TOOL: Accessible Automated Reasoning for Human Robot Collaboration

Ivan Gavran
Max Planck Institute for Software Systems
Kaiserslautern, Germany
gavran@mpi-sws.org

Ortwin Mailahn
Research Group: Assembly Planning
Zentrum für Mechatronik und Automatisierungstechnik gGmbH
Saarbrücken, Germany
o.mailahn@zema.de

Rainer Müller
Research Group: Assembly Planning
Zentrum für Mechatronik und Automatisierungstechnik gGmbH
Saarbrücken, Germany
rainer.mueller@zema.de

Richard Peifer
Research Group: Assembly Planning
Zentrum für Mechatronik und Automatisierungstechnik gGmbH
Saarbrücken, Germany
r.peifer@zema.de

Damien Zufferey
Max Planck Institute for Software Systems
Kaiserslautern, Germany
zufferey@mpi-sws.org

Abstract

We present an expressive, concise, and extendable domain specific language for planning of assembly systems, such as industrial human robot cooperation. Increased flexibility requirements in manufacturing processes call for more automation at the description and planning stages of manufacturing. Procedural models are good candidates to meet this demand as programs offer a high degree of flexibility and are easily composed.

Furthermore, we aim to make our programs close to declarative specification and integrate automatic reasoning tools to help the users. The constraints come both from specific programs and preexisting knowledge base from the target domain. The case of human robot collaboration is interesting as there is a number of constraints and regulations around this domain. Unfortunately, automated reasoners are often too unpredictable and cannot be used directly by non-experts.

In this paper, we present our domain specific language "Tool Ontology and Optimization Language" (TOOL) and describe how we integrated automated reasoners and planners in a way that makes them accessible to users which have little programming knowledge, but expertise in manufacturing domain and no previous experience with or knowledge about the underlying reasoners. We present encouraging results by applying TOOL to a case study from the automotive and aerospace industry.

ACM ISBN 978-1-4503-6031-9/18/11...$15.00
https://doi.org/10.1145/3276954.3276961

CCS Concepts • **Theory of computation** → *Automated reasoning*; *Description logics*; • **Software and its engineering** → **Domain specific languages**; *Software usability*;

Keywords assembly planning, automated reasoning, cyber-physical systems, domain specific language, human-robot cooperation, industry 4.0, knowledge integration, robotics and automation

ACM Reference Format:
Ivan Gavran, Ortwin Mailahn, Rainer Müller, Richard Peifer, and Damien Zufferey. 2018. TOOL: Accessible Automated Reasoning for Human Robot Collaboration. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18), November 7–8, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages.
https://doi.org/10.1145/3276954.3276961

1 Introduction

Software used to be limited to purely logical tasks but it has become pervasive and now controls many aspects of the systems around us. With the increasing speed at which cyber-physical systems get deployed, software interacts with the real world in a more and more autonomous way.

The waterfall development model for software was adapted from the more established engineering fields including the manufacturing industry [Benington 1983], but it didn't match the reality of most software projects. Therefore, iterative and incremental development processes are now the de facto standard in software development. Increasing digitization, automation, and competitive pressure to reduce the time to market makes the manufacturing sector move toward to iterative and decentralized processes. This is part of a larger trend referred to as "Industry 4.0" [McKinsey & Company

2015]. The challenges arising there are very similar to challenges of software development.

In this paper, we look at some of the decision making in human robot collaboration (HRC) in the context of task assignment for assembly planning. This leads to challenging questions regarding the choice of the best system setup for the lifecycle sequence of the product and commissioning related aspects like the safety of humans when working in cooperation with automated systems. Processes that are dangerous for humans (when a robot welds, for instance) should exclude humans. There are also tasks that particular robots cannot execute due to their limitations. Our final goal is to deliver a correct-by-construction approach to HRC in assembly planning. Instead of relying on humans to know all the regulations and not making any mistake, we rely on automated reasoning tools integrated in a domain specific language (DSL) to make the critical decisions. We present our first steps in that direction. While the idea of using ontologies for assembly planning knowledge isn't new (see [Raza and Harrison 2011], for instance), we realized quickly that current formal languages are hardly usable for non computer scientists. Developing a DSL became an important step to make the underlying formal model more understandable and usable.

We have developed the Tool Ontology and Optimization Language (Tool). The goal of Tool is to provide a lightweight interface which helps non-experts to effectively use automated reasoning and planning tools. In a sense, Tool is a restriction over existing software, as it only exposes tractable features and emulates some more advanced features by paraphrasing them with simpler ones when possible. The programmer using Tool first formalizes products, processes and resources in Tool's DSL. Then, Tool checks that suitable resources are provided to perform all processes using an automated reasoner. Finally, the resources selected by the reasoner are ordered into a schedule. The schedule is optimized according to a linear combination of four objective functions: cost, duration, probability of stable processes, and quality of work. The optimization includes time and cost for scheduled tooling of workers when necessary. While the final stage of Tool is a planning problem that subsumes all the steps which come before, we still decide to decompose the problem in smaller chunks and solve them separately. The overall complexity remains the same but, from a practical perspective, it makes the toolchain more predictable and usable. Therefore, the classification stage at which an automated reasoner picks the resources which can be used during the scheduling is critical. This part processes the largest amount of information and the input elements come directly from the user with little filtering.

Automated reasoners have become staple tools when reasoning about properties of programs and, more widely, any kind of system which has a precise mathematical model. While these tools can give impressive results in the hand of experts, they can be unpredictable for people not familiar with their underlying working principles. For instance, the common understanding of worst case computational complexity is not helpful for algorithms based on proof search. Sometimes, having more constraints, i.e. a larger instance, makes the solver faster (and sometimes not . . .). This has led to a renewed interest for more predictable systems based on tractable logics. For instance, the Ivy systems [Padon et al. 2016] uses "only" the Bernays-Schönfinkel class, also known as EPR, which is NEXPTIME-complete [Lewis 1980]. In this work, we aim for even more tractable logics. The downside of using simple logics is that encoding a problem can become less intuitive as some constraints need to be reduced to more restricted classes.

Instead of using SMT solvers which are pervasive in the programming language and verification community, knowledge representation in Tool is based on Owl 2 [Motik et al. 2012] and the associated reasoners. This choice is motivated by the fact that Owl has been chosen by standardization authorities [IEEE 2015; Schlenoff et al. 2012] to provide a formal basis of robotics. We use the Owl ecosystem to benefit from these efforts.

Tool is being tested on examples of assembly processes from the aerospace and automotive industries at Zentrum für Mechatronik und Automatisierungstechnik gemeinnützige GmbH (ZeMA) within a project aiming to develop more efficient cyber-physical production technologies.

1.1 A (De)Motivating Example

Among this push for formalization and automation of the assembly processes, a key component of our approach are automated reasoners which transform a declarative specification of the problem into a workable solution that meets the specification. The goal of the following experiment was to assess how usable automated reasoners are out-of-the-box for non-experts.

We asked a fellow at ZeMA to formulate a small product structure in the Web Ontology Language (Owl 2) with the help of the Protégé editor [Knublauch et al. 2005]. That person had some programming experience, but is not a computer science professional. The given task was modeling the component hierarchy of an aircraft fuselage with 1 skinner, 4 formers, 5 stringers and 16 clips. It is a small subset of the case study that we will present in Section 4. No assembly processes, workers or skills were to be modeled. He was neither briefed nor restricted in any other way how he should design the ontology. He searched the internet for examples to get an intuition how to deal with Owl 2. The knowledge base that he then designed is an interesting example of tractability problems that arise when users are not aware of the computational expensiveness of some operators and constraints. The result was expressed in a logic that is decidable but quite powerful, the description logic $\mathcal{SROIQ}$ [Horrocks et al. 2006] which is N2ExpTime complete [Kazakov 2008].

What made the example so complicated is the use of cardinality constraints and function inverses. Unsurprisingly, checking if the ontology is consistent with the HermiT reasoner [Glimm et al. 2010] aborted inconclusively after a timeout of 120 minutes. With the help of TOOL the same person was able to encode the same example and the same reasoner was able to perform the classification in less than a second.

1.2 Contributions

We developed TOOL as a lightweight DSL targeted at assembly planning for human-robot collaboration in the manufacturing sector. TOOL’s goal is to make off-the-shelf automated reasoners accessible to non-expert users, only requiring some programming background. Using off-the-shelf tools makes our approach simple to implement and, therefore, easy to adapt to other domains. A key challenge to face is the unpredictable nature of automated reasoners. Therefore, we designed the DSL to be a small layer around a tractable description logic. We present early results for the application of TOOL to examples from the aerospace and automotive industries.

2 Preliminaries

Background on HRC for manufacturing. Due to the increasingly available sensitive robots, which fulfill the criteria and requirements for a secure cooperation with humans, previously not automatable tasks can be handed over to robots. We target the assembly part of manufacturing because it still has a high proportion of manual activities compared to other areas of production, particularly for large component assembly. We are not looking at the operational part (as it is the domain of operators), but at the planning stage, where discrete decisions regarding the operations are made.

Planning roughly goes along the following steps. First, the data required for planning are collected, the tasks are specified, and the assembly system is basically planned. The plan is detailed and worked out in a refinement phase. The planning concludes with the deployment and evaluation.

Assembly planning for large components such as airplane parts takes days, even weeks when done manually. In the rarest cases, planning starts from scratch. With shorter product life-cycles, the uncertainties of the global economy, and the dependencies of supplier networks, the number of necessary planning adjustments is steadily increasing. Furthermore, numerous regulations involved with HRC make this problem hard to manage for humans.

Description Logics (DL). DL are a family of formal logics, which correspond to decidable fragments of First Order Logic with set theory. An ontology, also called knowledge base, is a set of axioms in a DL. Common to all DL is that predicates are at most of arity two and only special forms of quantification may occur. Predicate symbols of arity zero are called *nominals* or *individuals*, those of arity one are called *concepts* and those of arity two are called *roles*. Nominals are interpreted as elements of a nonempty universe $\mathfrak{U}$, concepts as subsets of $\mathfrak{U}$ and roles as subsets of $\mathfrak{U} \times \mathfrak{U}$. Let I denote the function that interprets DL predicate symbols over $\mathfrak{U}$ and let X_I be shorthand for $I(X)$.

Concepts can either be named or they are *concept expressions*, which are composed of at least one named concept and one or more of the following *constructors*: *concept union* ($(A \sqcup B)_I = A_I \cup B_I$), *intersection* ($(A \sqcap B)_I = A_I \cap B_I$) and *complement* ($(\neg A)_I = \neg(A_I)$).

Quantification can only be used via constructors that take a role and a concept as arguments and return a concept. It is therefore called *role restriction*. The *existential restriction* $\exists R.B$, where B denotes a concept and R a role, is defined as $a \in (\exists R.B)_I \Leftrightarrow \exists b \in B_I.(a,b) \in R_I$, and the *value restriction* $\forall R.B$ as $a \in (\forall R.B)_I \Leftrightarrow \forall b \in B_I.(a,b) \in R_I \Rightarrow b \in B_I$.

Local reflexivity ($\exists R.\mathsf{self}$) construct concept from roles by returning the concept which contains all individuals that are related to themselves via the role ,i.e., $a \in (\exists R.\mathsf{self})_I \Leftrightarrow (a,a) \in R_I$.

All DL allow to state axioms about inclusions ($A \sqsubseteq B$) and equivalences of concepts ($A \equiv B$) Their semantics is $A \sqsubseteq B \Leftrightarrow A_I \subseteq B_I$ and $A \equiv B \Leftrightarrow A_I = B_I$. To preserve tractability, concept inclusion is often restricted. Inclusion is also applicable to roles with *role inclusion* of the form $R \sqsubseteq S$ with semantics $R \sqsubseteq S \Leftrightarrow R_I \subseteq S_I$, and *role chain inclusion* (RCI) of the form $R_1 \circ \ldots \circ R_k \sqsubseteq S$ with $k > 1$ where $R \circ R'$ denotes the composition of binary relations.

Some DL also offer a bottom concept $\bot$ with semantics $\bot_I = \emptyset$ and a top concept $\top$ with $\top_I = \mathfrak{U}$. Disjoint concepts A, B can be expressed as $A \sqcap B \equiv \bot$. DL may also offer *concrete datatypes* through roles R whose interpretation involves an a priori defined set $\mathfrak{D}$, e.g. $\mathfrak{D} = \mathbb{Q}$, and obeys the condition that $R_I \subseteq \mathfrak{U} \times \mathfrak{D}$. Such roles are commonly called *data-properties*.

Very few DL allow *concept cross product inclusion* (XPI) axioms $A \times B \sqsubseteq R$ or $R \sqsubseteq A \times B$, where A, B are concepts and R is a role. The concept cross product semantics is $(A \times B)_I = A_I \times B_I$. Cross product inclusions can be simulated if a DL is capable of RCI, local reflexivity and a top role $\top \times \top$.

An interpretation I is called valid for an ontology $\mathcal{K}$ if it respects the given inductive equations for interpretations. Important reasoning tasks are concept satisfiability (given $\mathcal{K}$ and a concept C, find I s.t. C_I is nonempty), concept subsumption (given $\mathcal{K}$ and concepts B, C, does $B \sqsubseteq C$ hold for every valid I) and classification (given $\mathcal{K}$ return the graph of all concept subsumptions $B \sqsubseteq C$).

Polynomial DL Fragments. The DL fragment which we employed is called $\mathcal{SROEL}(\mathcal{D})$. It supports nominals, $\exists$ restrictions, $\bot, \top, \sqcap, \neg$, local reflexivity, GCI, RCI, range restriction axioms and some concrete datatypes. It is PTIME tractable if all RCI $R_1 \circ \ldots \circ R_k \sqsubseteq S$ obey the context condition that $\mathrm{ran}(R_k) \sqsubseteq \ldots \sqsubseteq \mathrm{ran}(S)$ is asserted directly or by an chain of concept inclusions [Krötzsch 2011]. $\mathcal{SROEL}(\mathcal{D})$ is an extension of $\mathcal{EL}$++ [Baader et al. 2008] with local reflexivity. The suffix '$\mathcal{D}$' indicates the support for the concrete datatypes of $\mathcal{EL}$++ [Baader et al. 2005a,b]. The Owl EL standard is based on $\mathcal{SROEL}$ and $\mathcal{EL}$++, but less powerful with respect to concrete datatypes.

The rational numbers are presented through data-properties R with $R_I \subseteq \mathfrak{U} \times \mathbb{Q}$. The logic becomes more expressive when these data-properties can be restricted to intervals of $\mathbb{Q}$. Let $\exists R.J$ denote the restriction to Interval $J \subseteq \mathbb{Q}$ with semantics $a \in (\exists R.J)_I \Leftrightarrow \exists b \in J.(a,b) \in R_I$. If F_I is a function, then $(\exists F.J)_I$ is the preimage of J. In the following, all data-properties are interpreted as functions and $\exists R.J$ is therefore called a *rational function restriction*.

Let $F(C) \equiv J$ be short for $C \equiv \exists F.J$, $F(C) \sqsubseteq J$ for $C \sqsubseteq \exists F.J$ and let $F(C) = q$ denote $\exists F.\{q\} \equiv C$ for $q \in \mathbb{Q}$. Owl EL only offers the restrictions $F(C) \equiv \mathbb{Q}$ and $F(C) = q$ whereas $\mathcal{EL}$++ also offers $F(C) \equiv (-\infty, q)$ which is written as $F(C) < q$. A range restriction of F to an interval $[q, \infty)$ can be expressed by the axiom $\exists F.(-\infty, q) \sqsubseteq \bot$, an additional axiom $\exists F.\{q\} \sqsubseteq \bot$ restricts the range to (q, ∞). Owl 2 DL provides restrictions to arbitrary intervals of $\mathbb{Q}$, but the combination of $F(C) < q$ and $F(C) > q$ restrictions with $\mathcal{EL}$++ makes reasoning ExpTime complete [Baader et al. 2005b]. In Section 3.1.3 we give a weaker semantics to keep the reasoning tractable.

Although Owl 2 has an extension with linear equations [Parsia and Sattler 2012]. Unfortunately, it is not supported by any of the reasoners we tried and, therefore, we cannot rely on it.

3 Tool DSL

The core of Tool is a domain specific language (DSL) that exposes a PTime tractable fragment of Owl and uses a PDDL reasoner for the scheduling of assembly processes. An user of Tool goes through description, classification, and planning:

1. The programmer encodes the problem in the DSL. The DSL is a declarative language in which the programmer describes elements, their types, and how they are connected.
2. Then Tool *classifies* the problem. Workers, Actions, Skills and Assembly Items are translated into an Owl ontology, which is classified by the HermiT reasoner [Glimm et al. 2010]. In this step, all the constraints are checked and the reasoner tries to find a model which connects the skills of available resources to skills required by actions that need to be taken. This step may already fail if there are not enough resources, or if some action requires incompatible resources, e.g. a welding robot along an unprotected human worker. When classification fails, Tool returns a set of unresolved skills. If the classification succeeds, the resulting model can go to planning.
3. Dependence constraints and cost functions are specified at the planning stage. They, and knowledge from the classified ontology, are translated into PDDL and fed into the LPG-*TD* reasoner [Gerevini et al. 2004].

As we will see, the DSL is a thin layer above ontologies and DL. Rather than using abstract DL operators, we reframe them into the corresponding concepts from the target application domain. Also, and more importantly, we only expose tractable constraints. Tool can additionally import existing ontologies, which makes it possible to import knowledge which is not specific to the current instance, e.g., the workers, robots, and tools available.

In the rest of this section, we explain the trade-offs we had to do to preserve tractability. Rather than explaining every operator in the language, we explain how the different *kinds* of operations in the language are encoded into DL. While the terminology is specific to our application domain, the fact that the DSL is a relatively thin layer above the ontology means a similar approach can be applied to other domains.

High-level structure and usage. An simplified abstract syntax of the DSL is shown in Figure 1. We assume that the DSL is embedded in an general purpose programming language and describes the Tool specific elements. The host language provides the control structure, e.g., branch, loop and procedure, and type system. The DSL is articulated among the following elements:

Workers are *Humans* and *Robots* that may have *Tools*, they contribute *Skills*.

Assembly Items are the *Components*, organized in *Assembly Groups*, to assemble.

Skills are capabilities offered by workers and their tools to accomplish some *Actions*.

Actions are the *Operations*, grouped in a *Process* hierarchy, to schedule.

Assessments give cost, duration, probability of success, and quality to tasks and workers

Schedules assign operations to workers, start and end times for operations, and tool changes.

Example 3.1. Below is a Tool formulation of a small example showing the DSL as currently implemented as a Kotlin library. To hide the implementation details assembly items, actions, skills, workers, tools, etc., are created using planning factory:

```
val factory = PlanningFactory(
      "urn:absolute:tool/AircraftFuselage")
val bonding = factory.skill("bonding")
val shredding = factory.skill("shredding")
val gluing = factory.skill(name = "gluing",
      containedInAny = setOf(bonding),
      disjointWithAny = setOf(shredding))
val clip = factory.component("clip")
```

$$\begin{array}{rcl}
\mathit{name} & ::= & \text{a string identifier} \\
\mathit{unit} & ::= & \text{unit of measurement, e.g., meter} \\
\mathit{interval} & ::= & \text{closed, half-open, or open interval of } \mathbb{Q} \\
\mathit{function} & ::= & \text{PlanningFunction}(\mathit{name},\ \mathit{unit},\ \text{domain} = \mathit{name},\ \text{range} = \mathit{interval}) \\
\mathit{constraint} & ::= & (\text{containedIn} \mid \text{disjointFrom})\ \mathit{skill} \\
\mathit{skill} & ::= & \text{Skill}(\mathit{name},\ \mathit{constraint}^*) \\
 & \mid & \mathit{function}\ \mathit{interval}\ \mathit{constraint}^* \\
\mathit{position} & ::= & \text{AssemblyPosition}(\mathit{name},\ \mathit{skill},\ \text{pre} = \mathit{skill}?,\ \text{post} = \mathit{skill}?) \\
\mathit{assemblyItem} & ::= & \text{Component}(\mathit{name},\ \text{weight} \in \mathit{interval}) \\
 & \mid & \text{AssemblyGroup}(\mathit{name},\ (\mathit{position}\ \mathit{assemblyItem})^*) \\
\mathit{actionMatching} & ::= & \text{MultipleProcess}(\mathit{assemblyItem}+,\ \mathit{action}+) \\
\mathit{action} & ::= & \text{Operation}(\mathit{name},\ \mathit{skill}^*,\ \mathit{assemblyItem}^*) \\
 & \mid & \text{Process}(\mathit{name},\ \mathit{action}^*,\ \mathit{assemblyItem}^*) \\
 & \mid & \text{Process}(\mathit{name},\ \mathit{actionMatching},\ (\mathit{position}\ \mathit{action})^*) \\
\mathit{tool} & ::= & \text{Tool}(\mathit{name},\ \mathit{skill}^*) \\
\mathit{worker} & ::= & \text{Human}(\mathit{name},\ \mathit{skill}^*,\ \mathit{tool}^*) \\
 & \mid & \text{Robot}(\mathit{name},\ \mathit{skill}^*,\ \mathit{tool}^*) \\
\mathit{skillMatching} & ::= & \text{Classification}(\mathit{action}+,\ \mathit{worker}+,\ \text{ImportedOntology}(\mathit{name})^*) \\
\mathit{actionCondition} & ::= & \text{DependsOn}(\mathit{action},\ \mathit{action}) \\
\mathit{objective} & ::= & \text{cost} \in \mathbb{Q},\ \text{duration} \in \mathbb{Q},\ \text{capability} \in \{0,\dots,10\},\ \text{quality} \in \{0,\dots,10\} \\
\mathit{assessment} & ::= & \text{Assessment}(\mathit{objective},\ \mathit{action},\ \mathit{worker}?,\ \mathit{tool}?) \\
 & \mid & \text{ActionCostPerHour}((\mathit{worker} \mid \mathit{tool})\ \text{value} \in \mathbb{Q}) \\
\mathit{schedule} & ::= & \text{Plan}(\mathit{skillMatching},\ \mathit{actionCondition}^*,\ \mathit{assessment}^*,\ \mathit{objective})
\end{array}$$

Figure 1. Tool abstract syntax (simplified)

```
val gluing_clip = factory.operation(
        "gluing of a clip",
        treats = setOf(clip),
        requiresSkills = listOf(gluing))
val assembleClips = factory.process(
        name = "assemble clips",
        subActions = setOf(
                gluing_clip totalCopies 16))
val robot1 = factory.robot(
        name = "robot1",
        skills = listOf(gluing))
```

Rather general skills 'bonding' and 'shredding' are defined, followed by the skill 'gluing', which is a sub-skill of bonding. In the ontology, sub-skills are expressed through concept inclusions. A parameter *containedInAny* can be provided while creating a new skill, which is telling that the Owl concept representing the skill will be included in any of the concept-expressions from *containedInAny*. It can also be stressed that gluing is not a type of shredding, by telling that it is disjoint with any shredding.

A component is created (potential parameters such as temperature are left out). An operation gluing of a clip is defined and linked to the clip, which it "treats" (this can be handy if the operation needs data such as the temperature of the component). The following definition of the process assemble clips gets 16 copies of the `gluing_clip` operation. At last, a robot without tools is defined, and the only skill it provides is gluing.

3.1 From the DSL to Constraints

The result of a Tool program is a schedule. However, computing the schedule is computationally hard. Therefore, most of the effort of Tool is to structure the problem description such that we can split the problem in two parts. The *skill matching* phase pre-process the problem and only keeps the relevant elements. Then a smaller planning problem is generated using the skill matching informations. In this subsection, we focus on how the skill matching is performed (element of Figure 1 up to *skillMatching*). We discuss how the main elements of the DSL map to DL constraints focusing on the element where we had to adopt unexpected encodings to preserve tractability.

3.1.1 Hierarchy and Relations

Objects created in Tool extend a category or an existing element from a category. This hierarchy is used to share some constraints and specialize other. For instance, human workers and robot workers are similar. The presence of two separate classes follows from the need of having different axioms applied to them.

In the example above, "`factory.skill("bonding")`" introduces a new *bonding* concept. The factory methods also provide access to operators for constraints between concepts.

The more complicated "`factory.skill(name = "gluing", containedInAny = setOf(bonding), disjointWithAny = setOf(shredding))`" creates a *gluing* concept and adds the following two constraints between concepts:

$$\mathit{gluing} \sqsubseteq \mathit{bonding}$$
$$\mathit{gluing} \sqcap \mathit{shredding} \equiv \bot$$

Next, each top level class comes with specific attributes and relation to other classes. For instance, workers have skills, and a property indicating all tool-types that a worker is able to use. The maximum weight that a worker is capable of handling is also a skill. When a worker uses a tool, he gets the tool's skills on top of his own skills.

From the attributes and relations, we produce Owl constraints and axioms. Binary relations are directly supported by Owl EL and we can directly translate such constraints. For instance, Tool generates relation between skills and workers (`canBeHandledByWorker`) or between skills and tools (`canBeHandledByTool`) and gives them to the reasoner.

The range of `canBeHandledByWorker` are all workers; its domain $\exists\texttt{canBeHandledByWorker}.\top$ is defined as the concept of all skills that can be handled by any worker. The generated ontology gets one XPI axiom $\texttt{skillOf\_}W \times W \sqsubseteq \texttt{canBeHandledByWorker}$ for each worker W to model the overall skills $\texttt{skillOf\_}W$ that the worker possesses. An ordinary concept inclusion $s \sqsubseteq \texttt{skillOf\_}W$ is added to the ontology for any skill s that W possesses. In the example the generated axioms are:

$$\texttt{skillOf\_robot1} \times \texttt{robot1} \sqsubseteq \texttt{canBeHandledByWorker}$$
$$\texttt{gluing} \sqsubseteq \texttt{skillOf\_robot1}$$

Unfortunately, XPI axioms are not directly supported in Owl. Instead, we simulate an inclusion $S \times W \sqsubseteq R$ by turning the concept into roles (*rolification*) and then using role inclusion. The rolification uses two fresh names R_S, R_W which get constrained by the axioms $\exists R_S.\mathrm{self} \equiv S$ and $\exists R_W.\mathrm{self} \equiv W$ and their domain and range are restricted to S or W respectively. Then the inclusion becomes $R_S \circ (\top \times \top) \circ R_W \sqsubseteq R$.

3.1.2 Numerical Constraints and Relaxed Semantics

Numerical constraints over rational numbers are tractable on their own. However, we need to be careful when combining them with other axioms as the unrestricted combination of constraints has a much greater expressive power. Numerical reasoning appears, for instance, with ranges that describe the weights of components to assemble or the reach of tools. Cardinality constraints occur when specifying the number of elements or resources like `gluing_clip` in the example above.

Cardinality constraints. In the example 3.1, Tool does not encode the number of clips as cardinality constraints. Exact cardinalities larger than 1 cannot be specified in Owl EL++. On the other hand, it is possible to have lower bounds. Such lower bounds are asserted by creating enough individuals. Instead of the exact constraints, Tool asserts that there are as many different components of the given type as it was specified, but leaves open if there are more. For our target domain, this relaxation is harmless and keeps the problem tractable.

Rational function restrictions. Tool's knowledge base is Owl 2 which allows data-properties F to have the rational numbers as image, and it's possible to use any rational interval $J \subseteq \mathbb{Q}$ in a restriction $\exists F.J$. Tool employs these Owl 2 DL data-property-restrictions on rational numbers, as Owl 2's PTime tractable fragment Owl EL supports only the restrictions $\exists F.\mathbb{Q}$ and $\exists F.\{q\}$ for $q \in \mathbb{Q}$. $\mathcal{EL}$++ offers a less then operator $<$ which allows to express half open intervals $\exists F.[q,\infty)$ and open intervals $\exists F.(q,\infty)$ for any $q \in \mathbb{Q}$. A simple use case in Tool is given below. A rational function is created through a `planningFunction` factory call, which takes a name and a comment about its intended use as arguments. In the second line a skill is created by restricting the function to the closed rational interval $[0, \frac{100}{36}]$.

```
val f_velocity = factory.planningFunction(
        "velocity", "in m/s")
val s_velocity_0_10kmh = f_velocity.skill(
        C[!0, !100/!36])
```

To create a rational number, we overload the '`!`' operator to use it as a conversion method from integer to Tool's rational number and square brackets can be redefined through appropriate getters.

3.1.3 Weakened Interval Semantics

As shown in the code above, skills often range over intervals. Furthermore, actions can require specific values for a skill or, in some cases, also a range. Therefore, ontologies created by Tool require reasoning about intervals. Tool's DL knowledge bases already includes rational function restrictions ($\exists F.J$) to arbitrary closed, open and half-open intervals over $\mathbb{Q}$. But mixing intervals and rational functions with the standard DL semantics makes it possible to express a concept union $C \equiv A \sqcup B$ with the axioms $C \equiv \exists F.[x,z]$, $A \equiv \exists F.[x,y]$ and $B \equiv \exists F.(y,z]$. This results in a more powerful DL where subsumption is ExpTime hard [Baader et al. 2005b]. Since we want that reasoning over Tool ontologies stays in PTime, we need to weaken the semantics for rational interval restrictions. The weakened semantics can be described as *incomplete reasoning* in $\mathcal{EL}$++$(\mathbb{Q},\mathfrak{F})$, where $\mathfrak{F}$ is the family of rational function restrictions $\exists F.J$. The only deductions which won't be performed are those which correspond to concept unions.

Some definitions are needed to show concept unions are not deduced. Let $\exists p(F_1,\ldots,F_n)$ be the general notation for a restriction of n-ary operators p on concrete datatypes, e.g.

$\exists{+}q(F, G)$ for the binary-operator restriction $\exists F + q = G$. A concrete domain $\mathfrak{D}$ is *p-admissible* if (1) satisfiability and implication in $\mathfrak{D}$ are decidable in polynomial time, and (2) $\mathfrak{D}$ is *convex*: if a conjunction of atoms of the form $\exists p(F_1, \ldots, F_n)$ implies a disjunction of such atoms, then it also implies one of its disjuncts. Conjunction, disjunction and implies can be read as the logical operators $\wedge, \vee, \rightarrow$ and as the DL operators $\sqcap, \sqcup, \sqsubseteq$. Subsumption in $\mathcal{EL}$++ is PTime tractable for any p-admissible concrete domain [Baader et al. 2005a,b].

The standard semantics for interval restrictions is $a \in (\exists R.J)_I \Leftrightarrow \exists b \in J.(a,b) \in R_I$ *and* it is interpreted *conformant* with concept unions $\sqcup$, i.e. under the standard semantics $\exists F.[x,z] \equiv \exists F.[x,y] \sqcup \exists F.(y,z]$ for $x < y < z \in \mathbb{Q}$. Therefore, the concrete domain $(\mathbb{Q}, \leq_q, >_q)$ with restriction schemes $\exists F.(-\infty, q]$ and $\exists F.(q, \infty)$ isn't PTime tractable.

For instance, consider $\exists F.(-\infty, 1] \sqsubseteq \exists F.(-\infty, 0] \sqcup \exists F.(0, \infty)$. This constraint is true but breaks convexity. $\exists F.(-\infty, 1]$ implies the disjunction $\exists F.(-\infty, 0] \sqcup \exists F.(0, \infty)$. However, neither $\exists F.(-\infty, 1] \sqsubseteq \exists F.(-\infty, 0]$ nor $\exists F.(-\infty, 1] \sqsubseteq \exists F.(0, \infty)$ holds.

We can enforce convexity by requiring that no deductions incorporate non-trivial concept unions over restrictions, with $\exists p_i(F_i, G_i)$, $\exists F_i.J_i$, or $\exists F_i + q_i = G_i$ and $J_i \subseteq \mathbb{Q}$, $q_i \in \mathbb{Q}$:

$$\exists p_1(F_1, G_1) \sqsubseteq p_2(F_2, G_2) \sqcup p_3(F_3, G_3) \quad \Leftrightarrow$$
$$p_1(F_1, G_1) \sqsubseteq p_2(F_2, G_2) \ \vee\ p_1(F_1, G_1) \sqsubseteq p_3(F_3, G_3)$$

Then $C \equiv \exists F.[x,z] \wedge A \equiv \exists F.[x,y] \wedge B \equiv \exists F.(y,z]$ doesn't imply that $C \equiv A \sqcup B$. It may be counterintuitive that $\exists F.(-\infty, 1]$ isn't recognized as a subconcept of $\exists F.(-\infty, 0] \sqcup \exists F.(0, \infty)$, but this should not be too surprising in a language which doesn't offer an union operator $\sqcup$.

The extension by local reflexivity $\exists R$.self doesn't affect the PTime tractability of $\mathcal{EL}$++$(\mathbb{Q}, \mathfrak{F})$, as local reflexivity does not apply to concrete datatypes.

3.1.4 Concept Inclusion Lattice for Intervals

HermiT and almost all Owl reasoners do not support the Owl datarange extension for linear equations, Tool therefore doesn't employ arithmetic operators on rational function restrictions. All rational function restrictions can still be encoded as concepts which respect the inclusion lattice for rational intervals without using linear programming.

We observe that there are at most $k(n) = 2n + 4 \cdot \binom{2n}{2}$ many intersections of n given intervals. This follows from (a) interval intersections are themselves intervals; (b) n intervals have at most $2n$ distinct endpoints, so there is at most $2n$ many singleton intervals; (c) there are four different proper interval types, open, closed and half open left / right and each resulting proper interval is described by one of the four types and two of the given $\leq 2n$ endpoints.

The classification of n given interval restrictions is done by adding the at most $k(n)^2$ many subsumption and disjointness axioms to the original knowledge base. Therefore, the subsumption and classification stay PTime tractable w.r.t. the size of the original knowledge base.

3.2 Tool Implementation

We originally planned to deliver Tool as a stand-alone DSL with its own syntax and toolchain. For practical use this requires not just a plain lexer and parser, but rather integration into a rich IDE like Eclipse and support for features like syntax highlighting, code folding, or navigation through the code. We explored using XText [Efftinge 2008], Jetbrains MPS [Dmitriev 2004] and Spoofax [Wachsmuth et al. 2014] as ways to obtain such an IDE integration. However, the time and effort required to develop a full-fledged DSL made us reconsider and opt for a shallow embedding in a general purpose language.

Kotlin embedding. The major reasons for the decision to implement Tool in Kotlin are that Tool needs to integrate with the JVM in the context of the IProGro[2] project at ZeMA. Kotlin is in both directions fully interoperable with Java. Furthermore, given that Kotlin is supported by Google, it was considered to be a viable option for long term development. Embedding in a general purpose language means that we can focus on the domain-specific elements and rely on the host language for the control structure and type system. Kotlin is shorter and more concise than plain Java, allows a limited form of operator overloading which is employed by Tool. All the elements from the DSL are offered as Kotlin types to describe assembly planning problems. We use the factory method design pattern to hide the implementation details of the DSL objects. Each skill, worker and action object contains a representation as Owl concept, and the planning factory also creates an ontology with all axioms that are explicitly or implicitly imposed on these concepts. The concepts that represent skills, workers and actions provide methods to obtain concept intersections and to tell explicitly inclusion, equivalence or disjointness relations. There are also classes that represent Owl roles and data-properties, which can be restricted to any of the concepts. The structure of the DSL also creates constraints, e.g., skills and workers are disjoint and robot is included in worker.

Owl reasoner and PDDL planner. We looked at the Owl Reasoner Evaluation 2015 (ORE 2015) [Parsia et al. 2017]. In addition to HermiT [Glimm et al. 2010] which we initially chose because it reliably provides good performances, we considered the following reasoners. CEL [Baader et al. 2006], ELepHant [Sertkaya 2013], Fact++ [Tsarkov and Horrocks 2006], and ELK [Kazakov et al. 2013] do not support the operators we use. Pellet [Sirin et al. 2007] is slow on our examples due to many cross products. Konclude [Steigmiller et al. 2014], unfortunately, does not support the OWL-API directly.

Since the standardization of plan definition language PDDL [Ghallab et al. 1998], many planners have been developed.

Tool's chosen axiomatization requires the features *:adl*, *:fluents* and *:typing* and the support for durative actions. When choosing the planner — as in the Owl reasoner case — our priority was predictability over raw performance. This is required as our target users are not aware of the inner working of the planner and we cannot expect them to restructure the problem into forms that are easier to solve. The planner of choice was LPG-*TD* [Gerevini et al. 2004]. In comparison to other planners that we tried ([Benton et al. 2012; Edelkamp and Jabbar 2008; Eyerich and Röger 2009; Helmert 2006; Hoffmann 2003; Vidal 2014]), LPG-*TD* stood out because of its stability, high-quality error messages, and ease of use.

User interface and PLM integration. We plan to use Tool in further projects within ZeMA and encourage its adoption outside of ZeMA. However, a survey of assembly planners among ZeMA's industrial partners has shown that code is not widely accepted as planning support. Graphical user interfaces (GUI) are the standard in the planning domain. Therefore, we are in the process of building a GUI (MoPlaTo) on top of Tool which mimic the style of existing product life-cycle management (PLM) systems. However, we already observed the use of capabilities of programming languages that would be hard to express in a GUI. For instance, some precedence constraints were directly extracted from the component structure by writing an auxiliary method that traverses the assembly groups and generate action conditions according to the component hierarchy.

In the first iteration, the plans encoded in the GUI will correspond to straight-line programs. Later, we are considering including in the GUI blocks encode more complex control flow in the style of visual programming languages like Scratch [Marji 2014]. Hopefully, the GUI will have the same capabilities as Tool and but in a form closer to what planners are familiar with. Wider deployment of Tool and MoPlaTo inside ZeMA and industry partners that would allow us to compare both is still future work. More information on the GUI and the integration of Tool in the wider industrial ecosystem can be found in [Müller et al. 2018].

4 Case Studies: Assembly of Large and Medium Size Components

We evaluate Tool on two case studies derived from research projects at ZeMA, which focus on the assembly of an Airbus A350 fuselage shell (large size) and the assembly of an underbody panel in the end of line car assembly (medium size). The difference in size implies different organization of the assembly. The assembly of large components at Airbus is designed as construction site assembly, whereas the assembly of the car underbody plates takes place in flow operation. The big advantage of the large component assembly—that usually enough space is available to work with several actors simultaneously on one component—is also a disadvantage, because the accessibility is often difficult to guarantee. For the flow operation, the vehicle is transported in the assembly section with the aid of a hanger attached to the hall ceiling. As a result, the car hangs above the heads of the workers in all process steps, which improves accessibility, but places particular strain on the cardiovascular system of workers. This circumstance makes it necessary to consider use of robots to help human workers. Tool is used in this context to analyze the product, the required processes and the resources and to evaluate a potential human-robot cooperation. In the following paragraphs we give use those two examples to give an overview of the planning process with Tool and present its results.

Rough planning of assembly tasks with Tool takes place in seven steps:

1. The product analysis, with the goal to transform the product structure into Tool. Here, all product and process-related design specifications are considered by the planner.
2. The first matching is performed by Tool, in which the product structure is transferred to a process structure with the aid of a process database and a comparison with the design-required processes.
3. During the process analysis the planner applies his knowledge and designs a suitable precedence graph.
4. The second matching is performed by Tool, which searches for suitable resources that meet the process requirements.
5. A suitability assessment of the proposed process resource allocation is performed by the planner. Tool gathers the result.
6. A scheduling according to the specifications of the assembly precedence graph is performed by Tool.
7. Optimization according to weighted target criteria in terms of time, costs and process capability is performed by Tool.

Subsequently, a geometric and ergonomic evaluation is recommended, which is currently not part of Tool[1]. Tool does not take care of complex geometric and ergonomic validation. An extension of Tool would be to connect existing software that solve this task. Nevertheless, for some areas of planning, such as the design of dynamic safety aspects, it is necessary that the result of the rough planning is personally assessed and supplemented by the planners. The technical changes and expressions that result from this evaluation can then be subsequently included in the model and evaluated iteratively. The integration of those aspects is part of ongoing research.

The product to be planned in the context of large component assembly is a fuselage shell element of section 13/14 of an Airbus A350 (Figure 2). The fuselage shell under consideration is a quarter of the lower fuselage shell, which is

[1]There exist appropriate computer-aided tools which can simulate schedules and validate the geometric constraints that doesn't occur during the high-level planning.

connected directly behind the cockpit section. Its dimensions comprise a length of 7 meters and a circumference of approximatively 2.5 meters. Three product variants are taken into account during planning which results in 18 assembly groups when nine different component types and its variants to be assembled. Stringer, clips, frames, brackets, tube connectors, plate holders are designed to be glued together. The tube should be threaded into the tube holder and the side plate is screwed into the plate holder. In comparison, the car underbody panel (Figure 3) is an approximately 2 square meter thin panel with low rigidity. It protects the engine and electronics from the underside of the vehicle against moisture and debris.

At the beginning of the planning process, the information is gathered from the perspective of a product designer. Nowadays many companies use PLM (product life-cycle management) systems from which these data are readily accessible. In some cases, a designer already specifies in detail, e.g. which joining process is to be selected with which parameters. However, detailed process design remains a task of the process planner.

The product structures are mapped with the aid of Tool: each component that finds its way into an assembly group is assigned a location and a required process type in addition to product attributes. With this information, a process planner can start designing the assembly processes. The process structure derived from the product structure serves the planner as a basis for this. Tool searches for suitable processes according to the process requirements from a process database and suggests them to the planner. The planner can then modify or accept the proposed processes, and create new processes or operations.

The sketch of the fuselage shell assembly looks like this: (1) stringers and clips are glued to the shell, (2) adhesive is then applied, the component is joined and fixed and the curing process is activated with the help of heat, (3) the frames are then assembled using the same procedure, followed by the add-on parts, (4) the brackets, tube connectors and plate holders of the add-on parts are also glued. Once the attachments are cured and free of defects after the inspection, the tube can be inserted into the connectors and the side plate screwed to the plate holders.

Creating a process or operation requires a description of the skills required by an action for the skill matching and dependencies for the scheduling. As the product structure follows a hierarchy (components, assembly groups), the analysis of processes follows a similar structure (operations, processes):

1. *handling* (feeding, transporting, fixing),
2. *joining* (adhesive bonding, fastening, pressing on),
3. *auxiliary processes* (monitoring of the adhesive bonding, final inspection), and
4. *special processes* (priming).

In the example of the assembly of the underbody plates, the naked car is continuously conveyed through the station. First, the c-clips (item 1.3 in Figure 3) and the expanding nuts (1.4) must be fed and installed. Appropriate screw points are provided for the nuts. Then the underbody plate (1.1) is fed in and threaded into the flap in the front area. It has screw feedthroughs that must afterwards be aligned with the positions of the nuts. Hexagonal bolts (1.2) must then be fed in and used to fix the underbody plate in the aligned position. Now the remaining screws can be inserted and the assembly must be checked visually. The output comprises of the following:

1. *handling* (feeding, transporting, aligning, fixing),
2. *joining* (threading, fastening),
3. *auxiliary processes* (visual inspection).

This is followed by an analysis of the resources The hierarchy starts here at the lowest level with resources, such as actors and tools, which can be aggregated to system groups. These in turn can be used to build system modules that can be integrated into a station. With regard to the system groups, the ZeMA relies on exchangeable modules. In both cases, the planner describes these modules and their skills.

In the aircraft context, it is possible to integrate up to three workers in a station. We model four robots which are physically available at ZeMA. As the station can only contain three workers, the planner must consider both quantity and qualifications. The *priming*, *adhesive application*, *joining*, and *screwing* processes have both manual and automated tools. *Fixing* is a human worker's skill. Only an automated tool is available for *thermographic inspection* and *curing*. On the other hand, *transport* requires a human. For underbody assembly, a maximum of two people or one person and one robot are permitted in the station. Appropriate workers were added and the robots from the existing database were used. An electric overhead conveyor was created for *transporting*, as there is no alternative equipment here. For all of the above processes, manual tools from different manufacturers with correspondingly different configurations have been stored. An automated tool was created exclusively for *screwing*, which can also feeds the screws at the same time.

After all the elements have been modeled, Tool creates an ontology of the planning data and a reasoner is used to evaluate the skill-matching. The task assignment takes place on the lowest reasonable level. In most cases these are the tasks of the operations. For instance, the assembly process *adhesive application* contains six operations: fixing of the base component, surface activation, application of adhesive, joining, fixing of the applied component, and curing. Exploring the operation further, we find the tasks: attach tool, move to target position, apply adhesive, move to next target position or move to tool holder, and detach tool. Each task has some properties. These can be the attributes of the task itself or requested methods. Aggregated with component

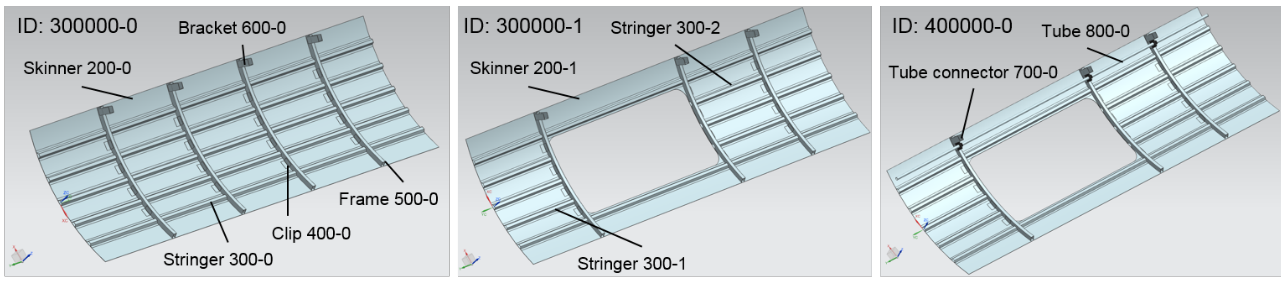

Figure 2. Variations of the left upper shell of an Airbus A350

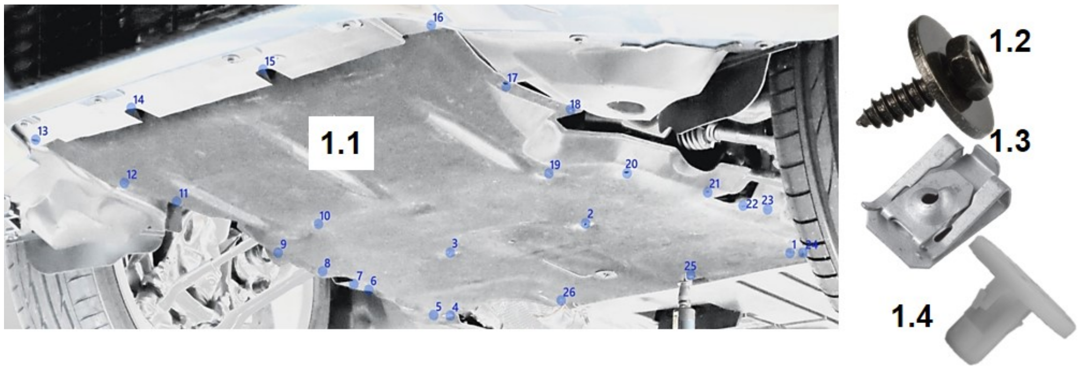

Figure 3. Car underbody panel assembly

properties, these task properties form the requirements. In many cases, it makes sense to group the tasks and assign them to one resource, e.g., if an actor lifts an object we can reasonably assume it will transport it afterwards. In Tool these activities (operations, processes) are flagged as *atomic processes*. Taking all the possible options into account, the automated reasoner is much faster than the manual effort of a planner. Nevertheless, since ontologies handle only basic geometric constraints and many safety-restrictions become relevant within the overall context, a suitability assessment is done by the planner. This assessment filters the infeasible resource combinations. In this example all possible combinations were feasible as well, so that the scheduling and optimization was able to select from all entities.

Without the support of Tool, the planner would pick the most complex product structure and plan a system manually, taking the other variants into account based on personal experience. With Tool, the planner must assign process times and, in a simplified form, costs to the permitted process resource combinations, after which the PDDL solver search a schedule. In most cases time is not the exclusive optimization target. Planners may have to meet cost, probability of stable processes and work quality targets among others. Thus, the assembly planning procedure becomes a multidimensional problem. At this stage of development, Tool can optimize time and cost on a cardinal scale and probability of stable processes and work quality on an ordinal scale. Thus, the optimization has four targets to meet. The ordinal scales must be translated into a cardinal one for optimization, and the weights may be changed in order to observe their impact. The PDDL-solver then searches for an assignment of operations to resources which respects the precedence relations of processes, operations and tasks. It does automatically schedule *tool-up* operations or *strip-down* operations when a worker needs no use a different tool to perform an action. The schedule is optimized according to a linear combination of the four objective functions.

In Table 1, we present an evaluation of the effort and result of the case study. Concerning the user's effort, the wide majority of the work has been done by a planning expert without prior experience in using automated reasoner. He had to learn the basic of Kotlin programming but did not had any extra knowledge about the underlying reasoner and planner. Most of the work was dedicated to studying the example and formalizing it. The overall process of studying, analyzing, and decomposing the planning problem can take from days to weeks depending on the complexity of the example. This effort is required whether Tool is used or not. For our case study, the formalization in Tool took one day for the underbody and two days for the fuselage. The reward for this extra effort is that Tool then helps in finding a solution. For a new project, we estimate that about two third of the development time is for the initial formalization and the remaining 3rd dedicated to solving, adapting, and optimizing the model to get better results. For replanning, the formalization effort will be drastically reduced. As we build libraries of components, skills, and processes, we expect to improve in that respect. The case study resulted in 169 lines of code for the underbody (excluding comments) and 501 lines of code for the fuselage. Most of this code is directly related to the formalization of the example and there is around 40 lines of boilerplate code to put everything together, handle the solving, and store the solution. The underbody classification problem had more that 700 axioms and 2500 for the fuselage. HermiT was able to handle them in less than 20 minutes. This stands in sharp contrast with our example from Section 1.1. That example was a very small subset of this case study. Then for the scheduling part, the scheduling problem for the underbody had around 250 actions to choose from and more than 5000 actions for the fuselage. The planner was able to find a schedule in about 15 minutes.

5 Related Work

The idea that *less is more* has gained momentum in programming languages and verification. The aim is to provide more predictable tools at the cost of removing features. This approach was taken by Ivy [Padon et al. 2016], which is limited to EPR, and Liquid Haskell [Vazou 2016], which only automates the reasoning for decidable, quantifier-free first-order theories. Tool subscribes to similar philosophy. By restricting the expressiveness of the language, we gain predictability and ease of use which more often than not is a crucial component especially for non-expert users. A similar trend is visible in different contexts of automation, aiming to achieve successful end-user programming. [Alexandrova et al. 2015] creates a flow-based language for programming general-purpose robots. [Thomason et al. 2015] builds a natural-language dialog system to communicate user's intent to a robot. Finally, [Finucane et al. 2010] shows how to make existing formal languages more usable by using structured English instead of temporal logic in robotics applications.

Planning is a classical problem in AI and robotics [Choset et al. 2005; Russell and Norvig 2009]. The planning algorithms and tools such as [Helmert 2006; Hoffmann and Nebel 2001] have already been used in the context of robotic and multi-robot systems [Desai et al. 2017; Gavran et al. 2017; Lin and Mitra 2015]. They start from declarative specification and synthesize a detailed (multi) robot plan. While dealing with multi-robot systems they — unlike Tool— only observe identical robots, with equal capabilities. While a lot of work has been done in (reactive) planning for temporal goals [DeCastro et al. 2015; Kress-Gazit et al. 2009; Wongpiromsarn et al. 2012], it is assumed that the assignment is provided to the planning algorithm beforehand. Tool automates this step but does not support reactive planning — it assumes a controlled environment such as a factory.

Planning of assembly systems is an active research field, where a computer aid can be used in different stages of the process thanks to the formalizations of planning knowledge [Rudolf 2006; Weidner 2014]. The deployment of ever more connected and reconfigurable production systems significantly increases the complexity of the tasks given to (human) workers in charge of planning. [Jonas 2000] demonstrates how planning in assembly system can be aided by computers. [Glawe et al. 2015] shows how AutomationML [Schmidt and Lüder 2015] can be connected to OWL reasoning in an application-independent fashion. The importance of using knowledge bases in industry has been recognized in recent years [Huang et al. 2015; Raza and Harrison 2011]. Tool, with its domain-specific language, enables engineers to take the full advantage of it. Recently, it has be shown how basic geometric information can be included into ontologies [Qiao et al. 2018]. We omit this part as Tool resides on a higher level of abstraction, performing skill-based matching. In the area of human-robot collaboration, [Beumelburg 2005] focuses on determining whether a human or a robot would be more suitable for an operation at hand. Tool considers human-robot cooperation at the level of the whole assembly rather than single operations. This property is crucial for encoding safety requirements of the cooperation.

6 Conclusion

We developed Tool, a DSL targeted at assembly planning for human-robot collaboration in the manufacturing sector and we present promising early results that show it can be applied to real examples. A key challenge one faces when using description logic is the unpredictable nature of automated reasoners. This is not avoidable when the reasoner has to deal with high-complexity logics. Therefore, the logics underlying Tool is limited to a polynomial time tractable fragment of description logic. Because the DSL is implemented as a shallow embedding and we use off-the-self tools, we expect that the approach can easily be adapted to other domains. We plan to continue to explore the embedding of low complexity logics in shallow DSLs as a bridge between declarative programming and imperative programming.

As part of this project, we realized that many of the challenges faced by "traditional" industries are very similar to the challenges of software development. This is due to the integration of automated and decentralized robotic systems which allow a degree of flexibility that used to be only seen in software system. That offers an opportunity to reuse and rethink the established solutions from the field of software development in order to improve industrial processes better.

Acknowledgments

Rainer Müller, Richard Peifer, and Ortwin Mailahn are supported in part by the research project IProGro[2], which is funded by the European Union (EU). The project is part

Table 1. Evaluation of Tool on the fuselage and underbody panel case studies

Metric	Underbody	Fuselage	
Implementation effort	1	2	man-days
Tool input size	169	501	#lines of code
Classification time	0.3	18.7	minutes
Planning time	0.1	14.8	minutes
Schedule size	113	170	#grounded temporal actions

of initiative "Investment in growth and jobs", which takes place within the structural fund for development (EFRE). It is supervised by the state of Saarland in Germany. Ivan Gavran and Damien Zufferey are supported in part by the European Research Council under the grant number 610150 (http://www.impact-erc.eu/), ERC Synergy Grant ImPACT.

References

Sonya Alexandrova, Zachary Tatlock, and Maya Cakmak. 2015. RoboFlow: A Flow-Based Visual Programming Language for Mobile Manipulation Tasks. In *ICRA*. https://doi.org/10.1109/ICRA.2015.7139973

Franz Baader, Sebastian Brandt, and Carsten Lutz. 2005a. Pushing the $\mathcal{EL}$ Envelope. In *IJCAI*, Vol. 5. 364–369. https://lat.inf.tu-dresden.de/research/papers/2005/BaaderBrandtLutz-IJCAI-05.pdf

Franz Baader, Sebastian Brandt, and Carsten Lutz. 2005b. *Pushing the $\mathcal{EL}$ Envelope*. LTCS-Report LTCS-05-01. Chair for Automata Theory, Institute for Theoretical Computer Science, Dresden University of Technology, Germany. See http://lat.inf.tu-dresden.de/research/reports.html.

Franz Baader, Sebastian Brandt, and Carsten Lutz. 2008. Pushing the $\mathcal{EL}$ Envelope Further. In *Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions*, Kendall Clark and Peter F. Patel-Schneider (Eds.). http://lat.inf.tu-dresden.de/~clu/papers/archive/owled08dc.pdf

Franz Baader, Carsten Lutz, and Boontawee Suntisrivaraporn. 2006. *CEL — A Polynomial-Time Reasoner for Life Science Ontologies*. Springer Berlin Heidelberg, Berlin, Heidelberg, 287–291. https://doi.org/10.1007/11814771_25

H. D. Benington. 1983. Production of Large Computer Programs. *Annals of the History of Computing* 5, 4 (Oct 1983), 350–361. https://doi.org/10.1109/MAHC.1983.10102

J Benton, Amanda Jane Coles, and Andrew Coles. 2012. Temporal Planning with Preferences and Time-Dependent Continuous Costs.. In *ICAPS*, Vol. 77. 78.

Katharina Beumelburg. 2005. Fähigkeitsorientierte Montageablaufplanung in der direkten Mensch-Roboter-Kooperation. (2005).

H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. 2005. *Principles of Robot Motion*. A Bradford Book.

J. A. DeCastro, J. Alonso-Mora, V. Raman, D. Rus, and H. Kress-Gazit. 2015. Collision-Free Reactive Mission and Motion Planning for Multi-Robot Systems. In *ISRR*.

A. Desai, I. Saha, J. Yang, S. Qadeer, and S. A. Seshia. 2017. DRONA: A Framework for Safe Distributed Mobile Robotics. In *ICCPS*. 239–248.

Sergey Dmitriev. 2004. Language Oriented Programming: The Next Programming Paradigm. (2004). http://resources.jetbrains.com/storage/products/mps/docs/Language_Oriented_Programming.pdf

Stefan Edelkamp and Shahid Jabbar. 2008. MIPS-XXL: Featuring external shortest path search for sequential optimal plans and external branch-and-bound for optimal net benefit. *6th. Int. Planning Competition Booklet (ICAPS-08)* (2008).

Sven Efftinge. 2008. Xtext - Language Engineering Made Easy! (2008). http://www.eclipse.org/Xtext/

Patrick Eyerich and Robert Mattmüller Gabriele Röger. 2009. Temporal Fast Downward. (2009).

C. Finucane, Gangyuan Jing, and H. Kress-Gazit. 2010. LTLMoP: Experimenting with language, Temporal Logic and robot control. In *IROS*. 1988–1993.

Ivan Gavran, Rupak Majumdar, and Indranil Saha. 2017. Antlab: A Multi-Robot Task Server. *ACM Trans. Embedded Comput. Syst.* 16, 5 (2017), 190:1–190:19. https://doi.org/10.1145/3126513

Alfonso Gerevini, Alessandro Saetti, Ivan Serina, and Paolo Toninelli. 2004. LPG-TD: a fully automated planner for PDDL2.2 domains. In *In Proc. of the 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04) International Planning Competition abstracts*.

M. Ghallab, C. Aeronautiques, C. K. Isi, and D. Wilkins. 1998. *PDDL: The Planning Domain Definition Language*. Technical Report CVC TR98003/DCS TR1165. Yale Center for Computational Vision and Control.

Matthias Glawe, Christopher Tebbe, Alexander Fay, and Karl-Heinz Niemann. 2015. Knowledge-based Engineering of Automation Systems Using Ontologies and Engineering Data. In *Proceedings of the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2015)*. SCITEPRESS - Science and Technology Publications, Lda, Portugal, 291–300. https://doi.org/10.5220/0005614502910300

Birte Glimm, Ian Horrocks, and Boris Motik. 2010. Optimized Description Logic Reasoning via Core Blocking. In *Proc. of the 5th Int. Joint Conf. on Automated Reasoning (IJCAR 2010) (LNCS)*, Jürgen Giesl and Reiner Hähnle (Eds.), Vol. 6173. Springer, Edinburgh, UK, 457–471.

Malte Helmert. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26 (2006), 191–246.

Jörg Hoffmann. 2003. The Metric-FF Planning System: Translating"Ignoring Delete Lists"to Numeric State Variables. *Journal of Artificial Intelligence Research* 20 (2003), 291–341.

J. Hoffmann and B. Nebel. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR* 14 (2001), 253–302.

Ian Horrocks, Oliver Kutz, and Ulrike Sattler. 2006. The Even More Irresistible SROIQ. In *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, Patrick Doherty, John Mylopoulos, and Christopher A. Welty (Eds.). AAAI Press, 57–67. http://www.aaai.org/Library/KR/2006/kr06-009.php

Zhicheng Huang, Lihong Qiao, Nabil Anwer, and Yihua Mo. 2015. Ontology Model for Assembly Process Planning Knowledge. In *Proceedings of the 21st International Conference on Industrial Engineering and Engineering Management 2014*, Ershi Qi, Jiang Shen, and Runliang Dou (Eds.). Atlantis Press, Paris, 419–423.

IEEE. 2015. Standard Ontologies for Robotics and Automation – IEEE Std 1872-2015. (2015).

Christian Jonas. 2000. *Konzept einer durchgängigen, rechnergestützten Planung von Montageanlagen*. Vol. 145. Herbert Utz Verlag.

Yevgeny Kazakov. 2008. SRIQ and SROIQ are Harder than SHOIQ. In *Proceedings of the 21st International Workshop on Description Logics*

(DL2008), Dresden, Germany, May 13-16, 2008 (CEUR Workshop Proceedings), Franz Baader, Carsten Lutz, and Boris Motik (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-353/Kazakov.pdf

Yevgeny Kazakov, Markus Krötzsch, and František Simančík. 2013. The Incredible ELK: From Polynomial Procedures to Efficient Reasoning with $\mathcal{EL}$ Ontologies. *Journal of Automated Reasoning* 53 (2013), 1–61. Issue 1. https://doi.org/10.1007/s10817-013-9296-3

Holger Knublauch, Matthew Horridge, Mark A. Musen, Alan L. Rector, Robert Stevens, Nick Drummond, Phillip W. Lord, Natalya Fridman Noy, Julian Seidenberg, and Hai Wang. 2005. The Protege OWL Experience. In *Proceedings of the OWLED*05 Workshop on OWL: Experiences and Directions, Galway, Ireland, November 11-12, 2005 (CEUR Workshop Proceedings)*, Bernardo Cuenca Grau, Ian Horrocks, Bijan Parsia, and Peter F. Patel-Schneider (Eds.), Vol. 188. CEUR-WS.org. http://ceur-ws.org/Vol-188/sub14.pdf

H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. 2009. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics* (2009).

Markus Krötzsch. 2011. Efficient Rule-Based Inferencing for OWL EL. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, Toby Walsh (Ed.). AAAI Press/IJCAI. 2668–2673.

Harry R. Lewis. 1980. Complexity results for classes of quantificational formulas. *J. Comput. System Sci.* 21, 3 (1980), 317 – 353. https://doi.org/10.1016/0022-0000(80)90027-6

Y. Lin and S. Mitra. 2015. StarL: Towards a Unified Framework for Programming, Simulating and Verifying Distributed Robotic Systems. In *LCTES*.

Majed Marji. 2014. *Learn to Program with Scratch.* No Starch Press.

McKinsey & Company. 2015. Industry 4.0: How to navigate digitization of the manufacturing sector. (2015).

Boris Motik, Peter F. Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, and Michael Smith. 2012. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition).* Technical Report. W3C.

Rainer Müller, Richard Peifer, and Ortwin Mailahn. 2018. Objectification of Assembly Planning for the Implementation of Human-Robot Cooperation. In *Advances in Artificial Intelligence, Software and Systems Engineering*, Tareq Z. Ahram (Ed.). Springer International Publishing, Cham, 24–34.

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 614–630. https://doi.org/10.1145/2908080.2908118

Bijan Parsia, Nicolas Matentzoglu, Rafael S. Gonçalves, Birte Glimm, and Andreas Steigmiller. 2017. The OWL Reasoner Evaluation (ORE) 2015 Competition Report. *Journal of Automated Reasoning* (21 Feb 2017). https://doi.org/10.1007/s10817-017-9406-8

Bijan Parsia and Uli Sattler. 2012. OWL 2 Web Ontology Language Data Range Extension: Linear Equations. (2012). https://www.w3.org/TR/owl2-dr-linear/

Lihong Qiao, Yifan Qie, Zuowei Zhu, Yixin Zhu, Uzair Khaleeq uz Zaman, and Nabil Anwer. 2018. An ontology-based modelling and reasoning framework for assembly sequence planning. *The International Journal of Advanced Manufacturing Technology* 94, 9 (01 Feb 2018), 4187–4197. https://doi.org/10.1007/s00170-017-1077-4

Muhammad Baqar Raza and Robert Harrison. 2011. Design, development & implementation of ontological knowledge based system for automotive assembly lines. *International Journal of Data Mining & Knowledge Management Process* 1, 5 (2011), 21–40.

Henning Rudolf. 2006. *Wissensbasierte Montageplanung in der digitalen Fabrik am Beispiel der Automobilindustrie.* Utz.

S. Russell and P. Norvig. 2009. *Artificial Intelligence: A Modern Approach.* Pearson.

Craig Schlenoff, Edson Prestes, Raj Madhavan, Paulo J. S. Gonçalves, Howard Li, Stephen Balakirsky, Thomas R. Kramer, and Emilio Miguelanez. 2012. An IEEE standard Ontology for Robotics and Automation. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, October 7-12, 2012.* IEEE, 1337–1342. https://doi.org/10.1109/IROS.2012.6385518

N Schmidt and A Lüder. 2015. AutomationML in a Nutshell Automation ML e. V. *Office* (2015).

Baris Sertkaya. 2013. The ELepHant Reasoner System Description. In *CEUR Workshop Proceedings*, Vol. 1015.

Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. 2007. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web* 5, 2 (2007), 51 – 53. https://doi.org/10.1016/j.websem.2007.03.004 Software Engineering and the Semantic Web.

Andreas Steigmiller, Thorsten Liebig, and Birte Glimm. 2014. Konclude: System Description. *Web Semantics: Science, Services and Agents on the World Wide Web* 27 (2014), 78–85.

Jesse Thomason, Shiqi Zhang, Raymond J. Mooney, and Peter Stone. 2015. Learning to Interpret Natural Language Commands through Human-Robot Dialog. In *IJCAI.* http://ijcai.org/Abstract/15/273

Dmitry Tsarkov and Ian Horrocks. 2006. *FaCT++ Description Logic Reasoner: System Description.* Springer Berlin Heidelberg, Berlin, Heidelberg, 292–297. https://doi.org/10.1007/11814771_26

Niki Vazou. 2016. *Liquid Haskell: Haskell as a theorem prover.* University of California, San Diego.

Vincent Vidal. 2014. YAHSP3 and YAHSP3-MT in the 8th international planning competition. *Proceedings of the 8th International Planning Competition (IPC-2014)* (2014), 64–65.

Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Software* 31, 5 (2014), 35–43. https://doi.org/10.1109/MS.2014.100

R Weidner. 2014. Wissensbasierte Planung und Beurteilung von Montagesystemen in der Luftfahrtindustrie, vol. 32. *von Berichte aus dem Institut für Konstruktions-und Fertigungstechnik, Shaker, Aachen* 18 (2014).

T. Wongpiromsarn, U. Topcu, and R. M. Murray. 2012. Receding Horizon Temporal Logic Planning. *IEEE Trans. Automat. Contr.* (2012).