2023 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)
June 28-30, 2023. Seattle, Washington, USA

Generating Synthetic Data Using a Knowledge-based Framework for Autonomous Productions

Oliver Petrovic[1], David Leander Dias Duarte[1], and Werner Herfs[1]

***Abstract*— Instead of taking images with a camera, synthetic data is generated in a computer simulation. One advantage of this is that training data can be generated on-demand, e.g., to automatically retrain robots when a task changes. While this makes synthetic data a promising approach for autonomous productions, realizing such autonomous setups is difficult with current systems for generating synthetic data, which usually require a programmer for every dataset to be generated.**

To overcome this problem, we present a novel framework for generating synthetic data. This framework restructures the generation process into asynchronous phases to increase the level of autonomy in two ways. First, by letting programmers write parameterized scripts, many different datasets can be autonomously generated. Secondly, by introducing a user interface, domain experts are enabled to influence the generation process on their own without a programmer. Furthermore, by being built as a new layer on top of existing systems for generating synthetic data, our framework shows a new way to maximize compatibility with other research on synthetic data generation.

To test our framework, we have developed a fully functional prototype based on it. Successfully using this prototype for an example experiment, we conclude that our ideas work. Future research can use our prototype for more elaborate experiments on autonomous productions and to further assess its usability.

***Index Terms*— Machine Learning, Computer Vision, Expert Knowledge, User Interfaces, Ontologies, Industry 4.0**

I. Introduction

Modern computer vision techniques offer many advantages, such as performing better or enabling functionalities that would not be possible otherwise [1], [2]. While deep learning-based techniques often achieve the best results, they require big datasets, which can be prohibitively expensive to create [3]–[6]. The reason for this expensiveness is the usually very manual process in which datasets are created [1]–[3], [5], [7], [8], which in computer vision involves taking images with a camera and then labeling them by hand, e.g., marking the positions of objects on all images. As this can take several minutes or more per image, it quickly becomes very costly for big datasets and thus puts powerful deep learning techniques outside the scope of many projects.

Synthetic data is a promising approach to solve this problem. With synthetic data, instead of taking images with a camera, both the images and their labels are generated in a computer simulation. Thus, human work is reduced to modeling a task in this simulation. After that, new images can be rendered in a fraction of the time of creating real images, allowing also big datasets to be cost-efficiently generated. [3], [8]

Synthetic data is especially attractive in the context of production. For instance, 3D models that can be used to model the computer simulation often already exist in production in the form of CAD models, leading to a lower barrier to generate synthetic data [9], [10]. Moreover, synthetic data can be used to enable autonomous production processes. For instance, new synthetic datasets can be automatically generated and used to retrain machine learning (ML) models when a task changes. That way, for example, robots can learn to handle new parts or to cope with changed environments without any human intervention. [7], [10]

Despite this potential, implementing autonomous production processes is difficult with current systems for generating synthetic data. On the one hand, such systems are designed for a workflow in which a programmer writes a script for every dataset to be generated. This workflow stands in the way of autonomy as it heavily relies on manual human work. On the other hand, the scripts that the programmer writes usually include hard-coded process knowledge. Thus, when a process expert notices that such knowledge has changed, they have to ask the programmer to update the script instead of being able to input the change themselves. As a result, two humans' attention is necessary in such situations, further shifting such systems away from autonomy.

In this paper, we present a novel approach for generating synthetic data specifically targeted to autonomous productions in two ways. First, by letting programmers write reusable, parameterized scripts, many different datasets can be autonomously generated without further human input. Secondly, our approach introduces a user interface to enable process experts to update knowledge themselves when it changes, reducing the minimum number of humans involved in these situations from two to one.

To test the applicability of our approach, we have developed a fully functional prototype based on it. Using our prototype as part of a computer vision experiment, we demonstrate that it can be successfully used to generate synthetic data. To enable further research and developments in the field of autonomous and knowledge-based synthetic data generation, we have published the entirety of our prototype's source code online for others to use or extend.

II. State of the Art

A. Synthetic Data

While there are many ways to generate image data synthetically, in this paper, we're specifically focused on methods rendering it in a simulation. Advantages of such methods are that they can be executed autonomously and don't require any real images. With them, human work is reduced to modeling the simulation. After that, a computer can render an arbitrary number of images of that kind without further human input. [3]

[1]Laboratory for Machine Tools WZL, RWTH Aachen University, Aachen, Germany. Email: o.petrovic@wzl.rwth-aachen.de, david.dias.duarte@rwth-aachen.de, w.herfs@wzl.rwth-aachen.de

978-1-6654-7633-1/23/$31.00 ©2023 IEEE

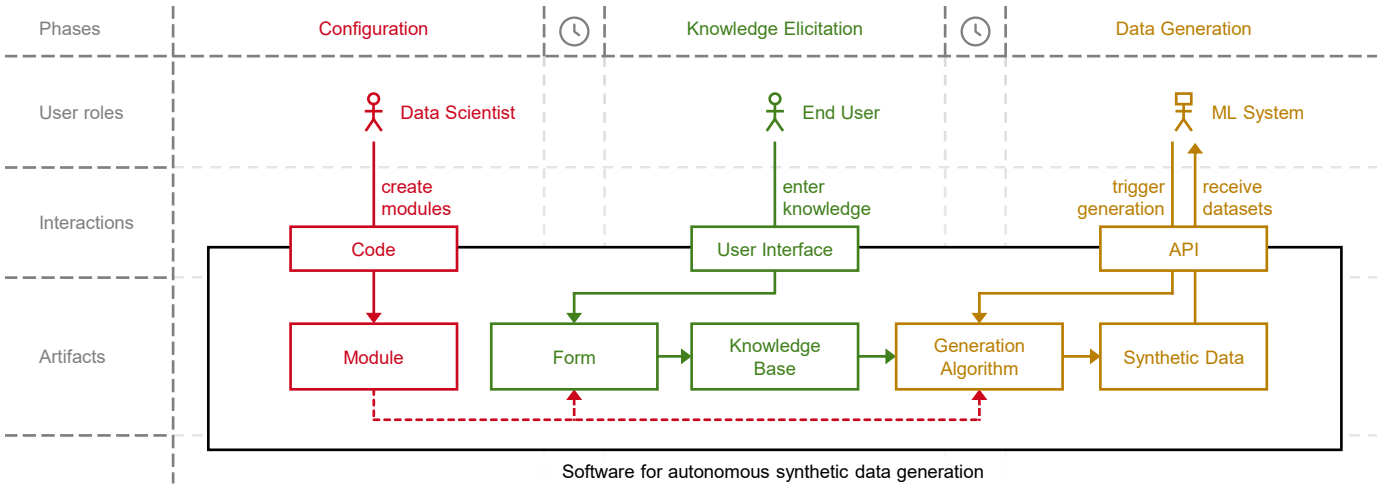

Fig. 1. Overview of our concept for autonomously generating synthetic data. Each user role creates artifacts for the succeeding phases. Because artifacts can be reused, the level of autonomy is increased: the End User can create many knowledge bases without further assistance from the Data Scientist, and the ML System can create many datasets without further human input. The colors indicate in which phases the artifacts are most relevant: While Forms and Generation Algorithms are created during the Configuration phase, they're used during the later phases.

Even when generated in a simulation, the purpose of synthetic datasets usually is to train ML models on them and then to deploy these models in real settings. This domain transfer can be a problem because ML models often overfit on peculiarities of the simulation and then perform badly when given real images [11]. To overcome this so-called sim2real gap, several approaches can be utilized. For instance, the simulation can be heavily randomized so that ML models become more robust to changes [9], [12], or the simulation's level of realism can be increased, e.g., by employing a more realistic rendering method [9], [13], [14]. Another approach to increase the level of realism that is especially relevant in the context of this paper is using knowledge from domain experts to model more detailed simulations [4], [7], [9], [15].

B. Synthetic Data Generators

Synthetic data generators (SDGs) are software tools that simplify generating synthetic data by providing functionalities that many synthetic data projects need [3, pp. 170-171]. For example, they may already implement methods for rendering common label types or support advanced realistic rendering methods like path tracing.

Many SDGs offer similar functionalities. For instance, BlenderProc [16] is a typical example of a SDG: With BlenderProc, images can be generated via a Python API, and it supports the rendering of images with path tracing. It also includes the typical problems of many SDGs, such as limiting the potential for autonomy by requiring a programmer to write a script for every kind of dataset to be generated and mixing algorithms and knowledge in these scripts. Despite these shortcomings, existing SDGs also play a big role in the software framework presented in this paper. This is because our framework is built as a new layer on top of existing SDGs, and, in particular, we use BlenderProc for this in our prototype.

While many SDGs only offer a script-based workflow, one SDG with similar aims to ours is Minervas [17]. Minervas also includes a Python API, but in addition to this, it also has a user interface to enable experts to generate synthetic data themselves without a programmer. Furthermore, it is described as more high-level than existing SDGs such as BlenderProc. Despite these similarities, Minervas is more limited than our approach in several ways. First, it can only generate synthetic data for interior designs, whereas our approach is extendible to any domain. Secondly, Minervas' user interface only offers a subset of the features of its API and consists of only one predefined form that cannot be changed. In contrast to this, any kind of user interface with any kind of rendering functionality can be implemented with our system. Thirdly, despite offering high-level functionalities, Minervas also implements low-level functionalities, such as rendering, itself instead of reusing existing low-level SDGs for this. Lastly, Minervas also doesn't provide any mechanisms for autonomous generation.

C. Knowledge-based Generation of Synthetic Data

Knowledge plays an important role in many synthetic data projects. It can define the basic task to be solved, e.g., which objects are to be recognized, and incorporating more detailed expert knowledge increases the probability of successful domain transfers [7], [9]. In this context, in [18], the topic of knowledge-based synthetic data generation is examined. First, user roles relevant to such systems are formalized. Thus, for example, programmers are formalized as a *Data Scientist* role. Secondly, six implications for knowledge-based synthetic data generation are derived, such as knowledge being also implicitly added by the Data Scientist role. The framework presented by us in this paper builds upon the work from [18], e.g., we try to incorporate the six derived implications in our prototype and we use the same user roles in our concept.

III. Conceptual Overview

To enable autonomous production processes, our approach restructures synthetic data generation into separate phases and introduces two new entry points besides the script from the programmer. These entry points are a form-based user

interface, which allows domain experts to enter knowledge about a task without help from the programmer, and an API, with which synthetic data generation can be automatically triggered. While Fig. 1 shows a first overview of our concept, the following subsections explain further characteristics of this concept and how they're integrated to achieve more autonomy.

A. User Roles

Following [18], our approach knows three user roles: *Data Scientist*, *End User*, and *ML System*. The *Data Scientist* role programs the algorithms for generating synthetic data and designs the form-based user interfaces. The *End User* role is a domain expert who uses the user interfaces created by the Data Scientist to enter knowledge about specific tasks. Finally, the *ML System* is an external software system that uses our system's API to trigger synthetic data generation. When triggering the generation, the ML System can attach parameters, e.g., the 3D model to be recognized, so that many different datasets can be autonomously generated without further human intervention.

With these user roles, the degree of autonomy reached depends on how much a production task changes over time. If only parameters change that the ML System can send via the API, datasets can be generated fully autonomously. If aspects of the task change that can be set via the user interface, then the End User can input the changes themself. Only when a completely new functionality is needed the Data Scientist becomes necessary and has to adapt their scripts.

B. Phases

Our approach splits the generation process into three successive phases. In each phase, another of the three user roles is active. First, the *Configuration* phase is where the Data Scientist is active, creating new algorithms or user interfaces. Following that, the *Knowledge Elicitation* phase is where the End User enters knowledge into the forms. Lastly, in the *Data Generation* phase, the ML System triggers the generation of synthetic datasets.

Each phase creates artifacts that the following phases build upon. For instance, the result of the Configuration phase is a module defining both a form for the Knowledge Elicitation phase and a generation algorithm for the Data Generation phase. The result of the Knowledge Elicitation phase is a knowledge base, which the Data Generation phase also uses. Finally, the result of the Data Generation phase is the finished synthetic dataset, which is then transferred to the ML System.

One important aspect is that the artifacts can be reused. Thus, the End User can create many knowledge bases based on one module, and the ML System can generate many datasets based on one knowledge base. Furthermore, the phases also don't have to be executed contiguously, but any arbitrary time interval can pass between them. For example, the End User can enter knowledge whenever they want to once the Data Scientist has saved one module, and the ML System can trigger the generation process whenever it needs to once the End User has saved one knowledge base.

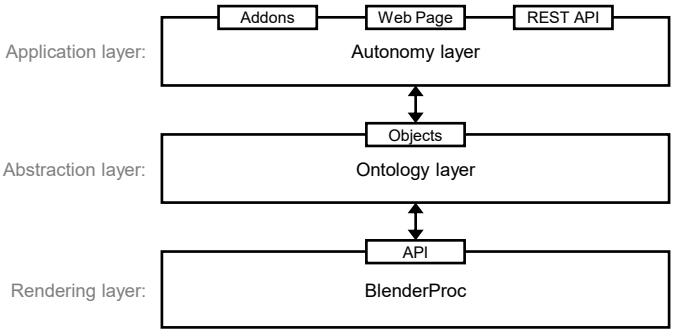

Fig. 2. Overview of the three-layer architecture implemented in our prototype.

IV. Implementation

Building upon the concept, we have developed a prototypical implementation, which we're describing in this section. The entire source code of our prototype can be found online so that others can view, use or extend it: https://github.com/davidd7/knowledge-based-synthetic-data-generation.

A. Basic Structure

At its core, our prototype is a web application that offers different interfaces for the three user roles. First, the Data Scientist can program addons and load them into the software to extend it with any rendering functionalities or user interfaces they need. Secondly, the End User can interact with the software via a web page, in which they can choose from the forms created by the Data Scientist and enter knowledge into them. Finally, the ML System can use a REST API to trigger synthetic data generation and to download the resulting datasets.

To implement these functionalities, our prototype follows a three-layer architecture, as shown in Fig. 2. At the top of this architecture is the *autonomy layer*, which implements all features of our concept, such as the three phases and entry points. Below that is the *ontology layer*, which simplifies all rendering-related aspects by abstracting a low-level SDG: instead of using the SDG itself, modules of the autonomy layer specify how synthetic data should be generated using functionalities defined in the ontology layer. Thus, less effort is needed, as general algorithms can be reused while task-specific knowledge is confined to the autonomy layer. Moreover, this setup also has advantages for the six implications for knowledge-based synthetic data generation (cf. Section II-C), which will be explained later. At the bottom of our three-layer architecture is the SDG itself, for which we've used BlenderProc in our prototype. In the following, the two top layers are explained in more detail.

B. Autonomy Layer

The autonomy layer contains the entire process of the concept, i.e., all three phases and entry points are implemented here. As can be seen in Fig. 3, the Data Scientist can extend the autonomy layer by creating two types of custom code: *modules* and *form components*. While form components are single elements, modules are made up of up to four parts specifying

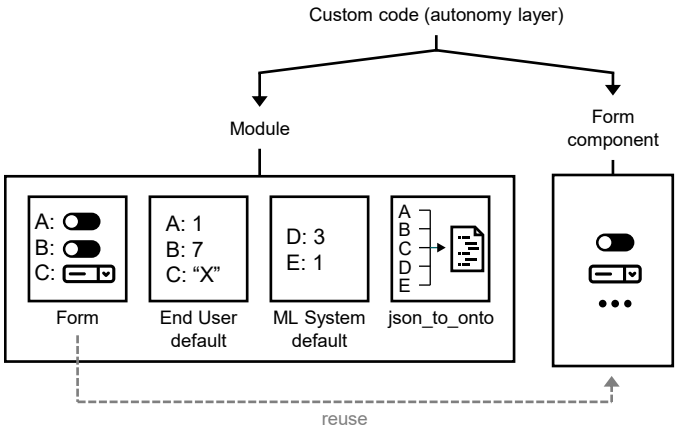

Fig. 3. Overview of custom code that the Data Scientist can create for the autonomy layer. Each module defines a specific form and generation algorithm. Because form components can be reused in many forms, the effort of creating new forms is minimized.

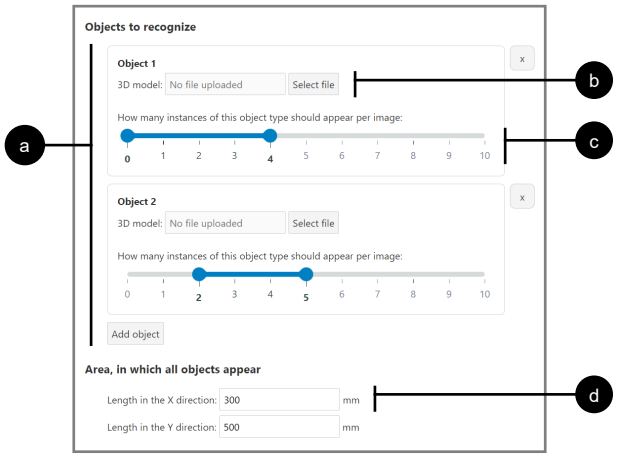

Fig. 4. Example form with form components highlighted: a) DynamicList, b) 3DModelInput, c) RangeInput, d) NumberInput. The DynamicList component allows End Users to create an arbitrarily long list of objects, e.g., so that they can specify objects to be recognized in a task without the Data Scientist having to specify how many objects there will be beforehand. For each object, all form components nested within the DynamicList component are loaded so that the End User can set attributes for each object.

different aspects of their behavior: a form file defining a form, two files with default values, and a function called json_to_onto that specifies how synthetic data is generated based on knowledge entered into the form.

Form creation: To design a form, the Data Scientist interacts with two elements of the custom code: the form components and the form file. The form components are single, reusable user interface elements, e.g., a dropdown menu. Once the Data Scientist has created a form component, it can be reused in any module's form file without further work. In the form file, the form components can be freely combined and nested to define a form. That way, even complex forms like the one shown in Fig. 4 can be easily created.

Saving form input: Knowledge entered into the form is saved in JSON format. Instead of custom load and save functions, this is done in a more automatic way to minimize the amount of effort required from the Data Scientist. First, the Data Scientist must create an end_user_default file as part of every module. This file consists only of a JSON object that specifies keys for the different knowledge points that the End User can enter into the form as well as default values that should be used when no knowledge is entered. These keys are then used in the form file, where all used form components must be bound to one of the keys. Using these bindings, the prototype can then automatically save knowledge entered into the form in JSON format and also automatically load data back into the form so that End Users can edit their input later on.

Generating data based on the input: The last required part of every module is a Python function called *json_to_onto*. This function is executed in the Data Generation phase and determines how to generate the synthetic dataset. As seen in the code snippet shown in Fig. 5, this is done by creating objects of different classes and setting their attributes. Crucially, the json_to_onto function is also where the knowledge from the different sources is integrated. On the one hand, the function receives two parameters for this: *end_user_data* contains the values entered by the End User via the form, and *ml_system_data* contains values sent by the ML System via the API. On the other hand, the function also makes knowledge added by the Data Scientist themself visible, namely all hard-coded values and the decisions which classes and attributes are used. Fig. 5 also shows what this integration can look like: the End User enters the depth and width of an area in which objects can appear, but the Data Scientist is deciding that they always appear 500mm above the ground without asking the End User for a value for this.

When the json_to_onto function is finished, the created objects are passed down to the ontology layer, which then renders the synthetic dataset solely based on the information in them. At the end of this, the autonomy layer makes the generated dataset available for the ML System to download.

```
def json_to_onto(onto_classes, end_user_data, ml_system_data):
    volume = onto_classes.SimpleVolume(
        Has_XCoordinate = [-end_user_data["area_length_x"]/2],
        Has_YCoordinate = [-end_user_data["area_length_y"]/2],
        Has_ZCoordinate = [500.0],
        Has_XLength = [end_user_data["area_length_x"]],
        Has_YLength = [end_user_data["area_length_y"]]
    )
    # ...
    root = onto_classes.GenerationRoot(
        Has_NumberOfImagesToRender = [ml_system_data["count"]],
        Has_Volume = [volume],
        Has_Object = obj,
        Has_Camera = [camera],
        Has_Label = [label]
    )
```

Fig. 5. Example json_to_onto function. Most knowledge is hard-coded by the Data Scientist, but for the volume's depth and width, values entered by an End User via the form are used (see a), and for the dataset size, a parameter sent by the ML System is used (see b). A root object is created to bind everything together.

C. Ontology Layer

As described in the preceding section, the Data Scientist defines in the json_to_onto function how to generate synthetic

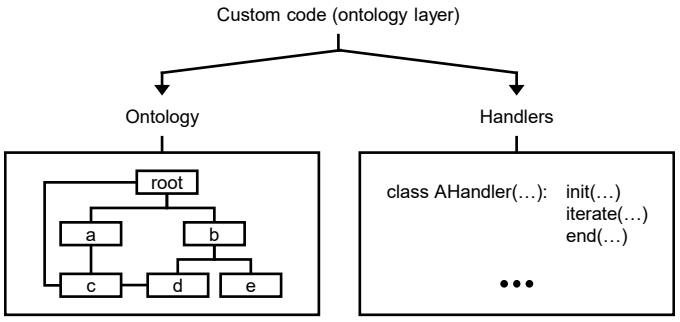

Fig. 6. Overview of custom code that the Data Scientist can create for the ontology layer. The ontology defines which classes and attributes can be used in the json_to_onto function and how they can be connected. The handlers specify how objects of these classes influence the rendering process.

datasets by creating objects of different classes. Which classes can be used in this function is defined in the ontology layer.

Similar to the autonomy layer, the ontology layer can be extended by the Data Scientist with custom code. As can be seen in Fig. 6, there are two kinds of custom code in the ontology layer: the ontology itself and so-called handlers. While the ontology defines which classes and attributes can be used in the json_to_onto function, the handlers define how synthetic data is generated based on them. Both the ontology and the handlers are reusable so that functionality implemented once is available to all modules in the autonomy layer. Thus, the ontology layer only has to be adapted when completely new rendering functionalities are required, e.g., when a special method for rendering steam is needed for a new task and hasn't been implemented before.

The ontology: Ontologies are a tool to standardize how data is structured. An important decision in creating ontologies is what exactly they describe (cf. [19]). In the context of our prototype, our ontology describes how to render synthetic datasets. Thus, unlike an ontology that might describe the real world, our ontology also includes classes for things that don't exist in reality, like rendering random backgrounds. Furthermore, the classes in our ontology are not specific to a particular SDG but are meant as general descriptions that can be translated into API commands for many different SDGs.

In the following, we want to go over a few characteristics of our ontology. First, because ontology objects can be referenced by multiple other objects, the ontology enables reusability of information. For instance, an exemplary class out of our ontology is the Volume class, which defines 3D areas in a scene, e.g., to coarsely specify where objects can appear. By referencing the same Volume object in attributes from multiple other objects, it can be modeled that all these objects should appear in the same area without having to repeat that area's parameters for every object.

A second pattern of our ontology is that because it supports inheritance, one can define a general parent class for a feature and child classes for different strategies how to implement it. For instance, the class *Rotation* in our ontology defines how objects connected to it should be rotated. As part of our prototype, we have implemented two child classes for it: *RandomRotation* chooses an entirely random rotation for every image, whereas *LookAtVolumeRotation* rotates the object so that its front side is always oriented towards a random point in a specified volume, which could be used for example to orient lights towards random points on the ground.

The final aspect of our ontology that we want to pick out is the GenerationRoot class. In the json_to_onto function, the Data Scientist must create an object of this class that is connected directly or transitively to every other object that should influence the module's generation procedure. The reason for this is that GenerationRoot objects are used as the starting points of the generation process. Each GenerationRoot object describes how to generate one specific kind of dataset, and what's not connected to it will be ignored.

Handler classes: Besides defining which classes are available in the json_to_onto function, one also must specify how synthetic data is generated based on them. For that, the Data Scientist writes handler classes in Python. While the ontology is not related to a specific SDG, the handler classes can be seen as adapters that transform abstract information from ontology objects into commands of a specific SDG.

Usually, for each class in the ontology, there's a handler class that specifies how to render it. During the Data Generation phase, the handlers search through the objects created in the json_to_onto function and adapt the simulation based on their attributes. For instance, a LightHandler may specifically search for light objects. To adapt the simulation, each handler implements three functions: init, iterate, and end. At the start of the rendering process, every handler's init function is executed, and at the end, all end functions are executed. Between this, all iterate functions are executed in a loop as often as how many images should be generated. Within these functions, the handlers directly interact with a specific SDG's API to influence the simulation. In the prototype, all handlers were implemented using BlenderProc, but which SDG is used is interchangeable as long as all registered handlers use the same one.

V. Evaluation

To test the applicability of our approach, we have used our prototype as part of an example experiment. This example experiment revolves around the topic of expert knowledge, because a central aspect of our approach is that it simplifies the integration of expert knowledge in synthetic data generation. Crucially, the central aim of the experiment is to see whether the components of our prototype can be successfully used to generate synthetic data for such a task. Thus, the experiment itself is designed to be rather simple, and while the results open up several interesting questions, the further investigation of these questions is outside the scope of this paper.

In the following, first, an overview of the example experiment's setup is given. After that, we describe how our prototype was used as part of the experiment. Finally, the experimental results are discussed.

A. Experimental Setup

Our experiment examines the influence of two types of knowledge on the performance of ML models and whether more realistic knowledge leads to better performances. The

examined knowledge types are the number and location of two kinds of objects that are to be recognized on images. These knowledge types were selected because they are simple to understand and enter into a form for humans, leading to the interesting question whether such simple knowledge types may still have a positive effect on ML model performance.

At the start of our experiment, we created a test dataset T_{Real} by manually taking images with a camera. Each image in T_{Real} contains exactly 3 gears with a few big teeth (class 1) and 3 gears with many small teeth (class 2). Furthermore, all gears are located within the left half of each image. After taking these images, we manually labeled them with segmentation masks marking the two gear classes.

Following the creation of T_{Real}, we generated six synthetic datasets D_1, ..., D_6. Each of these datasets was generated exactly the same except for the number of gears per image and the location of the gears within the images:

- *Number of gears per image:* D_1 and D_2 have exactly 3 gears of each class per image (same as T_{Real}), D_3 and D_4 have between 0 and 6 gears of each class per image (same average as T_{Real}), and D_5 and D_6 have exactly 20 gears of each class per image.
- *Location of gears:* In D_1, D_3, and D_5, all gears are located on the left half of every image (same as T_{Real}), whereas in D_2, D_4, and D_6, the gears appear on both halves.

The consequence of these different values is that some of the synthetic datasets are more similar to T_{Real}, and others are further away from it. For instance, in D_1, the number of gears and their location are exactly as in T_{Real}, whereas D_6 is most different from T_{Real}, with different locations and the gear numbers not even being the same on average. Overall, the different value combinations thus represent different sets of expert knowledge with varying degrees of realism.

In the last phase of our experiment, we first trained a different ML model on each synthetic dataset, resulting in the ML models $M_1, ..., M_6$. Then, we tested their performances on T_{Real}. Based on the assumption that expert knowledge improves the performance of the resulting ML models, we expected the models trained on more realistic datasets to achieve better scores on T_{Real}.

B. Creating the Synthetic Datasets

To generate the six synthetic datasets, we used our prototype, going over all steps of our concept. First, we took on the role of the Data Scientist to create a new module that specifies the form shown in Fig. 7 and a corresponding *json_to_onto* function. As part of this *json_to_onto* function, we specified a few fixed knowledge points without asking the End User for values for them, such as setting camera properties similar to the camera used for creating T_{Real} and enabling random background images and lights. Moreover, we set a physical plausibility effect, which simulates gravity to let the gears fall into random but realistic poses on each image (cf. physical plausibility, e.g., [20]). Because we mostly used components that we had already created for a previous experiment, we could

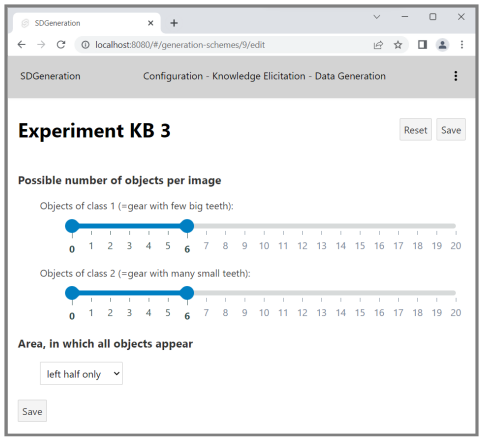

Fig. 7. Form created as part of our experiment with knowledge base for generating D_3 loaded.

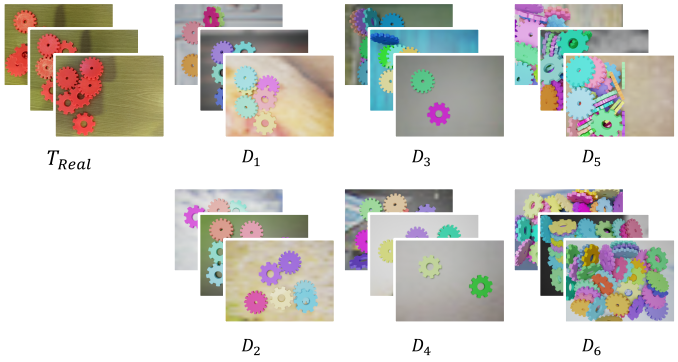

Fig. 8. Example images from the different created datasets. In D_3 and D_4, the number of gears per image is random, ranging from 0-12. Each synthetic dataset contains 1000 images.

take advantage of their reusability and didn't have to create any new ones, except for a drop-down form component.

Once the module was ready, we switched roles and used our software as an End User. Using the web app, we created six knowledge bases by entering the respective gear numbers and locations into the form. Finally, taking the role of the ML System, we triggered the generation of one dataset for each knowledge base via the API, resulting in the six datasets $D_1, ..., D_6$, of which example images can be seen in Fig. 8.

C. Training and Results

For our ML models' architecture, we selected U-Net [21]. We left the training parameters as predefined in the PyTorch implementation that we used, i.e., we trained each model in 5 epochs with a batch size of 1, a learning rate of $1e{-}5$, and 10% of the respective dataset split into a validation set [22]. The resulting performances of our trained ML models on T_{Real} can be seen in Table I.

The highest scores are achieved by M_1, ..., M_4, i.e., the models trained on 3 or 0-6 gears per class per image, with scores between 0.82 and 0.89. Notably, the best-performing model is M_2, even though it has less realistic knowledge than M_1. Following these four highest scores come M_6, with a score of 0.65, and M_5, which appears to be an outlier, with a score of 0.39.

TABLE I

PERFORMANCE OF THE TRAINED MACHINE LEARNING MODELS ON T_{Real}

Dataset	Model	Performance on T_{Real} (dice score)
D_1	M_1	0.86
D_2	M_2	**0.89**
D_3	M_3	0.86
D_4	M_4	0.82
D_5	M_5	0.39
D_6	M_6	0.65

Overall, it appears as if the number of gears per image is more important than their location. In each instance, a more realistic gear number improved performance, whereas models trained on more realistic locations performed worse in two cases. While this suggests that modeling the location has no benefit at all, we're cautious about such a conclusion because, except for the outlier M_5, realistic locations did lead to better performances in a preparatory experiment of ours. Thus, to gain more clarity, we suggest repeating the experiment several times and then comparing the average scores and standard deviations.

While M_5 also performs worse than the less realistic M_6, we suspect this rather has something to do with peculiarities of M_5. One hypothesis is that the high number of gears in a smaller space might move them closer to the camera, making them appear bigger than in T_{Real}. To test this, a dataset could be generated similar to D_5 but with a lower ground. Another hypothesis for the low performance of M_5 is that models may need more examples to understand the complex relations coming from 40 gears per image, and every image seen by M_6 having examples of such poses on both halves might be more valuable here than M_5 seeing where the gears are located in the real dataset. Further experiments could increase the training set size to see if M_5's performance improves if it sees more examples.

Once these more basic open questions have been tackled, the experiment could be expanded on in several ways. For instance, a bigger number of knowledge types could be compared, different ML tasks could be used, or it could be measured whether bigger dataset sizes can substitute more realistic knowledge at the expense of taking longer to generate.

VI. DISCUSSION

As seen in the preceding sections, we were able to realize our concept in a fully functional prototype. This prototype implements all features mentioned at the beginning of this paper. For instance, data can be automatically generated via an API, experts can influence the generation via forms, and the prototype is easily extendible to new domains and algorithms. Many of these features were used in our experiment, where we first created a parameterized script and then used this one script and the form-based user interface to generate six datasets with different expert knowledge. As this worked seamlessly, we conclude that our concept works at a basic level and that the prototype can be used to generate synthetic data.

Using our prototype, further research on autonomous productions can be conducted. On the one hand, because the experiment presented in this paper only assessed that the prototype works as expected, further experiments should more deeply evaluate the value that the prototype actually adds to autonomous productions. To do this, one could conduct user experiments or experiments in the style of [7], in which datasets are automatically generated to retrain robots. On the other hand, our prototype can also be used as one building block of a more complex autonomous process. Such a process as a whole can then be the focus of another line of experiments, e.g., testing how robust it works and how well the building blocks can be combined into such a system. Our prototype can thus help answer questions about our concept as well as about autonomous production in general.

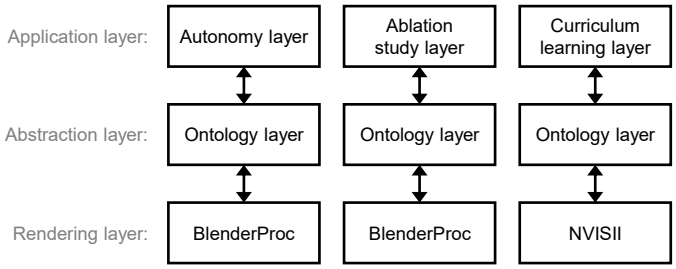

Fig. 9. Examples how other applications could be created following the three-layer architecture.

While our prototype adds features for autonomous productions, its rendering capabilities are entirely based upon an existing SDG. The main advantages of this are in reusability: Not only did we save effort by reusing functionalities from BlenderProc, but our software can also be easily extended with algorithms from other researchers using BlenderProc. Furthermore, as outlined in Fig. 9, the layers of our architecture are exchangeable so that another SDG than BlenderProc can be used, or the autonomy features be replaced with alternative applications. Overall, this approach works surprisingly well in our prototype, especially considering that BlenderProc was not designed for such use. We thus think that this three-layer architecture represents a promising new way to generate synthetic data, enabling higher levels of reusability.

Another feature of our prototype is that it implements features relevant to knowledge-based generation of synthetic data. In [18], six implications for such generation approaches were named. Table II shows how our prototype incorporates these implications. Except for one implication, all are part of the prototype, and for all of these, the ontology plays a central role. Our prototype thus shows that the implications can be realized and that ontologies appear to be an appropriate tool for this task.

To better understand our approach's efficiency, further experiments could test what amount of overhead is introduced by it. As the ontology layer creates and searches through ontology objects, the overall rendering time likely increases. To measure by what margin it increases, the same dataset could be rendered once with our prototype and once by directly using the underlying SDG. Comparing the respective rendering times can then show whether the features introduced by our software come with an acceptable overhead, and be a basis for optimizations in future prototypes.

TABLE II

HOW THE IMPLICATIONS FROM [18] ARE INCORPORATED IN OUR PROTOTYPE

Implication	Incorporation in prototype
Data Scientists implicitly add own knowledge	Knowledge from Data Scientists becomes visible because they define it using ontology objects. Saving these objects along with the datasets, it can be examined how a dataset was created later on, e.g., to rule out human errors or for analyses.
Some knowledge is only relevant for a user-friendly experience	Knowledge only relevant to user experience can be put in the JSON object and not saved in the ontology objects
End Users should be tasked with modeling the real world	Data Scientists can decide to only ask for real-world knowledge from End Users and define advanced knowledge themselves in the json_to_onto function
End Users and ML Systems contribute similar knowledge types	(For both, JSON is used, but no mechanism makes use of this fact at the moment)
There are different representations for the same knowledge	Can be realized with inheritance
Knowledge can be reused in several places	Ontology objects can be connected to many other objects

Another aspect to optimize is how exactly the ontology layer works. Currently, open questions include how to maximize the ontology classes' modular combinability and how best to implement the binding of the handlers to the ontology classes. To better understand the ontology-based approach and find adequate implementations, we suggest trying out different designs in further prototypes.

VII. SUMMARY

Our aim was to achieve higher levels of autonomy in synthetic data generation. To this end, we have presented a concept that allows automatic generation of synthetic data via an API and enables domain experts to influence the generation process via form-based user interfaces. Based on this concept, we have developed a prototypical implementation. To demonstrate its applicability, we have successfully used this prototype to generate synthetic data for an example experiment. From this successful usage of our prototype, we conclude that our ideas work and achieve the targeted goals, although further experiments are needed to assess our concept's full potential for autonomous production. One further observation of our prototype is that it is built on top of an existing tool for generating synthetic data. This leads us to the conclusion that synthetic data generation can be structured in exchangeable layers to maximize reusability of implemented functionalities.

To support further progress in the field of knowledge-based and autonomous synthetic data generation, we have published the entire source code of our prototype online: https://github.com/davidd7/knowledge-based-synthetic-data-generation. We invite other researchers to use our prototype to generate synthetic data for their projects or to try out adaptations of our approach.

ACKNOWLEDGEMENT

The IGF-project 22648 N/2 (ROOKIE) of the research association FVP was supported via the AiF within the funding program "Industrielle Gemeinschaftsforschung und -entwicklung (IGF)" by the Federal Ministry for Economic Affairs and Climate Action (BMWK) due to a decision of the German Parliament.

REFERENCES

[1] L. Zhou, L. Zhang, and N. Konz, "Computer vision techniques in manufacturing," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 53, no. 1, pp. 105–117, 2022.

[2] A. Mayr, D. Kißkalt, M. Meiners, B. Lutz, F. Schäfer, R. Seidel, A. Selmaier, J. Fuchs, M. Metzner, A. Blank, and J. Franke, "Machine learning in production – potentials, challenges and exemplary applications," *Procedia CIRP*, vol. 86, pp. 49–54, 2019.

[3] S. I. Nikolenko, *Synthetic Data for Deep Learning*, vol. 174 of *Springer Optimization and Its Applications*. Springer, 2021.

[4] A. Prakash, S. Boochoon, M. Brophy, D. Acuna, E. Cameracci, G. State, O. Shapira, and S. Birchfield, "Structured domain randomization: Bridging the reality gap by context-aware synthetic data," 2018.

[5] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, and X. Wang, "A survey of deep active learning," *ACM Computing Surveys*, vol. 54, no. 9, 2022.

[6] C. Chen, X. Jiang, S. Miao, W. Zhou, and Y. Liu, "Texture-less shiny objects grasping in a single rgb image using synthetic training data," *Applied Sciences*, vol. 12, no. 12, 2022.

[7] K. Alexopoulos, N. Nikolakis, and G. Chryssolouris, "Digital twin-driven supervised machine learning for the development of artificial intelligence applications in manufacturing," *International Journal of Computer Integrated Manufacturing*, vol. 33, no. 5, pp. 429–439, 2020.

[8] J. Arents and M. Greitans, "Smart industrial robot control trends, challenges and opportunities within manufacturing," *Applied Sciences*, vol. 12, no. 2, 2022.

[9] L. Eversberg and J. Lambrecht, "Generating images with physics-based rendering for an industrial object detection task: Realism versus domain randomization," *Sensors*, vol. 21, no. 23, 2021.

[10] J. Dümmel, V. Kostik, and J. Oellerich, "Generating synthetic training data for assembly processes," in *Advances in Production Management Systems. Artificial Intelligence for Sustainable and Resilient Production Systems*, vol. 633 of *IFIP AICT*, pp. 119–128, Springer, 2021.

[11] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb, "Learning from simulated and unsupervised images through adversarial training," 2016.

[12] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," 2017.

[13] N. Morrical, J. Tremblay, Y. Lin, S. Tyree, S. Birchfield, V. Pascucci, and I. Wald, "Nvisii: A scriptable tool for photorealistic image generation," 2021.

[14] P. Yudkin, E. Friedman, O. Zvitia, and G. Elbaz, "Hands-up: Leveraging synthetic data for hands-on-wheel detection," 2022.

[15] A. Tsirikoglou, J. Kronander, M. Wrenninge, and J. Unger, "Procedural modeling and physically based rendering for synthetic data generation in automotive applications," 2017.

[16] M. Denninger, M. Sundermeyer, D. Winkelbauer, Y. Zidan, D. Olefir, M. Elbadrawy, A. Lodhi, and H. Katam, "Blenderproc," 2019.

[17] H. Ren, H. Zhang, J. Zheng, J. Zheng, R. Tang, R. Wang, Y. Huo, and H. Bao, "Minervas: Massive interior environments virtual synthesis," 2021.

[18] O. Petrovic, D. L. Dias Duarte, S. Storms, and W. Herfs, "Towards knowledge-based generation of synthetic data by taxonomizing expert knowledge in production," 2023.

[19] N. F. Noy and D. L. McGuinness, "Ontology development 101: A guide to creating your first ontology," 2001.

[20] J. Tremblay, T. To, B. Sundaralingam, Y. Xiang, D. Fox, and S. Birchfield, "Deep object pose estimation for semantic robotic grasping of household objects," 2018.

[21] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," 2015.

[22] A. Milesi, "Pytorch-unet." https://github.com/milesial/Pytorch-UNet, 2017. Accessed: 2023-01-15.