triangleq Task(t).group = \text{“non-destructive”}$ e.g. $non\_destr("EVI") = true$

$A \prec B,\ B \succ A$ ... The partial order relation between tasks

$A \prec B \triangleq ConRel[A, B]$ e.g. $"EVI" \prec "XRAY"$

$B \succ A \triangleq ConRel[A, B]$ e.g. $"IVI" \succ "XRAY"$

$A \preceq B,\ B \succeq A$ ... The reflexive transitive closure over the partial order relation

$A \preceq B \triangleq TransConRel[A, B]$ e.g. $"EVI" \preceq "IVI"$

$B \succeq A \triangleq TransConRel[A, B]$ e.g. $"IVI" \succeq "EVI"$

$A| - B,\ B - |A$ ... The relation describing whether one task requires another task

$A \vdash B \triangleq RequiresRel[A, B]$ e.g. $"XRAY"| - "EVI"$

$B \dashv A \triangleq RequiresRel[A, B]$ e.g. $"XRAY" - |"IVI"$

W ... the sequence of workflow tasks W

$W \triangleq Workflow$ e.g. $W =<< "EVI", "IVI", "EVI" >>$

$T(W)$ ... The set of all workflow tasks T(W)

$T(W) \triangleq RAN(W)$ e.g. $T(W) = \{"EVI", "IVI"\}$

$W\#\#t$ ... The amount of occurrences of task t in the given workflow sequence

$W \#\# t \triangleq Count(W, t)$ e.g. $W\#\#"EVI" = 2$

$W@@t$ ... The set of all indexes of a task t in the given workflow sequence

$W \mathbin{@@} t \triangleq \mathit{Indexes}(W,\ t)$ e.g. $W@@\text{"}EVI\text{"} = \{1, 3\}$

$W\hat{}\hat{}t$ ... The first index of a task t in the given workflow sequence

$W\hat{}\hat{}t \triangleq \mathit{FirstIndex}(W,\ t)$ e.g. $W\hat{}\hat{}\text{"}IVI\text{"} = 2$

$W\$\$t$ ... The last index of a task t in the given workflow sequence

$W \$\$ t \triangleq \mathit{LastIndex}(W,\ t)$ e.g. $W\$\$\text{"}EVI\text{"} = 3$

The previously introduced notation is now used to describe the system's *validate* functionality, by precisely defining the aforementioned constraints that each workflow is checked against:

MODULE *WorkflowValidation*

All tasks must be known

$\mathit{KnownTaskConstraint} \triangleq$

$$\forall\, t\_w \in T(W):\quad \exists\, t \in T:\ t\_w = t$$

Non-repeatable tasks must not be repeated

$\mathit{RepeatabilityConstraint} \triangleq$

$$\forall\, t \in T:\quad t \in T(W) \land \neg t^* \implies W \mathbin{\#\#} t \leq 1$$

Destructive tasks must come after non-destructive ones

$\mathit{DestructiveOrderConstraint} \triangleq$

$$\forall\, d,\ n \in T:$$

$$d \neq n \land d \in T(W) \land n \in T(W) \land \mathit{destr}(d) \land \mathit{non\_destr}(n) \implies W\hat{}\hat{}d > W \$\$ n$$

Task order must adhere to (transitive) partial order

$\mathit{PartialOrderConstraint} \triangleq$

$$\forall\, a,\ b \in T:$$

$$a \neq b \land a \preceq b \land a \in T(W) \land b \in T(W) \land (\neg a^* \lor \neg b^*) \implies W \$\$ a < W\hat{}\hat{}b$$

Required tasks must be done before / after dependent tasks

$\mathit{MandatoryDependencyConstraint} \triangleq$

$$\forall\, a,\ b \in T:$$

$$\land\ a \neq b \land a \preceq b \land a \vdash b \land a \in T(W) \implies b \in T(W) \land W \$\$ a < W \$\$ b$$

$$\land\ a \neq b \land a \preceq b \land a \dashv b \land b \in T(W) \implies a \in T(W) \land W\hat{}\hat{}a < W\hat{}\hat{}b$$

Required tasks must be done in between dependent tasks, in case of repetition

$MandatoryRepetitionConstraint \triangleq$

$\quad \forall\, a, b \in T :$

$\quad \land$

$\qquad \land\, a \neq b \land a \preceq b \land a \vdash b \land a \in T(W) \land b \in T(W)$

$\qquad \implies \forall\, i, j \in (W \mathbin{@@} a) : \quad i < j \quad \implies \quad \exists\, k \in (W \mathbin{@@} b) : \; i < k \land k < j$

$\quad \land$

$\qquad \land\, a \neq b \land a \preceq b \land a \dashv b \land a \in T(W) \land b \in T(W)$

$\qquad \implies \forall\, i, j \in (W \mathbin{@@} b) : \quad i < j \quad \implies \quad \exists\, k \in (W \mathbin{@@} a) : \; i < k \land k < j$

Workflow validation can then be performed on an input workflow, for which any found errors are returned. Here, we highlight a single test from the complete validation test suite, in which the actual error output is asserted to match the expected error output for the given workflow input.

MODULE *WorkflowValidationTest*

LET *Validation* $\triangleq$ INSTANCE *WorkflowValidation* WITH

Connections $\leftarrow$

{ [*name* ↦ "has_successor", *srcName* ↦ "EVI", *dstName* ↦ "XRAY"]

, [*name* ↦ "has_successor", *srcName* ↦ "XRAY", *dstName* ↦ "IVI"]

},

Tasks $\leftarrow$

{ [*name* ↦ "EVI", *repeatable* ↦ FALSE, *group* ↦ "both"]

, [*name* ↦ "XRAY", *repeatable* ↦ FALSE, *group* ↦ "both"]

, [*name* ↦ "IVI", *repeatable* ↦ FALSE, *group* ↦ "both"]

},

Workflow $\leftarrow$

⟨"IVI", "EVI"⟩

ACTUAL $\triangleq$ *Validation*!*Errors*

EXPECTED $\triangleq$ [*Validation*!*NoErrors* EXCEPT

!.*ErrorPartialOrderViolations* = {

[*name* ↦ "has_successor", *srcName* ↦ "EVI", *dstName* ↦ "IVI"]

}

]

IN

$Assert($

$ACTUAL = EXPECTED,$

"should report errors for violating transitive partial order between tasks"

$)$

The full specification of the workflow validation functionality, including the precise input and output encoding, is given in appendix Section A.3. The complete suite of workflow validation tests is listed in appendix Section A.4.

3.2.2 Problem reasoning: Workflow repair

Afterwards, we want to repair an invalid workflow, in case it is considered invalid and does not pass the set of validation constraints. Namely, we want to recommend an alternative, valid workflow, that is most similar to the given invalid workflow according to a *similarity measure*.

For example, in the context of the example FA process given in Subsection 3.1.1, the workflow $\langle EPT, EVI \rangle$ is invalid and needs to be repaired. The expected alternative workflow recommendation is $\langle EVI, EPT \rangle$, since it is valid according to the business constraints and is (intuitively) also the most similar workflow to the original one.

Generation

Hence, the goal is to generate workflows that satisfy the given ground rules and choose the "best" among them. As the "building blocks of a workflow are tasks [...] with well-defined input–output interfaces, [...] [and] the design of constructs for workflows only needs to accommodate most control flows in practice, [...] the state space of a workflow is finite and loops take a limited form, as a result, automatic verification and generation become possible" [LuBL06]. Moreover, thanks to the elicitation of requirements with experts from Infineon, we assume an upper limit of 15 tasks in a *typical* failure analysis workflow for that specific application area.

As an aside, any generated workflow is trivially sound, formally proven by converting any such linear sequence of tasks into a corresponding Petri net.

Similarity measure

Although the definition of a *similarity measure* s is, in general, not as straightforward for entities from any kind of universe of discourse [Tver77], in this case *distance functions* d can be used to model the inverse of such similarity, its *dissimilarity*, between workflows.

A *distance function* d, also known as a *distance metric*, is a function mapping a pair of elements (a, b) from the *metric space*'s (S, d) set S to a non-negative numerical value n[15], such that $d(a, b) = n$ with $d : S \times S \rightarrow \mathbb{N}_{\geq 0}$ [ScSo60][Tver77]. Furthermore, a distance metric must satisfy the following conditions $\forall a, b \in S$ [ScSo60][Tver77]:

- *Identity*: $d(a, b) = 0 \Leftrightarrow a = b$
- *Positivity*: $d(a, b) \geq 0$

[15]A metric's range is defined as the set of real numbers $\mathbb{R}$. However, for the purpose of this work, we limit the range to the set of non-negative integers $\mathbb{N}_{\geq 0}$.

- *Symmetry*: $d(a, b) = d(b, a)$
- *Triangle Inequality*: $d(a, b) + d(b, c) \geq d(a, c)$

One such distance metric is the *string edit distance*, which counts the amount of character-manipulating operations that need to be performed on one string, in order for it to exactly match another string. There exist different variants of this string edit distance, based on the type of operations to be performed, as well as the respective operation cost functions [HyNI10]. For example, the *Damerau-Levenshtein* string edit distance computes the amount of insertion, deletion, substitution or transposition operations (of adjacent characters) that need to be performed [Bard07]. The *longest common subsequence distance* measures the amount of character insertions and deletions needed to be made so that the longest sequence of in-common characters is extended to the given strings [Nava01]. On the other hand, there exists a variant that allows substring moves, where such operation type facilitates the move of a whole sequence of characters around [CoMu07], as a more general version of adjacent character transposition.

As a workflow is a sequence of tasks (e.g. $\langle B, A, A, B, B, B, A \rangle$), a slightly adapted string edit distance could seemingly be used to compute the similarity between a given invalid workflow and a recommended workflow. It makes sense to compare workflows in this manner, since one can consider the amount of modifications as the amount of "fixes" to such invalid workflow. The recommended, alternative & valid workflow can be considered as the one with the least amount of "fixes" compared to the original, invalid & to-be-repaired one. For the string edit distance to be applicable in this case, each workflow task name can conceptually be thought of as a character, so that this function can be applied on such string (e.g. "$BAABBBA$").

The use of the string edit distance is also inspired by Leake and Kendall-Morwick's work [LeKe08], which is further described in related work Section 6.2.1. The authors use the string edit distance for (partly) assessing the similarity between process models by looking at the required amount of addition and deletion operations needed to "convert" one process model to the other.

Distance measures

However, after further analysis of the problem domain, we postulate how different edit operations have to be penalized differently. Namely, a custom-tailored workflow ordered by a client contains tasks that the client explicitly wants to be included in this workflow[16]. Therefore, the repaired, alternative workflow shall augment the invalid workflow with additional tasks, as opposed to removing tasks from it. Furthermore, we postulate how a client typically requests specific tasks to be performed on samples, but does not necessarily give importance to the task order or required dependency tasks.

Instead of opting for significantly different operation weights in a single string edit distance metric, we opt to rank recommended workflows by a priority list of different distance metrics, each inspired by one of the available string edit operations described before. A workflow is thus recommended only if it is the best candidate according to the highest priority distance metric or the best candidate according to subsequently lower priority metrics, in case of ties. However, in order to be able to split the compound string edit distance into respective distances

[16]We acknowledge the possibility of clients mistakenly adding a task to a workflow, but argue how a mistake should occur less frequently than an intentional inclusion in practice.

for each edit operation and determine the amount of "fixes" required for such an operation type, asymmetric *distance measures* d must be used in some cases instead, which violate a metric's symmetry condition, with $d(a, b) \neq d(b, a)$. In particular, the amount of removed tasks between a source and target workflow is necessarily asymmetric, with e.g. $d_{rm}(\langle A, B\rangle, \langle A\rangle) = 1$ and $d_{rm}(\langle A\rangle, \langle A, B\rangle) = max(0, -1) = 0$. Note that these distance measures are computed with the alternative, recommended workflow as such target and the original, invalid workflow as such source, since the reverse is of no interest.

Optimal alternative

We identified the priority list of distance measures through experimentation, as well as by writing tests to confirm whether the result of repairing various test workflows matches our expectations. These measures penalize the following differences between the invalid source and recommended target workflow, according to the specified priority:

1. the amount of removed task types
2. the amount of added task types
3. the reduced amount of task repetitions
4. the increased amount of task repetitions
5. the sum of task position differences in the sequence
6. the difference in workflow length

The former distance measures thus correspond to specialized versions of insertion and deletion distance measures. The penultimate distance measures the amount of task transpositions. The metric with the least priority acts as a catch-all distance measure, to resolve any remaining ties between available alternatives, if possible.

The "best" alternative workflow recommendation is thus the one that is the most optimal according to this priority list of distance measures, among all other alternatives.

Specification

As before, we formalize these constraints using an unambiguous `TLA+` specification, starting with some preliminary definitions:

MODULE *WorkflowRepair*

W ... the sequence of workflow tasks to repair

$W \triangleq Workflow$ e.g. $W =<< "EVI", "IVI" >>$

$ValidWorkflows$ ... the set of all valid workflows that act as candidate recommendations

$ValidWorkflows$ e.g. $ValidWorkflows = \{<<>>, << "EVI" >>, ...\}$

W_best ... the best alternative sequence of workflow tasks

W_best e.g. $W\_best =<< "EVI", "XRAY", "IVI" >>$

W_worse ... any other worse alternative sequence of workflow tasks

W_worse e.g. $W\_worse =<< "EVI" >>$

The previously introduced notation is now used to describe the system's *repair* functionality, by precisely defining the order of distance measures that each recommendation candidate workflow is evaluated against:

MODULE *WorkflowRepair*

```
$\forall\ W\_worse \in ValidWorkflows : W\_best \neq W\_worse \implies$
 1. minimize deletion of tasks while going from old to new wf
 $\lor\ missingTaskTypes(W\_worse,\ W) > missingTaskTypes(W\_best,\ W)$
 $\lor$
  $\land\ missingTaskTypes(W\_worse,\ W) = missingTaskTypes(W\_best,\ W)$
  $\land$
   2. minimize addition of tasks while going from old to new wf
   $\lor\ additionalTaskTypes(W\_worse,\ W) > additionalTaskTypes(W\_best,\ W)$
   $\lor$
    $\land\ additionalTaskTypes(W\_worse,\ W) = additionalTaskTypes(W\_best,\ W)$
    $\land$
     3. minimize reduction of task repetitions while going from old to new wf
     $\lor\ missingTaskAmount(W\_worse,\ W) > missingTaskAmount(W\_best,\ W)$
     $\lor$
      $\land\ missingTaskAmount(W\_worse,\ W) = missingTaskAmount(W\_best,\ W)$
      $\land$
       4. minimize increase of task repetitions while going from old to new wf
       $\lor\ additionalTaskAmount(W\_worse,\ W) > additionalTaskAmount(W\_best,\ W)$
       $\lor$
        $\land\ additionalTaskAmount(W\_worse,\ W) = additionalTaskAmount(W\_best,\ W)$
        $\land$
         5. minimize ordering difference of matching tasks in old and new wf
         $\lor\ diffTaskOrder(W\_worse,\ W) > diffTaskOrder(W\_best,\ W)$
         $\lor$
          $\land\ diffTaskOrder(W\_worse,\ W) = diffTaskOrder(W\_best,\ W)$
          $\land$
           6. minimize length difference between old and new wf
           $\lor\ diffWorkflowLength(W\_worse,\ W) > diffWorkflowLength(W\_best,\ W)$
           $\lor\ diffWorkflowLength(W\_worse,\ W) = diffWorkflowLength(W\_best,\ W)$
```

Workflow repair can then be performed on an input workflow, for which the "closest" valid workflow is returned, in case the input workflow contains any validation errors. If the input workflow is valid, it is returned instead. Here, we highlight a single test from the complete repair test suite, in which the actual workflow recommendation output is asserted to match the expected workflow recommendation output for the given workflow input.

```
MODULE WorkflowRepairTest

LET Repair ≜ INSTANCE WorkflowRepair WITH
Connections ←
{  [name ↦ "has_successor", srcName ↦ "EVI", dstName ↦ "XRAY"]
,  [name ↦ "has_successor", srcName ↦ "XRAY", dstName ↦ "IVI"]
},
Tasks ←
{  [name ↦ "EVI", repeatable ↦ FALSE, group ↦ "both"]
,  [name ↦ "XRAY", repeatable ↦ FALSE, group ↦ "both"]
,  [name ↦ "IVI", repeatable ↦ FALSE, group ↦ "both"]
},
Workflow ← ⟨"IVI", "EVI"⟩,
MaxDepth ← 3

ACTUAL ≜ Repair!Recommendation

EXPECTED ≜ ⟨ "EVI", "IVI"⟩
IN
    Assert(
        ACTUAL = EXPECTED,
        "should repair invalid workflow with transitive partial order violations"
    )
```

The full specification of the workflow repair functionality, including the distance measure definitions and precise input & output encoding, is given in appendix Section A.5. The complete suite of workflow repair tests is listed in appendix Section A.6.

3.2.3 Ontology

The example failure analysis process modeled as an OWL2-DL ontology, as described in Section 3.1.6, serves as the basis for reasoning about the latter part of the problem statement. The former Subsection explores how Apache Jena rules can be used for user-defined reasoning in ontologies, while the latter investigates the Shapes Constraint Language for the same purpose. Common to both approaches is the definition of an input workflow that is checked for errors, which is used for ontology reasoning, alongside the knowledge base of previously defined tasks and their connections. Such reasoning then results in a validation result or recommended alternative workflow output.

Apache Jena rules

In this Subsection, we explore how Apache Jena rules can be used for reasoning in ontologies in order to perform workflow validation & repair.

```
[PartialOrderConstraintA:
    (?typeA :PartialOrderViolation ?typeB)
    <-
    (?typeA :connected ?typeB), notEqual(?typeA, ?typeB),
    (?typeA :repeatable "false"^^xsd:boolean), (?typeB :repeatable "false"^^xsd:boolean),
    (?typeA :lastIndex ?a), (?typeB :firstIndex ?b),
    greaterThan(?a, ?b)
]
[PartialOrderConstraintB:
    (?typeA :PartialOrderViolation ?typeB)
    <-
    (?typeA :connected ?typeB), notEqual(?typeA, ?typeB),
    (?typeA :repeatable "true"^^xsd:boolean), (?typeB :repeatable "false"^^xsd:boolean),
    (?typeA :lastIndex ?a), (?typeB :firstIndex ?b),
    greaterThan(?a, ?b)
]
[PartialOrderConstraintC:
    (?typeA :PartialOrderViolation ?typeB)
    <-
    (?typeA :connected ?typeB), notEqual(?typeA, ?typeB),
    (?typeA :repeatable "false"^^xsd:boolean), (?typeB :repeatable "true"^^xsd:boolean),
    (?typeA :lastIndex ?a), (?typeB :firstIndex ?b),
    greaterThan(?a, ?b)
]
```

Src. 3.8: Excerpt from Jena inference rules used for workflow validation of example FA process

Listing 3.8 shows an excerpt of executable Apache Jena rules that perform workflow validation[17]. Here, the partial order constraint from Section 3.2.1 is modeled by three separate backward rules. As seen in these rule definitions, the rules' (implicitly) universally-qualified bodies consist of triples and built-in primitives that closely resemble the mathematical terms used in the specification of this (explicitly) universally-qualified validation constraint, after converting the constraint's implication to the equivalent conjunction of terms $P \rightarrow Q \equiv P \land \neg Q$ instead. As Jena inference rules do not support disjunction, three variations of this constraint rule are needed in order to handle all truthful cases of `(?typeA :repeatable "true"^^xsd:boolean)` $\lor$ `(?typeB :repeatable "true"^^xsd:boolean)`. Whenever either of those rules' bodies is satisfied for a suitable variable instantiation, the respective constraint violation is produced in the form of an inferred triple. The rest of such Jena inference rules are similarly constructed for the remaining constraints, as well as for some utility, derived RDF triples used in those constraints. For example, for the workflow

[17] Validation performed on Apache Fuseki (`https://jena.apache.org/documentation/fuseki2/`), a SPARQL server.

input `:Workflow rdf:type ([a :EPT] [a :EVI])` the validation error output `:EVI :PartialOrderViolation :EPT` is inferred[18].

However, to our knowledge, it is not feasible to infer "best" alternative valid workflows using Apache Jena rules, since this rule engine does not support any form of weak constraints.

Summary:

+ declarative approach to workflow validation using rules that reason directly on ontology triples
– non-standard rule format that only works on Jena's built-in reasoner
– inability to use this approach for repair of workflows, as well as forced vendor lock for workflow validation

Shapes Constraints Language

In this Subsection, we explore how the Shapes Constraint Language can be used for reasoning in ontologies in order to perform workflow validation & repair.

```
# PartialOrderConstraint
:TaskTypeShape sh:sparql :PartialOrderConstraint .
:PartialOrderConstraint
   rdf:type sh:SPARQLConstraint ;
   sh:message "(Transitive) partial order must be upheld." ;
   sh:prefixes <http://ainf.aau.at/lottraveler/> ;
   sh:select """
      SELECT ?this (:connected AS ?path) (?other AS ?value)
      WHERE {
         ?this :connected ?other .
         FILTER (?this != ?other).

         {
            ?this :repeatable \"false\"^^xsd:boolean .
            ?other :repeatable \"false\"^^xsd:boolean .
         } UNION {
            ?this :repeatable \"true\"^^xsd:boolean .
            ?other :repeatable \"false\"^^xsd:boolean .
         } UNION {
            ?this :repeatable \"false\"^^xsd:boolean .
            ?other :repeatable \"true\"^^xsd:boolean .
         }

         ?this :firstIndex ?t .
         ?other :lastIndex ?o .
         FILTER (?t > ?o).
      }
   """ .
```

Src. 3.9: Excerpt from SHACL RDF graph used for workflow validation of example FA process

Listing 3.9 shows an excerpt from a SHACL RDF graph used to perform workflow validation on the example FA process[19]. This excerpt represents the SHACL *shapes*-file, which models expectations on data. Workflows can be validated by combining this *shapes*-file with a *data*-file,

[18]Besides creating the knowledge base dataset using one of the previously shown OWL2-DL ontologies, additional queries for inserting this workflow input and for appropriately querying the inferred validation error output are needed.

[19]Validation performed via SHACL API (`https://github.com/TopQuadrant/shacl`)

which supplies RDF graphs for the definition of tasks and connections, as well as for the workflow to check. Here, a SPARQL-based constraint is defined on a *shape* that targets every task (type) which is supplied by the *data*-file. The contained `SELECT` query finds all violations of this partial order constraint, if any, by returning the offender tasks and the (transitive) connections to other tasks that are violated in the given workflow. The query consists of RDF triples and `FILTER` statements that define the desired graph pattern, which is appropriately derived from the constraint definition from Section 3.2.1, after converting the constraint's implication to the equivalent conjunction of terms $P \rightarrow Q \equiv P \wedge \neg Q$ instead, as seen before in the approach using Apache Jena rules. However, via the use of the `UNION` operator several alternative graph patterns can be explored at once, such that those alternatives correspond to respective terms used in the disjunction of this constraint specification. As this query loops over all connected tasks (types) of each shape target task (type), it triggers every such violation in the *validation report* as the evaluation of the universally-qualified partial order constraint. The rest of such SHACL constraints are similarly constructed for the remaining constraint specifications. Furthermore, the highlighted constraint and remaining constraints depend on additional utility RDF triples, which are entailed by appropriate SHACL rules. For example, for the workflow input `:Workflow rdf:type ([a :EPT] [a :EVI])` the *validation report* output in Listing 3.10 is produced[20].

```
[ rdf:type    sh:ValidationReport ;
  sh:conforms false ;
  sh:result   [ rdf:type                    sh:ValidationResult ;
                sh:focusNode                :EVI ;
                sh:resultMessage            "(Transitive) partial order must be upheld." ;
                sh:resultPath               :connected ;
                sh:resultSeverity           sh:Violation ;
                sh:sourceConstraint         :PartialOrderConstraint ;
                sh:sourceConstraintComponent sh:SPARQLConstraintComponent ;
                sh:sourceShape              :TaskTypeShape ;
                sh:value                    :EPT
              ]
] .
```

Src. 3.10: SHACL validation report RDF graph for the given input workflow of the example FA process

However, to our knowledge, it is not feasible to infer "best" alternative valid workflows using SHACL or its advanced features, as these W3C recommendations do not support any form of optimization statements.

Summary:

+ declarative approach to workflow validation using SPARQL queries that reason directly on ontology triples
+ SPARQL queries provide significant expressive power
− SHACL rules require developer to specify rule evaluation ordering, which mixes operational and declarative semantics
+ SHACL is a W3C recommendation, its advanced features are implemented in a few products[21]

[20] Validation via SHACL additionally requires the knowledge base to be embedded alongside this workflow input into the data-file, as well as helper SHACL rules in the shapes-file that produce utility RDF triples, which the validation constraints depend on.

– inability to use this approach for repair of workflows

3.2.4 Answer set programming

The example failure analysis process modeled as an answer set program, as described in Section 3.1.7, serves as the basis for reasoning about the latter part of the problem statement. The next Subsection explores how ASP can be used for workflow validation, followed by an investigation of this approach for workflow repair in the latter one. Common to both application areas is the definition of a workflow that is checked for errors and for which a valid “best” alternative is recommended. The workflow forms, alongside the previously defined tasks and their connections, the complete *problem instance*. This *problem instance* acts as the input to the *validate* and *repair problem class*, for which an ASP reasoner produces the respective output validation and recommendation.

```
workflow("Input").

orderNumber(0..1).

% example invalid workflow
workflowTaskAssignment(workflow("Input"), task("EPT"), orderNumber(0)).
workflowTaskAssignment(workflow("Input"), task("EVI"), orderNumber(1)).
```

Src. 3.11: Invalid workflow input for example FA process in ASP

Listing 3.11 shows an invalid workflow input to the answer set program that checks the input workflow for errors or recommends an alternative valid workflow. This part of the problem instance is analogously encoded as NLP facts. The workflow input consists of predicates $workflowTaskAssignment$ associating each workflow input's $workflow(''Input'')$ task $task(taskName)$ with its corresponding position in the workflow's sequence of tasks $orderNumber(n)$. These predicate arguments are again wrapped in function symbols, for added clarity on the meaning of these arguments. Necessarily, the sequence of tasks modeled via $workflowTaskAssignment$ must be contiguous (without holes) and functional (without multiple assignments for an index).

Workflow validation

In this Subsection, we explore how ASP can be used to perform workflow validation.

```
% check that task order for non-repeatable predecessor tasks and non-repeatable sucessor tasks is
    satisfied
error(workflow(W), reason(partial_order_violation), task(TA), task(TB)) :-
    not repeatable(task(TA)),
    workflowTaskAssignment(workflow(W), task(TA), orderNumber(OA)), orderNumber(OA),
    connected(task(TA), task(TB)), TA != TB,
    not repeatable(task(TB)),
    workflowTaskAssignment(workflow(W), task(TB), orderNumber(OB)), orderNumber(OB),
    OA > OB.

% check that task order for repeatable predecessor tasks and non-repeatable sucessor tasks is satisfied
error(workflow(W), reason(partial_order_violation), task(TA), task(TB)) :-
    repeatable(task(TA)),
    latestWorkflowTaskAssignment(workflow(W), task(TA), orderNumber(OAMax)), orderNumber(OAMax),
    connected(task(TA), task(TB)), TA != TB,
    not repeatable(task(TB)),
```

[21] See https://w3c.github.io/data-shapes/data-shapes-test-suite/ for a list of these products.

```
    workflowTaskAssignment(workflow(W), task(TB), orderNumber(OB)), orderNumber(OB),
    OAMax > OB.

% check that task order for non-repeatable predecessor tasks and repeatable sucessor tasks is satisfied
error(workflow(W), reason(partial_order_violation), task(TA), task(TB)) :-
    not repeatable(task(TA)),
    workflowTaskAssignment(workflow(W), task(TA), orderNumber(OA)), orderNumber(OA),
    connected(task(TA), task(TB)), TA != TB,
    repeatable(task(TB)),
    firstWorkflowTaskAssignment(workflow(W), task(TB), orderNumber(OBMin)), orderNumber(OBMin),
    OA > OBMin.
```

Src. 3.12: Excerpt from workflow validator in ASP

Listing 3.12 shows an excerpt of ASP rules that perform workflow validation[22]. This excerpt represents the *problem class*'s rules NLP_C, which form in combination with the previously discussed *problem instance*'s facts NLP_I a typical *uniform* problem definition [GKKL+15][GKKS12]. Here, the partial order constraint from Section 3.2.1 is modeled by three separate ASP rules. As seen in these rule definitions, the rules' (implicitly) universally-qualified bodies consist of user-defined predicates and built-in comparison predicates that closely resemble the mathematical terms used in the specification of this (explicitly) universally-qualified validation constraint, after converting the constraint's implication to the equivalent conjunction of terms $P \to Q \equiv P \land \neg Q$ instead. As we opt not to use the *pooling* feature for succinctly representing multiple rule variants[23], three variations of this ASP rule are needed in order to handle all truthful cases of $\mathit{repeatable}(\mathit{task}(TA)) \lor \mathit{repeatable}(\mathit{task}(TB))$. However, instead of classical negation and known-to-be false atoms, default negation is used to similarly reason about assumed-to-be false atoms, such as e.g. $\sim\!\mathit{repeatable}(\mathit{task}(T))$. Whenever either of those rules' bodies is satisfied for a suitable variable instantiation, the respective constraint violation is produced in the form of an entailed *error* predicate. The rest of such NLP rules are similarly constructed for the remaining constraints, as well as for some utility predicates used in those rules. Any encountered existentially-qualified formulae $(\exists X)F$ can be translated into appropriate count aggregates $\#\mathit{count}\{X : f\} > 0$. For example, for the workflow input from Listing 3.11 the validation output in Listing 3.13 is produced[24].

```
workflowTaskAssignment(workflow("Input"),task("EPT"),orderNumber(0))
workflowTaskAssignment(workflow("Input"),task("EVI"),orderNumber(1))

error(workflow("Input"),reason(partial_order_violation),task("EVI"),task("EPT"))
```

Src. 3.13: ASP workflow validation output for the given input workflow of the example FA process

Workflow repair

In this Subsection, we explore how ASP can be used to perform workflow repair.

An intuitive way of encoding problems is the so-called *generate-and-test*, or also known as *guess-and-check*, methodology [GKKS12][EiIK09]. It consists of the *generation* of possible solution

[22]Validation tested via `clingo` (`https://potassco.org/clingo/`), an answer set grounder and solver.

[23]*Pooling* is syntactic sugar for representing multiple rule variants with different predicate arguments succinctly $p(a;b). \equiv p(a).p(b).$, as further detailed in `clingo`'s user manual [GKKL+15]. However, we opt not to use this feature as it makes the rules harder to read in this scenario.

[24]Validation via ASP additionally requires the problem domain of tasks and their connections to be embedded alongside this workflow input into the problem instance. The problem class encoding furthermore depends on helper ASP rules that produce utility predicates, which the validation rules depend on.

candidates by means of non-determinism, followed by *tests* which check whether these candidates fulfill constraints for them to be considered proper solutions. Auxiliary concepts (predicates) may additionally be introduced to help in defining these parts [EiIK09]. Among multiple proper solutions, the “best” one may then be selected according to some optimization criteria [GKKS12]. Programs encoded in this way are, due to the non-determinism, necessarily unstratified [EiIK09].

Such generate-and-test methodology is used for generating valid workflow alternatives, as seen in the following source code listing.

```
#const maxDepth = 15.

workflow("Output").

orderNumber(0..maxDepth).

% Generate for each orderNumber potential task assignments to the workflow
{ workflowTaskAssignment(workflow(W), task(T), orderNumber(O)) : task(T) } :-
    workflow(W),
    orderNumber(O),
    W = "Output".

% Ordernumbers must start at 0 and must be continuous
:- workflowOrderNumber(workflow(W), orderNumber(O)),
    not workflowOrderNumber(workflow(W), orderNumber(O2)),
    orderNumber(O2), O2 < O.

% Workflow breadth = 1
:- workflow(W),
    orderNumber(O),
    workflowOrderNumber(workflow(W), orderNumber(O)),
    #count { task(T) : workflowTaskAssignment(workflow(W), task(T), orderNumber(O)) } != 1.

% No errors must occur
:- error(workflow("Output"), _, _).
:- error(workflow("Output"), _, _, _).
```

Src. 3.14: Valid workflow generation for example FA process in ASP

Listing 3.14 shows ASP rules that generate valid workflows[25]. As mentioned previously, this answer set program generates arbitrary task sequences and then tests whether they adhere to a specific structure and whether they raise no validation errors. Only those solution candidates that pass these tests are considered for further reasoning. In the guess part, the rule head consists of the short-form of a *count* aggregate, with default lower bound 0 and default upper bound 1. This *count* aggregate operates on a set resulting from the previous evaluation of the contained condition literal, which produces all combinations of $workflowTaskAssignment$ predicates for all possible tasks $task(T)$. Therefore, for each possible position in the generated workflow's sequence of tasks $orderNumber(n)$ either none or exactly one task assignment $workflowTaskAssignment$, out of all possible assignments to any task $task(taskName)$, is associated to the generated workflow $workflow(''Output'')$ at that position. In the test part, each such workflow candidate is then checked whether it adheres to the expected sequence structure. The first integrity constraint discards all generated workflows $workflow(''Output'')$ that do not contain a contiguous sequence of tasks $workflowTaskAssignment$. The second integrity constraint rejects all workflow candidates $workflow(''Output'')$ that contain multiple

[25]Generation tested via `clingo` (`https://potassco.org/clingo/`), an answer set grounder and solver.

task assignments $workflowTaskAssignment$ at a single sequence position $orderNumber(n)$. Note that the *count* aggregate is necessary here, as it checks for each workflow $workflow(W)$ and sequence position $orderNumber(n)$ that there exists exactly one task $task(T)$ that is generated from each such $workflowTaskAssignment$ combination. The final integrity constraints forbid any occurrences of an *error* predicate which are entailed by invalid generated workflows $workflow("Output")$, as presented in the former Subsection on workflow validation. Put differently, only generated workflows that get through workflow validation without errors are considered subsequently. Therefore, any workflow candidates passing both constraints are deemed properly-structured and valid workflow alternatives.

However, we are interested in an alternative workflow most similar to the given invalid workflow, as also motivated by the guess-and-check methodology.

```
% determine tasks that are differently ordered in both workflows
diffTaskOrder(workflow("Output"), workflow("Input"), task(T), | OOutput - OInput |) :-
    workflowTaskAssignment(workflow("Output"), task(T), orderNumber(OOutput)),
    workflowTaskAssignment(workflow("Input"), task(T), orderNumber(OInput)).


% @3: minimize difference of different task order in instance and generated workflow
#minimize { ODiff@3,T : diffTaskOrder(workflow("Output"), workflow("Input"), task(T), ODiff) }.
```

Src. 3.15: Excerpt from workflow repairer in ASP

Listing 3.15 shows an excerpt of ASP rules that perform workflow repair[26]. This excerpt combined with the previous listing represents the *problem class*'s rules NLP_C, which form in combination with the previously discussed *problem instance*'s facts NLP_I a typical *uniform* problem definition [GKKL+15][GKKS12]. Here, the task order distance measure $diffTaskOrder$ from Section 3.2.2 is entailed by an ASP rule. It computes the absolute difference of sequence positions $|OOutput - OInput|$ for each task $task(T)$ occurring in both input $Workflow("Input")$ and generated $Workflow("Output")$ workflow. Comparing each workflow alternative with every other workflow alternative, as done in the specification of workflow repair in Section 3.2.2, would require recording all workflow alternatives in an appropriate data structure in the answer set program. However, optimization statements are the idiomatic way of comparing and soliciting the "best" solution alternative, as part of such generate-and-test methodology. Therefore, this comparison of multiple distance measures between an invalid input workflow and a valid alternative workflow is broken up into multiple appropriately prioritized optimization statements on these distance measures. As seen in this excerpt, the distance measure $diffTaskOrder$ is subjected to such an optimization statement at a lower priority level. This rule contains an inner condition literal that is evaluated for each entailed $diffTaskOrder(workflow("Output"), workflow("Input"), task(T), ODiff)$ into the corresponding variable instantiation of $ODiff@3, T$. The amounts $ODiff$ in the former part of this construct $ODiff@3$ are subject to a minimization function at priority level @3, whereas the latter part T is used to uniquely tag potentially non-unique amounts (e.g. $ODiff = 1$) with the respective task name T, such that all these potentially duplicate amounts are considered in such aggregation set. Therefore, given that multiple generated workflow alternatives are tied in terms of higher priority optimization statements, this third-to-last optimization statement aims to find such a workflow alternative that contains the smallest difference in task positions to the

[26]Repair tested via `clingo` (`https://potassco.org/clingo/`), an answer set grounder and solver.

given input workflow. The rest of such NLP rules are similarly constructed for the remaining distance measures and optimization statements, as well as for some utility predicates used in those rules. For example, for the workflow input from Listing 3.11 the repair output in Listing 3.16 is produced[27].

```
workflowTaskAssignment(workflow("Input"),task("EPT"),orderNumber(0))
workflowTaskAssignment(workflow("Input"),task("EVI"),orderNumber(1))

error(workflow("Input"),reason(partial_order_violation),task("EVI"),task("EPT"))

workflowTaskAssignment(workflow("Output"),task("EVI"),orderNumber(0))
workflowTaskAssignment(workflow("Output"),task("EPT"),orderNumber(1))
```

Src. 3.16: ASP workflow repair output for the given input workflow of the example FA process

Summary

+ declarative approach to workflow validation using ASP rules that reason on domain facts
+ declarative approach to workflow repair using ASP optimization statements
∘ although some features are solver specific, significant subset of notation is based on ASP-CORE-2 reasoner input language standard [CFGI+20]

The full listing of ASP reasoning source codes, including all utility and helper rules, is given in appendix Chapter B.

3.2.5 Other approaches

Other approaches used for workflow meta-modeling, like flow-based process models and DSMLs, do not intrinsically support workflow validation or repair.

However, a naive & straightforward way to perform workflow validation with flow-based models is to first record all possible workflows resulting from every possible execution path taken in such process models. Afterwards, any workflow to-be-validated must then match (at least) one such previously found valid workflow. Furthermore, any workflows not found in this collection of valid workflows can then be repaired by finding the most similar workflow to the given input workflow, according to some similarity measure. Multiple algorithms for workflow validation & repair have been explored previously, which are presented in detail in Chapter 6 on related work. However, all these methods are described in an imperative way so that both the *logic* and the *control* part of these presented algorithms need to be implemented [Lloy94].

Although DSMLs have no intrinsic support for workflow validation & repair either, DSMLs can be used to generate further artifacts, as motivated by the software-engineering practice of domain-specific modeling in background Section 2.5. The following Section explores this aspect of DSMLs.

3.3 Combining modeling & reasoning

Having examined multiple alternatives and how they can be used to solve both parts of the problem statement, a solution must now be chosen to implement a system for workflow manage-

[27]Repair via ASP additionally requires the problem domain of tasks and their connections to be embedded alongside this workflow input into the problem instance. The problem class encoding furthermore depends on helper ASP rules that produce utility predicates, which the repair rules depend on.

ment in *typical* production environments. However, as shown in the previous sections, there is no one-solution-fits-all approach that satisfies the needs for both the design-time experts, which need an easy-to-use and easy-to-understand workflow meta-model for expressing their domain knowledge about valid workflow models, as well as for the service providers that prefer to implement workflow validation & repair based on a declarative approach stating only "*what* is to be computed, but not necessarily *how* it is to be computed" [Lloy94].

Therefore, we can divide-and-conquer and combine the best of both worlds by handling each part of the problem with a different sub-approach. For workflow meta-modeling, a graphical model notation is easier to understand, as also corroborated by subjects of a study on the differences between graphical and textual declarative process models [HaZu14]. Furthermore, declarative *constraint-based* process models allow for more flexibility in representing workflows with many alternative execution paths than imperative *flow-based* ones [Slaa20][ABSK$^+$20], and are more suited "for tailored solutions to unique problems" [Slaa20]. In agreement to this sentiment, Grambow, Oberhauser and Reichert state how "traditional workflow modeling for [their business domain is] difficult since many different activity sequences matching different situations would have to be integrated in one vast workflow model" [GrOR11]. On the other hand, for workflow validation & repair, a solution is required that supports both reasoning on constraints and optimization of preferences among multiple alternatives.

The implementation shall thus use a DSML for encoding the design-time requirements of the domain[28], and answer set programs for run-time reasoning on this domain. As both sub-approaches are based on the same problem domain concepts, task and connection objects specified in such domain-specific model can directly be mapped to facts of a NLP's problem instance. These concepts can also be mapped from & to OWL ontology entities and properties, in the same vein. Figure 3.7 depicts this combination of sub-approaches.

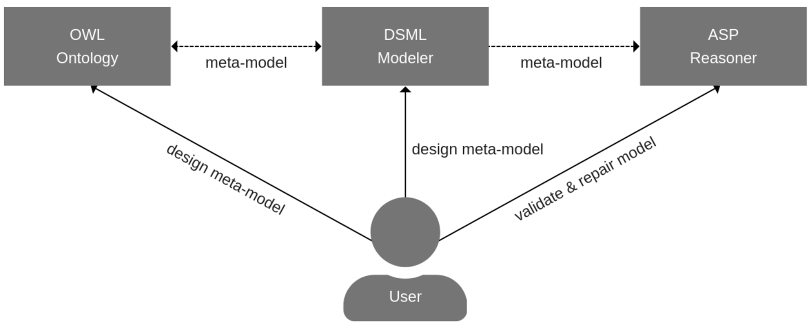


Figure 3.7: LotTraveler's combined approach for workflow meta-modeling and workflow reasoning

The next Chapter 4 outlines how these different parts of the implementation operate and interact with each other.

[28]Although CMMN would be a valid, more standardized alternative for representing design-time requirements of processes, we opt not to use it due to limited tool support and difficulty of reasoning about the significantly expressive semantics of (a subset of) this notation.

4 Implementation

This Chapter describes the implementation details of the system, which is realized according to the combined approach presented in approach Section 3.3, that is linked to a class-based OWL2-DL conform ontology, as discussed in approach Section 3.1.6. The complete implementation source code is hosted alongside setup and run instructions on a publicly available source code repository[1]. Additional, auxiliary materials are available on a private repository, made accessible to select stakeholders from Infineon and university.

The introductory Section provides an overview of the system and its modules, followed by more detailed descriptions of each module's implementation. In conclusion, the system's deployment possibilities are examined.

4.1 High-level overview of the implementation

In this Section, we give an overview of components implemented in the system, how those are connected to each other and the communication flow between them. Subsequent sections elaborate on specific implementation details of these modules and how they can be used.

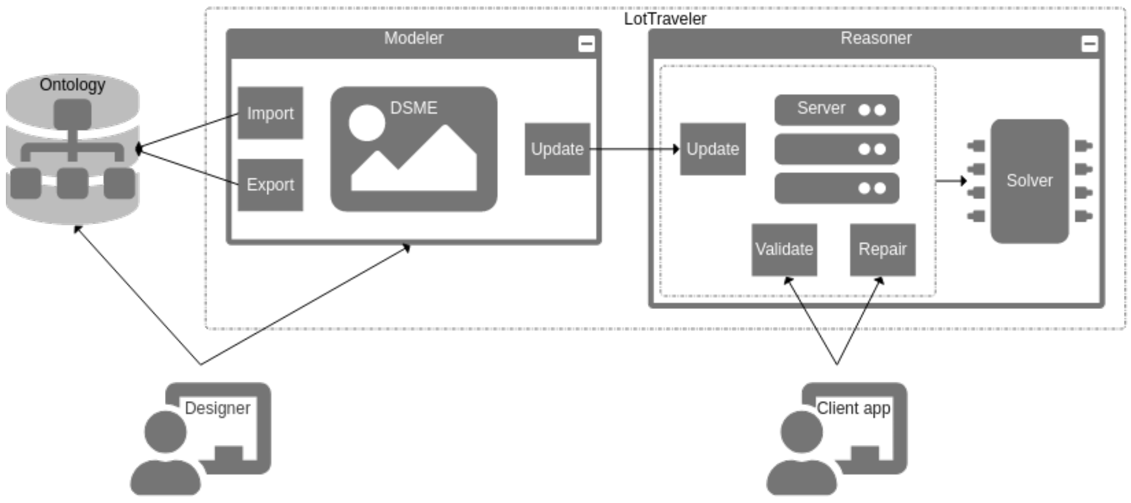


Figure 4.1: Overview of LotTraveler's implementation, its modules, users & external components, and the available interactions between them

[1]The source code repository is available at `https://github.com/mucaho/lottraveler`.

Figure 4.1 shows a conceptual overview of LotTraveler's system implementation, its modules, users & external components. Furthermore, directed edges represent the control-flow of available interactions between external interfaces of those modules and users & components.

The *modeler* module enables domain experts and designers to create a workflow meta-model which captures all ground rules for any custom-tailored or template workflow. Experts list all possible tasks, which tasks may follow other tasks, as well as mandatory tasks that must appear alongside other ones, in such domain-specific modeling environment. On one hand, the modeler interfaces with an ontology which relates tasks with each other and with additional domain concepts of the business domain. This includes supplementary information about samples that are analyzed by such tasks, as well as any broader machine-processable documentation of interest to the company. This taxonomy, that relates domain concepts with each other, acts as the single source of truth for the workflow meta-model. Designers can thus directly encode workflow ground rules in this ontology[2], before *importing* it into the modeler module. However, since the DSME of the modeler is specialized on modeling this particular slice of the whole business domain, domain experts get a more focused visual overview and are guided by additional meta-model sanity checks that forbid invalid designs. Therefore, the modeler supports *exporting* a modified meta-model, or even one built from scratch, back into the ontology. Do note that *importing* and *exporting* are optional functionalities. The LotTraveler system is self-contained and works without interfacing with such ontology. On the other hand, the modeler interfaces with another internal module used for workflow reasoning. Once the domain experts are satisfied by the initial design of ground rules or want to modify prior such meta-model of tasks and their connections, they can push an *update* to the other internal module. Any further automated processing on the other module will then use this updated set of workflow constraints.

The *reasoner* module enables users to validate and repair any custom-tailored workflows against these constraints. A user can supply a sequence of tasks to check for errors and possibly get an alternative recommendation via a request from an external application[3] or directly from the web interface of this reasoner. It consists of a web *server* that acts as a *facade* for easier interaction with the answer set *solver* in the background [EGGGB$^+$95]. After an *update* request is triggered from the modeler with the new meta-model of tasks and connections, the solver is supplied with such updated problem domain instance. Client applications can trigger *validate* and *repair* requests, such that the input workflow problem instance is forwarded analogously to this solver. After the solver is done reasoning on the combined problem instance, its validation & repair output is returned as a (synchronous) response to such requests.

A guide to using the system is available in appendix Chapter C.

4.2 Workflow meta-modeling environment

Here, we present the implementation of the *modeler* module where ground rules for all workflows can be modeled.

[2]For example, Protégé (`https://protege.stanford.edu/`) may be used to load and modify the ontology hosted on an Apache Fuseki server (`https://jena.apache.org/documentation/fuseki2/`).

[3]For example, Postman (`https://www.postman.com/`) may be used to explore and test the API surface of this web service.

The *modeler* module is based on a web application known as `WebGME`, "a web-based, collaborative meta-modeling environment"[4], with customized additions that suit our use-case. As it is a web application, most of the business logic is run on the server, whereas the browser acts as the client and takes care of the presentation and dispatch of appropriate requests to the server. `WebGME`'s server runs on a NodeJs backend, which is a JavaScript[5] runtime[6], whereas its client runs on most modern desktop web browsers[7]. Taking the open-source implementation of `WebGME` as the foundation, we thus implemented additional addons[8], plugins[9] and decorators[10] in order to get the desired system behavior. The JavaScript implementation of these extensions was mainly written using the 5th JavaScript standard[11], whereas some parts follow the newer 8th JavaScript standard[12].

A *TaskDecorator* runs on the client and wraps the original display code in order to "attach[] additional responsibilities" [EGGGB+95] to the representation of task objects. Namely, a task's respective *group* and *repeatable* attributes are visualized as part of task names in the form of a suffix to distinguish between [N]on-destructive and [D]estructive tasks, as well as in the form of an asterisk suffix * for repeatable tasks. This decorator uses the internal client API provided by `WebGME`[13] to adapt the task name display appropriately.

A *CheckDAGAddon* is run on the client whenever the meta-model is changed in the application and checks whether the meta-model still passes some sanity checks. Firstly, as also discussed in Section 3.1.5 on the problem domain, this addon verifies whether the meta-model contains any cycles. Secondly, it checks whether multiple tasks with the same name are created in the modeling environment. In case of violations, an error message is displayed in the editor. This addon uses the internal core API provided by `WebGME`[14] to query tasks and their connections.

The *ImportFromOntology* and *ExportToOntology* plugins are run on the server following a manual user activation and allow the user to import tasks and their connections from an ontology, modify them or create new ones from scratch, and export them back to this ontology. The ontology needs to be a OWL2-DL conform class-based process ontology variant, where tasks are modeled as classes and connections are represented as class expressions on specific data & object property restrictions, as presented in excerpt 3.6. When an import is performed, all existing tasks and connections are first wiped in the modeler before the imported ones are created from this ontology. Connections that do not adhere to expected class expressions on property restrictions, cannot be imported. Namely, the modeler can only import simple connections in the form `srcTask SubClassOf :has_predecessor some dstTask` or `srcTask SubClassOf :has_predecessor min 1 dstTask` (analogous for the other connection types). More complex statements, like e.g. `srcTask SubClassOf :has_predecessor some (dstTaskA and dstTaskB)` cannot currently be imported. However, similar (but not exact) semantics can be achieved by splitting up the connection into multiple

[4]See `https://webgme.readthedocs.io/en/latest/introduction/what_is_webgme.html` for more details.
[5]See `https://developer.mozilla.org/en-US/docs/Web/JavaScript` for more details.
[6]See `https://nodejs.org` for more details.
[7]See `https://caniuse.com/usage-table` for a list of popular web browsers.
[8]See `https://github.com/webgme/webgme/wiki/GME-Add-Ons` for more details.
[9]See `https://github.com/webgme/webgme/wiki/GME-Plugins` for more details.
[10]See `https://github.com/webgme/webgme/wiki/GME-Decorators` for more details.
[11]See `https://262.ecma-international.org/5.1/` for more details.
[12]See `https://262.ecma-international.org/8.0/` for more details.
[13]See `https://github.com/webgme/webgme/wiki/GME-Client-API` for more details.
[14] See `https://github.com/webgme/webgme/wiki/GME-Core-API` for more details.

ones, with e.g. `srcTask SubClassOf :has_predecessor some dstTaskA` and `srcTask SubClassOf :has_predecessor some dstTaskB`. As the ontology is the single source of truth of domain knowledge, the export functionality was implemented cautiously as to not overwrite unrelated concepts or otherwise corrupt the data integrity in this ontology. Furthermore, the export plugin adheres to the following modus operandi in this deliberate order of execution:

1. Optionally renames tasks that were renamed in modeler. This feature is opt-in, such that the user cannot rename tasks in the ontology accidentally. Can be enabled by ticking the appropriate option when running the *ExportToOntology* plugin.
2. Drops all connections between tasks that are managed by the modeler. Connections that cannot currently be imported, due to the previously stated import limitation, will not be modified or deleted.
3. Optionally deletes all tasks that are not contained in the modeler. This feature is opt-in, such that the user cannot delete tasks in the ontology accidentally. Can be enabled by ticking the appropriate option when running the *ExportToOntology* plugin. Do note that deleting tasks in the middle of class hierarchies may leave children of deleted tasks "stranded". Such non-trivial task class hierarchies need to be manually corrected afterwards[15].
4. Inserts new and augments existing tasks.
5. Inserts connections between tasks.

These plugins use the internal core API provided by `WebGME`[14] to query and update the meta-model. Communication with a SPARQL server[16] is done via the library `sparql-http-client`[17]. The *ImportFromOntology* plugin posts multiple SPARQL queries to the endpoint and parses each result sequentially. On the other hand, the *ExportToOntology* plugin bundles all SPARQL delete & insert queries into a single one by combining individual queries with a statement separator semi-colon ";", so that this combined query is executed in a single atomic transaction. If an error occurs in one of those bundled queries, the SPARQL endpoint automatically rolls-back any previous queries and thus leaves the ontology in a consistent state.

The *UpdateReasoner* plugin runs on the server following a manual user activation and allows the user to update the problem domain instance in the *reasoner* module. Namely, it finds all tasks and their connections in the meta-model, builds a problem domain instance as specified in Section 3.1.5, and then forwards this data to the reasoner. This plugin uses the internal core API provided by `WebGME`[14] to query the meta-model. Afterwards, the library `axios`[18] is used to execute a `POST` HTTP request against the reasoner web-service using `JSON`-encoded tasks and their connections from this meta-model. The `POST` payload's body is made available for download from the modeling environment's interface, using the internal blob storage API provided by `WebGME`[19].

[15]This can be done with e.g. Protégé (`https://protege.stanford.edu/`).
[16]In our implementation an Apache Fuseki server (`https://jena.apache.org/documentation/fuseki2/`) is used.
[17]See `https://github.com/zazuko/sparql-http-client` for more details.
[18]See `https://github.com/axios/axios` for more details.
[19]See `https://github.com/webgme/webgme/wiki/GME-Blob-Storage-API` for more details.

Beside these extensions to `WebGME`, the DSME available to the user is further customized through the use of a read-only library[20], which defines the base building blocks available for workflow meta-modeling. This library encapsulates the meta-model of the workflow meta-modelling environment and thus contains the definition of the *Task* and the different *Connection* base objects, as showcased in Figure 3.5. A workflow meta-model then imports this library and the contained building blocks, and can thus instantiate various *Task* and *Connection* objects that point to their respective prototypes from this library, such as depicted in Figure 3.6. The extraction of meta-information into a separate library prevents accidental modification of these base prototypes and allows the reuse of this library across multiple workflow meta-models, if desired. However, every such workflow meta-model still needs to have the aforementioned extensions to `WebGME` enabled for its *Root* object, since these extensions cannot be enabled through the import of this library. Refer to the user guide in appendix Chapter C for more details regarding the usage of these meta-modeling building blocks.

The *modeler* implementation was tested using manual system tests, by trying different scenarios & use-cases and comparing the obtained results to our expectations. This includes checking for the desired effect in the modeling environment itself, as well as for the expected effect on the *reasoner* and ontology. One particular test checks whether the amount of triples in the ontology stays constant through repeated imports and exports from this ontology, such that no triples "leak" in such idempotent import / export cycle.

4.3 Workflow validation and repair service

Here, we present the implementation of the *reasoner* module where workflows can be validated and repaired against a previously stated set of ground rules.

The *reasoner* module consists of two main parts —a web *server* and an answer set *solver*. The server hosts a web-service, which in turn provides service methods */updateKnowledgeBase* for updating tasks and their connections, */validate* for validating a workflow against these constraints and */repair* for repairing an invalid workflow by providing a valid, alternative workflow recommendation. The server forwards these service method calls to an answer set solver, that is used to reason about the respective problem instance in the background. Namely, */updateKnowledgeBase* can only succeed if the provided tasks and connections pass some sanity checks in the respective NLP. The workflow input for */validate* and */repair* is similarly forwarded to respective NLPs, after which the resulting validation errors and recommended alternative workflow are returned from these service calls.

The web server is built on ExpressJs, a NodeJs web application framework[21]. It was written using the 5th JavaScript standard[22]. Each of the web-service methods was implemented as an ExpressJs route[23] that responds to `POST` requests on the aforementioned route path. HTTP requests and responses are validated according to an OpenAPI specification via the `express-openapi-validator` middleware[24], that "defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand

[20]See `https://github.com/webgme/webgme/wiki/GME-Libraries` for more details.
[21]See `https://expressjs.com/` for more details.
[22]See `https://262.ecma-international.org/5.1/` for more details.
[23] See `https://expressjs.com/en/guide/routing.html` for more details.
[24]See `https://github.com/cdimascio/express-openapi-validator` for more details.

the capabilities of the service"[25]. This specification thus defines all possible input and outputs that these methods can receive and return, and includes appropriate examples for each such case. If an HTTP request does not adhere to the expected structure, an erroneous HTTP response is returned outright. Otherwise, given that the incoming request matches the expected format of this specification, its `JSON`-encoded body is parsed and written to a `tmp` file[26], with which the answer set solver called, alongside other persistent files that constitute the problem instance and class encoding of such NLP. Afterwards, the output of the solver is parsed and converted to an appropriate successful or erroneous HTTP response, depending on whether the input workflow passes the validation checks, containing a `JSON`-encoded payload. As NodeJs runs on a single-threaded event loop[27], all involved I/O operations throughout the entire call stack were implemented in a non-blocking way so that multiple requests may be executed concurrently. This includes request routes[23], file operations[28] and solver child process executions[29]. These operations were implemented using asynchronous callbacks[30], as well as promises[31]. Extra care was taken to ensure possible I/O & parsing errors are handled properly and result in a corresponding error HTTP response, while also cleaning up any dangling I/O handles. Do note that although these methods use an asynchronous implementation internally, the HTTP request-response cycle is still implemented synchronously, such that the HTTP response from the server is returned in the same HTTP connection as the request from the client. Beside the aforementioned web-service methods defined in the specification, the server exposes utility routes */api-docs*, */getKnowledgeBase* and */spec*. The */getKnowledgeBase* route can be used to `GET` the current set of tasks and connections, whereas */spec* statically hosts[32] the OpenAPI spec. Route */api-docs* facilitates an auto-generated SwaggerUI[33] API documentation via the use of the `swagger-ui-express` module[34]. This API documentation serves as an interactive overview over available service methods and presents expected requests and responses from the OpenAPI specification in a human-readable way. Furthermore, it allows the execution of predefined or custom requests and shows the corresponding responses. It thus facilitates a web-based "playground" interface to the solver, beside more *typical* access scenarios via external client applications.

`Clingo`[35] is the ASP grounder & solver used for reasoning about different NLPs, which are partly constructed by the corresponding service methods. The logic programs are based on the ASP approach to workflow meta-modeling and workflow validation & repair, which is described in sections 3.1.7 and 3.2.4, respectively. As explained in these sections, each NLP consists of the problem instance NLP_I facts, common to both validation & repair, and the problem class encoding NLP_C rules, which are different between the validation NLP_{C_V} & re-

[25]See `https://swagger.io/specification/` for more details.
[26]See `https://github.com/raszi/node-tmp` for more details
[27]See `https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/` for more details.
[28]See `https://nodejs.org/api/fs.html` for more details.
[29]See `https://nodejs.org/api/child_process.html` for more details.
[30]See `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing#async_callbacks` for more details.
[31]See `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing#promises` for more details.
[32]See `https://expressjs.com/en/starter/static-files.html` for more details.
[33]See `https://swagger.io/tools/swagger-ui/` for more details.
[34]See `https://github.com/scottie1984/swagger-ui-express` for more details.
[35]See `https://potassco.org/clingo/` for more details. At implementation time `clingo` version 5.4.0 was used.

pair NLP_{C_R} functionality. Moreover, the problem instance NLP_I is split into the problem domain instance NLP_{I_D}, comprised of available tasks and their connections, and the problem workflow instance NLP_{I_W}, providing the sequence of tasks to reason about. The web-service method */updateKnowledgeBase* thus updates the problem domain instance NLP_{I_D} such that these workflow constraints are used in all further */validation* and */repair* requests. On the other hand, each call to methods */validate* and */repair* thus consist of its own, request-specific problem workflow instance NLP_{I_W} so that the given workflow is checked against these previous constraints. The implementation of the *validate* and *repair* system functionality was done exactly according to the ASP approach presented beforehand in Section 3.2.4, except for an additional optimization criteria that was added to workflow *repair*. This additional optimization statement was added at the lowest priority and provides an additional tie-breaker in case of multiple "best" workflow recommendation alternatives. Namely, it penalizes workflow alternatives that repeat a single task multiple times before the first occurrence of other tasks, in lieu of additional ordering or dependency restrictions. We found that this criterion produces more natural workflow alternatives, such that $\langle A, B, A, B\rangle$ is recommended over $\langle A, A, B, B\rangle$, for example. Furthermore, the implementation of *updateKnowledgeBase* introduced additional sanity checks to provided tasks and connections. Implicit assumptions made in the problem domain specification, as seen in appendix Section A.1, are explicitly modeled as respective constraint violations that produce additional *error* predicates for such invalid knowledge bases. These constraints check whether the given task and connection facts are *consistent* and thereby follow the expected structure, and whether the provided connections induce a directed *acyclic* graph. The web-service takes care of calling `clingo` with appropriate temporarily-created & persistent logic program files, which together build the complete uniform NLP definition. Appendix B lists all these logic program files and declares exact file and other arguments supplied to the solver, depending on the desired reasoning mode.

The *reasoner* implementation was tested using automated functional system tests, in which post requests are made to the three available web-service methods and the actual responses are compared with the expected responses from the web-service. Validation & repair tests were directly translated from the complete set of problem specification tests, as listed in appendix sections A.4 and A.6. Further tests were written for the behavior of the */updateKnowledgeBase* method, to check whether invalid meta-models comprised of duplicate tasks and cyclic connections are properly rejected by the service.

4.4 Deployment

Both system modules can either be run locally, by setting up necessary dependencies and external test components, or by integrating them into a company's existing IT infrastructure, where these modules share access to such dependencies and components.

The *modeler* module's `WebGME` web application internally relies on a Mongo database[36], into which application data is saved permanently. This includes workflow meta-models that are being worked on and used, such that these persist through restarts. Furthermore, the *modeler* requires an external SPARQL server to be configured into the system's configuration, for the optional *import* and *export* features to function. On the other hand, the *reasoner* module requires

[36]See `https://www.mongodb.com/` for more details.

a (virtual) file storage system, onto which the current workflow constraints are saved as a file. This file storage needs to be persistent, in case the knowledge base of tasks and connections are to be preserved between system restarts.

In order to run the system in both a local environment for testing, and through deployment and integration as part of an existing IT infrastructure for production use, we chose to package the system using the Docker platform and related technologies[37]. Each of the system's modules, dependencies and external components is packaged into a separate Docker image. A Docker image specifies all its required dependencies, including the operating system, the operating system's configuration, required system packages, as well as the application itself that is run once such self-contained image is started as a Docker container. A Docker container is thus a running instance of a Docker image in a (loosely) isolated environment. It can be thought of as a lightweight virtual machine, such that the user-space is kept separate for each container, whereas the system-kernel is shared among all containers. Multiple such Docker containers are connected to each other as services inside a Docker stack. A Docker stack allows otherwise isolated containers to discover and talk to each other, either via the stack's virtual network or via persistent virtual file systems, called volumes. Furthermore, when a Docker stack is spun up, containers that are required for other containers to function are started in the appropriate order.

We thus packaged the *modeler*, *reasoner*, a Mongo database and an Apache Fuseki SPARQL server as separate services in such Docker stack, for use in a local development environment. When the Docker stack is run locally, all these services are in turn spun up. As an anonymous persistent volume is implicitly created for each service, the Docker stack can safely be stopped and restarted without loss of application data. Alternatively, both system modules can be run locally as "plain" NodeJs processes using appropriate `npm` scripts[38]. Furthermore, the *modeler* and *reasoner* are additionally packaged as separate Docker images. This allows such container package definitions to be used on production-grade container orchestration platforms, like Kubernetes[39] and OpenShift[40]. For example, with proper configuration of appropriate environment variables, these Docker images can be run as part of OpenShift pods[41].

[37]See `https://docs.docker.com/` for more details.
[38]See `https://www.npmjs.com/` for more details.
[39]See `https://kubernetes.io/` for more details.
[40]See `https://www.redhat.com/en/technologies/cloud-computing/openshift` for more details.
[41]See `https://docs.openshift.com/container-platform/4.9/cicd/builds/build-strategies.html` for more details.

5 Results and discussion

In this Chapter, we reflect on the results of our work by recalling the general steps that led us here. Afterwards, we discuss those results and the thus presented approach.

5.1 Results

We used a declarative process model for workflow meta-modeling, with which we validated whether a workflow model's sequence of tasks passed all the rules in such meta-model. Furthermore, we generated valid workflow models and then used a notion of similarity to find the best possible alternative to a given invalid workflow model. In particular, we leveraged the modeling concepts of *tasks* and *connections* as a domain-specific alternative for both modeling and reasoning on allowed workflow models.

However, there was no one-solution-fits-all approach that satisfied the needs for both the design-time experts, which need an easy-to-use and easy-to-understand workflow meta-model for expressing their domain knowledge about valid workflow models, as well as for the service providers that prefer to implement workflow validation & repair based on a declarative approach stating only "*what* is to be computed, but not necessarily *how* it is to be computed" [Lloy94]. To combat this, we thus applied divide-and-conquer and proposed an approach that combines the best of both worlds by handling each part of the problem with a different sub-approach. This combined approach thus uses a DSML for encoding the design-time requirements of the domain, and answer set programs for run-time reasoning on this domain.

We evaluated the proposed approach by applying it to the area of failure analysis (FA) in Infineon AG Villach. To this end, we used the visual modeler to construct a workflow meta-model of tasks and their connections encountered in a *typical* failure analysis department. Using this meta-model, we were able to properly represent ground rules common to 14 elicited template workflows. All these workflows successfully passed workflow validation in the reasoner, according to those previously defined constraints.

5.2 Discussion

This Section shows our findings, presents lessons learned along the way, and discusses threats to validity of the proposed approach and limitations of the implementation.

5.2.1 Findings

Here, we come to and end of our pursuit of finding answers to the research questions stated in the introductory Section 1.2. As the combined approach was also implemented into a working system, these answers are founded on both a theoretical and practical basis. This is what we learned:

RQ1 What do these workflows *typically* look like? How can we model all potential workflows that are requested by clients and that are subsequently checked by experts in order to determine whether they follow business rules and domain restrictions? Which notation, with formally defined semantics, can we use to represent such valid workflows? Can we identify common modeling elements to capture ground rules which all custom-tailored workflows need to adhere to?

A1 Workflows are linear bounded[1] sequences of tasks, where these tasks are executed one after another. Business rules restrict available tasks, state which tasks may follow other tasks, as well list any mandatory tasks that must appear alongside other ones. Examples are provided in Subsection 3.1.1.

We can formally define these ground rules by constructing an appropriate (business) process model that (implicitly or explicitly) captures all allowed execution sequences (workflows) adhering to these ground rules. We can use the concepts of *tasks* and *connections* as a domain-specific alternative for workflow (meta-)modeling. A complete description and specification is given in Subsection 3.1.5.

RQ2 How can we define and verify these validness constraints of workflows? In which way can these previously identified modeling elements be used to encode such constraints? How can we check workflow consistency automatically?

A2 For a workflow to be valid, its sequence of tasks must fit the workflow (meta-)model. This sequence must thus completely align with at least a single process (meta-)model execution. Those previously identified (meta-)modeling elements of *tasks* and *connections* hereby directly encode respective validness constraints. If the input sequence of tasks passes all such (meta-) model constraints it is (implicitly) completely aligned. A complete description and specification is given in Subsection 3.2.1.

RQ3 How can an automated repair procedure construct the "best" valid workflow for an inconsistent one? How can we find such alternative efficiently?

A3 A "best" valid alternative to an inconsistent workflow is chosen according to some notion of similarity. We define an appropriate notion of similarity usually via a similarity metric. However, in our case, we use multiple distance metrics instead. We then generate all valid workflows and choose the "best" (most optimal) according to this notion of similarity. As the "generati[on of] the optimal recovery of missing events is [] NP-hard" [WSZL13], we argue how our repair procedure is sufficiently efficient for small workflow state spaces. A complete description and specification is given in Subsection 3.2.2.

Despite the base ingredients being well known and researched, we believe the proposed approach is a novel combination of procedures for event log repair in small process state spaces, not only for incomplete but also for incorrect event traces.

[1] A workflow sequence typically contains no more than 15 tasks in failure analysis, for example.

5.2.2 Lessons learned

Investing additional time in writing the system specification helped identifying problems in the reasoner implementation. Furthermore, insights gained from writing the implementation propagated back to the refinement of the system specification. It was an iterative refinement and streamlining process. In particular, writing the specification helped us understand how to implement proper order checking between repeatable tasks. Furthermore, we uncovered a major flaw in our initial implementation where we did not account for the transitivity of the partial order relation. Tests helped us assert that the actual system behavior matches the expected one. Therefore, due to both system specification tests and reasoner tests matching our expectations, we have high confidence that both represent the expected system behavior. As an aside, writing behavioral tests using the given-when-then layout helped us structure these tests. Furthermore, SPARQL test scripts helped us in identifying issues related to leaky triples that would accumulate in the ontology after repeated import & export cycles. We fixed the problem by making the import & export cycles idempotent.

We find a domain-specific declarative process modeling language to be practical in allowing users to express their domain knowledge using a manageable set of modeling elements inside a graphical editor. The underlying domain concepts seem understandable for domain experts, whereas constraints allow for more flexibility in the model design. These are advantages that must typically be weighed against each other when choosing between an imperative or declarative process modeling language.

More expressive SHACL-AF rules and all Apache Jena rules mix operational semantics with declarative ones by forcing the user to specify rule execution ordering. We intended to evaluate the use of SPIN magic properties instead, which seemingly do not suffer from this limitation, but found a single proprietary & commercial vendor supporting it only. More to this point, we find reasoning approaches that operate on an ontology directly and write the inferred triples back into the ontology convenient. Due to favorable computational properties, we would have preferred to use ontology reasoning instead of the latter sub-approach. However, it was unsuitable for performing workflow repair with it.

5.2.3 Threats to validity

In this Subsection, we look at different threats to validity of the proposed approach, as well as limitations encountered in the implementation.

Our proposed approach is not scientifically evaluated against other approaches in the area of event log repair. As our approach generates only perfectly fitting event sequences, a comparison on the similarity between the repaired event log and the given event log would be required instead. This would either disprove or prove our hypothesis that current event log repair approaches do not properly account for event transpositions, in contrast to the addition of missing events. As the proposed approach is intended for process models with small state spaces, a comparison on the performance characteristics would be required as well to find the “sweet spot” of the state space cardinality, for which our approach is feasibly applicable.

We found an additional constraint for the FA example process presented in Section 3.1.1, which we cannot accurately model using the current notion of tasks and connections from the workflow meta-model: *Following a layer stripping at least one of the two available analysis tasks on that* *layer, `TEM` or `SAM`, is to be performed.* By looking at the process represented as a BPMN model in Figure 3.1, we see that both `TEM` and `SAM` tasks are completely optional in a process execution. The same is true in the respective domain-specific model in Figure 3.6, because the `STRIP` task has no mandatory successor. We can modify the BPMN model to represent this behavior by removing the bottom two branches from the enclosing exclusive gateway and adding the respective counterpart as an optional successor task in the top two branches. However, the DSM cannot be extended in any way to correctly model this behavior using the currently available domain concepts. Adding mandatory successor connections from `STRIP` to both `TEM` and `SAM` would require both tasks to be executed after layer stripping. There exist multiple solutions to the extension of current concepts so that these constraints can be modeled, like the inclusion of XOR, AND & OR gateways for grouping tasks and allowing connections between tasks and such task groups, but that is left to future work.

Furthermore, we examined a non-exhaustive collection of process modeling languages only. Other languages may fare better in representing the business constraints. In particular, we need to further analyze YAWL (and its worklets), and various declarative & hybrid process modeling languages for their understandability and flexibility in representing valid workflow executions. More to this point, the evaluation of quality criteria for each approach corresponds to our subjective opinion, based on our own experiences and findings.

Although initial user reception was positive, the implemented system has not been scientifically evaluated for its usability according to a proper analysis of user feedback. There are additional non-critical improvements that can be made to the current implementation, which are discussed in more detail in future work Chapter 7.1.

6 Related work

This Chapter lists related works encountered in literature. The first Section lists the state of the art of alternative process modeling approaches not explored in this thesis, while the latter examines scientific articles about workflow validation & repair.

6.1 Workflow modeling

In the ensuing subsections, we present related works in the context of workflow modeling.

6.1.1 Flow-based workflow modeling

Lu, Bernstein and Lewis summarize how "several formal methods have been proposed for specifying and modeling workflows. These include event algebra[], state charts[], Petri nets[], temporal logic[], and concurrent transaction logic[]" [LuBL06]. Therefore, this Subsection explores a sample of related works that use formal methods for workflow modeling.

Van Der Aalst motivates the use of Petri nets as a "well-founded process modeling technique" [Aals99]. The author argues how the formal semantics and properties of Petri nets facilitate various types of analyses that can be performed on such process models, such as checking for safety properties and deadlocks[1]. This work furthermore presents an algorithm for the generation of such correct Petri nets from a bill-of-materials product structure.

In their work [KoOb05], Koschmider and Oberweis build an ontology of Petri net concepts. It captures the formal semantics of places, transitions and their relationships to other modeling elements of Petri nets. The authors provide a graphical editor, which allows a user to model instances to such Petri net modeling element classes exposed by this ontology. The editor is operated in an intuitive manner as typical Petri net modelers, by adding places and transitions and drawing arcs between them, which triggers the creation of these individuals in the background. However, the user can additionally define values of data properties on these individuals in a pop-up dialog. The representation of Petri net process models via respective OWL individuals opens up the possibility for reasoning on this semantic data, via built-in OWL entailment or other rule-based languages that operate on ontologies.

Roser, Lautenbacher and Bauer suggest the use of common process modeling format that "enables people with no or little experience in [...] workflow technology" to generate executable workflows from higher-level, more abstract representations [RoLB07]. Namely, by writing appropriate *adapters*, a given process description written using a process-oriented domain-specific

[1]These properties are examined in background Section 2.1.4.

language (like e.g. UML activities) can be transformed to this common process modeling format. On the basis of this common format, the process is afterwards optimized and transformed into constructs of the target executable workflow language (like e.g. BPEL), leveraging graph transformation algorithms and appropriate code *generators*.

6.1.2 Constraint-based workflow modeling

This Subsection explores related works which model workflows by declaring general-purpose or domain-specific constraints between tasks. This is of particular interest to workflow modeling in production environments, since declarative constraint-based process models allow for more flexibility in representing workflows with many alternative execution paths than imperative flow-based ones [Slaa20][ABSK$^+$20].

In their work [GrOR11], Grambow, Oberhauser and Reichert propose a constraint-based process model, which refines their previous approach to declarative process modeling. In particular, *sequencing constraints* and *existence constraints* enforce an execution order and dependencies between tasks, respectively. Given tasks A and B, the constraint A *hasSuccessor* B declares that, if both A and B occur in a workflow, then A must be executed before B. Alternatively, relationship A *hasParallel* B denotes that, if both tasks A and B are present in the workflow, they must be executed in parallel. The sequencing constraint *hasSuccessor* is furthermore transitively applied to subsequent tasks (or task groups connected via *hasParallel*) throughout such process model. The association A *requires* B dictates that A can never be executed without B being also present in the workflow. On the other hand, A *mutualExclusion* B forbids the inclusion of task B in a workflow that contains task A. The proposed process model also supports sequential, parallel and repeatable execution of task groups through the declaration of so-called *building blocks*. Building blocks can be nested inside other building blocks, fostering such hierarchical composition of these workflow constructs. The proposed workflow model thus consists of the set of available tasks, the given set of constraints on these tasks and the provided set of building blocks. Using OWL-DL entailment and SWRL rules (similar to Apache Jena rules), the workflow model encoded as an ontology is first checked for its consistency, such that e.g. it contains no cyclic ordering dependencies. Afterwards, such consistent workflow specification is then used for generating workflows that adhere to the given constraints and hierarchical composition.

The definition tasks and connections relating those tasks, as presented in detail in this thesis' approach Section 3.1.5, is to a great extent inspired by Grambow, Oberhauser and Reichert's findings. The connection types *has_successor* & *has_predecessor* exhibit similar semantics as the constraint *has_successor*, by imposing a transitive order relation between two associated tasks. Furthermore, connections *has_mandatory_successor* & *has_mandatory_predecessor* facilitate a similar declaration of a task's dependency as the constraint *requires*. Task (group) repetition is handled seemingly analogously via a task's *repeatable* attribute. Cyclic task dependencies are likewise forbidden in our work's domain specification. On the other hand, the exact semantics of sequencing and existence constraints are distinctly different from the formal meaning of workflow ground rules comprised of tasks and connections. Namely, while the constraint *has_successor* specifies a total order, the corresponding connections *has_successor* & *has_predecessor* declare a partial order between tasks. Moreover, *has_mandatory_successor* & *has_mandatory_predecessor* purposefully inhibit both task sequencing and task dependency semantics, which is a limitation of Grambow, Oberhauser and Reichert's prior work for modelling the respective bug fixing problem domain and is thus addressed in their refined approach. While the repeatability of (hierarchically) composed groups of tasks is handled explicitly by appropriate building blocks, this thesis' solution uses the *repeatable* task property to implicitly model flat sub-processes. The authors impose additional restrictions on allowed constraints between tasks, such as requiring at least one constraint between all tasks, as well as limiting each task (group) to have at most one successor and predecessor. Such restrictions are currently not enforced by our implementation. Such enhancements to our workflow meta-model were explored and considered. However, because "[t]he *Rule of Least Power* suggests choosing the least powerful language suitable for a given purpose" [Coot06b], we argue how such extended modeling capabilities are not needed and how the increased amount of required connections complicates the modeling process for users [ABSK$^+$20] needlessly for the current set of elicited requirements. We explore these enhancements for future work, for when more feedback is collected on the usage of the current system and business requirements change or become more refined.

6.2 Workflow validation & repair

In what follows, related works are examined in the context of workflow reasoning.

6.2.1 Reasoning on flow-based process models

This Subsection explores reasoning approaches that operate directly on flow-based process models.

The string edit distance, as considered in this thesis' approach Section 3.2.2, is used in Leake and Kendall-Morwick's work for (partly) assessing the similarity between process models, by looking at the required addition and deletion operations needed to "convert" one process model to the other [LeKe08]. As "the problem of comparing workflows becomes the same as the subgraph isomorphism problem, which is NP-complete" [LeKe08], the authors additionally employ "a greedy algorithm to search for an optimal mapping between the nodes of [one workflow] and the nodes of [another workflow]" [LeKe08] for workflow comparison, instead.

In the context of process mining, Sun, Du and Li propose a methodology for repairing reference process models based on real process execution traces that may differ from the original process definition [SuDL17]. Namely, because businesses need to react to constantly changing environmental factors and requirements, companies need to make ad hoc changes to their process, which makes the reference process model outdated and not a true reflection of the new status quo. Therefore, event logs are examined, which "reflect the real process directly and truly" [SuDL17]. However, since "the cost of mining a new model from a large number of event logs is very high[, a]nd the discovered model is likely to bear no similarity with the original model" [SuDL17], existing process model repair is preferred instead. To this end, the authors construct mirroring matrices, which denote pair-wise ordering relations (*footprints*) between the available process activities. This includes (direct or transitive) successor relations, (immediate or eventual) casual relations, as well as mutual exclusion or parallel execution relations. These mirroring matrices are created for both real execution traces and execution traces generated from the (potentially) outdated process model. The authors then provide an algorithm for the inclusion of missing activities into the process model at appropriate positions.

On the other hand, Rogge-Solti et al. evaluate an approach for accomplishing the reverse, that is, repairing incomplete event logs through the examination of the reference process model [RSMAW13]. Although services typically support the automatic generation of events upon completion of their assigned tasks, manual tasks performed by users may not be always logged properly. These documentation errors can happen due to a number of different reasons in e.g. a surgery process, where a patient's arrival or departure from an operation room may not properly be tracked. However, since many analysis methods in process mining depend on execution traces being complete representations of the real behavior of a process, Rogge-Solti et al.'s work deals with the repair of such incomplete event logs. To this end, the authors propose a methodology for, firstly, inserting missing tasks into the event log according to a "stochastically enriched process model" [RSMAW13], and for secondly, guessing the most likely time when these added events occurred. Missing events, that need to be added, are determined by cost-minimal fitness alignments, as introduced in Subsection 1.3.2. The alignment algorithm is hereby configured with a high cost for log moves, a low cost for model moves and a zero cost for synchronous moves. Model moves need to be weighed with a non-zero cost as to avoid infinite paths in cyclic process models. Path-dependent task probabilities from the stochastically enriched process model are used as a tie-breaker for choosing the most-likely model execution sequence among all optimal alignments. The given task sequence from the event log is then repaired with respective tasks from such model execution sequence.

Wang et al. show that "finding the minimum recovery of missing events" is `NP`-hard [WSZL13]. In order to more efficiently solve the problem of repairing incomplete event logs, the authors devise a branch and prune technique. The presented technique leads to a reduction of the search space for model execution sequence alternatives that are similar to the given incomplete event sequence. Moreover, they extend their proposed approach for finding the top k most similar execution sequence recommendations. As "we cannot always automatically make correct recovery of missing events by only using specification constraints" [WSZL13], the most similar execution sequence may not be the one that occurred in reality. Namely, if multiple events are missing from the log, the real execution sequence may be less similar to the given incomplete event sequence than another alternative sequence.

6.2.2 Reasoning via pre- & post-conditions of tasks

This Subsection explores related works which represent tasks in terms of their pre- & post-conditions, such that planners and similar reasoner implementations can find a suitable sequence of tasks that results in a goal state which exhibits the desired post-conditions.

Lu, Bernstein and Lewis propose an alternate method of verifying whether a flow-based workflow model is correct based on expected workflow pre- & post-conditions, and a method for generating valid workflows based on given pre- & post-conditions [LuBL06]. Such a workflow model consists of tasks that are connected to each other through typical flow-based process model elements, such as exclusive-choice and conditional gateways. Similar to how all tasks in a workflow need to adhere to a set of ground rules in this thesis' problem statement, each task in the aforementioned workflow model defines a set of its own pre- & post-conditions. Workflow verification thus evaluates each "path" in the workflow model step-by-step via inference rules, that check whether the current workflow state matches the currently evaluated task or construct's pre-conditions, before applying the respective post-conditions onto this workflow state[2], and then moving on to the subsequent task or construct on this "path". Starting with the given pre-conditions as the initial workflow state, this state is subsequently altered until all possible alternative post-condition outcomes of each "path" are reached. On the other hand, workflow generation starts with the given workflow post-conditions and tries to find tasks or constructs that would produce those post-conditions. The pre-conditions of suitable tasks or constructs are then in turn considered as post-condition alternatives for finding subsequent tasks or constructs. This iterative process is repeated for all "branches" in such search space tree until the given workflow pre-conditions are implied by the current workflow. The current workflow is thereby the composition of all tasks and constructs from the current node up to the root node. Workflow generation can thus be regarded as the inverse to workflow verification, because it starts at the final post-conditions and works its way backwards to the starting pre-conditions.

Chun, Atluri and Adam adapt the previously introduced notion of pre- & post-conditions of tasks in order to generate customized workflows that adhere to regulations of the business domain [ChAA02]. Namely, for the purpose of creating custom, client-specific intra-government-agency workflows for e.g. new business registrations, these authors explore a domain knowledge-based approach for said workflow generation. A task (governmental service) ontology serves as the description of knowledge about tasks and their hierarchical relationship to each other, whereby broader compound tasks are recursively broken down into their finer-grained contained tasks. A rule (governmental regulation) ontology models interrelated rules of pre- & post-conditions, whereas these so-called compositional rules are divided into selection rules, whose activation adds tasks to a workflow, as well as into coordination rules, which enforce ordering dependencies between workflow tasks when triggered. Additionally, these ontologies model the required client-specific data (user-profile preferences) that must be supplied for generating such customized workflows. The client-specific data resembles form-fields (key-value mappings) that must adhere to a data schema. It is the dynamic part that changes for each synthesis, in constrast to the static definition of these domain ontologies. Starting from the given broad goal task, the proposed algorithm constructs an individualized workflow of tasks and dependencies between those tasks, such that it considers only relevant component tasks, adheres to the aforementioned regulations and is furthermore specialized to the given client-specific data. In particular, this algorithm breaks down the goal task into its more fine-grained component tasks and considers only those tasks and related rules for workflow generation. The client-specific data is then evaluated against each such rule's pre-condition, which is a logical expression based on terms from such data. If the pre-condition is satisfied, the rule's post-condition is applied to the workflow. This includes adding the respective tasks from the post-condition to the workflow, and imposing order restrictions between tasks as specified in such post-condition. "In addition to the tasks selected based on [client data], there exist obligatory tasks required for all users" [ChAA02]. These obligatory tasks are added through rules with empty pre-conditions which thus always "fire". The produced workflow is a tuple consisting of the selected tasks and the dependencies that "glue [these] tasks together in order" [ChAA02].

[2] Atomic formulas are made up of predicates or conditional expressions, which, when combined with other atoms through conjunction, disjunction, existential & universal quantification, build up well-formed formulas. These well-formed formulas represent a set of workflow states, which are in turn "an assignment of values to variables and propositions" [LuBL06].

In their work, Kumara et al. explore an ontology-based workflow generation method [KPZS⁺15]. Similar to previous approaches, at the core a reasoner (planner) is used to find a linear sequence of actions (plan) that, starting from the given initial state, results in the desired goal state. Each available action defines the pre-condition for it to be triggered, as well as the effect it applies to the workflow state once it is triggered. The initial & goal state, as well as pre-conditions & effects are hereby described as expressions of logic atoms. However, the goal state itself is determined by a collection of SWRL rules which operate on a domain ontology that encodes domain concepts and relationships between them. These rules, which are similar to Jena rules presented in Section 3.2.3, infer the necessary tasks that need to be included in the generated workflow, based on the ontology concepts' data & object properties. These necessary tasks generated from SWRL rules are then appropriately encoded into a goal state, for which the planner finds a suitable action sequence that results in such state. The planner thus takes care of finding a proper ordering between those tasks, and adds any intermediate dependency tasks that must be included in this action sequence.

Chen and Yang use an event-calculus[3] based planner for workflow generation [ChYa05]. Event calculus uses *fluents* (properties that change over time), *events* (that may change the state of the world) and *time points*. Multiple built-in EC *predicate*s are provided to relate these building blocks, e.g. to assert a fluent to hold from a time-point on, given that an event occurs $Initiates(event, fluent, time)$ [ChYa05].

6.2.3 Workflow management in Grid computing

Grid applications can be understood in terms of abstract workflows that combine different logical *activities* offered by a Grid of distributed computing resources in order to achieve a desired goal by respecting data- & control-flow dependencies of the activities' exposed interfaces [DBGK⁺04b][SiVF07]. Such abstract activities are, upon execution, then dynamically mapped to available resources across different Grid nodes [SiVF07].

Gil, Deelman et al. describe the *Pegasus* system in their works, which is a planning system that, among other features, generates workflows based on available resources on a computer grid [GDBK⁺04][DBGK⁺04b]. Pegasus "addresses key areas for Grid computing such as allocating resources for higher quality workflows and maintaining the workflow in a dynamic environment" [GDBK⁺04]. Pegasus is developed as part of the *GriPhyN* project, which "aims to support large-scale data management in physics experiments such as high-energy physics, astronomy, and gravitational wave physics in the wide area" [DBGK04a]. The mapping of an abstract workflow to currently available resources on the Grid is explored in another of Deelman et al.'s works [DBGK⁺03].

Siddiqui, Villazon and Fahringer explore an ontology-driven approach to the generation of such abstract Grid workflows in their work [SiVF07]. Their approach leverages Apache Jena rules in order to generate valid workflows by combining them sequentially or parallelly and linking outputs of one activity to matching inputs of a subsequent activity. These rules run on an ontology which describes activities, their inputs & outputs and other concepts in such "machine-processable specification with formally defined meaning" [HiKR09].

[3] The Event Calculus (EC) is a highly developed logic-based formalism based on many-sorted first-order predicate calculus augmented with circumscription [ChYa05].

6.2.4 Workflow management for adaptive workflows

Van Der Aalst et al. describe in their work challenges encountered in adaptive workflows that need to accommodate for changes in the process model, even during execution [ABVV^{+}00]. Any such workflow transformation has to preserve both syntactic and semantic correctness in the resulting workflow model. While semantic-preserving changes are context-specific, syntax-preserving changes can be formally defined for all workflows using Petri net analysis. For example, the transformed workflows need to be sound[4], which is a “general purpose sanity check for workflows” [HeSW13].

The topic of dynamic workflow change is also addressed in Chun and Atluri’s work [ChAt03], where the authors use appropriate migration rules identified by an ontology in order to adapt workflows at run-time.

6.2.5 Other approaches

Shepelev and Director present an approach for the automatic generation of design flows in their work [ShDi96]. Central to the synthesis algorithm is a task schema graph, which “captures the dependencies between all design entities (both tools and data) available to the designer and specifies construction rules to be used to create design flows that when executed realize design tasks” [ShDi96]. Tasks require a set of entities as their inputs and produce a set of entities as their output. The proposed algorithm generates a *flow* that produces the desired target entities. A flow is thus an alternating sequence of entities and tasks, whereby the output entities of one task are fed as input to the next task.

Doshi et al. explore a decision-theoretic planning approach for the generation of workflows [DGAV05]. Their approach “models both, the stochastic nature of services, and the dynamic nature of the environment producing workflows that are robust and adaptive” [DGAV05]. A decision policy is found via a Markov Decision Process, which chooses the optimal action, with the lowest expected cost, based on the current workflow state. Such policy is used to generate an output sequence of actions that results in the desired workflow state. Bayesian Model Learning is in turn used to iteratively update estimates of certainty and thus the optimal decision policy, after receiving new information from such dynamic environment.

[4]See 2.1.4 for a background on this process model property.

7 Conclusion

We answer all research questions by implementing a customized approach, which combines different technologies that suit the needs of service users by being easy-to-use, as well as being easy-to-implement-and-verify for service providers.

Despite the base ingredients being well known and researched, we believe the proposed approach is a novel combination of procedures for event log repair in small process state spaces, not only for incomplete but also for incorrect event traces.

During the process of finding such answers, we make the following contributions, in order of significance:

- A customized approach that addresses the particular requirements of workflow management in *typical* production environments, alongside a formal specification thereof.
- A working and tested system implementing the proposed approach[1], with auxiliary materials, such as setup and run instructions, as well as a comprehensive user guide.
- Usage examples of various technologies for addressing not only the problem statement but also for further application areas that may be of interest in the pursuit of Industry 4.0.
- An overview of theoretical foundations of the underlying concepts behind this work so that interested readers may gain a deeper understanding of such concepts.
- Bug reports for two defects in the reference implementation of the W3C Shapes Constraint Language, which were fixed by the authors in the meantime[2].

[1]The source code repository containing the system implementation is available at `https://github.com/mucaho/lottraveler`.

[2]See `https://github.com/TopQuadrant/shacl/issues/111` and `https://github.com/TopQuadrant/shacl/issues/112` for the respective issue details.

7.1 Future work

This Section is split into two parts: In the former part, we examine future work related to the problem statement, opportunities for refining the presented approach and possible additional evaluations. The latter part introduces another potential application area for improving the day-to-day operations in *typical* production environments, which was additionally elicited during meetings with experts from Infineon. Various reasoning approaches to solve such new problem statement can be in detail examined in subsequent work.

7.1.1 Future work: Workflow management in FA

This Subsection presents opportunities for improving the current approach & implementation and explores alternative methodologies for addressing the problem of workflow management in *typical* production environments.

Improvements to current approach and implementation

In order for our proposed approach to qualify as a *significant* discovery in the area of event log repair, we need to perform a proper scientific evaluation of the proposed method, as mentioned in the threats to validity Section 5.2.3.

As mentioned in approach Section 3.2.2, the current notion of similarity between workflows is based on a priority list of distance measures. These distance measures are inherently asymmetric and represent individual operations that are typically evaluated in tandem as part of a (compound) edit distance instead. An alternative, more elegant solution is to directly use the string edit distance and apply proper weights to different operations types instead, which is the topic of Hyyrö, Narisawa and Inenaga's work [HyNI10]. For example, the deletion operation must be significantly more penalized than any other operation, since the removal of tasks from an invalid workflow is "frowned upon" in contrast to the inclusion of additional tasks that would make that workflow valid.

Additional modeling concepts can be introduced in future to properly handle ground rules that require more flexibility, such that a wider spectrum of the *as-is* behavior can be modeled accurately. A motivating example is given in Section 5.2.3. For example, Grambow, Oberhauser and Reichert [GrOR11] present workflow modeling capabilities that are more expressive than those available in this thesis' approach, and thus provide the workflow designer more fine-grained control in representing a greater variety of valid workflow sequences. In particular, this includes sequencing & existence constraints and building blocks for declaratively describing such workflows with flexible business requirements. As argued in Subsection 6.1.2, such modeling capabilities are considered for future work. Namely, this includes the separation of sequencing and dependency existence connections, parallelism and mutual exclusion as appropriate counterparts to existing connections, as well as grouping of tasks into hierarchically composed blocks for which those relations can be specified as well. Additional correctness conditions need to be enforced on models, such as preventing the declaration of conflicting constraints between tasks.

Future work can also look into refining the current approach further, by sitting down with stakeholders and gathering new insights and feedback, after the system has been used for some time. Such meetings can lead to the elicitation of new or changed requirements, both functional and non-functional. The latter type of requirements, like performance expectations, are currently not specified for the system.

There are additional non-critical improvements that can be made to the current implementation:

- Inform the user whenever the induced directed acyclic graph is not minimal with regard to its transitive reduction. These redundant connections may thus be removed.
- Inform the user whenever a *has_successor* connection has been defined alongside a *has_mandatory_successor* between the same two tasks. As the latter connection is a specialization of the former, the former is redundant and may thus be removed.
- There are additional limitations regarding the import / export logic that may corrupt class hierarchies that are defined on tasks. These can be addressed in the future.
- The implementation of the distance measure for task ordering can be further improved in the reasoner module.
- The validation error output encoding in the specification can be refined to fully match the more natural one from the implementation.

Workflow modeling via other languages

As the thesis' approach to workflow (meta-)modeling in Section 3.1 considers only a select collection of modeling techniques, additional process modeling languages can be considered in future to represent ground rules for valid workflows instead.

In the family of imperative process modeling languages, "where all execution alternatives must be explicitly specified" [Pesi08][ABSK$^+$20], future work can examine the use of Yet Another Workflow Language (YAWL) instead. It supports the typical gateways and iteration constructs as flow-based process-model (like e.g. in BPMN), but also offers design elements that allow the addition and removal of specific tasks dynamically from a process instance at run-time [THAAR09].

On the other hand, declarative process modeling languages provide an "implicit specification of [all] execution alternatives" [Pesi08] which satisfy the given constraints [ABSK$^+$20]. Such alternatives to the previously showcased declarative process modeling language CMMN include Declare, Dynamic Conditional Response (DCR), and Extended Tabular Tree version 2 (XTT2) [ABSK$^+$20].

However, "considering the rich and complex semantics of declarative languages [...] and all the possible ways in which constraints can interact, the understandability of declarative process models gets quickly hampered when dealing with complex processes" [ABSK$^+$20]. To this end, various hybrid process modeling languages have been proposed that combine the advantages of both imperative and declarative modeling paradigms [ABSK$^+$20]. These languages can be categorized into ones that mix semantic concepts from both paradigms, like for example nesting YAWL or Declare models as sub-models of each other [Pesi08], and ones integrating business rules into imperative process models [ABSK$^+$20].

Reasoning on flow-based process models

An alternative combined approach to the one presented in approach Section 3.3 is to convert flow-based process models into the `ABOX` of an appropriate ontology. In this scenario, a given process model is converted into individuals of appropriate TBOX OWL classes that capture the formal semantics of all notation elements offered by such process modeling language, as motivated in Koschmider and Oberweis's work [KoOb05]. Based on this combined ABOX and TBOX of such ontology, custom rules can be used to infer any potential constraint violations, as seen in approach Section 3.2.3. Future work can explore this alternative combined approach for workflow validation.

Workflow validation & repair via planning
Lu, Bernstein and Lewis state how "the problem of automatic workflow generation is closely related to the problem of propositional STRIPS planning in artificial intelligence, which is to find a sequence of actions [with pre- & post-conditions] to achieve a goal [state] from a given initial state" [LuBL06]. Although developed planners are not suited to generate workflows with complex structures and constructs [LuBL06], this work's problem statement deals with simple workflows for which the application of such planners can be examined in future work. As a correct workflow trivially requires each task's pre- & post-conditions to be met in the respective workflow state, this thesis problem statement's ground rules can be modeled as such instead. Furthermore, since this work's workflows correspond to a sequence of tasks, sequentially combined tasks must only be allowed in such workflow models. Namely, a task's pre-conditions can include any completion markers of mandatory dependency tasks that need to be done before this task. These mandatory dependency tasks then mark the workflow state with their completion, as part of their post-conditions. A similar effect can be achieved for the ordering requirement between non-destructive and destructive tasks. Any destructive task leaves a marker denoting the completion of at least a single destructive task, as part of its post-condition. Pre-conditions of non-destructive tasks then forbid any such prior completion. However, in order to capture the exact semantics of the transitively induced partial order between tasks, the entire (reflexive) transitive closure needs to be encoded into appropriate pre- & post-conditions. This includes stating all task markers that must not be done before a respective task, as part of its pre-conditions. Like before, a task's post-condition would then apply the respective task marker on the workflow state. Therefore, by defining pre-conditions that require specific task completion markers to be present, and by declaring post-conditions that apply such markers to the workflow state, a planner can then find a sequence of tasks that leads to the goal state comprised of task completion markers requested by clients. The planner thereby respects the (transitive) partial order and adds any mandatory dependency tasks to the resulting workflow.

The proposed workflow verification method in Lu, Bernstein and Lewis' work [LuBL06] can additionally be applied to any such workflow, without any expectations on the workflow's pre- & post-conditions themselves in order to check whether each individual task's pre- & post-conditions have been fulfilled. Similarly, their workflow generation algorithm can be used to find similar, valid workflows for a given invalid workflow. Namely, the goal is to find the smallest superset of the invalid tasks' completion markers for which correct workflows can be generated. This superset of task completion markers serves as the workflow's post-condition, alongside an empty set of workflow pre-conditions, as part of such synthesis method. The "best" workflow repair recommendation is then chosen according to some similarity measure between the given and all generated workflows, as discussed in this thesis' approach.

The approach proposed by Chun, Atluri and Adam [ChAA02] can be considered as well to solve part of this thesis' problem statement, in particular for generating valid workflow alternatives to a given (potentially) invalid workflow. Tasks can analogously be represented in a task ontology without an imposed hierarchy. The rule ontology then contains rules that fire when a task is requested by the client as part of such client-specific data. Selection rules add the respective task and all its mandatory dependency tasks to the workflow. Coordination rules enforce the (transitive) partial order between the requested task and all other affected tasks in form of appropriate control-flow dependencies. Finally, the client request consists of all desired tasks, for which the algorithm finds a superset of these tasks[3] and a set of inter-task dependencies. The produced workflow is a tuple of tasks and dependencies that needs to be "linearized" first in order to resolve a sequence of tasks that can be recommended as an alternative to the given invalid sequence of tasks. Each available execution path of the produced workflow corresponds to one such task sequence alternative. The best alternative is again chosen according to some similarity measure, such as one defined in this thesis' approach Section 3.2.2.

Future work can examine the inclusion of SWRL rules (or analogously Jena rules) in workflow generation to further individualize the desired goal states based on additional client needs, as proposed in Kumara et al.'s work [KPZS+15].

7.1.2 Future work: Job management in FA

Through elicitation of requirements with the help of experts from Infineon's failure analysis lab we have identified another potential application area for improving the day-to-day operations in *typical* production environments. As labs *typically* function on a simple high-prio jobs come before low-prio jobs basis, there is potential for improvement. As introduced in Chapter 1, more fine-grained evaluation of job priority via respecting deadlines, having tools run without idle time, consideration of worker preferences, and other factors come into play when trying to come up with a more suitable assignment of available resources to the tasks at hand. We refer to this problem as job optimization and aim to find a suitable solution in the future.

Jobs can be understood as specific instances of workflow models, which must necessarily be valid by, e.g., checking the validity using the system implemented as part of this thesis. However, a job box contains multiple samples, where each sample must potentially be analyzed using a different workflow in order to detect all possible flaws of the component. Moreover, in order to detect such faults with statistical significance, multiple such samples need to be analyzed using the same workflow.

Although significant strides have already been made towards a preliminary definition of the problem as well as a prototype implementation of the job optimizer, the specification of the exact problem statement, approach and implementation is left for future work.

[3]The produced workflow contains a superset of the given tasks, because it is composed of the requested tasks themselves and all their mandatory dependency tasks.

A Specification

This Chapter contains the complete & executable[1] system specification, written in the state-based specification language `TLA+`[2], which is based on mathematical set theory [MaKE18]. The use of an unambiguous written description of what a system is supposed to do is motivated in approach Section 3.1.5. Here, we complete the definition of the problem domain from Section 3.1.5, the workflow validation functionality from Section 3.2.1 and the workflow repair functionality from Section 3.2.2.

The rest of this Chapter is broken up into multiple sections, each containing the appropriate specification listing of the respective system `MODULE`[3]. Splitting the system specification into multiple modules facilitates sharing of common specification statements among other modules that `EXTEND` such common module. Furthermore, a container module can create multiple module `INSTANCE`s by providing the necessary module construction parameters known as `CONSTANT`s[4]. Afterwards, a container module can use the exposed functions (*operators*) of such module instances[5]. These specifications describe:

A.1: the expected structure of tasks, of their connections between them, and of workflows, as well as utility operators that depend on those constants

A.2: an example instance of the previous module for the example FA process given in Section 3.1.1

A.3: the workflow validation functionality, by precisely defining the constraints that each workflow is checked against, and the structure of produced validation errors

A.4: multiple tests, which assert whether the actual workflow validation error output matches the expected workflow validation error output for the given workflow input

A.5: the workflow repair functionality, by precisely defining the order of distance measures that each recommendation candidate workflow is evaluated against, as well as the definitions of the distance measures themselves

A.6: multiple tests, which assert whether the actual workflow recommendation output matches the expected workflow recommendation output for the given workflow input

A.7: an executable entry-point to the complete test suite

[1]Of note are the system specification tests, which can be run to verify whether the actual test output matches the expectations on the behavior of the system.
[2]See `https://lamport.azurewebsites.net/tla/tla.html` for more details.
[3]See `http://lamport.azurewebsites.net/tla/newmodule.html` for more details.
[4]See `https://learntla.com/models/constants/` for more details.
[5]See `https://learntla.com/tla/operators/` for more details.

A.8: a collection of utility operators, which are leveraged in previous modules

These specifications are run using the `TLA+` Toolbox[6] and require the additional installation of `TLA+` community modules[7]. Execution is supported via the GUI or the command-line. For the former, the community modules need to be added as an external `TLA+` library in the appropriate Toolbox preferences dialog. For the latter, these modules need to be linked likewise in the Java classpath. For example, the complete test suite can be run with the command `java -cp /PATH/TO/CommunityModules.jar:/PATH/TO/toolbox/tla2tools.jar tlc2.TLC MetaModelReasonerTest.tla`.

Although we hope the mathematical notation feels intuitive to the reader, the exact syntax and semantics of the used specification language are beyond the scope of this work, thus, if needed, the reader is advised to take a look at e.g. Lamport's [Lamp02] or Wayne's book [Wayn18] on this topic.

A.1 Workflow definition

MODULE *WorkflowDefinition*

LOCAL INSTANCE *Utilities*

GIVEN the set of tasks:

CONSTANT *Tasks*

```
e.g. Tasks ==
{   [ name |-> "EVI", repeatable |-> FALSE, group |-> "non-destructive" ]
,   [ name |-> "IVI", repeatable |-> FALSE, group |-> "non-destructive" ]
}
```

such that it adheres to the expected structure

ASSUME $IsFiniteSet(Tasks)$

ASSUME $\forall task \in Tasks : (\forall id \in DOM(task) : id \in \{\text{“name”}, \text{“repeatable”}, \text{“group”}\})$

ASSUME $\forall task \in Tasks : task.name \in \text{STRING}$

ASSUME $\forall task, otherTask \in Tasks : task.name = otherTask.name \implies task = otherTask$

ASSUME $\forall task \in Tasks : task.repeatable \in \text{BOOLEAN}$

ASSUME $\forall task \in Tasks : task.group \in \{\text{“destructive”}, \text{“non-destructive”}, \text{“both”}\}$

with the following helper definitions

$TaskNames \triangleq \{task.name : task \in Tasks\}$

$unknownTask \triangleq [name \mapsto \text{“UNKNOWN”}, repeatable \mapsto \text{FALSE}, group \mapsto \text{“both”}]$

ASSUME $unknownTask \notin Tasks$

[6]See `https://lamport.azurewebsites.net/tla/toolbox.html` for more details. At implementation time version 1.7.1 was used.

[7]See `https://github.com/tlaplus/CommunityModules` for more details. At implementation time version 202009162135 was used.

$\text{ASSUME } \forall\, task \in Tasks : task.name \neq unknownTask.name$

$Task(t) \triangleq ChooseOrDefault(Tasks, \text{LAMBDA } task : task.name = t, unknownTask)$

GIVEN the set of connections between tasks:

$\text{CONSTANT } Connections$

```
e.g. Connections ==
{   [ name |-> "has_successor", srcName |-> "EVI", dstName |-> "IVI" ]
}
```

such that it adheres to the expected structure

$\text{ASSUME } IsFiniteSet(Connections)$

$\text{ASSUME } \forall\, conn \in Connections :$

$\forall\, id \in DOM(conn) : id \in \{\text{“name”}, \text{“srcName”}, \text{“dstName”}\}$

$\text{ASSUME } \forall\, conn \in Connections : conn.name \in$

$\{\ \text{“has\_successor”}$

$,\ \text{“has\_predecessor”}$

$,\ \text{“has\_mandatory\_predecessor”}$

$,\ \text{“has\_mandatory\_successor”}$

$\}$

$\text{ASSUME } \forall\, conn \in Connections : \exists\, task \in Tasks : task.name = conn.srcName$

$\text{ASSUME } \forall\, conn \in Connections : \exists\, task \in Tasks : task.name = conn.dstName$

$\text{ASSUME } \forall\, conn \in Connections : conn.srcName \neq conn.dstName$

such that the induced binary relation satisfies some properties

$RequiresRel[x \in TaskNames, y \in TaskNames] \triangleq$

$\quad \exists\, conn \in Connections :$

$\quad\quad \land\ conn.name = \text{“has\_mandatory\_successor”} \lor conn.name = \text{“has\_mandatory\_predecessor”}$

$\quad\quad \land\ conn.srcName = x$

$\quad\quad \land\ conn.dstName = y$

$ConRel[x \in TaskNames, y \in TaskNames] \triangleq$

$\quad \lor\ \exists\, conn \in Connections :$

$\quad\quad \land\ conn.name = \text{“has\_successor”} \lor conn.name = \text{“has\_mandatory\_successor”}$

$\quad\quad \land\ conn.srcName = x$

$\quad\quad \land\ conn.dstName = y$

$\quad \lor\ \exists\, conn \in Connections :$

$\wedge\ conn.name = $ "has_predecessor" $\vee\ conn.name = $ "has_mandatory_predecessor"

$\wedge\ conn.srcName = y$

$\wedge\ conn.dstName = x$

$TransConRel \triangleq ReflexiveTransitiveClosure(ConRel,\ TaskNames)$

ASSUME $IsIrreflexive(ConRel,\ TaskNames)$

ASSUME $Is2Acyclic(TransConRel,\ TaskNames)$

with the following helper definitions

$unknownConnection \triangleq [name \mapsto$ "UNKNOWN", $srcName \mapsto$ "UNKNOWN", $dstName \mapsto$ "UNKNOWN"$]$

ASSUME $unknownConnection \notin Connections$

ASSUME $\forall\ conn \in Connections : conn.name \neq unknownConnection.name$

```
$ErrorConn(s,\ d,\ CONSTRAINT(\_,\ \_)) \triangleq$
  IF $\neg CONSTRAINT(s,\ d)$
   THEN $\{[name \mapsto$ "has_successor", $srcName \mapsto s,\ dstName \mapsto d]\}$
   ELSE $\{\}$
```

```
$ErrorConns(s,\ d,\ CONSTRAINT(\_,\ \_)) \triangleq \{conn \in Connections : \neg($
  $\vee$
      $\wedge\ conn.name = $ "has_successor"
       $\vee\ conn.name = $ "has_mandatory_successor"
      $\wedge\ conn.srcName = s \wedge conn.dstName = d$
  $\vee$
      $\wedge\ conn.name = $ "has_predecessor"
       $\vee\ conn.name = $ "has_mandatory_predecessor"
      $\wedge\ conn.srcName = d \wedge conn.dstName = s$
   $\implies CONSTRAINT(s,\ d)$
$)\}$
```

WHEN checking for errors in an input workflow

CONSTANT $Workflow$

```
e.g. Workflow == << "EVI", "IVI" >>
```

such that it adheres to the expected structure

ASSUME $DOM(Workflow) = 1\ ..\ Len(Workflow)$ a proper tuple

$\textsc{assume}\ \forall\, t \in \mathit{RAN}(\mathit{Workflow}) : t \in \textsc{string}$

A.2 Workflow example

$\textsc{module}\ \mathit{WorkflowExample}$

$\mathit{WorkflowExample} \triangleq \textsc{instance}\ \mathit{WorkflowDefinition}\ \textsc{with}$

$\quad \mathit{Tasks} \leftarrow$

$\quad \{\ [\mathit{name} \mapsto \text{“EVI”}, \mathit{repeatable} \mapsto \textsc{false}, \mathit{group} \mapsto \text{“non-destructive”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“SEM”}, \mathit{repeatable} \mapsto \textsc{false}, \mathit{group} \mapsto \text{“non-destructive”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“XRAY\_MIC”}, \mathit{repeatable} \mapsto \textsc{false}, \mathit{group} \mapsto \text{“non-destructive”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“EPT”}, \mathit{repeatable} \mapsto \textsc{false}, \mathit{group} \mapsto \text{“non-destructive”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“DECAP”}, \mathit{repeatable} \mapsto \textsc{false}, \mathit{group} \mapsto \text{“destructive”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“IVI”}, \mathit{repeatable} \mapsto \textsc{false}, \mathit{group} \mapsto \text{“destructive”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“STRIP”}, \mathit{repeatable} \mapsto \textsc{true}, \mathit{group} \mapsto \text{“destructive”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“XRAY\_SPEC”}, \mathit{repeatable} \mapsto \textsc{true}, \mathit{group} \mapsto \text{“destructive”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“SAM”}, \mathit{repeatable} \mapsto \textsc{true}, \mathit{group} \mapsto \text{“destructive”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“TEM”}, \mathit{repeatable} \mapsto \textsc{true}, \mathit{group} \mapsto \text{“destructive”}]$

$\quad \},$

$\quad \mathit{Connections} \leftarrow$

$\quad \{\ [\mathit{name} \mapsto \text{“has\_predecessor”}, \mathit{srcName} \mapsto \text{“SEM”}, \mathit{dstName} \mapsto \text{“EVI”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_predecessor”}, \mathit{srcName} \mapsto \text{“XRAY\_MIC”}, \mathit{dstName} \mapsto \text{“EVI”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_predecessor”}, \mathit{srcName} \mapsto \text{“EPT”}, \mathit{dstName} \mapsto \text{“SEM”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_predecessor”}, \mathit{srcName} \mapsto \text{“EPT”}, \mathit{dstName} \mapsto \text{“XRAY\_MIC”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_predecessor”}, \mathit{srcName} \mapsto \text{“DECAP”}, \mathit{dstName} \mapsto \text{“EPT”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_mandatory\_successor”}, \mathit{srcName} \mapsto \text{“DECAP”}, \mathit{dstName} \mapsto \text{“IVI”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”}, \mathit{srcName} \mapsto \text{“IVI”}, \mathit{dstName} \mapsto \text{“DECAP”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”}, \mathit{srcName} \mapsto \text{“STRIP”}, \mathit{dstName} \mapsto \text{“IVI”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”}, \mathit{srcName} \mapsto \text{“XRAY\_SPEC”}, \mathit{dstName} \mapsto \text{“STRIP”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”}, \mathit{srcName} \mapsto \text{“SEM”}, \mathit{dstName} \mapsto \text{“STRIP”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”}, \mathit{srcName} \mapsto \text{“TEM”}, \mathit{dstName} \mapsto \text{“STRIP”}]$

$\quad ,\ [\mathit{name} \mapsto \text{“has\_predecessor”}, \mathit{srcName} \mapsto \text{“SEM”}, \mathit{dstName} \mapsto \text{“XRAY\_SPEC”}]$

$, \quad [name \mapsto$ “has_predecessor”, $srcName \mapsto$ “TEM”, $dstName \mapsto$ “XRAY_SPEC”$]$

$\},$

$Workflow \leftarrow$

$\langle$“EVI”, “EPT”, “DECAP”, “IVI”, “STRIP”, “SAM”, “STRIP”, “XRAY_SPEC”, “TEM”$\rangle$

A.3 Workflow validation

MODULE *WorkflowValidation*

EXTENDS *WorkflowDefinition*

LOCAL INSTANCE *Utilities*

THEN an error is returned for unknown tasks

```
$ErrorUnknownTasks \triangleq$
    LET
        $KnownTaskConstraint(t) \triangleq$
            $\exists\, task \in Tasks : task.name = t$
    IN
        $\{t \in RAN(Workflow) : \neg KnownTaskConstraint(t)\}$
```

such that it adheres to the expected structure

ASSUME $IsFiniteSet(ErrorUnknownTasks)$

ASSUME $\forall\, t \in ErrorUnknownTasks : t \in$ STRING

THEN an error is returned for non-repeatable tasks being repeated

```
$ErrorNonRepeatableTasks \triangleq$
    LET
        $RepeatabilityConstraint(t) \triangleq$
            $\land\ Contains(Workflow, t)$
            $\land\ \neg Task(t).repeatable$
             $\implies Count(Workflow, t) \leq 1$
    IN
        $\{t \in TaskNames : \neg RepeatabilityConstraint(t)\}$
```

such that it adheres to the expected structure

ASSUME $IsFiniteSet(ErrorNonRepeatableTasks)$

ASSUME $\forall\, t \in ErrorNonRepeatableTasks : t \in$ STRING

THEN an error is returned for destructive tasks coming before non-destructive ones

```
$ErrorDestructiveBeforeNonDestructive \triangleq$
    LET
        $DestructiveOrderConstraint(d) \triangleq$
            $\forall n \in TaskNames :$
              $\land d \neq n$
              $\land Contains(Workflow, d)$
              $\land Contains(Workflow, n)$
              $\land Task(d).group = \text{“destructive”}$
              $\land Task(n).group = \text{“non-destructive”}$
              $\implies FirstIndex(Workflow, d) > LastIndex(Workflow, n)$
    IN
        $\{t \in TaskNames : \neg DestructiveOrderConstraint(t)\}$
```

such that it adheres to the expected structure

ASSUME $IsFiniteSet(ErrorDestructiveBeforeNonDestructive)$

ASSUME $\forall t \in ErrorDestructiveBeforeNonDestructive : t \in \text{STRING}$

THEN an error is returned for partial-order violations

```
$ErrorPartialOrderViolations \triangleq$
    LET
        $PartialOrderConstraint(s, d) \triangleq$
            $\land s \neq d \land TransConRel[s, d]$
            $\land Contains(Workflow, s) \land Contains(Workflow, d)$
            $\land \neg Task(s).repeatable \lor \neg Task(d).repeatable$
            $\implies LastIndex(Workflow, s) < FirstIndex(Workflow, d)$
    IN
        UNION $\{ErrorConn(s, d, PartialOrderConstraint) : s, d \in TaskNames\}$
```

such that it adheres to the expected structure

ASSUME $IsFiniteSet(ErrorPartialOrderViolations)$

ASSUME $\forall conn \in ErrorPartialOrderViolations :$

$\forall id \in DOM(conn) : id \in \{\text{“name”}, \text{“srcName”}, \text{“dstName”}\}$

ASSUME $\forall conn \in ErrorPartialOrderViolations : conn.name \in$

$\{$ “has_successor”

, “has_predecessor”

```
    ,  "has_mandatory_predecessor"
    ,  "has_mandatory_successor"
    }
ASSUME $\forall\, conn \in ErrorPartialOrderViolations : conn.srcName \in$ STRING
ASSUME $\forall\, conn \in ErrorPartialOrderViolations : conn.dstName \in$ STRING
```

THEN an error is returned for missing mandatory dependency tasks

```
$ErrorMissingMandatoryDependencies \triangleq$
    LET
        $MandatoryDependencyConstraint(s, d) \triangleq$
        $\land$
            $\land\ s \neq d \land TransConRel[s, d]$
            $\land\ RequiresRel[s, d] \land Contains(Workflow, s)$
             $\implies \land\ Contains(Workflow, d)$
             $\land\ LastIndex(Workflow, s) < LastIndex(Workflow, d)$
        $\land$
            $\land\ s \neq d \land TransConRel[s, d]$
            $\land\ RequiresRel[d, s] \land Contains(Workflow, d)$
             $\implies \land\ Contains(Workflow, s)$
             $\land\ FirstIndex(Workflow, s) < FirstIndex(Workflow, d)$
    IN
        UNION $\{ErrorConns(s, d, MandatoryDependencyConstraint) : s, d \in TaskNames\}$
```

such that it adheres to the expected structure

```
ASSUME $IsFiniteSet(ErrorMissingMandatoryDependencies)$
ASSUME $\forall\, conn \in ErrorMissingMandatoryDependencies :$
 $\forall\, id \in DOM(conn) : id \in \{$"name", "srcName", "dstName"$\}$
ASSUME $\forall\, conn \in ErrorMissingMandatoryDependencies : conn.name \in$
    {  "has_successor"
    ,  "has_predecessor"
    ,  "has_mandatory_predecessor"
    ,  "has_mandatory_successor"
    }
ASSUME $\forall\, conn \in ErrorMissingMandatoryDependencies : conn.srcName \in$ STRING
ASSUME $\forall\, conn \in ErrorMissingMandatoryDependencies : conn.dstName \in$ STRING
```

THEN an error is returned for missing mandatory dependency repetitions

```
$ErrorMissingMandatoryDependencyRepetitions \triangleq$
  LET
    $MandatoryRepetitionConstraint(s, d) \triangleq$
    $\land$
        $\land s \neq d \land TransConRel[s, d] \land RequiresRel[s, d]$
        $\land Contains(Workflow, s) \land Contains(Workflow, d)$
        $\implies \forall i, j \in Indexes(Workflow, s) :$
        $i < j \implies \exists k \in Indexes(Workflow, d) : i < k \land k < j$
    $\land$
        $\land s \neq d \land TransConRel[s, d] \land RequiresRel[d, s]$
        $\land Contains(Workflow, s) \land Contains(Workflow, d)$
        $\implies \forall i, j \in Indexes(Workflow, d) :$
        $i < j \implies \exists k \in Indexes(Workflow, s) : i < k \land k < j$
  IN
    UNION $\{ErrorConns(s, d, MandatoryRepetitionConstraint) : s, d \in TaskNames\}$
```

such that it adheres to the expected structure

```
ASSUME $IsFiniteSet(ErrorMissingMandatoryDependencyRepetitions)$
ASSUME $\forall conn \in ErrorMissingMandatoryDependencyRepetitions :$
 $\forall id \in DOM(conn) : id \in \{$“name”, “srcName”, “dstName”$\}$
ASSUME $\forall conn \in ErrorMissingMandatoryDependencyRepetitions : conn.name \in$
  {  “has_successor”
  ,  “has_predecessor”
  ,  “has_mandatory_predecessor”
  ,  “has_mandatory_successor”
  }
ASSUME $\forall conn \in ErrorMissingMandatoryDependencyRepetitions : conn.srcName \in$ STRING
ASSUME $\forall conn \in ErrorMissingMandatoryDependencyRepetitions : conn.dstName \in$ STRING
```

FINALLY a structure of all errors is returned

```
$Errors \triangleq [$
  $ErrorUnknownTasks \mapsto ErrorUnknownTasks,$
  $ErrorNonRepeatableTasks \mapsto ErrorNonRepeatableTasks,$
  $ErrorDestructiveBeforeNonDestructive \mapsto ErrorDestructiveBeforeNonDestructive,$
```

$\quad \mathit{ErrorPartialOrderViolations} \mapsto \mathit{ErrorPartialOrderViolations},$

$\quad \mathit{ErrorMissingMandatoryDependencies} \mapsto \mathit{ErrorMissingMandatoryDependencies},$

$\quad \mathit{ErrorMissingMandatoryDependencyRepetitions} \mapsto \mathit{ErrorMissingMandatoryDependencyRepetitions}$

]

```
e.g. Errors ==
[ ErrorUnknownTasks |-> {"IVI"}
, ErrorNonRepeatableTasks |-> {"EVI"}
, ErrorDestructiveBeforeNonDestructive |-> {"IVI"}
, ErrorPartialOrderViolations |->
    { [ name |-> "has_successor", srcName |-> "EVI", dstName |-> "IVI" ] }
, ErrorMissingMandatoryDependencies |->
{ [ name |-> "has_mandatory_predecessor", srcName |-> "IVI", dstName |->
"EVI" ] }
, ErrorMissingMandatoryDependencyRepetitions |->
{ [ name |-> "has_mandatory_predecessor", srcName |-> "IVI", dstName |->
"EVI" ] }
]
```

WHILE the structure containing no errors matches

$\mathit{NoErrors} \triangleq [$

$\quad \mathit{ErrorUnknownTasks} \mapsto \{\},$

$\quad \mathit{ErrorNonRepeatableTasks} \mapsto \{\},$

$\quad \mathit{ErrorDestructiveBeforeNonDestructive} \mapsto \{\},$

$\quad \mathit{ErrorPartialOrderViolations} \mapsto \{\},$

$\quad \mathit{ErrorMissingMandatoryDependencies} \mapsto \{\},$

$\quad \mathit{ErrorMissingMandatoryDependencyRepetitions} \mapsto \{\}$

]

A.4 Workflow validation tests

MODULE *WorkflowValidationTest*

LOCAL INSTANCE *TLC*

$\mathit{ValidationTests} \triangleq$

should report no errors for trivial valid workflow

$\langle$LET *Validation* $\triangleq$ INSTANCE *WorkflowValidation* WITH

Connections $\leftarrow$

{

```
 },
 Tasks ←
 {  [name ↦ “EVI”, repeatable ↦ FALSE, group ↦ “both”]
 },
 Workflow ←
 ⟨“EVI”⟩

 ACTUAL ≜ Validation!Errors

 EXPECTED ≜ Validation!NoErrors
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, “should report no errors for trivial valid workflow”)
    )
```

should not report errors for complex, valid workflow

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
  Connections ←
  {  [name ↦ “has_successor”, srcName ↦ “EVI”, dstName ↦ “IVI”]
  ,  [name ↦ “has_mandatory_successor”, srcName ↦ “EVI”, dstName ↦ “IVI”]
  ,  [name ↦ “has_successor”, srcName ↦ “IVI”, dstName ↦ “DES”]
  },
  Tasks ←
  {  [name ↦ “EVI”, repeatable ↦ FALSE, group ↦ “non-destructive”]
  ,  [name ↦ “IVI”, repeatable ↦ TRUE, group ↦ “destructive”]
  ,  [name ↦ “DES”, repeatable ↦ TRUE, group ↦ “destructive”]
  },
  Workflow ←
  ⟨“EVI”, “IVI”, “DES”, “IVI”, “DES”⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ Validation!NoErrors
IN
    Assert(
        $Print(ACTUAL, ACTUAL = EXPECTED)$,
        $Print(EXPECTED$, “should not report errors for complex, valid workflow”)
    )
```

should report errors for unknown task

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  {
  },
  $Tasks \leftarrow$
  {  [$name \mapsto$ “EVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
  },
  $Workflow \leftarrow$
  ⟨“IVI”⟩

  $ACTUAL \triangleq Validation!Errors$

  $EXPECTED \triangleq [Validation!NoErrors$ EXCEPT
      $!.ErrorUnknownTasks = \{$“IVI”$\}$
  ]
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED)$,
        $Print(EXPECTED$, “should report errors for unknown task”)
    )
```

should report errors for repeating unrepeatable task

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  {
  },
  $Tasks \leftarrow$
  {  [$name \mapsto$ “EVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
  },
  $Workflow \leftarrow$
  ⟨"EVI", "EVI"⟩
  $ACTUAL \triangleq Validation!Errors$
  $EXPECTED \triangleq [Validation!NoErrors$ EXCEPT
      $!.ErrorNonRepeatableTasks = \{$"EVI"$\}$
  ]
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED),$
        $Print(EXPECTED,$ "should report errors for repeating unrepeatable task")
    )
```

should report no errors for mixing tasks that can be both destructive and non-destructive

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  {
  },
  $Tasks \leftarrow$
  {  [$name \mapsto$ "EVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"]
  ,  [$name \mapsto$ "IVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"]
  },
  $Workflow \leftarrow$
  ⟨"EVI", "IVI"⟩
  $ACTUAL \triangleq Validation!Errors$
  $EXPECTED \triangleq Validation!NoErrors$
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED),$
        $Print(EXPECTED,$ "should report no errors for mixing tasks that can be both destructive and non-destruc
    )
```

should report no errors for doing non-destructive task before destructive one

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
```

```
  $Connections \leftarrow$
  $\{$
  $\}$,
  $Tasks \leftarrow$
  $\{\ [name \mapsto \text{“EVI”},\ repeatable \mapsto \text{FALSE},\ group \mapsto \text{“non-destructive”}]$
  $,\ \ [name \mapsto \text{“IVI”},\ repeatable \mapsto \text{FALSE},\ group \mapsto \text{“destructive”}]$
  $\}$,
  $Workflow \leftarrow$
  $\langle \text{“EVI”}, \text{“IVI”} \rangle$

  $ACTUAL \triangleq Validation!Errors$

  $EXPECTED \triangleq Validation!NoErrors$
IN
    $Assert($
        $Print(ACTUAL,\ ACTUAL = EXPECTED)$,
        $Print(EXPECTED,\ \text{“should report no errors for doing non-destructive task before destructive one”})$
    $)$
```

should report errors for doing destructive task before non-destructive one

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  $\{$
  $\}$,
  $Tasks \leftarrow$
  $\{\ [name \mapsto \text{“EVI”},\ repeatable \mapsto \text{FALSE},\ group \mapsto \text{“non-destructive”}]$
  $,\ \ [name \mapsto \text{“IVI”},\ repeatable \mapsto \text{FALSE},\ group \mapsto \text{“destructive”}]$
  $\}$,
  $Workflow \leftarrow$
  $\langle \text{“IVI”}, \text{“EVI”} \rangle$

  $ACTUAL \triangleq Validation!Errors$

  $EXPECTED \triangleq [Validation!NoErrors$ EXCEPT
      $!.ErrorDestructiveBeforeNonDestructive = \{\text{“IVI”}\}$
  $]$
```

```
IN
  $Assert($
    $Print(ACTUAL,\ ACTUAL = EXPECTED),$
    $Print(EXPECTED,$ “should report errors for doing destructive task before non-destructive one”$)$
  $)$
```

should report no errors for containing a mandatory successor task

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
 $Connections \leftarrow$
 { [$name \mapsto$ “has_mandatory_successor”, $srcName \mapsto$ “EVI”, $dstName \mapsto$ “IVI”]
 },
 $Tasks \leftarrow$
 { [$name \mapsto$ “EVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
 ,  [$name \mapsto$ “IVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
 },
 $Workflow \leftarrow$
 ⟨“EVI”, “IVI”⟩
 $ACTUAL \triangleq Validation!Errors$
 $EXPECTED \triangleq Validation!NoErrors$
IN
  $Assert($
    $Print(ACTUAL,\ ACTUAL = EXPECTED),$
    $Print(EXPECTED,$ “should report no errors for containing a mandatory successor task”$)$
  $)$
```

should report errors for missing mandatory successor task

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
 $Connections \leftarrow$
 { [$name \mapsto$ “has_mandatory_successor”, $srcName \mapsto$ “EVI”, $dstName \mapsto$ “IVI”]
 },
 $Tasks \leftarrow$
 { [$name \mapsto$ “EVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
 ,  [$name \mapsto$ “IVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
```

```
  },
  $Workflow \leftarrow$
  $\langle$"EVI"$\rangle$

  $ACTUAL \triangleq Validation!Errors$

  $EXPECTED \triangleq [Validation!NoErrors$ EXCEPT
      $!.ErrorMissingMandatoryDependencies = \{$
        $[name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"$]$
      $\}$
  $]$
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED),$
        $Print(EXPECTED,$ "should report errors for missing mandatory successor task"$)$
    $)$
```

should report no errors for containing a mandatory predecessor task

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  $\{$  $[name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"$]$
  $\},$
  $Tasks \leftarrow$
  $\{$  $[name \mapsto$ "EVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"$]$
  ,   $[name \mapsto$ "IVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"$]$
  $\},$
  $Workflow \leftarrow$
  $\langle$"EVI", "IVI"$\rangle$

  $ACTUAL \triangleq Validation!Errors$

  $EXPECTED \triangleq Validation!NoErrors$
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED),$
        $Print(EXPECTED,$ "should report no errors for containing a mandatory predecessor task"$)$
  )
```

should report errors for missing mandatory predecessor task

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  { [$name \mapsto$ “has_mandatory_predecessor”, $srcName \mapsto$ “IVI”, $dstName \mapsto$ “EVI”]
  },
  $Tasks \leftarrow$
  { [$name \mapsto$ “EVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
  , [$name \mapsto$ “IVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
  },
  $Workflow \leftarrow$
  ⟨“IVI”⟩
  $ACTUAL \triangleq Validation!Errors$
  $EXPECTED \triangleq$ [$Validation!NoErrors$ EXCEPT
      $!.ErrorMissingMandatoryDependencies = \{$
        [$name \mapsto$ “has_mandatory_predecessor”, $srcName \mapsto$ “IVI”, $dstName \mapsto$ “EVI”]
      }
  ]
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED)$,
        $Print(EXPECTED$, “should report errors for missing mandatory predecessor task”)
    )
```

should report no errors for respecting partial order between non-repeatable tasks

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  { [$name \mapsto$ “has_successor”, $srcName \mapsto$ “EVI”, $dstName \mapsto$ “IVI”]
  },
  $Tasks \leftarrow$
  { [$name \mapsto$ “EVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
  , [$name \mapsto$ “IVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
  },
  Workflow ←
  ⟨"EVI", "IVI"⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ Validation!NoErrors
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, "should report no errors for respecting partial order between non-repeatable tasks")
    )
```

should report errors for violating partial order between non-repeatable tasks

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
  Connections ←
  {  [name ↦ "has_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
  },
  Tasks ←
  {  [name ↦ "EVI", repeatable ↦ FALSE, group ↦ "both"]
  ,  [name ↦ "IVI", repeatable ↦ FALSE, group ↦ "both"]
  },
  Workflow ←
  ⟨"IVI", "EVI"⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ [Validation!NoErrors EXCEPT
      !.ErrorPartialOrderViolations = {
        [name ↦ "has_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
      }
  ]
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, "should report errors for violating partial order between tasks")
    )
```

should automatically infer partial order from other connection types - has_predecessor

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  {  [$name \mapsto$ "has_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"]
  },
  $Tasks \leftarrow$
  {  [$name \mapsto$ "EVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"]
  ,  [$name \mapsto$ "IVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"]
  },
  $Workflow \leftarrow$
  ⟨"IVI", "EVI"⟩
  $ACTUAL \triangleq Validation!Errors$
  $EXPECTED \triangleq$ [$Validation!NoErrors$ EXCEPT
      $!.ErrorPartialOrderViolations = \{$
        [$name \mapsto$ "has_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"]
      }
  ]
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED),$
        $Print(EXPECTED,$ "should automatically infer partial order from other connection types - has_predecesso
    )
```

should automatically infer partial order from other connection types - has_mandatory_predecessor

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  {  [$name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"]
  },
  $Tasks \leftarrow$
  {  [$name \mapsto$ "EVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"]
  ,  [$name \mapsto$ "IVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"]
  },
  Workflow ←
  ⟨"IVI", "EVI"⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ [Validation!NoErrors EXCEPT
      !.ErrorPartialOrderViolations = {
        [name ↦ "has_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
      }
  ]
IN
    Assert(
        Print(ACTUAL, ACTUAL.ErrorPartialOrderViolations = EXPECTED.ErrorPartialOrderViolations),
        Print(EXPECTED, "should automatically infer partial order from other connection types - has_mandatory
    )
```

should automatically infer partial order from other connection types - has_mandatory_successor

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
  Connections ←
  {  [name ↦ "has_mandatory_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
  },
  Tasks ←
  {  [name ↦ "EVI", repeatable ↦ FALSE, group ↦ "both"]
  ,  [name ↦ "IVI", repeatable ↦ FALSE, group ↦ "both"]
  },
  Workflow ←
  ⟨"IVI", "EVI"⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ [Validation!NoErrors EXCEPT
      !.ErrorPartialOrderViolations = {
        [name ↦ "has_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
      }
  ]
```

```
IN
  $Assert($
    $Print(ACTUAL,\ ACTUAL.ErrorPartialOrderViolations = EXPECTED.ErrorPartialOrderViolations),$
    $Print(EXPECTED,$ “should automatically infer partial order from other connection types - has_mandatory
  $)$
```

should report no errors for respecting partial order between non-repeatable and repeatable task

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
 $Connections \leftarrow$
 { [$name \mapsto$ “has_successor”, $srcName \mapsto$ “EVI”, $dstName \mapsto$ “IVI”]
 },
 $Tasks \leftarrow$
 { [$name \mapsto$ “EVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
 ,  [$name \mapsto$ “IVI”, $repeatable \mapsto$ TRUE, $group \mapsto$ “both”]
 },
 $Workflow \leftarrow$
 ⟨“EVI”, “IVI”⟩
 $ACTUAL \triangleq Validation!Errors$
 $EXPECTED \triangleq Validation!NoErrors$
IN
  $Assert($
    $Print(ACTUAL,\ ACTUAL = EXPECTED),$
    $Print(EXPECTED,$ “should report no errors for respecting partial order between non-repeatable and repea
  $)$
```

should report errors for violating partial order between non-repeatable and repeatable task

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
 $Connections \leftarrow$
 { [$name \mapsto$ “has_successor”, $srcName \mapsto$ “EVI”, $dstName \mapsto$ “IVI”]
 },
 $Tasks \leftarrow$
 { [$name \mapsto$ “EVI”, $repeatable \mapsto$ FALSE, $group \mapsto$ “both”]
 ,  [$name \mapsto$ “IVI”, $repeatable \mapsto$ TRUE, $group \mapsto$ “both”]
```

```
  },
  $Workflow \leftarrow$
  $\langle$"IVI", "EVI"$\rangle$

  $ACTUAL \triangleq Validation!Errors$

  $EXPECTED \triangleq [Validation!NoErrors$ EXCEPT
      $!.ErrorPartialOrderViolations = \{$
        $[name \mapsto$ "has_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"$]$
      $\}$
  ]
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED)$,
        $Print(EXPECTED$, "should report errors for violating partial order between non-repeatable and repeatable
    )
```

should report no errors for respecting partial order between repeatable and non-repeatable task

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  $\{$  $[name \mapsto$ "has_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"$]$
  $\}$,
  $Tasks \leftarrow$
  $\{$  $[name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$
  ,   $[name \mapsto$ "IVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"$]$
  $\}$,
  $Workflow \leftarrow$
  $\langle$"EVI", "IVI"$\rangle$

  $ACTUAL \triangleq Validation!Errors$

  $EXPECTED \triangleq Validation!NoErrors$
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED)$,
        $Print(EXPECTED$, "should report no errors for respecting partial order between repeatable and non-repea
    )
```

should report errors for violating partial order between repeatable and non-repeatable task

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
  Connections ←
  {  [name ↦ "has_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
  },
  Tasks ←
  {  [name ↦ "EVI", repeatable ↦ TRUE, group ↦ "both"]
  ,  [name ↦ "IVI", repeatable ↦ FALSE, group ↦ "both"]
  },
  Workflow ←
  ⟨"IVI", "EVI"⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ [Validation!NoErrors EXCEPT
      !.ErrorPartialOrderViolations = {
        [name ↦ "has_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
      }
  ]
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, "should report errors for violating partial order between non-repeatable and repeatable
    )
```

should report no errors for repeating non-mandatory tasks in any order

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
  Connections ←
  {  [name ↦ "has_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
  },
  Tasks ←
  {  [name ↦ "EVI", repeatable ↦ TRUE, group ↦ "both"]
  ,  [name ↦ "IVI", repeatable ↦ TRUE, group ↦ "both"]
  },
  Workflow ←
  ⟨"EVI", "IVI", "EVI", "EVI", "IVI", "IVI", "IVI"⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ Validation!NoErrors
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, "should report no errors for repeating non-mandatory tasks in any order")
    )
```

should report no errors for repeating a task and its mandatory successor properly

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
  Connections ←
  {  [name ↦ "has_mandatory_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
  },
  Tasks ←
  {  [name ↦ "EVI", repeatable ↦ TRUE, group ↦ "both"]
  ,  [name ↦ "IVI", repeatable ↦ TRUE, group ↦ "both"]
  },
  Workflow ←
  ⟨"EVI", "IVI", "EVI", "IVI"⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ Validation!NoErrors
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, "should report no errors for repeating a task and its mandatory successor properly")
    )
```

should report errors for repeating a task more often than its mandatory successor at the end

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
```

$Connections \leftarrow$

$\{\ [name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"]

$\},$

$Tasks \leftarrow$

$\{\ [name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]

, $[name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]

$\},$

$Workflow \leftarrow$

⟨"EVI", "IVI", "EVI"⟩

$ACTUAL \triangleq Validation!Errors$

$EXPECTED \triangleq [Validation!NoErrors$ EXCEPT

$!.ErrorMissingMandatoryDependencies = \{$

$[name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"]

$\}$

]

IN

$Assert($

$Print(ACTUAL, ACTUAL = EXPECTED),$

$Print(EXPECTED,$ "should report errors for repeating a task more often than its mandatory successor at

)

should report errors for repeating a task more often than its mandatory successor in between

, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH

$Connections \leftarrow$

$\{\ [name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"]

$\},$

$Tasks \leftarrow$

$\{\ [name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]

, $[name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]

$\},$

$Workflow \leftarrow$

⟨"EVI", "EVI", "IVI"⟩

$ACTUAL \triangleq Validation!Errors$

```
  EXPECTED ≜ [Validation!NoErrors EXCEPT
      !.ErrorMissingMandatoryDependencyRepetitions = {
        [name ↦ "has_mandatory_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
      }
  ]
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, "should report errors for repeating a task more often than its mandatory successor in l
    )
```

should report errors for repeating a task before its mandatory successor

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
  Connections ←
  {  [name ↦ "has_mandatory_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
  },
  Tasks ←
  {  [name ↦ "EVI", repeatable ↦ TRUE, group ↦ "both"]
  ,  [name ↦ "IVI", repeatable ↦ TRUE, group ↦ "both"]
  },
  Workflow ←
  ⟨"EVI", "EVI", "IVI", "IVI"⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ [Validation!NoErrors EXCEPT
      !.ErrorMissingMandatoryDependencyRepetitions = {
        [name ↦ "has_mandatory_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
      }
  ]
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, "should report errors for repeating a task before its mandatory successor")
    )
```

should report no errors for repeating a task and its mandatory predecessor properly

$, \textsc{let}\ Validation \triangleq \textsc{instance}\ WorkflowValidation\ \textsc{with}$

$\quad Connections \leftarrow$

$\quad \{\ [name \mapsto \text{“has\_mandatory\_predecessor”}, srcName \mapsto \text{“IVI”}, dstName \mapsto \text{“EVI”}]$

$\quad \},$

$\quad Tasks \leftarrow$

$\quad \{\ [name \mapsto \text{“EVI”}, repeatable \mapsto \textsc{true}, group \mapsto \text{“both”}]$

$\quad ,\ \ [name \mapsto \text{“IVI”}, repeatable \mapsto \textsc{true}, group \mapsto \text{“both”}]$

$\quad \},$

$\quad Workflow \leftarrow$

$\quad \langle \text{“EVI”}, \text{“IVI”}, \text{“EVI”}, \text{“IVI”} \rangle$

$\quad ACTUAL \triangleq Validation!Errors$

$\quad EXPECTED \triangleq Validation!NoErrors$

$\textsc{in}$

$\qquad Assert($

$\qquad\quad Print(ACTUAL, ACTUAL = EXPECTED),$

$\qquad\quad Print(EXPECTED, \text{“should report no errors for repeating a task and its mandatory predecessor properly”})$

$\qquad)$

should report errors for repeating a task more often than its mandatory predecessor at the beginning

$, \textsc{let}\ Validation \triangleq \textsc{instance}\ WorkflowValidation\ \textsc{with}$

$\quad Connections \leftarrow$

$\quad \{\ [name \mapsto \text{“has\_mandatory\_predecessor”}, srcName \mapsto \text{“IVI”}, dstName \mapsto \text{“EVI”}]$

$\quad \},$

$\quad Tasks \leftarrow$

$\quad \{\ [name \mapsto \text{“EVI”}, repeatable \mapsto \textsc{true}, group \mapsto \text{“both”}]$

$\quad ,\ \ [name \mapsto \text{“IVI”}, repeatable \mapsto \textsc{true}, group \mapsto \text{“both”}]$

$\quad \},$

$\quad Workflow \leftarrow$

$\quad \langle \text{“IVI”}, \text{“EVI”}, \text{“IVI”} \rangle$

$\quad ACTUAL \triangleq Validation!Errors$

$\quad EXPECTED \triangleq [Validation!NoErrors\ \textsc{except}$

```
      $!.ErrorMissingMandatoryDependencies = \{$
        $[name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"$]$
      $\}$
  $]$
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED),$
        $Print(EXPECTED,$ "should report errors for repeating a task more often than its mandatory predecessor a
    $)$
```

should report errors for repeating a task more often than its mandatory predecessor in between

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  $\{$  $[name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"$]$
  $\},$
  $Tasks \leftarrow$
  $\{$  $[name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$
  ,   $[name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$
  $\},$
  $Workflow \leftarrow$
  $\langle$"EVI", "IVI", "IVI"$\rangle$

  $ACTUAL \triangleq Validation!Errors$

  $EXPECTED \triangleq [Validation!NoErrors$ EXCEPT
      $!.ErrorMissingMandatoryDependencyRepetitions = \{$
        $[name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"$]$
      $\}$
  $]$
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED),$
        $Print(EXPECTED,$ "should report errors for repeating a task more often than its mandatory predecessor i
    $)$
```

should report errors for repeating a task before its mandatory predecessor

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  { [$name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"]
  },
  $Tasks \leftarrow$
  { [$name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]
  ,  [$name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]
  },
  $Workflow \leftarrow$
  ⟨"EVI", "EVI", "IVI", "IVI"⟩
  $ACTUAL \triangleq Validation!Errors$
  $EXPECTED \triangleq$ [$Validation!NoErrors$ EXCEPT
      $!.ErrorMissingMandatoryDependencyRepetitions = \{$
        [$name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"]
      }
  ]
IN
   $Assert($
       $Print(ACTUAL, ACTUAL = EXPECTED)$,
       $Print(EXPECTED$, "should report errors for repeating a task before its mandatory predecessor")
   )
```

should report no errors for repeating co-dependent tasks properly

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  { [$name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"]
  ,  [$name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"]
  },
  $Tasks \leftarrow$
  { [$name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]
  ,  [$name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]
  },
  $Workflow \leftarrow$
  $\langle$"EVI", "IVI", "EVI", "IVI"$\rangle$

  $ACTUAL \triangleq Validation!Errors$

  $EXPECTED \triangleq Validation!NoErrors$
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED)$,
        $Print(EXPECTED,$ "should report no errors for repeating co-dependent tasks properly"$)$
    $)$
```

should report errors for repeating one of the co-dependent tasks more often at the end

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  {  [$name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"]
  ,  [$name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"]
  },
  $Tasks \leftarrow$
  {  [$name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]
  ,  [$name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]
  },
  $Workflow \leftarrow$
  $\langle$"EVI", "IVI", "EVI", "IVI", "EVI"$\rangle$

  $ACTUAL \triangleq Validation!Errors$

  $EXPECTED \triangleq$ [$Validation!NoErrors$ EXCEPT $!.ErrorMissingMandatoryDependencies =$
      {  [$name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"]
      ,  [$name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "EVI"]
      }
  ]
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED)$,
        Print(EXPECTED, "should report errors for repeating one of the co-dependent tasks more often at the er
    )
```

should report errors for repeating one of the co-dependent tasks more often at the beginning

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
  Connections ←
  {  [name ↦ "has_mandatory_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
  ,  [name ↦ "has_mandatory_predecessor", srcName ↦ "IVI", dstName ↦ "EVI"]
  },
  Tasks ←
  {  [name ↦ "EVI", repeatable ↦ TRUE, group ↦ "both"]
  ,  [name ↦ "IVI", repeatable ↦ TRUE, group ↦ "both"]
  },
  Workflow ←
  ⟨"IVI", "EVI", "IVI", "EVI", "IVI"⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ [Validation!NoErrors EXCEPT !.ErrorMissingMandatoryDependencies =
      {  [name ↦ "has_mandatory_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
      ,  [name ↦ "has_mandatory_predecessor", srcName ↦ "IVI", dstName ↦ "EVI"]
      }
  ]
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, "should report errors for repeating one of the co-dependent tasks more often at the be
    )
```

should report errors for incorrectly repeating co-dependent tasks

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
  Connections ←
  {  [name ↦ "has_mandatory_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
  ,  [name ↦ "has_mandatory_predecessor", srcName ↦ "IVI", dstName ↦ "EVI"]
  },
  Tasks ←
  {  [name ↦ "EVI", repeatable ↦ TRUE, group ↦ "both"]
  ,   [name ↦ "IVI", repeatable ↦ TRUE, group ↦ "both"]
  },
  Workflow ←
  ⟨"EVI", "IVI", "EVI", "EVI", "IVI", "IVI"⟩

  ACTUAL ≜ Validation!Errors

  EXPECTED ≜ [Validation!NoErrors EXCEPT !.ErrorMissingMandatoryDependencyRepetitions =
      {  [name ↦ "has_mandatory_successor", srcName ↦ "EVI", dstName ↦ "IVI"]
      ,   [name ↦ "has_mandatory_predecessor", srcName ↦ "IVI", dstName ↦ "EVI"]
      }
  ]
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, "should report errors for incorrectly repeating co-dependent tasks")
    )
```

should report no errors for correctly repeating dependent tasks in a more complex example

```
, LET Validation ≜ INSTANCE WorkflowValidation WITH
  Connections ←
  {  [name ↦ "has_mandatory_successor", srcName ↦ "EVI", dstName ↦ "XRAY"]
  ,   [name ↦ "has_mandatory_predecessor", srcName ↦ "IVI", dstName ↦ "XRAY"]
  },
  Tasks ←
  {  [name ↦ "EVI", repeatable ↦ TRUE, group ↦ "both"]
  ,   [name ↦ "XRAY", repeatable ↦ TRUE, group ↦ "both"]
  ,   [name ↦ "IVI", repeatable ↦ TRUE, group ↦ "both"]
  },
  Workflow ←
  ⟨ "EVI", "XRAY"
  ,   "EVI", "XRAY", "IVI"
  ,   "XRAY"
```

$,\quad$ "EVI", "XRAY", "IVI"

$,\quad$ "XRAY", "IVI"

$\rangle$

$ACTUAL \triangleq Validation!Errors$

$EXPECTED \triangleq Validation!NoErrors$

IN

$Assert($

$Print(ACTUAL,\ ACTUAL = EXPECTED),$

$Print(EXPECTED,$ "should report no errors for correctly repeating dependent tasks in a more complex exa

$)$

should report errors for incorrectly repeating dependent tasks in a more complex example I

$,$ LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH

$Connections \leftarrow$

$\{\ [name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "XRAY"$]$

$,\quad [name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "XRAY"$]$

$\},$

$Tasks \leftarrow$

$\{\ [name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$

$,\quad [name \mapsto$ "XRAY", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$

$,\quad [name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$

$\},$

$Workflow \leftarrow$

$\langle$ "EVI"

$,\quad$ "EVI", "XRAY", "IVI"

$,\quad$ "IVI"

$\rangle$

$ACTUAL \triangleq Validation!Errors$

$EXPECTED \triangleq [Validation!NoErrors$ EXCEPT $!.ErrorMissingMandatoryDependencyRepetitions =$

$\{\ [name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "XRAY"$]$

$,\quad [name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "XRAY"$]$

$\}$

```
  ]
IN
    $Assert($
        $Print(ACTUAL,\ ACTUAL = EXPECTED),$
        $Print(EXPECTED,$ “should report errors for incorrectly repeating dependent tasks in a more complex exar
    $)$
```

should report errors for incorrectly repeating dependent tasks in a more complex example II

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  { [$name \mapsto$ “has_mandatory_successor”, $srcName \mapsto$ “EVI”, $dstName \mapsto$ “XRAY”]
  ,  [$name \mapsto$ “has_mandatory_predecessor”, $srcName \mapsto$ “IVI”, $dstName \mapsto$ “XRAY”]
  },
  $Tasks \leftarrow$
  { [$name \mapsto$ “EVI”, $repeatable \mapsto$ TRUE, $group \mapsto$ “both”]
  ,  [$name \mapsto$ “XRAY”, $repeatable \mapsto$ TRUE, $group \mapsto$ “both”]
  ,  [$name \mapsto$ “IVI”, $repeatable \mapsto$ TRUE, $group \mapsto$ “both”]
  },
  $Workflow \leftarrow$
  ⟨ “EVI”
  ,  “EVI”, “XRAY”, “IVI”
  ,  “XRAY”, “IVI”
  ⟩
  $ACTUAL \triangleq Validation!Errors$
  $EXPECTED \triangleq [Validation!NoErrors$ EXCEPT $!.ErrorMissingMandatoryDependencyRepetitions =$
      { [$name \mapsto$ “has_mandatory_successor”, $srcName \mapsto$ “EVI”, $dstName \mapsto$ “XRAY”]
      }
  ]
IN
    $Assert($
        $Print(ACTUAL,\ ACTUAL = EXPECTED),$
        $Print(EXPECTED,$ “should report errors for incorrectly repeating dependent tasks in a more complex exar
    $)$
```

should report no errors for correctly repeating co-dependent tasks in a more complex example

$, \textsc{let}\ \mathit{Validation} \triangleq \textsc{instance}\ \mathit{WorkflowValidation}\ \textsc{with}$

$\quad \mathit{Connections} \leftarrow$

$\quad \{\ [\mathit{name} \mapsto \text{“has\_mandatory\_successor”}, \mathit{srcName} \mapsto \text{“EVI”}, \mathit{dstName} \mapsto \text{“XRAY”}]$

$\quad ,\ \ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”}, \mathit{srcName} \mapsto \text{“XRAY”}, \mathit{dstName} \mapsto \text{“EVI”}]$

$\quad ,\ \ [\mathit{name} \mapsto \text{“has\_mandatory\_successor”}, \mathit{srcName} \mapsto \text{“XRAY”}, \mathit{dstName} \mapsto \text{“IVI”}]$

$\quad ,\ \ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”}, \mathit{srcName} \mapsto \text{“IVI”}, \mathit{dstName} \mapsto \text{“XRAY”}]$

$\quad \},$

$\quad \mathit{Tasks} \leftarrow$

$\quad \{\ [\mathit{name} \mapsto \text{“EVI”}, \mathit{repeatable} \mapsto \textsc{true}, \mathit{group} \mapsto \text{“both”}]$

$\quad ,\ \ [\mathit{name} \mapsto \text{“XRAY”}, \mathit{repeatable} \mapsto \textsc{true}, \mathit{group} \mapsto \text{“both”}]$

$\quad ,\ \ [\mathit{name} \mapsto \text{“IVI”}, \mathit{repeatable} \mapsto \textsc{true}, \mathit{group} \mapsto \text{“both”}]$

$\quad \},$

$\quad \mathit{Workflow} \leftarrow$

$\quad \langle\ \text{“EVI”}, \text{“XRAY”}, \text{“IVI”}$

$\quad ,\ \ \text{“EVI”}, \text{“XRAY”}, \text{“IVI”}$

$\quad ,\ \ \text{“EVI”}, \text{“XRAY”}, \text{“IVI”}$

$\quad \rangle$

$\quad \mathit{ACTUAL} \triangleq \mathit{Validation}!\mathit{Errors}$

$\quad \mathit{EXPECTED} \triangleq \mathit{Validation}!\mathit{NoErrors}$

$\textsc{in}$

$\qquad \mathit{Assert}($

$\qquad\qquad \mathit{Print}(\mathit{ACTUAL}, \mathit{ACTUAL} = \mathit{EXPECTED}),$

$\qquad\qquad \mathit{Print}(\mathit{EXPECTED},$ “should report no errors for correctly repeating co-dependent tasks in a more complex

$\qquad)$

should report errors for incorrectly repeating co-dependent tasks in a more complex example

$, \textsc{let}\ \mathit{Validation} \triangleq \textsc{instance}\ \mathit{WorkflowValidation}\ \textsc{with}$

$\quad \mathit{Connections} \leftarrow$

$\quad \{\ [\mathit{name} \mapsto \text{“has\_mandatory\_successor”}, \mathit{srcName} \mapsto \text{“EVI”}, \mathit{dstName} \mapsto \text{“XRAY”}]$

$\quad ,\ \ [\mathit{name} \mapsto \text{“has\_mandatory\_predecessor”}, \mathit{srcName} \mapsto \text{“XRAY”}, \mathit{dstName} \mapsto \text{“EVI”}]$

$\quad ,\ \ [\mathit{name} \mapsto \text{“has\_mandatory\_successor”}, \mathit{srcName} \mapsto \text{“XRAY”}, \mathit{dstName} \mapsto \text{“IVI”}]$

, $[name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "XRAY"$]$

},

$Tasks \leftarrow$

{ $[name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$

, $[name \mapsto$ "XRAY", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$

, $[name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$

},

$Workflow \leftarrow$

$\langle$ "EVI", "XRAY", "IVI"

, "EVI", "XRAY", "IVI"

, "XRAY", "IVI"

$\rangle$

$ACTUAL \triangleq Validation!Errors$

$EXPECTED \triangleq [Validation!NoErrors$ EXCEPT $!.ErrorMissingMandatoryDependencyRepetitions =$

{ $[name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "XRAY"$]$

, $[name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "XRAY", $dstName \mapsto$ "EVI"$]$

}

$]$

IN

$Assert($

$Print(ACTUAL, ACTUAL = EXPECTED)$,

$Print(EXPECTED$, "should report errors for incorrectly repeating co-dependent tasks in a more complex e

$)$

should report no errors for respecting transitive partial order between tasks

, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH

$Connections \leftarrow$

{ $[name \mapsto$ "has_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "XRAY"$]$

, $[name \mapsto$ "has_successor", $srcName \mapsto$ "XRAY", $dstName \mapsto$ "IVI"$]$

},

$Tasks \leftarrow$

{ $[name \mapsto$ "EVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"$]$

, $[name \mapsto$ "XRAY", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"$]$

```
  ,  $[name \mapsto \text{“IVI”}, repeatable \mapsto \text{FALSE}, group \mapsto \text{“both”}]$
  $\},$
  $Workflow \leftarrow$
  $\langle \text{“EVI”}, \text{“IVI”} \rangle$
  $ACTUAL \triangleq Validation!Errors$
  $EXPECTED \triangleq Validation!NoErrors$
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED),$
        $Print(EXPECTED, \text{“should report errors for violating transitive partial order between tasks”})$
    $)$
```

should report errors for violating transitive partial order between tasks

```
, LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
  $Connections \leftarrow$
  $\{$  $[name \mapsto \text{“has\_successor”}, srcName \mapsto \text{“EVI”}, dstName \mapsto \text{“XRAY”}]$
  ,  $[name \mapsto \text{“has\_successor”}, srcName \mapsto \text{“XRAY”}, dstName \mapsto \text{“IVI”}]$
  $\},$
  $Tasks \leftarrow$
  $\{$  $[name \mapsto \text{“EVI”}, repeatable \mapsto \text{FALSE}, group \mapsto \text{“both”}]$
  ,  $[name \mapsto \text{“XRAY”}, repeatable \mapsto \text{FALSE}, group \mapsto \text{“both”}]$
  ,  $[name \mapsto \text{“IVI”}, repeatable \mapsto \text{FALSE}, group \mapsto \text{“both”}]$
  $\},$
  $Workflow \leftarrow$
  $\langle \text{“IVI”}, \text{“EVI”} \rangle$
  $ACTUAL \triangleq Validation!Errors$
  $EXPECTED \triangleq [Validation!NoErrors$ EXCEPT
      $!.ErrorPartialOrderViolations = \{$
        $[name \mapsto \text{“has\_successor”}, srcName \mapsto \text{“EVI”}, dstName \mapsto \text{“IVI”}]$
      $\}$
  $]$
IN
```

```
    $Assert($
        $Print(ACTUAL,\ ACTUAL = EXPECTED),$
        $Print(EXPECTED,$ “should report errors for violating transitive partial order between tasks”$)$
    $)$
$\rangle$
```

A.5 Workflow repair

MODULE *WorkflowRepair*

EXTENDS *WorkflowDefinition*

LOCAL INSTANCE *Utilities*

GIVEN the max workflow length of:

CONSTANT *MaxDepth*

```
e.g. MaxDepth == 5
```

such that it is a non-negative number

ASSUME $MaxDepth \in Nat$

with the following workflow distance definitions

$missingTaskTypes(W\_new,\ W\_old) \triangleq Cardinality(RAN(W\_old) \setminus RAN(W\_new))$

$additionalTaskTypes(W\_new,\ W\_old) \triangleq Cardinality(RAN(W\_new) \setminus RAN(W\_old))$

$missingTaskAmount(W\_new,\ W\_old) \triangleq BagCardinality(SeqToBag(W\_old) \ominus SeqToBag(W\_new))$

$additionalTaskAmount(W\_new,\ W\_old) \triangleq BagCardinality(SeqToBag(W\_new) \ominus SeqToBag(W\_old))$

$diffWorkflowLength(W\_new,\ W\_old) \triangleq Abs(Len(W\_new) - Len(W\_old))$

```
$diffTaskOrder(W\_new,\ W\_old) \triangleq$
    LET
        $task\_order\_diffs \triangleq [t \in (RAN(W\_old) \cap RAN(W\_new)) \mapsto$
            $Abs(Sum(Indexes(W\_old,\ t)) - Sum(Indexes(W\_new,\ t)))$
        $]$
        $task\_order\_diffs\_bag \triangleq RestrictRangeWithPredicate(task\_order\_diffs,$ LAMBDA $n : n > 0)$
    IN
```

```
        $BagCardinality(task\_order\_diffs\_bag)$
```

and with the thus resulting set of all possible, valid workflows

```
$ValidWorkflows \triangleq \{$
    $W \in BoundedSeq(TaskNames, MaxDepth):$
     LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
     $Tasks \leftarrow Tasks,$
     $Connections \leftarrow Connections,$
     $Workflow \leftarrow W$
     IN  $Validation!Errors = Validation!NoErrors$
$\}$
```

THEN the closest valid workflow is recommended as an alternative to the given workflow, in case the given workflow is invalid

```
$Recommendation \triangleq$
    LET $Validation \triangleq$ INSTANCE $WorkflowValidation$ WITH
    $Connections \leftarrow Connections,$
    $Tasks \leftarrow Tasks,$
    $Workflow \leftarrow Workflow$
    IN
        IF $Validation!Errors = Validation!NoErrors$
         THEN $Workflow$
         ELSE CHOOSE $W\_best \in ValidWorkflows:$
          $\forall W\_worse \in ValidWorkflows : W\_best \neq W\_worse \implies$
              1. minimize deletion of tasks while going from old to new wf
             $\lor missingTaskTypes(W\_worse, Workflow)$
              $> missingTaskTypes(W\_best, Workflow)$
             $\lor$
               $\land missingTaskTypes(W\_worse, Workflow)$
                 $= missingTaskTypes(W\_best, Workflow)$
               $\land$
                 2. minimize addition of tasks while going from old to new wf
                 $\lor additionalTaskTypes(W\_worse, Workflow)$
                  $> additionalTaskTypes(W\_best, Workflow)$
                 $\lor$
```

$\land$ $\mathit{additionalTaskTypes}(W\_worse,\ Workflow)$
  $= \mathit{additionalTaskTypes}(W\_best,\ Workflow)$

$\land$

  3. minimize reduction of task repetitions while going from old to new wf

  $\lor$ $\mathit{missingTaskAmount}(W\_worse,\ Workflow)$
    $> \mathit{missingTaskAmount}(W\_best,\ Workflow)$

  $\lor$

    $\land$ $\mathit{missingTaskAmount}(W\_worse,\ Workflow)$
      $= \mathit{missingTaskAmount}(W\_best,\ Workflow)$

    $\land$

      4. minimize increase of task repetitions while going from old to new wf

      $\lor$ $\mathit{additionalTaskAmount}(W\_worse,\ Workflow)$
        $> \mathit{additionalTaskAmount}(W\_best,\ Workflow)$

      $\lor$

        $\land$ $\mathit{additionalTaskAmount}(W\_worse,\ Workflow)$
          $= \mathit{additionalTaskAmount}(W\_best,\ Workflow)$

        $\land$

          5. minimize ordering difference of matching tasks in old and new wf

          $\lor$ $\mathit{diffTaskOrder}(W\_worse,\ Workflow)$
            $> \mathit{diffTaskOrder}(W\_best,\ Workflow)$

          $\lor$

            $\land$ $\mathit{diffTaskOrder}(W\_worse,\ Workflow)$
              $= \mathit{diffTaskOrder}(W\_best,\ Workflow)$

            $\land$

              6. minimize length difference between old and new wf

              $\lor$ $\mathit{diffWorkflowLength}(W\_worse,\ Workflow)$
                $> \mathit{diffWorkflowLength}(W\_best,\ Workflow)$

              $\lor$ $\mathit{diffWorkflowLength}(W\_worse,\ Workflow)$
                $= \mathit{diffWorkflowLength}(W\_best,\ Workflow)$

```
e.g. Recommendation == << "EVI", "IVI" >>
```

such that it adheres to the expected structure

ASSUME $\mathit{DOM}(\mathit{Recommendation}) = 1 \mathrel{..} \mathit{Len}(\mathit{Recommendation})$ a proper tuple

ASSUME $\forall t \in RAN(Recommendation) : t \in$ STRING

ASSUME $\forall t \in RAN(Recommendation) : \exists\, task \in Tasks : task.name = t$

A.6 Workflow repair tests

```
MODULE WorkflowRepairTest
LOCAL INSTANCE TLC

RepairTests ≜

should not repair trivial, valid workflow

⟨LET Repair ≜ INSTANCE WorkflowRepair WITH
 Connections ←
 {
 },
 Tasks ←
 {  [name ↦ "EVI", repeatable ↦ FALSE, group ↦ "both"]
 },
 Workflow ← ⟨"EVI"⟩,
 MaxDepth ← 3
 ACTUAL ≜ Repair!Recommendation
 EXPECTED ≜ ⟨"EVI"⟩
IN
    Assert(
        Print(ACTUAL, ACTUAL = EXPECTED),
        Print(EXPECTED, "should not repair trivial, valid workflow")
    )

should repair trivial, invalid workflow

, LET Repair ≜ INSTANCE WorkflowRepair WITH
  Connections ←
  {  [name ↦ "has_successor", srcName ↦ "EVI", dstName ↦ "XRAY"]
  },
  $Tasks \leftarrow$
  $\{$  $[name \mapsto$ "EVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"$]$
  ,   $[name \mapsto$ "XRAY", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"$]$
  $\}$,
  $Workflow \leftarrow \langle$"XRAY", "EVI"$\rangle$,
  $MaxDepth \leftarrow 3$

  $ACTUAL \triangleq Repair!Recommendation$

  $EXPECTED \triangleq \langle$"EVI", "XRAY"$\rangle$
IN
    $Assert($
        $Print(ACTUAL, ACTUAL = EXPECTED)$,
        $Print(EXPECTED$, "should repair trivial, invalid workflow"$)$
    $)$
```

should repair invalid workflow with missing mandatory tasks I

```
, LET $Repair \triangleq$ INSTANCE $WorkflowRepair$ WITH
  $Connections \leftarrow$
  $\{$  $[name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "XRAY"$]$
  ,   $[name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "XRAY", $dstName \mapsto$ "EVI"$]$
  ,   $[name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "XRAY", $dstName \mapsto$ "IVI"$]$
  ,   $[name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "XRAY"$]$
  $\}$,
  $Tasks \leftarrow$
  $\{$  $[name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$
  ,   $[name \mapsto$ "XRAY", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$
  ,   $[name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"$]$
  $\}$,
  $Workflow \leftarrow \langle$"XRAY", "XRAY"$\rangle$,
  $MaxDepth \leftarrow 6$

  $ACTUAL \triangleq Repair!Recommendation$

  $EXPECTED\_A \triangleq$
  $\langle$ "EVI", "XRAY", "IVI"
```

$, \quad$ "EVI", "XRAY", "IVI"

$\rangle$

$EXPECTED\_B \triangleq$

$\langle$ "EVI", "XRAY"

$, \quad$ "EVI"

$, \quad$ "IVI"

$, \quad$ "XRAY", "IVI"

$\rangle$

IN

$\quad Assert($

$\qquad Print(ACTUAL, ACTUAL = EXPECTED\_A \lor ACTUAL = EXPECTED\_B),$

$\qquad Print(\langle EXPECTED\_A, EXPECTED\_B \rangle,$ "should repair invalid workflow with missing mandatory tasks I"

$\quad)$

should repair invalid workflow with missing mandatory tasks II

, LET $Repair \triangleq$ INSTANCE $WorkflowRepair$ WITH

$Connections \leftarrow$

$\{ \quad [name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "IVI"]

$, \quad [name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "XRAY"]

$, \quad [name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "XRAY", $dstName \mapsto$ "EVI"]

$, \quad [name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "XRAY", $dstName \mapsto$ "IVI"]

$, \quad [name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "XRAY"]

$\},$

$Tasks \leftarrow$

$\{ \quad [name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]

$, \quad [name \mapsto$ "XRAY", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]

$, \quad [name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]

$\},$

$Workflow \leftarrow \langle$"XRAY", "XRAY"$\rangle,$

$MaxDepth \leftarrow 6$

$ACTUAL \triangleq Repair!Recommendation$

$EXPECTED \triangleq$

$\langle$ "EVI", "XRAY", "IVI"

, "EVI", "XRAY", "IVI"

$\rangle$

IN

$Assert($

$Print(ACTUAL,\ ACTUAL = EXPECTED)$,

$Print(EXPECTED$, "should repair invalid workflow with missing mandatory tasks II")

$)$

should repair invalid workflow with transitive mandatory tasks

, LET $Repair \triangleq$ INSTANCE $WorkflowRepair$ WITH

$Connections \leftarrow$

{ [$name \mapsto$ "has_mandatory_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "XRAY"]

, [$name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "XRAY", $dstName \mapsto$ "EVI"]

, [$name \mapsto$ "has_mandatory_predecessor", $srcName \mapsto$ "IVI", $dstName \mapsto$ "XRAY"]

},

$Tasks \leftarrow$

{ [$name \mapsto$ "EVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]

, [$name \mapsto$ "XRAY", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]

, [$name \mapsto$ "IVI", $repeatable \mapsto$ TRUE, $group \mapsto$ "both"]

},

$Workflow \leftarrow \langle$"IVI"$\rangle$,

$MaxDepth \leftarrow 3$

$ACTUAL \triangleq Repair!Recommendation$

$EXPECTED \triangleq \langle$ "EVI", "XRAY", "IVI"$\rangle$

IN

$Assert($

$Print(ACTUAL,\ ACTUAL = EXPECTED)$,

$Print(EXPECTED$, "should repair invalid workflow with transitive mandatory tasks")

$)$

should repair invalid workflow with transitive partial order violations

, LET $Repair \triangleq$ INSTANCE $WorkflowRepair$ WITH

```
  $Connections \leftarrow$
  $\{$ $[name \mapsto$ "has_successor", $srcName \mapsto$ "EVI", $dstName \mapsto$ "XRAY"$]$
  $,$ $[name \mapsto$ "has_successor", $srcName \mapsto$ "XRAY", $dstName \mapsto$ "IVI"$]$
  $\}$,
  $Tasks \leftarrow$
  $\{$ $[name \mapsto$ "EVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"$]$
  $,$ $[name \mapsto$ "XRAY", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"$]$
  $,$ $[name \mapsto$ "IVI", $repeatable \mapsto$ FALSE, $group \mapsto$ "both"$]$
  $\}$,
  $Workflow \leftarrow \langle$"IVI", "EVI"$\rangle$,
  $MaxDepth \leftarrow 3$

  $ACTUAL \triangleq Repair!Recommendation$

  $EXPECTED \triangleq \langle$ "EVI", "IVI"$\rangle$
IN
   $Assert($
       $Print(ACTUAL, ACTUAL = EXPECTED)$,
       $Print(EXPECTED,$ "should repair invalid workflow with transitive partial order violations"$)$
   $)$

$\rangle$
```

A.7 Test suite

```
MODULE MetaModelReasonerTest
EXTENDS $WorkflowValidationTest, WorkflowRepairTest$
LOCAL INSTANCE $TLC$

ASSUME $PrintT(\langle ValidationTests, RepairTests\rangle)$
```

A.8 Utilities

```
MODULE Utilities
EXTENDS $FiniteSetsExt, Sequences, SequencesExt, Functions, Bags, Relation, Naturals, Integers$
```

Utility helpers for Sets

```
$ChooseOne(S, P(\_)) \triangleq \text{CHOOSE } x \in S : P(x) \land \forall y \in S : P(y) \implies y = x$

$ChooseOrDefault(S, P(\_), D) \triangleq \text{IF } \exists s \in S : P(s)$
    THEN
        $\text{CHOOSE } s \in S : P(s)$
    ELSE
        $D$

$AnyOf(S) \triangleq \text{CHOOSE } s \in S : \text{TRUE}$

$Min(S) \triangleq \text{CHOOSE } min \in S : \forall other \in S : min \leq other$
$Max(S) \triangleq \text{CHOOSE } max \in S : \forall other \in S : max \geq other$

$Abs(x) \triangleq Max(\{x, -x\})$
```

Utility helpers for Functions

```
$DOM(f) \triangleq \text{DOMAIN } f$
$RAN(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$

$RestrictRange(f, S) \triangleq$
   LET
        $DomainForRange \triangleq \{x \in \text{DOMAIN } f : f[x] \in S\}$
   IN
        $Restrict(f, DomainForRange)$

$RestrictRangeWithPredicate(f, P(\_)) \triangleq$
   LET
        $DomainForRange \triangleq \{x \in \text{DOMAIN } f : P(f[x])\}$
   IN
        $Restrict(f, DomainForRange)$
```

Utility helpers for Sequences

```
$Indexes(xs, x) \triangleq \{i \in \text{DOMAIN } xs : xs[i] = x\}$

$Index(xs, x) \triangleq$
   IF
        $IsInjective(xs)$  index non-sensical in non-injective sequences
```

```
    THEN
        $\text{CHOOSE } i \in 1 \,..\, Len(xs) : xs[i] = x$
    ELSE
        $\text{CHOOSE } b \in \text{BOOLEAN} : b \notin \text{BOOLEAN}$

$FirstIndex(xs, x) \triangleq$
    IF
        $\exists\, i \in 1 \,..\, Len(xs) : xs[i] = x$  index non-sensical iff x not in xs
    THEN
        $\text{CHOOSE } i \in 1 \,..\, Len(xs) :$
            $\land\, xs[i] = x$
            $\land\, \forall\, j \in 1 \,..\, Len(xs) : xs[j] = x \implies j \geq i$
    ELSE
        $\text{CHOOSE } b \in \text{BOOLEAN} : b \notin \text{BOOLEAN}$

$LastIndex(xs, x) \triangleq$
    IF
        $\exists\, i \in 1 \,..\, Len(xs) : xs[i] = x$  index non-sensical iff x not in xs
    THEN
        $\text{CHOOSE } i \in 1 \,..\, Len(xs) :$
            $\land\, xs[i] = x$
            $\land\, \forall\, j \in 1 \,..\, Len(xs) : xs[j] = x \implies j \leq i$
    ELSE
        $\text{CHOOSE } b \in \text{BOOLEAN} : b \notin \text{BOOLEAN}$

$Count(xs, x) \triangleq$
    LET
        $Cnt[i \in 1 \,..\, Len(xs)] \triangleq$
        LET
            $Eq \triangleq \text{IF } xs[i] = x \text{ THEN } 1 \text{ ELSE } 0$
        IN
            $\text{IF } i = 1 \text{ THEN } Eq \text{ ELSE } Eq + Cnt[i - 1]$
    IN
        $\text{IF } xs = \langle\rangle \text{ THEN } 0 \text{ ELSE } Cnt[Len(xs)]$

$SumSeq(xs) \triangleq ReduceSeq(\text{LAMBDA } x, acc : acc + x, xs, 0)$
```

```
$ReduceSeqPairs(op(\_, \_), xs, acc) \triangleq$
    LET $ReduceSeqPairs[i \in 1 \,..\, Len(xs)] \triangleq$
    IF $i = 2$
     THEN $op(\langle xs[i-1], xs[i] \rangle, acc)$
     ELSE $op(\langle xs[i-1], xs[i] \rangle, ReduceSeqPairs[i-2])$
    IN
        IF $xs = \langle\rangle \lor Len(xs) \% 2 \neq 0$ THEN $acc$ ELSE $ReduceSeqPairs[Len(xs)]$
```

see https://learntla.com/tla/functions/

$PermutationKey(n) \triangleq \{key \in [1 \,..\, n \rightarrow 1 \,..\, n] : Range(key) = 1 \,..\, n\}$

$PermutationsOf(T) \triangleq \{[x \in 1 \,..\, Len(T) \mapsto T[P[x]]] : P \in PermutationKey(Len(T))\}$

Utility helpers for Bags

$SeqToBag(xs) \triangleq [x \in RAN(xs) \mapsto Count(xs, x)]$

Utility helpers for Relations

$Is2Acyclic(R, S) \triangleq \forall x \in S : \neg(\exists y \in S : x \neq y \land R[x, y] \land R[y, x])$

B Answer set program

This Chapter contains complete & executable[1] answer set programs, written in the language used for the `clingo` reasoner[2], which is mainly based on the ASP-CORE-2 reasoner input language standard [CFGI$^+$20]. The ASP encoding for this thesis' problem statement is divided into problem instance facts NLP_I and appropriate rules for solving such problem class NLP_C. Such uniform problem definition [GKKL$^+$15][GKKS12] is motivated in Section 3.1.7. Here, we complete the listings for the domain model from Section 3.1.7, the workflow validation functionality from Section 3.2.4 and the workflow repair functionality from Section 3.2.4.

Splitting the NLPs into multiple files facilitates reuse of logic statements common to different uniform problem definitions exposed by the system. These files thus represent either (part of) a problem instance or its corresponding problem class, as described in detail in approach Section 3.1.7. The system provides three uniform problem definitions which are used for:

- workflow domain validation
- workflow validation
- workflow repair

The rest of this Chapter is broken up into multiple sections, each containing a listing of the respective NLP file. These source code listings describe:

B.1: the workflow domain of the example FA process given in Section 3.1.1, encoded as appropriate facts

B.2: the display of relevant atoms of the given workflow domain only, using utility directives

B.3: the workflow domain validation functionality, which produces errors if the given workflow domain contains cycles or is not correctly encoded

B.4: the generation of valid workflow candidates, such that they adhere to a linear sequence structure and contain no errors

B.5: the workflow validation functionality, which produces errors if the given workflow violates the imposed constraints

B.6: the workflow repair functionality, which computes appropriate distance measures between the given input workflow and generated output workflow candidate, and chooses the best such candidate according to differently prioritized optimization criteria of those measures

[1]Of note are the workflow validation and repair examples, for which the produced output can be examined, given the provided input workflow.

[2]See `https://potassco.org/clingo/` for more details.

B.7: an example invalid workflow input

B.8: the output of repairing the previous workflow input; slightly manually formatted for better visual clarity

B.9: an example parametrization of generated workflows, such that they adhere to a specified shape

B.10: the output of validating a generated workflow that adheres the previously specified shape; slightly manually formatted for better visual clarity

These NLP files are run via the `clingo` grounder & solver[3]. The example workflow validation can be run with the command `clingo --outf=1 --warn no-atom-undefined kb-facts.lp workflow-input-generated.lp workflow-validator.lp`.The example workflow repair can be run with the command `clingo --outf=1 --warn no-atom-undefined kb-facts.lp workflow-input-error.lp workflow-repairer.lp`.

B.1 Workflow domain

```
task("EVI").
group_non_destructive(task("EVI")).

task("SEM").
group_non_destructive(task("SEM")).
has_predecessor(task("SEM"), task("EVI")).

task("XRAY_MIC").
group_non_destructive(task("XRAY_MIC")).
has_predecessor(task("XRAY_MIC"), task("EVI")).

task("EPT").
group_non_destructive(task("EPT")).
has_predecessor(task("EPT"), task("SEM")).
has_predecessor(task("EPT"), task("XRAY_MIC")).

task("DECAP").
group_destructive(task("DECAP")).
has_predecessor(task("DECAP"), task("EPT")).
has_mandatory_successor(task("DECAP"), task("IVI")).

task("IVI").
group_destructive(task("IVI")).
has_mandatory_predecessor(task("IVI"), task("DECAP")).

task("STRIP").
group_destructive(task("STRIP")).
repeatable(task("STRIP")).
has_mandatory_predecessor(task("STRIP"), task("IVI")).

task("XRAY_SPEC").
group_destructive(task("XRAY_SPEC")).
repeatable(task("XRAY_SPEC")).
has_mandatory_predecessor(task("XRAY_SPEC"), task("STRIP")).

task("SAM").
group_destructive(task("SAM")).
repeatable(task("SAM")).
has_mandatory_predecessor(task("SAM"), task("STRIP")).
has_predecessor(task("SAM"), task("XRAY_SPEC")).
```

[3]See `https://potassco.org/clingo/` for more details. At implementation time version 5.4.0 was used.

```
task("TEM").
group_destructive(task("TEM")).
repeatable(task("TEM")).
has_mandatory_predecessor(task("TEM"), task("STRIP")).
has_predecessor(task("TEM"), task("XRAY_SPEC")).
```

Src. B.1: kb-facts.lp

B.2 Workflow domain display

```
% utility: add default transitive property to connections missing it
has_successor(task(A), task(B), transitive(true)) :- has_successor(task(A), task(B)).
has_predecessor(task(B), task(A), transitive(true)) :- has_predecessor(task(B), task(A)).
has_mandatory_successor(task(A), task(B), transitive(true)) :- has_mandatory_successor(task(A), task(B)).
has_mandatory_predecessor(task(B), task(A), transitive(true)) :- has_mandatory_predecessor(task(B), task(
     A)).

% display tasks
#show task/1.
#show group_non_destructive/1.
#show group_destructive/1.
#show repeatable/1.

% display connections
#show has_successor/3.
#show has_predecessor/3.
#show has_mandatory_successor/3.
#show has_mandatory_predecessor/3.
```

Src. B.2: kb-display.lp

B.3 Workflow domain validation

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% utility
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% utility: add default transitive property to connections missing it
has_successor(task(A), task(B), transitive(true)) :- has_successor(task(A), task(B)).
has_predecessor(task(B), task(A), transitive(true)) :- has_predecessor(task(B), task(A)).
has_mandatory_successor(task(A), task(B), transitive(true)) :- has_mandatory_successor(task(A), task(B)).
has_mandatory_predecessor(task(B), task(A), transitive(true)) :- has_mandatory_predecessor(task(B), task(
     A)).

% utility: construct a DAG by deriving implicit connections:
% has_predecessor and has_successor are inverse connections
edge(task(A), task(B)) :- has_successor(task(A), task(B), _).
edge(task(A), task(B)) :- has_predecessor(task(B), task(A), _).

% utility: construct a DAG by deriving implicit connections:
% mandatory task dependenies can be handled like non-mandatory task order
edge(task(A), task(B)) :- has_mandatory_successor(task(A), task(B), _).
edge(task(A), task(B)) :- has_mandatory_predecessor(task(B), task(A), _).

% utility: determine if connections are transitive
transitive(task(A), task(B)) :- has_successor(task(A), task(B), transitive(true)).
transitive(task(A), task(B)) :- has_predecessor(task(B), task(A), transitive(true)).
transitive(task(A), task(B)) :- has_mandatory_successor(task(A), task(B), transitive(true)).
transitive(task(A), task(B)) :- has_mandatory_predecessor(task(B), task(A), transitive(true)).

% utility: determine if task dependencies are mandatory
requires(task(A), task(B)) :- has_mandatory_successor(task(A), task(B), _).
requires(task(B), task(A)) :- has_mandatory_predecessor(task(B), task(A), _).

% utility: construct a the reflexive transitive closure for the DAG
connected(task(T), task(T)) :- task(T).
connected(task(A), task(B)) :- edge(task(A), task(B)).
connected(task(A), task(B)) :-
    connected(task(A), task(M)), A != M, transitive(task(A), task(M)),
    connected(task(M), task(B)), M != B, transitive(task(M), task(B)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% assumption checking
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% KB must be DAG <=> no 2-cycle in reflexive transitive closure
error(kb, reason(kb_not_a_directed_acyclic_graph)) :-
    connected(task(A), task(B)),
    connected(task(B), task(A)),
    A != B.

% all predicates operate on tasks, check proper references
error(kb, reason(kb_inconsistent)) :- group_non_destructive(Task), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- group_destructive(Task), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- repeatable(Task), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_successor(Task, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_successor(Task, _, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_successor(_, Task), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_successor(_, Task, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_successor(_, _, Transitive), Transitive != transitive(true),
     Transitive != transitive(false).
error(kb, reason(kb_inconsistent)) :- has_predecessor(Task, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_predecessor(Task, _, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_predecessor(_, Task), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_predecessor(_, Task, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_predecessor(_, _, Transitive), Transitive != transitive(true),
     Transitive != transitive(false).
error(kb, reason(kb_inconsistent)) :- has_mandatory_successor(Task, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_mandatory_successor(Task, _, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_mandatory_successor(_, Task), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_mandatory_successor(_, Task, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_mandatory_successor(_, _, Transitive), Transitive != transitive
     (true), Transitive != transitive(false).
error(kb, reason(kb_inconsistent)) :- has_mandatory_predecessor(Task, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_mandatory_predecessor(Task, _, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_mandatory_predecessor(_, Task), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_mandatory_predecessor(_, Task, _), Task != task(T) : task(T).
error(kb, reason(kb_inconsistent)) :- has_mandatory_predecessor(_, _, Transitive), Transitive !=
     transitive(true), Transitive != transitive(false).
```

Src. B.3: kb-validator.lp

B.4 Workflow generation

```
#const maxDepth = 15.

workflow("Output").

orderNumber(0..maxDepth).
```

```
% Generate for each orderNumber potential task assignments to the workflow
{ workflowTaskAssignment(workflow(W), task(T), orderNumber(O)) : task(T) } :- workflow(W), orderNumber(O)
     , W = "Output".

workflowOrderNumber(workflow(W), orderNumber(O)) :- workflowTaskAssignment(workflow(W), _, orderNumber(O)
     ).
workflowTask(workflow(W), task(T)) :- workflowTaskAssignment(workflow(W), task(T), _).

% Ordernumbers must start at 0 and must be continuous
:- workflowOrderNumber(workflow(W), orderNumber(O)),
    not workflowOrderNumber(workflow(W), orderNumber(O2)),
    orderNumber(O2), O2 < O.

% Workflow breadth = 1
:- workflow(W),
    orderNumber(O),
    workflowOrderNumber(workflow(W), orderNumber(O)),
    #count { task(T) : workflowTaskAssignment(workflow(W), task(T), orderNumber(O)) } != 1.

% No errors must occur
:- error(workflow("Output"), _, _).
:- error(workflow("Output"), _, _, _).
```

Src. B.4: workflow-generator.lp

B.5 Workflow validation

```
#include "kb-validator.lp".

% make sure these predicates are defined
workflowOrderNumber(workflow(W), orderNumber(O)) :- workflowTaskAssignment(workflow(W), _, orderNumber(O)
     ).
workflowTask(workflow(W), task(T)) :- workflowTaskAssignment(workflow(W), task(T), _).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% utility
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% utility: workflow length
workflowMaxOrderNumber(workflow(W), orderNumber(OMax)) :-
    workflowTaskAssignment(workflow(W), _, orderNumber(OMax)), orderNumber(OMax),
    #count { orderNumber(OAny) : workflowTaskAssignment(workflow(W), _, orderNumber(OAny)), orderNumber(
         OAny), OAny > OMax } = 0.

% utility: nth assignment of a task in workflow
nthWorkflowTaskAssignment(workflow(W), index(N), task(T), orderNumber(O)) :-
    workflowTaskAssignment(workflow(W), task(T), orderNumber(O)), orderNumber(O),
    #count { orderNumber(OAny) : workflowTaskAssignment(workflow(W), task(T), orderNumber(OAny)),
         orderNumber(OAny), OAny < O } = N.

% utility: first assignment of a task in workflow
firstWorkflowTaskAssignment(workflow(W), task(T), orderNumber(OMin)) :-
    workflowTaskAssignment(workflow(W), task(T), orderNumber(OMin)), orderNumber(OMin),
    #count { orderNumber(OAny) : workflowTaskAssignment(workflow(W), task(T), orderNumber(OAny)),
         orderNumber(OAny), OAny < OMin } = 0.

% utility: latest assignment of a task in workflow
latestWorkflowTaskAssignment(workflow(W), task(T), orderNumber(OMax)) :-
    workflowTaskAssignment(workflow(W), task(T), orderNumber(OMax)), orderNumber(OMax),
    #count { orderNumber(OAny) : workflowTaskAssignment(workflow(W), task(T), orderNumber(OAny)),
         orderNumber(OAny), OAny > OMax } = 0.
```

```
% utility: count occurences of task in workflow
workflowTaskCount(workflow(W), task(T), count(C)) :-
    workflow(W),
    task(T),
    workflowTask(workflow(W), task(T)),
    #count { orderNumber(O) : workflowTaskAssignment(workflow(W), task(T), orderNumber(O)) } = C.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% workflow validation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% check that all given workflowTasks exist
error(workflow(W), reason(unknown_task), task(T)) :-
    workflow(W),
    workflowTask(workflow(W), task(T)),
    not task(T).

% check that non-repeatable tasks can only occur once in workflow
error(workflow(W), reason(non_repeatable), task(T)) :-
    workflow(W),
    task(T),
    workflowTask(workflow(W), task(T)),
    not repeatable(task(T)),
    #count { orderNumber(O) : workflowTaskAssignment(workflow(W), task(T), orderNumber(O)) } != 1.

% check that no destructive tasks comes before non-destructive one
error(workflow(W), reason(destructive_before_non_destructive_task), task(TD)) :-
    workflow(W),
    task(TD), group_destructive(task(TD)), not group_non_destructive(task(TD)),
    orderNumber(OD), workflowTaskAssignment(workflow(W), task(TD), orderNumber(OD)),
    TD != TN,
    task(TN), group_non_destructive(task(TN)), not group_destructive(task(TN)),
    orderNumber(ON), workflowTaskAssignment(workflow(W), task(TN), orderNumber(ON)),
    OD < ON.

% check that task order for non-repeatable predecessor tasks and non-repeatable sucessor tasks is
     satisfied
error(workflow(W), reason(partial_order_violation), task(TA), task(TB)) :-
    not repeatable(task(TA)),
    workflowTaskAssignment(workflow(W), task(TA), orderNumber(OA)), orderNumber(OA),
    connected(task(TA), task(TB)), TA != TB, % not shortCircuit(task(TA), task(TB)),
    not repeatable(task(TB)),
    workflowTaskAssignment(workflow(W), task(TB), orderNumber(OB)), orderNumber(OB),
    OA > OB.

% check that task order for repeatable predecessor tasks and non-repeatable sucessor tasks is satisfied
error(workflow(W), reason(partial_order_violation), task(TA), task(TB)) :-
    repeatable(task(TA)),
    latestWorkflowTaskAssignment(workflow(W), task(TA), orderNumber(OAMax)), orderNumber(OAMax),
    connected(task(TA), task(TB)), TA != TB, % not shortCircuit(task(TA), task(TB)),
    not repeatable(task(TB)),
    workflowTaskAssignment(workflow(W), task(TB), orderNumber(OB)), orderNumber(OB),
    OAMax > OB.

% check that task order for non-repeatable predecessor tasks and repeatable successor tasks is satisfied
error(workflow(W), reason(partial_order_violation), task(TA), task(TB)) :-
    not repeatable(task(TA)),
    workflowTaskAssignment(workflow(W), task(TA), orderNumber(OA)), orderNumber(OA),
    connected(task(TA), task(TB)), TA != TB, % not shortCircuit(task(TA), task(TB)),
    repeatable(task(TB)),
    firstWorkflowTaskAssignment(workflow(W), task(TB), orderNumber(OBMin)), orderNumber(OBMin),
    OA > OBMin.

% check that there is always a mandatory successor task after the dependent task
```

```
error(workflow(W), reason(missing_mandatory_dependency_task), task(TA), task(TB)) :-
    TA != TB,
    connected(task(TA), task(TB)),
    requires(task(TA), task(TB)),
    workflowTaskAssignment(workflow(W), task(TA), orderNumber(OA)),
    #count { OB : workflowTaskAssignment(workflow(W), task(TB), orderNumber(OB)), OB > OA, orderNumber(OB
         ) } = 0.

% check that there is always a mandatory predecessor task before the dependent task
error(workflow(W), reason(missing_mandatory_dependency_task), task(TA), task(TB)) :-
    TB != TA,
    connected(task(TA), task(TB)),
    requires(task(TB), task(TA)),
    workflowTaskAssignment(workflow(W), task(TB), orderNumber(OB)),
    #count { OA : workflowTaskAssignment(workflow(W), task(TA), orderNumber(OA)), OA < OB, orderNumber(OA
         ) } = 0.

% check that the dependent task does not repeat itself before the mandatory successor task
error(workflow(W), reason(missing_mandatory_dependency_task_between_repetitions), task(TA), task(TB)) :-
    TA != TB,
    repeatable(task(TA)),
    repeatable(task(TB)),
    connected(task(TA), task(TB)),
    requires(task(TA), task(TB)),
    workflowTaskAssignment(workflow(W), task(TA), orderNumber(OA)),
    workflowTaskAssignment(workflow(W), task(TA), orderNumber(OA2)),
    OA < OA2,
    #count { OB : workflowTaskAssignment(workflow(W), task(TB), orderNumber(OB)), OA < OB, OB < OA2,
         orderNumber(OB) } = 0.

% check that the dependent task does not repeat itself after the mandatory predecessor task
error(workflow(W), reason(missing_mandatory_dependency_task_between_repetitions), task(TA), task(TB)) :-
    TB != TA,
    repeatable(task(TB)),
    repeatable(task(TA)),
    connected(task(TA), task(TB)),
    requires(task(TB), task(TA)),
    workflowTaskAssignment(workflow(W), task(TB), orderNumber(OB)),
    workflowTaskAssignment(workflow(W), task(TB), orderNumber(OB2)),
    OB2 < OB,
    #count { OA : workflowTaskAssignment(workflow(W), task(TA), orderNumber(OA)), OB2 < OA, OA < OB,
         orderNumber(OA) } = 0.

% display relevant facts
#show workflowTaskAssignment/3.
#show error/2.
#show error/3.
#show error/4.
```

Src. B.5: workflow-validator.lp

B.6 Workflow repair

```
#include "workflow-generator.lp".
#include "workflow-validator.lp".

% are there any errors in the provided workflow input?
hasError(workflow(W)) :- error(workflow(W), _, _).
hasError(workflow(W)) :- error(workflow(W), _, _, _).

% do not generate any alternate workflowTaskAssignment suggestions
% if there are not any errors in provided workflow input
```

```
:- workflowTaskAssignment(workflow("Output"), _, _),
    not hasError(workflow("Input")).

% on the other hand there must be alternate workflowTaskAssignment suggestions
% if there are any errors in provided workflow input
:- not workflowTaskAssignment(workflow("Output"), _, _),
    hasError(workflow("Input")).

% determine tasks that are not in both workflows
diffTaskExistencePos(workflow("Output"), workflow("Input"), task(T)) :-
    workflowTask(workflow("Output"), task(T)),
    not workflowTask(workflow("Input"), task(T)).
diffTaskExistenceNeg(workflow("Output"), workflow("Input"), task(T)) :-
    workflowTask(workflow("Input"), task(T)),
    not workflowTask(workflow("Output"), task(T)).

% determine tasks count difference between workflows

diffTaskCount(workflow("Output"), workflow("Input"), task(T), CGen - CInst) :-
    workflowTaskCount(workflow("Output"), task(T), count(CGen)),
    workflowTaskCount(workflow("Input"), task(T), count(CInst)).
diffTaskCount(workflow("Output"), workflow("Input"), task(T), -CInst) :-
    not workflowTaskCount(workflow("Output"), task(T), _),
    workflowTaskCount(workflow("Input"), task(T), count(CInst)).
diffTaskCount(workflow("Output"), workflow("Input"), task(T), CGen) :-
    workflowTaskCount(workflow("Output"), task(T), count(CGen)),
    not workflowTaskCount(workflow("Input"), task(T), _).

diffTaskCountPos(workflow("Output"), workflow("Input"), task(T), C) :-
    diffTaskCount(workflow("Output"), workflow("Input"), task(T), C), C > 0.
diffTaskCountPos(workflow("Output"), workflow("Input"), task(T), 0) :-
    diffTaskCount(workflow("Output"), workflow("Input"), task(T), C), C <= 0.

diffTaskCountNeg(workflow("Output"), workflow("Input"), task(T), C) :-
    diffTaskCount(workflow("Output"), workflow("Input"), task(T), C), C < 0.
diffTaskCountNeg(workflow("Output"), workflow("Input"), task(T), 0) :-
    diffTaskCount(workflow("Output"), workflow("Input"), task(T), C), C >= 0.

% determine tasks that are differently ordered in both workflows
% for a task with multiple instances this favours the same distance between task repetitions in new
     workflow,
%   e.g. for inst = abbab, gen = babba the differences are 1 and 2 two times respectively,
%   but since these are encoded as predicates, these differences are counted only once, instead of twice
diffTaskOrder(workflow("Output"), workflow("Input"), task(T), | OOutput - OInput |) :-
    workflowTaskAssignment(workflow("Output"), task(T), orderNumber(OOutput)),
    workflowTaskAssignment(workflow("Input"), task(T), orderNumber(OInput)).

% determine difference of workflow length
diffWorkflowLength(workflow("Output"), workflow("Input"), | OOutputMax - OInputMax |) :-
    workflowMaxOrderNumber(workflow("Output"), orderNumber(OOutputMax)),
    workflowMaxOrderNumber(workflow("Input"), orderNumber(OInputMax)).

% determine ordering difference of multiple occurrences of tasks inside output workflow

diffGenTaskOrderBetweenRepetitions(workflow("Output"), task(TA), task(TB), index(NA), index(NB), OB - OA)
      :-
    nthWorkflowTaskAssignment(workflow("Output"), index(NA), task(TA), orderNumber(OA)),
    nthWorkflowTaskAssignment(workflow("Output"), index(NB), task(TB), orderNumber(OB)),
    connected(task(TA), task(TB)), TA != TB,
    NB > NA.
diffGenTaskOrderBetweenRepetitions(workflow("Output"), task(TA), task(TB), index(NA), index(NB), OA - OB)
      :-
    nthWorkflowTaskAssignment(workflow("Output"), index(NA), task(TA), orderNumber(OA)),
    nthWorkflowTaskAssignment(workflow("Output"), index(NB), task(TB), orderNumber(OB)),
    connected(task(TA), task(TB)), TA != TB,
    NA > NB.

diffGenTaskOrderBetweenRepetitionsNeg(workflow("Output"), task(TA), task(TB), index(NA), index(NB), ODiff
     ) :-
    diffGenTaskOrderBetweenRepetitions(workflow("Output"), task(TA), task(TB), index(NA), index(NB),
         ODiff), ODiff < 0.
diffGenTaskOrderBetweenRepetitionsNeg(workflow("Output"), task(TA), task(TB), index(NA), index(NB), 0) :-
    diffGenTaskOrderBetweenRepetitions(workflow("Output"), task(TA), task(TB), index(NA), index(NB),
         ODiff), ODiff >= 0.

% @7: minimize missing tasks compared to input, in output workflow
#maximize { -1@7,T : diffTaskExistenceNeg(workflow("Output"), workflow("Input"), task(T)) }.
% @6: minimize additional tasks compared to input, in output workflow
#minimize { 1@6,T : diffTaskExistencePos(workflow("Output"), workflow("Input"), task(T)) }.

% @5: minimize less occurrences of a task between input and output workflow
#maximize { C@5,T : diffTaskCountNeg(workflow("Output"), workflow("Input"), task(T), C) }.
% @4: minimize more occurrences of a task between input and output workflow
#minimize { C@4,T : diffTaskCountPos(workflow("Output"), workflow("Input"), task(T), C) }.

% @3: minimize difference of different task order in input and output workflow
#minimize { ODiff@3,T : diffTaskOrder(workflow("Output"), workflow("Input"), task(T), ODiff) }.

% @2: minimize difference of workflow length of input and output workflow
#minimize { ODiff@2 : diffWorkflowLength(workflow("Output"), workflow("Input"), ODiff) }.

% @1: minimize occurrences of tasks of later repetitions before tasks of previous repetitions in the
     output workflow
#maximize { ODiff@1,TA,TB,NA,NB : diffGenTaskOrderBetweenRepetitionsNeg(workflow("Output"), task(TA),
     task(TB), index(NA), index(NB), ODiff) }.
```

Src. B.6: workflow-repairer.lp

B.7 Workflow invalid input

```
workflow("Input").

orderNumber(0..1).

% example invalid workflow
workflowTaskAssignment(workflow("Input"), task("EPT"), orderNumber(0)).
workflowTaskAssignment(workflow("Input"), task("EVI"), orderNumber(1)).
```

Src. B.7: workflow-input-error.lp

B.8 Workflow repair of invalid input

```
% clingo version 5.4.0
% Reading from kb-facts.lp ...
% Solving...
% Answer: 2
ANSWER
workflowTaskAssignment(workflow("Input"),task("EPT"),orderNumber(0)).
workflowTaskAssignment(workflow("Input"),task("EVI"),orderNumber(1)).
error(workflow("Input"),reason(partial_order_violation),task("EVI"),task("EPT")).
workflowTaskAssignment(workflow("Output"),task("EVI"),orderNumber(0)).
workflowTaskAssignment(workflow("Output"),task("EPT"),orderNumber(1)).
COST 0 0 0 0 2 0 0
```

```
OPTIMUM
%
% Models       : 2
%   Optimum    : yes
% Optimization : 0 0 0 0 2 0 0
% Calls        : 1
% Time         : 3.475s (Solving: 0.56s 1st Model: 0.11s Unsat: 0.43s)
% CPU Time     : 3.466s
```

Src. B.8: workflow-output-repair.lp

B.9 Workflow generated input

```
#include "workflow-generator.lp".
%%% EXAMPLE PARAMETRIZATION %%%

% Min workflow depth > 10
:- workflow(W),
   W = "Output",
   #count { orderNumber(O) : workflowOrderNumber(workflow(W), orderNumber(O)) } <= 10.

% Do not repeat task more than twice
:- workflow(W),
   W = "Output",
   task(T),
   workflowTask(workflow(W), task(T)),
   #count { orderNumber(O) : workflowTaskAssignment(workflow(W), task(T), orderNumber(O)) } > 2.
```

Src. B.9: workflow-input-generated.lp

B.10 Workflow validation of generated input

```
% clingo version 5.4.0
% Reading from kb-facts.lp ...
% Solving...
% Answer: 1
ANSWER
workflowTaskAssignment(workflow("Output"),task("EVI"),orderNumber(0)).
workflowTaskAssignment(workflow("Output"),task("SEM"),orderNumber(1)).
workflowTaskAssignment(workflow("Output"),task("XRAY_MIC"),orderNumber(2)).
workflowTaskAssignment(workflow("Output"),task("EPT"),orderNumber(3)).
workflowTaskAssignment(workflow("Output"),task("DECAP"),orderNumber(4)).
workflowTaskAssignment(workflow("Output"),task("IVI"),orderNumber(5)).
workflowTaskAssignment(workflow("Output"),task("STRIP"),orderNumber(6)).
workflowTaskAssignment(workflow("Output"),task("XRAY_SPEC"),orderNumber(7)).
workflowTaskAssignment(workflow("Output"),task("TEM"),orderNumber(8)).
workflowTaskAssignment(workflow("Output"),task("SAM"),orderNumber(9)).
workflowTaskAssignment(workflow("Output"),task("STRIP"),orderNumber(10)).
%
% Models       : 1+
% Calls        : 1
% Time         : 0.122s (Solving: 0.01s 1st Model: 0.01s Unsat: 0.00s)
% CPU Time     : 0.122s
```

Src. B.10: workflow-output-validation.lp

C User guide

This Chapter provides a user guide to the implemented system described in Section 4. This user guide assumes access to the complete implementation source code, which is hosted alongside setup and run instructions on a publicly available source code repository[1]. Furthermore, this guides provides instructions for using a local system setup, but also applies to these LotTraveler modules when integrated into a company's existing IT infrastructure, save for differences in regards to maintaining service availability, interactions with an external ontology server and service web endpoint locations. These differences between a local test setup and a live production environment are highlighted in the following sections.

C.1 Setup

As described in implementation Section 4.4, the provided Docker stack spins up the modeler module, the reasoner module, a Mongo database and an Apache Fuseki SPARQL server as separate services, which can communicate with each other.

The local Docker stack is used as follows:

- Change working directory to the `docker-stack` directory
- Use `docker-compose build --no-cache` to build all docker images for this stack
- Use `docker-compose up` to start running all services; press `Ctrl+C` to stop
- Alternatively, use `docker-compose up -d` to run all services in the background; once done, stop them using `docker-compose stop`
- Open `localhost:3030`, `localhost:8888` and `localhost:8080` in your preferred web browser to access the ontology, modeler and reasoner service, respectively (listed in order of usage scenarios)
- You can restart again with `docker-compose up` or with `docker-compose up -d`, preserving the data from before
  - If fuseki complains about an outdated lock, connect to the running docker service via `docker exec -ti docker-stack_fuseki_1 /bin/sh`, run `rm /fuseki/system/tdb.lock` and then `exit`. Afterwards, stop and restart the services.
  - If you need to start from fresh use `docker-compose rm` to remove all service registrations and persistent volumes.

[1]The source code repository is available at `https://github.com/mucaho/lottraveler`.

In a live production environment, these services are automatically maintained and kept alive across server restarts by the container orchestration platform. Hence, these services need to be setup only once before their first use in a company's IT infrastructure.

C.2 Ontology component

After spinning up the relevant services, the Apache Fuseki ontology server can be configured thereafter. It serves a FA ontology which relates FA tasks with each other and additional domain concepts of the business domain. As noted in the implementation Section 4.1, this external component is optional and is not needed for the operation of the self-contained system.

The local Fuseki server is used as follows:

- Using a preferred web-browser navigate to `http://localhost:3030` and log-in with the test user & password `admin`.
- Navigate to the *manage datasets* page and *add a new dataset* there, naming it `FAOntology` and setting its type to *in-memory*. The name of the dataset dictates the Fuseki server endpoint URL from which the ontology can be queried.
- *Upload data* to this dataset by leaving the destination graph name blank, *select*ing the latest `contraints-ontology/FAOntologyV?_ex.rdf` file and *upload*ing it. This ontology corresponds to the class-based OWL-DL2 ontology presented in approach Section 3.1.6. It contains OWL classes & properties which represent the current FA process.
- Optionally, test whether the RDF graph was correctly uploaded by navigating to the dataset's *query* page and running the default `SELECT` query. It should output some RDF triples after running it via the play button ▶.
- The ontology can be opened directly in Protégé from the Fuseki endpoint `http://localhost:3030/FAOntology`. Changes need to be saved to a new RDF-encoded file, which can be re-uploaded to the previously cleared Fuseki dataset.

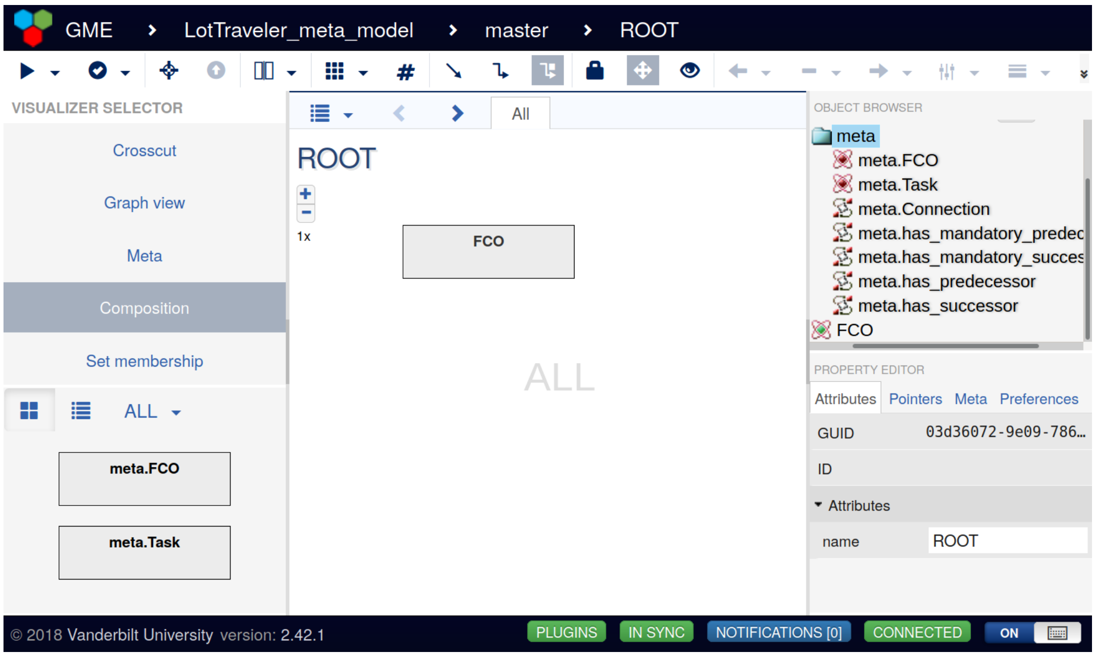
In a live production environment, the dataset needs to be configured only once before the first use of the system. By making sure to create a *persistent* dataset, the managed ontology data survives system restarts.

C.3 Modeler module

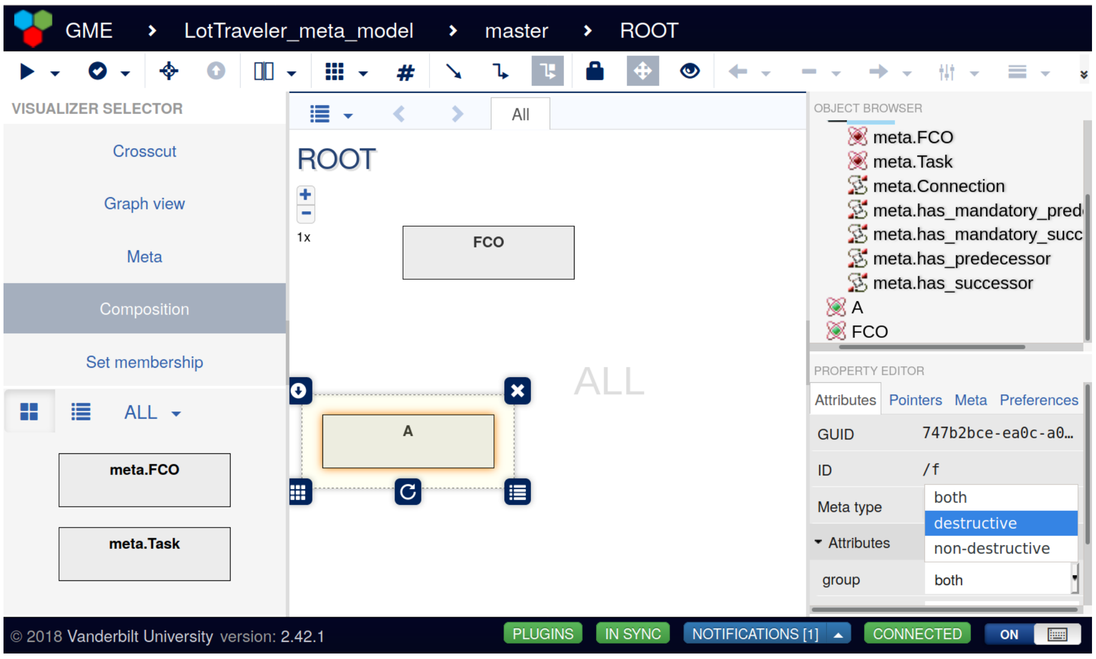
The modeler module enables domain experts and designers to create a workflow meta-model which captures all ground rules for any custom-tailored workflow.

In the following scenario, a new meta-model is created by adding task objects, configuring their attributes, and by associating these tasks via appropriate connections. Furthermore, the meta-model is exported to the ontology and re-imported back again. Finally, these tasks and connections are used to update the reasoner module. This scenario is achieved via the following steps in the locally running modeler module:

- Using a preferred web-browser navigate to `http://localhost:8888`, *create a new guest* project (e.g. call it `LotTraveler_meta_model`) by *importing it from the file* `constraints-modeler/model-template.webgmex`. The imported project comes pre-configured with the appropriate meta-model library which defines the base modeling elements available in this modeling environment. It also enables additional custom extensions in the modeling environment. However, it is otherwise a blank template for the creation of workflow meta-models. The modeling environment, after import of the template project, is showcased in the following figure.

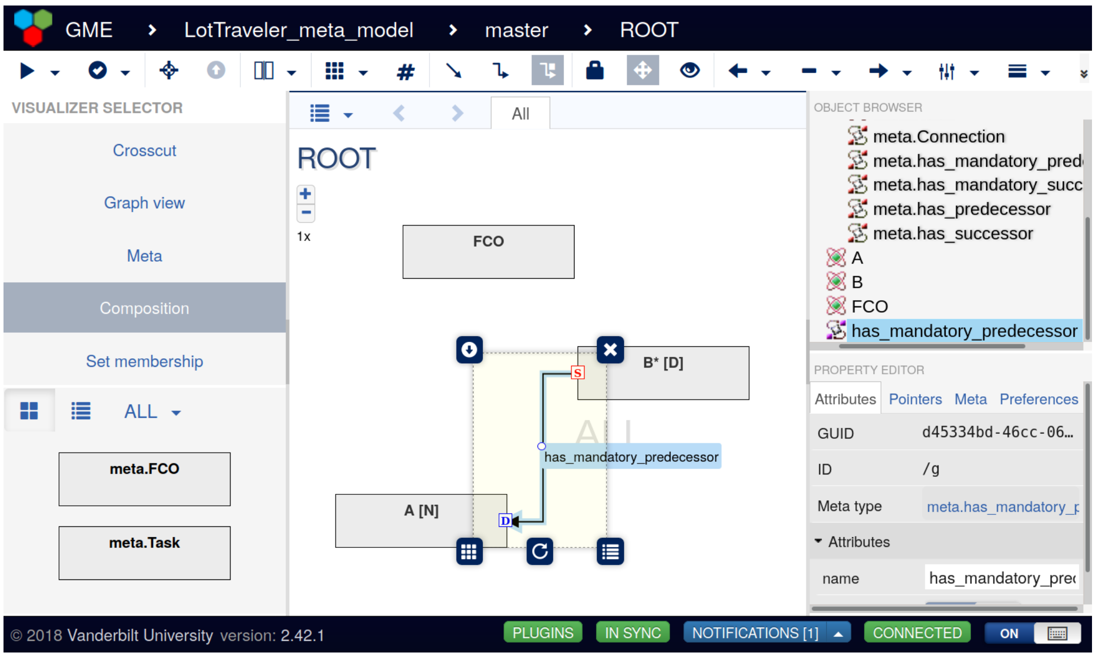


- Add a new task in the *`Root`* of the *composition* hierarchy view by dragging it from the *Part Browser* in the left panel into the main panel. Adapt the newly created task's name by double-clicking on the new object and changing the text to `A`. Change this task's group attribute to *non-destructive* by selecting it and changing the respective property in the *Attribute* tab of the *Property Editor* in the lower-right panel. The following figure shows the current state of the modeling editor during the modification of the task's attributes.



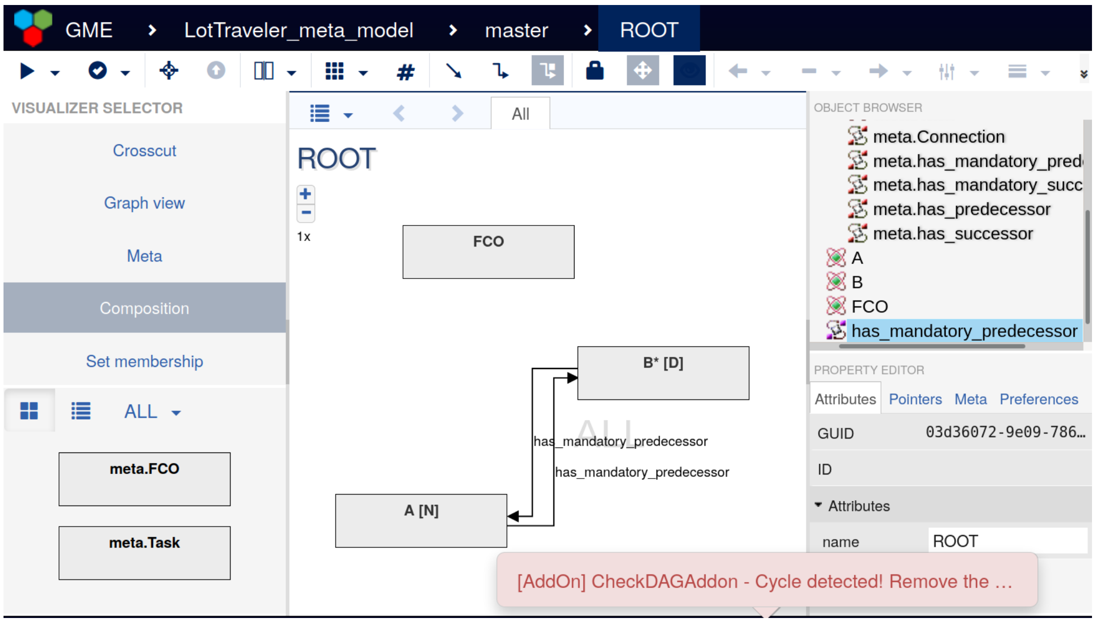
GUID 747b2bce-ea0c-a0…
ID /f
Meta type
Attributes
group both
both
destructive
non-destructive

- Add another task in the same vein, name it `B` and change its group attribute to *destructive* and set its repeatable attribute to *true*. You can move the newly created task by dragging it around the viewport.
- Connect these two tasks by declaring `A` a mandatory predecessor to B. Start by hovering over the second task until two ports pop-up. Click and drag a connection from either of those ports to either of the ports that pop-up while hovering over the first task. Select *has_mandatory_predecessor* as the connection type. The following figure shows the state of the modeling editor after adding the connection.



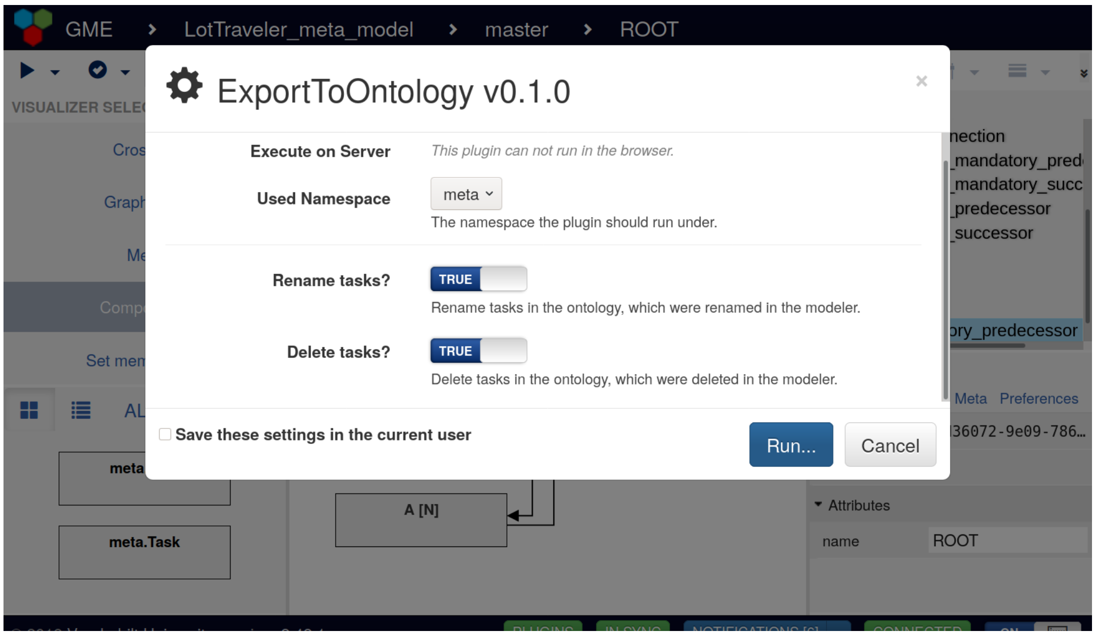
GUID d45334bd-46cc-06…
ID /g
Meta type meta.has_mandatory_p
Attributes
name has_mandatory_pre

- Associate `A` with `B` using an inverse *has_mandatory_predecessor* connection. Observe the displayed error message, which warns about the detected cycle. A similar error message is displayed for creating multiple tasks using the same name. Remove the invalid connection by clicking on it and pressing `DEL` or clicking on the cross icon ×. The following figure showcases the error message in the modeling environment.



[AddOn] CheckDAGAddon - Cycle detected! Remove the …

- Export the current workflow meta-model of tasks and connections to the ontology server by clicking on the *execute plug-in* play button ▶ in the title menu, choosing the *Export-ToOntology* plugin and ticking the appropriate options in the follow-up pop-up modal dialog. Set the *used namespace* to `meta` and choose the rest of options according to the desired export behavior, as described in the plugin implementation Section 4.2. In this case, the export should overwrite all existing tasks and connections in the ontology. A success notification is displayed after the plugin is finished *run*ning. The following figure showcases the chosen options in the pop-up modal dialog of the *ExportToOntology* plugin.

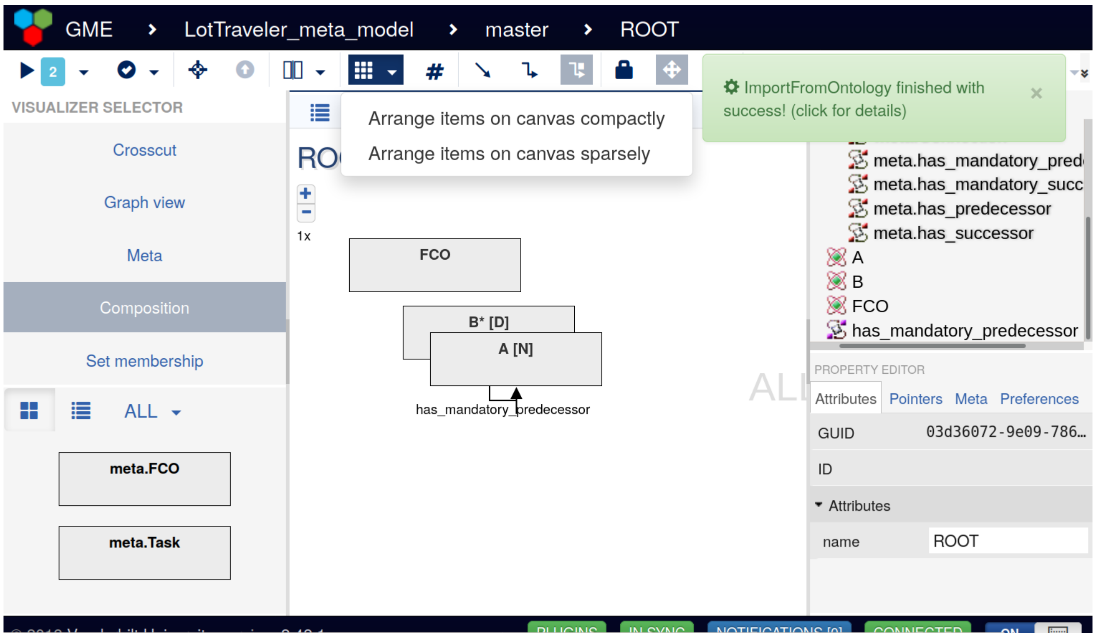


ExportToOntology v0.1.0

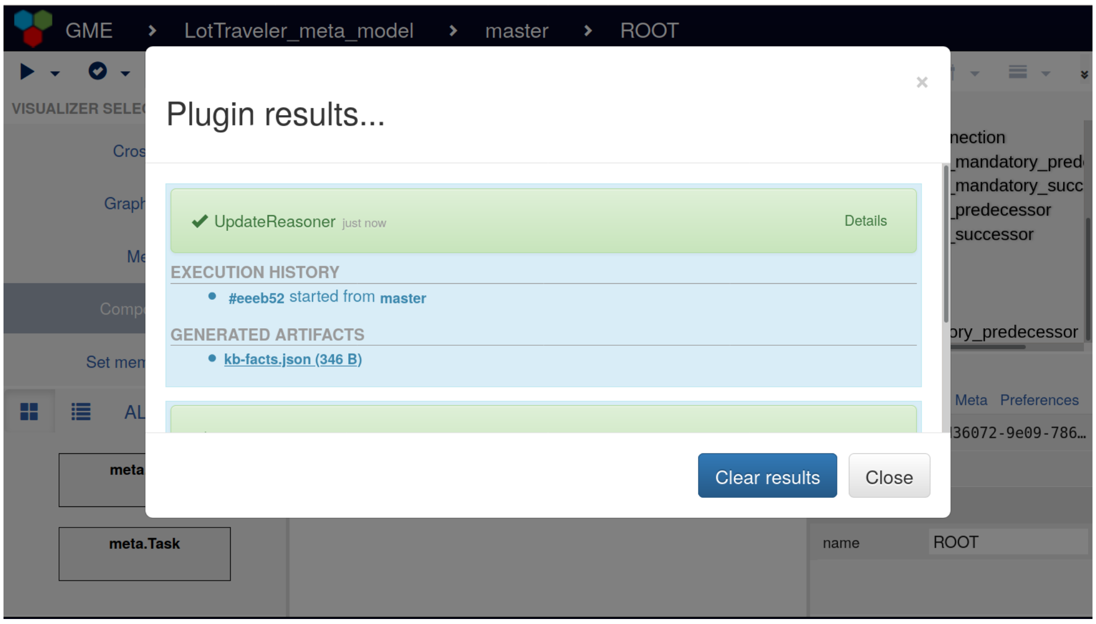
Execute on Server	This plugin can not run in the browser.
Used Namespace	meta The namespace the plugin should run under.
Rename tasks?	TRUE Rename tasks in the ontology, which were renamed in the modeler.
Delete tasks?	TRUE Delete tasks in the ontology, which were deleted in the modeler.

☐ Save these settings in the current user

- Re-import the previously exported workflow meta-model from the ontology server by *running* the *ImportFromOntology* plugin under the `meta` namespace. A success message signals the completion of the import. Choose the option to *arrange items on the canvas compactly* from the *arrange items* menu button. Alternatively, manually arrange these two tasks. The following figure showcases the state of the modeling environment after import.



- Update the knowledge base in the reasoner module by running the *UpdateReasoner* plugin under the `meta` namespace. *Click for details* on the success notification once the plugin finishes. The generated knowledge base, which has been transmitted to the reasoner, is available for download as a `kb-facts.json` file in this detail dialog. The following figure shows the details of the reasoner update.



Plugin results...

UpdateReasoner just now Details

EXECUTION HISTORY

- #eeeb52 started from master

GENERATED ARTIFACTS

- kb-facts.json (346 B)

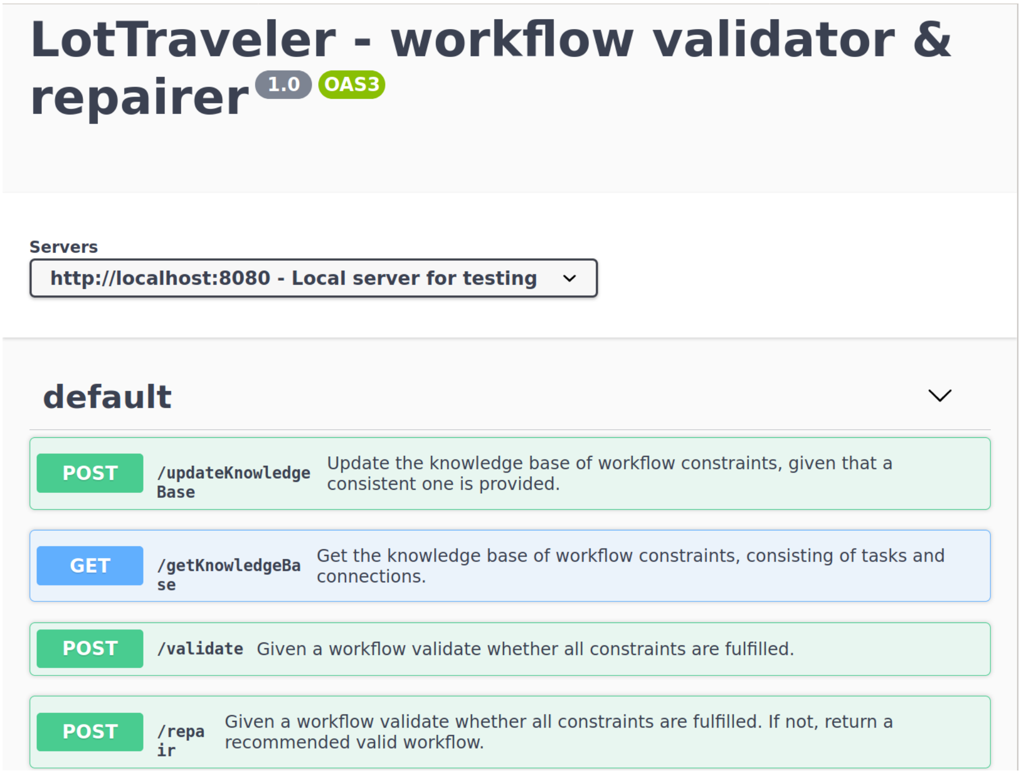
In a live production environment, the project needs to be created only once before the first use of the system, since this modeler service is automatically maintained and kept alive across server restarts by the container orchestration platform. However, the meta-model of the project must be modified each time the ground rules for all workflows change. Therefore, the meta-model's tasks & connections are adapted in the modeling editor, followed by exporting this new state to the ontology. Alternatively, this can be achieved by treating the external ontology as the single source of truth, applying necessary changes to the ontology as dictated by new requirements there, and importing new tasks and connections to the modeler afterwards. In either case, the reasoner module must be updated with the latest knowledge base.

C.4 Reasoner module

The *reasoner* module enables users to validate and repair any custom-tailored workflows against previously supplied ground rules, which must be satisfied by all workflows.

In the following scenario, the current meta-model is retrieved from the reasoner service and checked whether it matches the workflow constraints supplied previously by the modeler. Afterwards, the reasoner's knowledge base is extended with additional tasks & connections. The scenario is concluded by supplying a sequence of tasks to the reasoner, so that it is checked for errors and for which an alternative valid recommendation is returned. This scenario is achieved via the following steps in the locally running modeler module:

- Using a preferred web-browser navigate to `http://localhost:8080`. Follow the link to the *api-docs*. This page serves as a web interface and playground for interacting with the reasoner. Here, each of the reasoner's service methods can be examined. This web page specifies how each method can be called, what the input & output schema looks like and contains example requests. Furthermore, such example or custom requests can directly be executed and the returned responses can be examined. The following figure shows an overview of available service methods in the reasoner's web interface.


LotTraveler - workflow validator & repairer 1.0 OAS3

Servers

http://localhost:8080 - Local server for testing

default

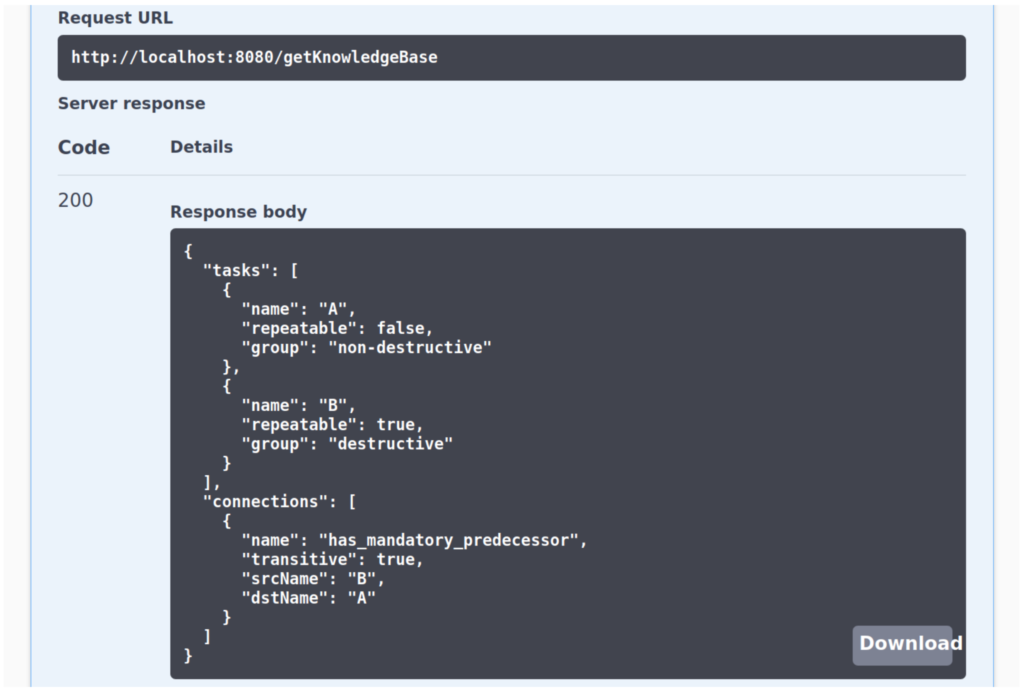
POST /updateKnowledgeBase Update the knowledge base of workflow constraints, given that a consistent one is provided.

GET /getKnowledgeBase Get the knowledge base of workflow constraints, consisting of tasks and connections.

POST /validate Given a workflow validate whether all constraints are fulfilled.

POST /repair Given a workflow validate whether all constraints are fulfilled. If not, return a recommended valid workflow.

- Query the reasoner's current knowledge base by scrolling to the *getKnowledgeBase* method, *trying it out* and thereby *executing* a parameterless example request. A successful response containing tasks and their connections is returned upon completion. *Download* the response body and save it as a `.json` file. The following figure shows the response of querying the reasoner's knowledge base.



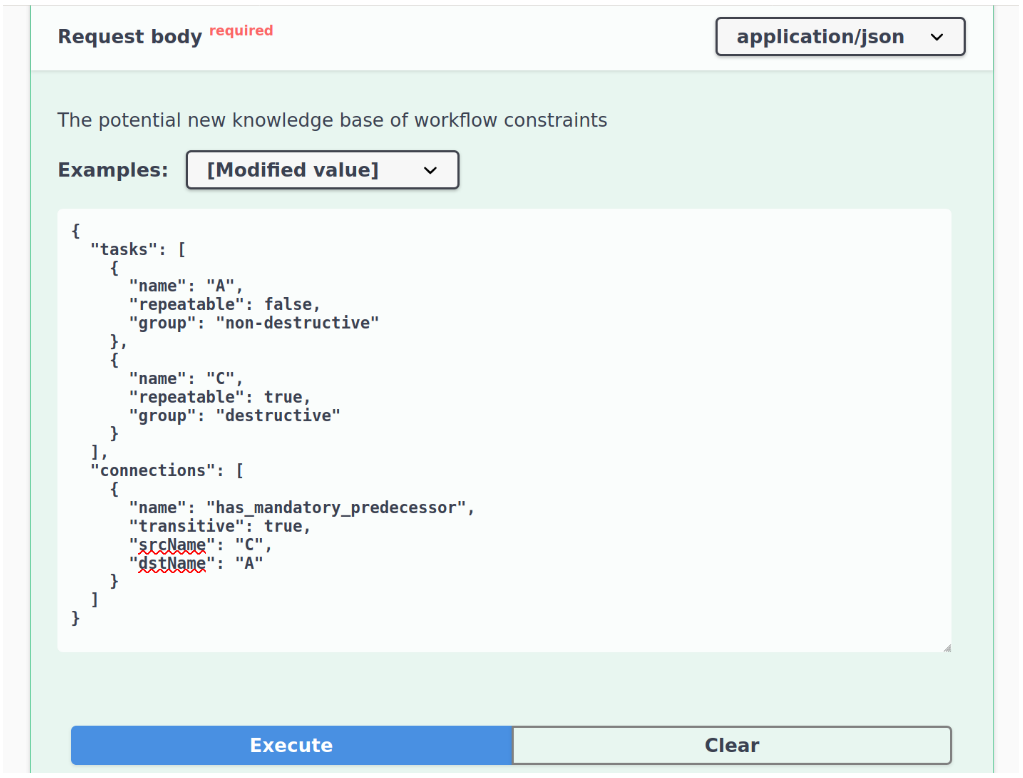
```
http://localhost:8080/getKnowledgeBase
```

200

Response body

```
{
  "tasks": [
    {
      "name": "A",
      "repeatable": false,
      "group": "non-destructive"
    },
    {
      "name": "B",
      "repeatable": true,
      "group": "destructive"
    }
  ],
  "connections": [
    {
      "name": "has_mandatory_predecessor",
      "transitive": true,
      "srcName": "B",
      "dstName": "A"
    }
  ]
}
```

- Update the reasoner's current knowledge base by supplying an altered list of tasks and connections. Open the downloaded `.json` file in a preferred code editor and alter the second task's name from `B` to `C`, in the task and connection list. Copy & paste the whole content of the `.json` file into the request body text-area of the *updateKnowledgeBase* method, after expanding that Section using the *try it out* button. Execute the request. A successful response containing an empty list of errors is returned upon completion. The following figure showcases the request body that the update method is called with.

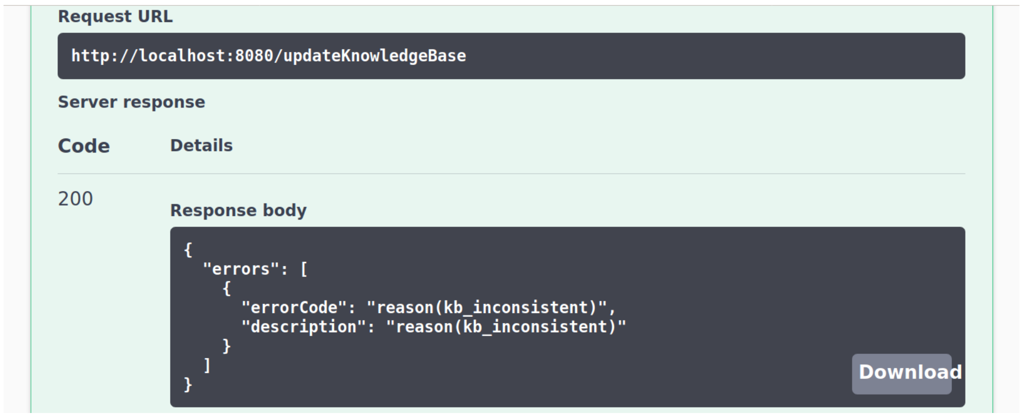


The potential new knowledge base of workflow constraints

Examples: [Modified value]

```
{
  "tasks": [
    {
      "name": "A",
      "repeatable": false,
      "group": "non-destructive"
    },
    {
      "name": "C",
      "repeatable": true,
      "group": "destructive"
    }
  ],
  "connections": [
    {
      "name": "has_mandatory_predecessor",
      "transitive": true,
      "srcName": "C",
      "dstName": "A"
    }
  ]
}
```

- Try to update the reasoner's current knowledge base by supplying an inconsistent knowledge base. Revert one of the occurrences of task name `C` to `B` in the task list, leaving the other occurrence inside the connection list unchanged. After executing this request, an error about an inconsistent knowledge base is returned. A similar error is returned when supplying connections that form a cycle between tasks. Verify the knowledge base has not changed by re-querying it using the *getKnowledgeBase* method. The following figure shows the response containing the error about an inconsistent knowledge base.



Request URL

```
http://localhost:8080/updateKnowledgeBase
```

Server response

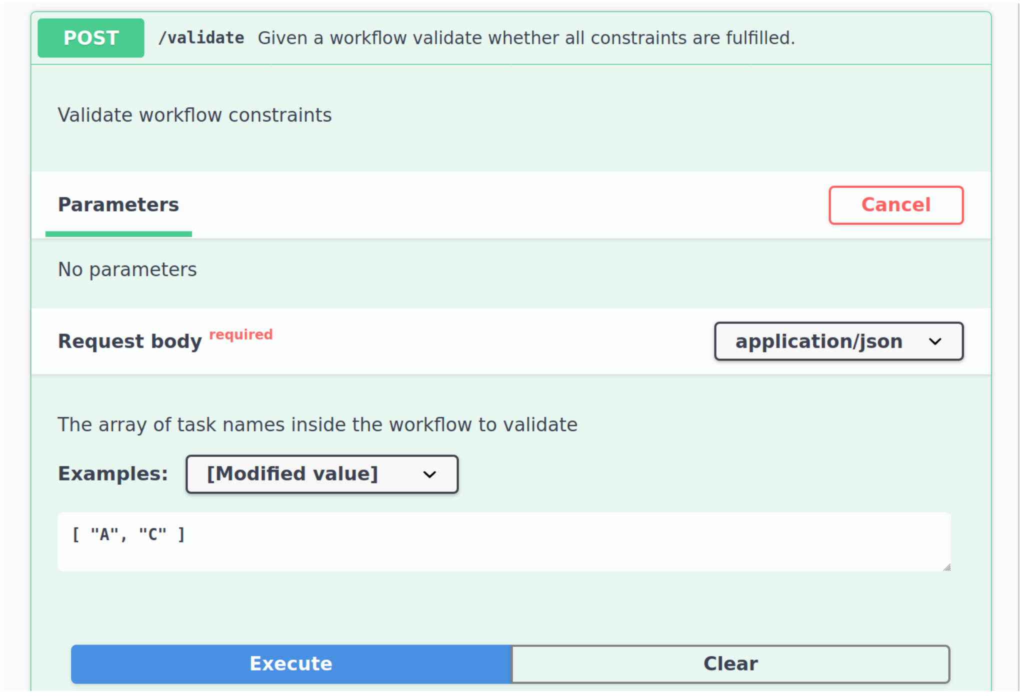
Code Details

200

Response body

```
{
  "errors": [
    {
      "errorCode": "reason(kb_inconsistent)",
      "description": "reason(kb_inconsistent)"
    }
  ]
}
```

- Validate a valid sequence of tasks. Execute a *validate* request using `["A", "C"]` as the request body. The returned response contains an empty error list. The following figure shows the executed *validate* request.



POST `/validate` Given a workflow validate whether all constraints are fulfilled.

Validate workflow constraints

Parameters

No parameters

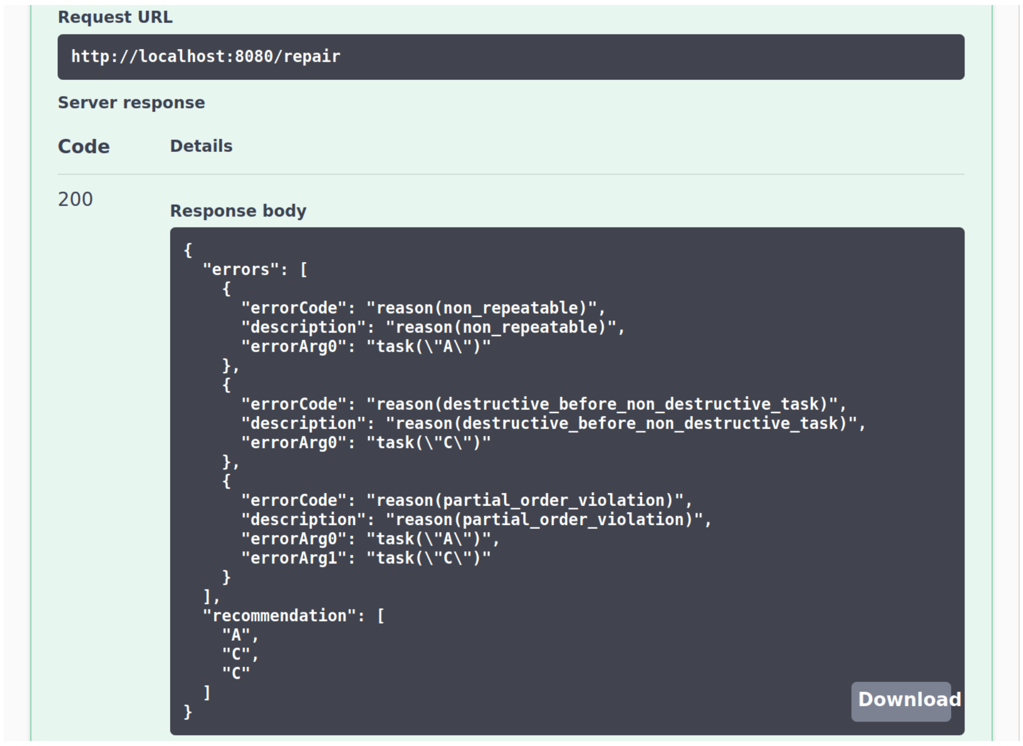
Request body required — application/json

The array of task names inside the workflow to validate

Examples: [Modified value]

```
[ "A", "C" ]
```

- Repair an invalid sequence of tasks. Execute a *repair* request using `["A", "C", "A", "C"]` as the request body. The returned response contains a list of multiple errors and a valid alternative workflow recommendation. The following figure shows the returned *repair* response.



Request URL

```
http://localhost:8080/repair
```

Server response

Code	Details
200	Response body

```
{
  "errors": [
    {
      "errorCode": "reason(non_repeatable)",
      "description": "reason(non_repeatable)",
      "errorArg0": "task(\"A\")"
    },
    {
      "errorCode": "reason(destructive_before_non_destructive_task)",
      "description": "reason(destructive_before_non_destructive_task)",
      "errorArg0": "task(\"C\")"
    },
    {
      "errorCode": "reason(partial_order_violation)",
      "description": "reason(partial_order_violation)",
      "errorArg0": "task(\"A\")",
      "errorArg1": "task(\"C\")"
    }
  ],
  "recommendation": [
    "A",
    "C",
    "C"
  ]
}
```

In a live production environment, the reasoner does not further need to be configured before the first use of the system. Additionally, the reasoner service is automatically maintained and kept alive across server restarts by the container orchestration platform. However, the meta-model of the reasoner must be modified each time the ground rules for all workflows change. Therefore, the modeler service needs to update the reasoner service with a new meta-model corresponding to new requirements. Alternatively, the meta-model can be altered directly in the reasoner by supplying it with a new knowledge base. A typical client application sends respective `POST` requests directly to the web service methods, in contrast to the usage scenario described above that involves manual interaction through the web interface.

Bibliography

[AaAD12] W. Van der Aalst, A. Adriansyah, B. van Dongen: Replaying history on process models for conformance checking and performance analysis. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2, 2 (2012), 182–192.

[AaHe04] W. van der Aalst, K. van Hee: Workflow management: models, methods, and systems. MIT press (2004).

[Aals99] W. van der Aalst: On the automatic generation of workflow processes based on product structures. In: *Computers in Industry*, 39, 2 (1999), 97–111, `https://www.sciencedirect.com/science/article/pii/S016636159900007X`.

[Aals16] W. M. Van der Aalst: Process mining: data science in action. Springer (2016).

[ABSK+20] A. Abbad Andaloussi, A. Burattin, T. Slaats, E. Kindler, B. Weber: On the declarative paradigm in hybrid business process representations: A conceptual framework and a systematic literature study. In: *Information Systems*, 91 (2020), 101505, `https://www.sciencedirect.com/science/article/pii/S0306437920300168`.

[ABVV+00] W. M. P. van der Aalst, T. Basten, H. M. W. Verbeek, P. A. C. Verkoulen, M. Voorhoeve: Adaptive Workflow, Springer Netherlands, Dordrecht (2000), 63–70, `https://doi.org/10.1007/978-94-015-9518-6_5`.

[AhGU72] A. V. Aho, M. R. Garey, J. D. Ullman: The Transitive Reduction of a Directed Graph. In: *SIAM Journal on Computing*, 1, 2 (1972), 131–137, `https://doi.org/10.1137/0201008`.

[AHHS+11] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, M. T. Wynn: Soundness of workflow nets: classification, decidability, and analysis. In: *Formal Aspects of Computing*, 23, 3 (2011), 333–363, `https://doi.org/10.1007/s00165-010-0161-4`.

[AlDC15] M. B. Alves, C. V. Damásio, N. Correia: SPARQL Commands in Jena Rules. *In: P. Klinov, D. Mouromtsev (Hrsg.), Knowledge Engineering and Semantic Web*, Springer International Publishing, Cham (2015), 253–262.

[ApBW88] K. R. Apt, H. A. Blair, A. Walker: Chapter 2 - Towards a Theory of Declarative Knowledge. *In: J. Minker (Hrsg.), Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann (1988), 89–148, `https://www.sciencedirect.com/science/article/pii/B9780934613408500063`.

[ArBa09] S. Arora, B. Barak: Computational complexity: a modern approach. Cambridge University Press (2009).

[ArSS15] R. Arp, B. Smith, A. D. Spear: Building ontologies with basic formal ontology. Mit Press (2015).

[BaHS08] F. Baader, I. Horrocks, U. Sattler: Chapter 3 Description Logics. *In: F. van Harmelen, V. Lifschitz, B. Porter (Hrsg.), Handbook of Knowledge Representation*, Elsevier, *Foundations of Artificial Intelligence*, Bd. 3 (2008), 135–179, `https://www.sciencedirect.com/science/article/pii/S1574652607030039`.

[Bard07] G. V. Bard: Spelling-Error Tolerant, Order-Independent Pass-Phrases via the Damerau-Levenshtein String-Edit Distance Metric. *In: Proceedings of the Fifth Australasian Symposium on ACSW Frontiers - Volume 68*, ACSW '07, Australian Computer Society, Inc., AUS (2007), 117–124.

[BDWVB14] S. K. L. M. vanden Broucke, J. De Weerdt, J. Vanthienen, B. Baesens: Determining Process Model Precision and Generalization with Weighted Artificial Negative Events. In: *IEEE Transactions on Knowledge and Data Engineering*, 26, 8 (2014), 1877–1889.

[BoGG01] E. Boerger, E. Graedel, Y. Gurevich: The classical decision problem. Springer Science & Business Media (2001).

[BrET11] G. Brewka, T. Eiter, M. Truszczyński: Answer Set Programming at a Glance. In: *Commun. ACM*, 54, 12 (2011), 92–103, `https://doi.org/10.1145/2043174.2043195`.

[Camp02] R. Camps: Transforming N-ary Relationships to Database Schemas: An Old and Forgotten Problem. In: *Research Repot LSI-5-02R of Universitat Politècnica de Catalunya (Spain)* (2002).

[CCIL08] F. Calimeri, S. Cozza, G. Ianni, N. Leone: Computable Functions in ASP: Theory and Implementation. *In: M. Garcia de la Banda, E. Pontelli (Hrsg.), Logic Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg (2008), 407–424.

[CFGI$^+$20] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, et al.: ASP-Core-2 Input Language Format. In: *Theory and Practice of Logic Programming*, 20, 2 (2020), 294–309.

[CGT$^+$89] S. Ceri, G. Gottlob, L. Tanca, : What you always wanted to know about Datalog(and never dared to ask). In: *IEEE transactions on knowledge and data engineering*, 1, 1 (1989), 146–166.

[ChAA02] S. A. Chun, V. Atluri, N. R. Adam: Domain Knowledge-Based Automatic Workflow Generation. *In: A. Hameurlain, R. Cicchetti, R. Traunmüller (Hrsg.), Database and Expert Systems Applications*, Springer Berlin Heidelberg, Berlin, Heidelberg (2002), 81–93.

[ChAt03] S. A. Chun, V. Atluri: Ontology-based workflow change management for flexible eGovernment service delivery. *In: Proceedings of the 2003 annual national conference on Digital government research*, Citeseer (2003), 1–4.

[ChYa05] L. Chen, X. Yang: Applying AI Planning to Semantic Web Services for Workflow Generation. *In: 2005 First International Conference on Semantics, Knowledge and Grid* (2005), 65–65.

[CoMu07] G. Cormode, S. Muthukrishnan: The String Edit Distance Matching Problem with Moves. In: *ACM Trans. Algorithms*, 3, 1 (2007), `https://doi.org/10.1145/1186810.1186812`.

[Coot06a] W. W. W. Consortium, : Defining N-ary Relations on the Semantic Web (2006), `https://www.w3.org/TR/2006/NOTE-swbp-n-aryRelations-20060412/`.

[Coot06b] W. W. W. Consortium, : The Rule of Least Power (2006), `https://www.w3.org/2001/tag/doc/leastPower-2006-02-23.html`.

[Coot12a] W. W. W. Consortium, : OWL 2 Web Ontology Language Direct Semantics (2012), `https://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/`.

[Coot12b] W. W. W. Consortium, : OWL 2 Web Ontology Language Profiles (2012), `https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/`.

[Coot12c] W. W. W. Consortium, : OWL 2 Web Ontology Language RDF-Based Semantics (2012), `https://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/`.

[Coot12d] W. W. W. Consortium, : OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (2012), `https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/`.

[Coot13] W. W. W. Consortium, : SPARQL 1.1 overview (2013), `https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/`.

[Coot14a] W. W. W. Consortium, : RDF 1.1 Semantics (2014), `https://www.w3.org/TR/2014/REC-rdf11-mt-20140225/`.

[Coot14b] W. W. W. Consortium, : RDF 1.1 Turtle: terse RDF triple language (2014), `https://www.w3.org/TR/2014/REC-turtle-20140225/`.

[Coot14c] W. W. W. Consortium, : RDF Schema 1.1 (2014), `https://www.w3.org/TR/2014/REC-rdf-schema-20140225/`.

[Coot17a] W. W. W. Consortium, : SHACL Advanced Features (2017), `https://www.w3.org/TR/2017/NOTE-shacl-af-20170608/`.

[Coot17b] W. W. W. Consortium, : Shapes Constraint Language (SHACL) (2017), `https://www.w3.org/TR/2017/REC-shacl-20170720/`.

[DBGK$^+$03] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, S. Koranda: Mapping Abstract Complex Workflows onto Grid Environments. In: *Journal of Grid Computing*, 1, 1 (2003), 25–39, `https://doi.org/10.1023/A:1024000426962`.

[DBGK04a] E. Deelman, J. Blythe, Y. Gil, C. Kesselman: Workflow Management in Griphyn, Springer US, Boston, MA (2004), 99–116, `https://doi.org/10.1007/978-1-4615-0509-9_7`.

[DBGK^{+}04b] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, M. Livny: Pegasus: Mapping Scientific Workflows onto the Grid. *In: M. D. Dikaiakos (Hrsg.), Grid Computing*, Springer Berlin Heidelberg, Berlin, Heidelberg (2004), 11–20.

[DEGV01] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov: Complexity and Expressive Power of Logic Programming. In: *ACM Comput. Surv.*, 33, 3 (2001), 374–425, `https://doi.org/10.1145/502807.502810`.

[DeZi04] J. Dehnert, A. Zimmermann: Making Workflow Models Sound Using Petri Net Controller Synthesis. *In: R. Meersman, Z. Tari (Hrsg.), On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, Springer Berlin Heidelberg, Berlin, Heidelberg (2004), 139–154.

[DGAV05] P. Doshi, R. Goodwin, R. Akkiraju, K. Verma: Dynamic workflow composition: Using markov decision processes. In: *International Journal of Web Services Research (IJWSR)*, 2, 1 (2005), 1–17.

[DSMB19] S. Dunzer, M. Stierle, M. Matzner, S. Baier: Conformance Checking: A State-of-the-Art Literature Review. *In: Proceedings of the 11th International Conference on Subject-Oriented Business Process Management*, S-BPM ONE '19, Association for Computing Machinery, New York, NY, USA (2019), `https://doi.org/10.1145/3329007.3329014`.

[DuKo11] D.-Z. Du, K.-I. Ko: Theory of computational complexity, Bd. 58. John Wiley & Sons (2011).

[EGGGB^{+}95] R. Erich Gamma, E. Gamma, E. Gamma, G. Booch, R. Helm, R. Johnson, Addison-Wesley, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley professional computing series, Addison-Wesley (1995), `https://books.google.hr/books?id=tmNNfSkfTlcC`.

[EiIK09] T. Eiter, G. Ianni, T. Krennwallner: Answer Set Programming: A Primer, Springer Berlin Heidelberg, Berlin, Heidelberg (2009), 40–110, `https://doi.org/10.1007/978-3-642-03754-2_2`.

[FaAa15] D. Fahland, W. M. van der Aalst: Model repair - aligning process models to reality. In: *Information Systems*, 47 (2015), 220–243, `https://www.sciencedirect.com/science/article/pii/S0306437913001725`.

[FFST^{+}18] A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E. C. Teppan: Industrial Applications of Answer Set Programming. In: *KI - Künstliche Intelligenz*, 32, 2 (2018), 165–176, `https://doi.org/10.1007/s13218-018-0548-6`.

[Fitt12] M. Fitting: First-order logic and automated theorem proving. Springer Science & Business Media (2012).

[Foot19] T. A. S. Foundation, : Apache Jena - Reasoners and rule engines: Jena inference support (2019), `https://jena.apache.org/documentation/inference/#rules`.

[Forg89] C. L. Forgy: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *In: J. Mylopolous, M. Brodie (Hrsg.), Readings in Artificial Intelligence and Databases*, Morgan Kaufmann, San Francisco (CA) (1989), 547–559, `https://www.sciencedirect.com/science/article/pii/B9780934613538500418`.

[Frey05] T. Freytag: Woped–workflow petri net designer. In: *University of Cooperative Education* (2005), 279–282.

[FrRu05] R. France, B. Rumpe: Domain specific modeling (2005).

[GDBK$^+$04] Y. Gil, E. Deelman, J. Blythe, C. Kesselman, H. Tangmunarunkit: Artificial intelligence and grids: workflow planning and beyond. In: *IEEE Intelligent Systems*, 19, 1 (2004), 26–33.

[GHVD03] B. N. Grosof, I. Horrocks, R. Volz, S. Decker: Description Logic Programs: Combining Logic Programs with Description Logic. *In: Proceedings of the 12th International Conference on World Wide Web*, WWW '03, Association for Computing Machinery, New York, NY, USA (2003), 48–57, `https://doi.org/10.1145/775152.775160`.

[GKKL$^+$15] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele: Potassco user guide. In: *Institute for Informatics, University of Potsdam, second edition edition* (2015), 69.

[GKKS12] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub: Answer Set Solving in Practice. In: *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6, 3 (2012), 1–238, `https://doi.org/10.2200/S00457ED1V01Y201211AIM019`.

[Glim11] B. Glimm: Using SPARQL with RDFS and OWL Entailment, Springer Berlin Heidelberg, Berlin, Heidelberg (2011), 137–201, `https://doi.org/10.1007/978-3-642-23032-5_3`.

[GNTG$^+$07] J. Gray, S. Neema, J.-P. Tolvanen, A. S. Gokhale, S. Kelly, J. Sprinkle: Domain-Specific Modeling. In: *Handbook of dynamic system modeling*, 7 (2007), 7–1.

[GrKP94] R. Graham, D. Knuth, O. Patashnik: Concrete Mathematics: A Foundation for Computer Science. A @foundation for computer science, Addison-Wesley (1994).

[GrOR11] G. Grambow, R. Oberhauser, M. Reichert: Semantically-Driven Workflow Generation Using Declarative Modeling for Processes in Software Engineering. *In: 2011 IEEE 15th International Enterprise Distributed Object Computing Conference Workshops* (2011), 164–173.

[GRot14] O. M. GROUP, : Business Process Model and Notation Specification Version 2.0.2 (2014), `https://www.omg.org/spec/BPMN/2.0.2`.

[GRot16] O. M. GROUP, : Case Management Model and Notation Specification Version 1.1 (2016), `https://www.omg.org/spec/CMMN/1.1`.

[GuMM81] J. E. Gunn, S. K. Malik, P. M. Mazumdar: Highly Accelerated Temperature and Humidity Stress Test Technique (HAST). *In: 19th International Reliability Physics Symposium* (1981), 48–51.

[HaZu14] C. Haisjackl, S. Zugal: Investigating Differences between Graphical and Textual Declarative Process Models. *In: L. Iliadis, M. Papazoglou, K. Pohl (Hrsg.), Advanced Information Systems Engineering Workshops*, Springer International Publishing, Cham (2014), 194–206.

[HBPS08] M. Horridge, J. Bauer, B. Parsia, U. Sattler: Understanding Entailments in OWL. *In: OWLED* (2008).

[HeGA20] J. Hendler, F. Gandon, D. Allemang: Semantic Web for the Working Ontologist: Effective Modeling for Linked Data, RDFS, and OWL. Morgan & Claypool (2020).

[HeSW13] K. Hee, N. Sidorova, J. M. Van der Werf: Business Process Modeling Using Petri Nets. In: , 7480 (2013).

[HFBPL09] J. Hebeler, M. Fisher, R. Blace, A. Perez-Lopez: Semantic web programming. In: *Notes*, 3 (2009), 4.

[HiKR09] P. Hitzler, M. Krotzsch, S. Rudolph: Foundations of semantic web technologies. CRC press (2009).

[HoKS06] I. Horrocks, O. Kutz, U. Sattler: The Even More Irresistible SROIQ. In: *Kr*, 6 (2006), 57–67.

[Holl95] D. Hollingsworth: Workflow management coalition: The workflow reference model. In: *Document Number TC00-1003*, 19, 16 (1995), 224.

[HyNI10] H. Hyyrö, K. Narisawa, S. Inenaga: Dynamic Edit Distance Table under a General Weighted Cost Function. *In: J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, B. Rumpe (Hrsg.), SOFSEM 2010: Theory and Practice of Computer Science*, Springer Berlin Heidelberg, Berlin, Heidelberg (2010), 515–527.

[KeTo08] S. Kelly, J.-P. Tolvanen: Domain-specific modeling: enabling full code generation. John Wiley & Sons (2008).

[KLPS16] B. Kaufmann, N. Leone, S. Perri, T. Schaub: Grounding and Solving in Answer Set Programming. In: *AI Magazine*, 37, 3 (2016), 25–32, `https://ojs.aaai.org/index.php/aimagazine/article/view/2672`.

[KoOb05] A. Koschmider, A. Oberweis: Ontology Based Business Process Description. *In: EMOI-INTEROP* (2005), 321–333.

[KPHJ19] M. Kocbek Bule, G. Polančič, J. Huber, G. Jošt: Semiotic clarity of Case Management Model and Notation (CMMN). In: *Computer Standards and Interfaces*, 66 (2019), 103354, `https://www.sciencedirect.com/science/article/pii/S0920548918302514`.

[KPZS$^+$15] B. T. G. S. Kumara, I. Paik, J. Zhang, T. H. A. S. Siriweera, K. R. C. Koswatte: Ontology-Based Workflow Generation for Intelligent Big Data Analytics. *In: 2015 IEEE International Conference on Web Services* (2015), 495–502.

[KSFL15] M. Kurz, W. Schmidt, A. Fleischmann, M. Lederer: Leveraging CMMN for ACM: Examining the Applicability of a New OMG Standard for Adaptive Case Management. *In: Proceedings of the 7th International Conference on Subject-Oriented*

Business Process Management, S-BPM ONE '15, Association for Computing Machinery, New York, NY, USA (2015), https://doi.org/10.1145/2723839.2723843.

[Lamp02] L. Lamport: Specifying systems, Bd. 388. Addison-Wesley Boston (2002).

[LeKe08] D. Leake, J. Kendall-Morwick: Towards Case-Based Support for e-Science Workflow Generation by Mining Provenance. *In: K.-D. Althoff, R. Bergmann, M. Minor, A. Hanft (Hrsg.), Advances in Case-Based Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg (2008), 269–283.

[LeRS97] N. Leone, P. Rullo, F. Scarcello: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics, and Computation. In: *Information and Computation*, 135, 2 (1997), 69–112, https://www.sciencedirect.com/science/article/pii/S0890540197926304.

[LiLi09] Y. Lierler, V. Lifschitz: One More Decidable Class of Finitely Ground Programs. *In: P. M. Hill, D. S. Warren (Hrsg.), Logic Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg (2009), 489–493.

[LiTK81] T.-W. Ling, F. W. Tompa, T. Kameda: An Improved Third Normal Form for Relational Databases. In: *ACM Trans. Database Syst.*, 6, 2 (1981), 329–346, https://doi.org/10.1145/319566.319583.

[Lloy94] J. W. Lloyd: Practical Advtanages of Declarative Programming. *In: M. Alpuente, R. Barbuti, I. Ramos (Hrsg.), 1994 Joint Conference on Declarative Programming, GULP-PRODE'94 Peñiscola, Spain, September 19-22, 1994, Volume 1* (1994), 18–30.

[LuBL06] S. Lu, A. Bernstein, P. Lewis: Automatic workflow verification and generation. In: *Theoretical Computer Science*, 353, 1 (2006), 71–92, https://www.sciencedirect.com/science/article/pii/S0304397505007243.

[MaHM16] M. A. Marin, M. Hauder, F. Matthes: Case Management: An Evaluation of Existing Approaches for Knowledge-Intensive Processes. *In: M. Reichert, H. A. Reijers (Hrsg.), Business Process Management Workshops*, Springer International Publishing, Cham (2016), 5–16.

[MaKE18] A. Mashkoor, F. Kossak, A. Egyed: Evaluating the suitability of state-based formal methods for industrial deployment. In: *Software: Practice and Experience*, 48, 12 (2018), 2350–2379, https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2634.

[MKKB+14] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, Á. Lédeczi: Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure. In: *MPM@ MoDELS*, 1237 (2014), 41–60.

[MLBZ16] Z. Ming, X. Ling, X. Bai, B. Zong: A review of the technology and process on integrated circuits failure analysis applied in communications products. In: *Journal of Physics: Conference Series*, 679 (2016), 012040.

[Nava01] G. Navarro: A Guided Tour to Approximate String Matching. In: *ACM Comput. Surv.*, 33, 1 (2001), 31–88, https://doi.org/10.1145/375360.375365.

[ObYu18] A. Oberai, J.-S. Yuan: Efficient Fault Localization and Failure Analysis Techniques for Improving IC Yield. In: *Electronics*, 7 (2018), 28.

[Papa94] C. H. Papadimitriou: Computational complexity. Addison-Wesley (1994).

[Pesi08] M. Pesic: Constraint-based workflow management systems : shifting control to users. Dissertation, Industrial Engineering and Innovation Sciences (2008), proefschrift.

[PKNŞ19] P. Pareti, G. Konstantinidis, T. J. Norman, M. Şensoy: SHACL Constraints with Inference Rules. *In: C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois, F. Gandon (Hrsg.), The Semantic Web – ISWC 2019*, Springer International Publishing, Cham (2019), 539–557.

[RoLB07] S. Roser, F. Lautenbacher, B. Bauer: Generation of workflow code from DSMs. *In: J. Sprinkle (Hrsg.), Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling, 21-22 October 2007, Montréal, Canada* (2007), `http://www.dsmforum.org/events/DSM07/papers/roser.pdf`.

[RSMAW13] A. Rogge-Solti, R. S. Mans, W. M. van der Aalst, M. Weske: Repairing event logs using stochastic process models, Bd. 78. Universitätsverlag Potsdam (2013).

[ScSo60] B. Schweizer, A. Sklar, : Statistical metric spaces. In: *Pacific J. Math*, 10, 1 (1960), 313–334.

[ShDi96] V. A. Shepelev, S. W. Director: Automatic workflow generation. *In: Proceedings EURO-DAC '96. European Design Automation Conference with EURO-VHDL '96 and Exhibition* (1996), 104–109.

[SiVF07] M. Siddiqui, A. Villazon, T. Fahringer: Semantic-Based On-demand Synthesis of Grid Activities for Automatic Workflow Generation. *In: Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)* (2007), 43–50.

[Slaa20] T. Slaats: Declarative and Hybrid Process Discovery: Recent Advances and Open Challenges. In: *Journal on Data Semantics*, 9, 1 (2020), 3–20, `https://doi.org/10.1007/s13740-020-00112-9`.

[SuDL17] Y. Sun, Y. Du, M. Li: A Repair of Workflow Models Based on Mirroring Matrices. In: *International Journal of Parallel Programming*, 45, 4 (2017), 1001–1020, `https://doi.org/10.1007/s10766-016-0438-1`.

[SyNi01] T. Syrjänen, I. Niemelä: The Smodels System. *In: T. Eiter, W. Faber, M. l. Truszczyński (Hrsg.), Logic Programming and Nonmotonic Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg (2001), 434–438.

[THAAR09] A. H. Ter Hofstede, W. M. Van der Aalst, M. Adams, N. Russell: Modern Business Process Automation: YAWL and its support environment. Springer Science & Business Media (2009).

[TsHo03] D. Tsarkov, I. Horrocks: DL Reasoner vs. First-Order Prover. In: *Description Logics*, 81 (2003), 152–159.

[Tver77] A. Tversky: Features of similarity. In: *Psychological review*, 84, 4 (1977), 327.

[Usch18] M. Uschold: Demystifying OWL for the Enterprise. In: *Synthesis Lectures on Semantic Web: Theory and Technology*, 8, 1 (2018), i–237.

[Wall11] W. Wallis: A Beginner's Guide to Discrete Mathematics. Birkhäuser Basel, 2nd Aufl. (2011).

[WaSi06] T. Wahl, G. Sindre: An analytical evaluation of BPMN using a semiotic quality framework. *In: Advanced Topics in Database Research, Volume 5*, IGI Global (2006), 94–105.

[Wayn18] H. Wayne: Practical TLA+: Planning Driven Development. Apress (2018).

[Wesk12] M. Weske: Business Process Management - Concepts, Languages, Architectures, 2nd Edition. Springer (2012), `https://doi.org/10.1007/978-3-642-28616-2`.

[WSZL13] J. Wang, S. Song, X. Zhu, X. Lin: Efficient Recovery of Missing Events. In: *Proc. VLDB Endow.*, 6, 10 (2013), 841–852, `https://doi.org/10.14778/2536206.2536212`.

[YaBD18] F. Yasmin, F. Bukhsh, P. De Alencar Silva: Process enhancement in process mining: A literature review. In: *CEUR workshop proceedings*, 2270 (2018), 65–72, `http://simpda2018.di.unimi.it/`, 8th International Symposium on Data-driven Process Discovery and Analysis 2018, SIMPDA 2018 ; Conference date: 13-12-2018 Through 14-12-2018.

[ZhDP09] Y. Zhao, J. Dong, T. Peng: Ontology Classification for Semantic-Web-Based Software Engineering. In: *IEEE Transactions on Services Computing*, 2, 4 (2009), 303–317.

[ZuZh94] R. Zurawski, M. Zhou: Petri nets and industrial applications: A tutorial. In: *IEEE Transactions on Industrial Electronics*, 41, 6 (1994), 567–583.