



Semi-Automatic Task Planning of Virtual Humans in Digital Factory Settings

Martin Winter¹ , Thomas Kronfeld² , Guido Brunnett³ 

¹Technische Universität Chemnitz, martin.winter@mathematik.tu-chemnitz.de

²Technische Universität Chemnitz, thomas.kronfeld@informatik.tu-chemnitz.de

³Technische Universität Chemnitz, guido.brunnett@informatik.tu-chemnitz.de

Corresponding author: Martin Winter, martin.winter@mathematik.tu-chemnitz.de

Abstract. In commercial software systems for production planning the movements of digital humans have to be programmed manually. To improve the usability of digital humans in such simulations, methods are needed that create work plans, actions and movements for digital humans at least semi-automatically. The “Smart Virtual Worker” (SVW) is an experimental software platform for the development of such methods. For given descriptions of transport or assembly tasks the SVW computes action sequences and movements to fulfill these tasks. An optimization procedure is used to find solutions that balance the requirements of efficiency and ergonomics according to the specifications of Method Time Measurement (MTM) and Rapid Upper Limb Assessment (RULA). Since these scores can only be computed as we traverse the state space, learning methods must be used to compute the solution. In this paper we present a way to systematically implement world knowledge in the form of an action pre-selection mechanism to enhance the performance of such strategies. To show the effectiveness of our method we demonstrate that even with a complete random action selection our method is capable of solving non-trivial planning problems.

Keywords: Virtual Humans, Digital Factory, Human Factors, Process Planning, Virtual Prototyping, State Space Search, Knowledge Modeling

DOI: <https://doi.org/10.14733/cadaps.2019.688-702>

1 INTRODUCTION

Digital factory planning and digital commissioning are important tools for process and production planning. Their main objective is to minimize costs by optimizing the factory layout in terms of process time and material flow. The focus of these planning methods recently has changed from the simulation of machine and material movements to a human factor oriented viewpoint – especially considering an aging workforce. In this process ergonomic methods are used for the evaluation and the identification of potential problem areas. At the moment, production simulations involving digital humans are mainly used in major enterprises, operating in

the automotive and engineering sector. The high operational costs of available software systems, e.g. for necessary staff training, in combination with the enormous expenditure of time for creating a simulation, are substantial shortcomings which make this approach nearly inaccessible for small and medium-sized companies [14] [11]. To improve the usability of digital humans in simulations, methods are needed that avoid the time consuming process of manual programming by automatically creating work plans, actions and movements for digital humans.

In this paper we present a software system “The Smart Virtual Worker” (SVW) that has been developed as an experimental platform to demonstrate the feasibility of such an approach. The SVW is capable of performing five different types of elementary actions (walkto, transport, align, assemble and disassemble) and for any particular action that is performed our software calculates ergonomic scores based on Method Time Measurement (MTM, [6]) and Rapid Upper Limb Assessment (RULA, [7]) for their evaluation. An Autodesk® Inventor® plugin to our system is used to create a realistic virtual environment for the SVW. A task description for the SVW can either be imported with this plugin or created with the GUI of our software. Tasks for the SVW are internally represented as objectives. This means that any task is defined by criteria that must be met (the terminal objectives, see Section 4.2), rather than a specific action sequence to reach it. The actions to achieve the terminal objectives are generated by the planning algorithm that intends to minimize an objective function specified as a convex combination of the MTM and RULA scores of the movements of the virtual worker. The user can influence the optimization by providing the weights to the convex combination.

The task of finding optimal action sequences can be modeled as a shortest path problem in an abstract state space. However, the state space is fairly large and unstructured (to keep generality and extensibility) and we can only compute costs and states as we traverse the space. Therefore, a world agnostic (i.e. problem independent) learning method seems appropriate, more precisely a reinforcement learning (RL) based state space search is used to determine an ergonomically optimal solution to the planning problem. See [13] for a detailed description of the RL agent.

The RL agent has no a priori knowledge about the world and consequently does not distinguish between reasonable and unreasonable actions in any particular situation. Therefore the agent creates a tremendous amount of useless action sequences (until enough knowledge has been build up in the reward mechanism of the learning strategy) which results in high expenses for run-time and memory consumption. In this paper we report on our approach to reduce the computational costs by combining the RL with a reasoning mechanism, which pre-selects reasonable actions, and therewith counteracts the combinatorial explosion of the search space. To formalize the selection mechanism we introduce the concepts of objective spaces and their predecessors. An objective space O represents those states of the environment that satisfy a given objective and a predecessor is the collection of all states which can be transformed into elements of O with a single action of the SVW. By recursively generating the predecessors of objective spaces and storing them in the nodes of the objective tree we create a data structure that represents preconditions for actions to be performed. A locking mechanism on this tree is used to specify objectives that have to stay valid in the course of actions. Actions are considered invalid, if the achieved states do not satisfy all locked objectives. In this way, we can avoid undoing achieved goals, and hence, avoid getting stuck in loops. By systematically locking/unlocking objectives, it is possible to narrow down the set of possible actions, and therewith avoid the combinatorial explosion on the state space search.

In each cycle of the planning process the elements of the action sequence are generated one after another. Each action is determined in a process consisting of three consecutive steps: selection of achievable objectives, action pre-selection and action selection. Achievable objectives can be fulfilled with a single action in the current state of the environment. For each achievable objective the action pre-selection determines the corresponding actions and sorts out unreasonable actions (see Section 4.4). This list of actions is then handed to the action selection based on reinforcement learning. It decides on a single action from the list, and sends it to the environment for application (see Section 4.5).

The main contribution of this paper lies in the formalization of the action selection mechanism which allows

to use it for different applications. The concept of elementary intermediate goals (objectives) is abstracted in a way, so that the world knowledge can be extended in an easy and structured manner. Whenever rules can be formulated that distinguish reasonable from unreasonable action sequences, these can be implemented with our framework. The pre-selection mechanism then provides a significant reduction of the size of the state space of the optimization problem.

The layout of this paper is as follows. The next section gives an overview of the related work. Following this, Section 3 gives an overview of the developed framework as well as a description of the object, scene and task definition. Section 4 is devoted to a discussion of the planning process, with focus on the pre-selection mechanisms. We describe how the SVW translates the problem into a processable form, how it finds a path to a terminal state and how this path can be optimized. For a compact description, we concentrate on the pre-processing and do not include the RL. In Section 5, we demonstrate the planning using a simplified random action selection instead.

2 RELATED-WORK

The application of digital human models in the context of factory planning not only reduces costs and time of the process, it also leads to a better understanding of human factors at an early stage of the planning process. A very comprehensive and detailed catalogue of human models in computer aided engineering applications can be found in [12]. Among these Human Builder (part of the manufacturing simulation package DELMIA by Dassault Systemes) and Jack (part of Siemens PLM software system Tecnomatix) are the most established commercial solutions [1] [10]. In [4] an immersive VR system is presented using a Redirect Walking controller. With the focus on human factors [15] realized a CAVE system for factory planning which is also able to visualize the sound intensity of noise. The implemented planning algorithm uses a condition based pre-selection that can be compared to the one presented in [8]. However, the focus of this paper is on the explicit knowledge modeling [2] and its implementation rather than learning methods. While machine learning based techniques seem to take over in many fields, knowledge engineering still has many applications, especially where the system must be well understood and reliable. The application of ontologies and knowledge design was considered before in automated safety planning [16], automated construction safety [5] and support for cyber physical systems [9].

3 OVERVIEW OF THE SMART VIRTUAL WORKER

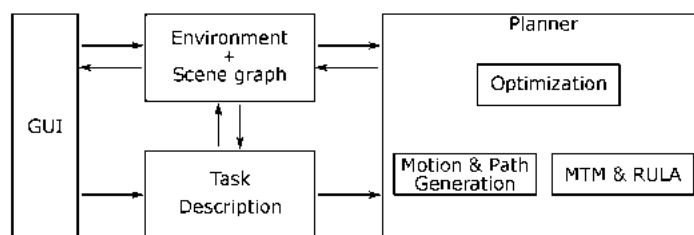


Figure 1: Scheme of the smart virtual worker (SVW) framework.

The smart virtual worker consists of several modules shown in Fig. 1. The GUI module is used for offline input, description and visualization of the scene, configuration of the modules, input and alignment of the tasks, and visualizing the results.

The central component is the environment module. It manages the scene graph (see Section 3.1), the current and all previously visited states of the environment, and the static and dynamic parameters of the

virtual worker. Thus it forms the data base for the operations of all other modules and holds their configuration data.

The SVW is capable of transforming abstract task descriptions into an action sequence for reaching these goals, as well as a keyframe animation to display the solution (see Fig. 2). This happens using the following pipeline in the planning module: the planning module receives a list of complex tasks and generates a discrete sequence of elementary actions (e.g. go, carry, assembly, ...; see Tab. 2). For this purpose, an optimization algorithm is used to optimize the order of elementary actions based on a predefined target function (see Section 4). The resulting set of actions can be divided into stationary actions (e.g. grab, release) and non-stationary actions (e.g. walk, carry). For non-stationary actions, the path planning module computes a shortest collision-free path between the virtual human and a given target position within the current scene. Based on the anthropometric parameters of the worker, the path module is able to determine the accessibility of objects. For stationary actions, further pre-defined keyframe animations are used to generate collision free motions for the agent. One of the more complex steps in the simulation of virtual human models is the generation of realistic and collision-free motions for each elementary action. Therefore, a method was developed which generates human motion based on parameterized motion spaces [3]. These motion spaces correspond to the elementary actions defined in the framework. The motion generation module uses these motion spaces in order to calculate a motion sequence (keyframes) based on the discrete sequence of elementary actions. This motion sequence is used to compute the costs of an action sequence. For each elementary action, the duration is given by the MTM method, used for time standardization. The ergonomic costs are computed by summing the ergonomic scores calculated for each pose of the motion sequence based on the RULA method. A customized convex combination of the MTM and RULA scores are now considered the costs of the action sequence. These costs are used in the planning module to optimize the actions sequence iteratively.



Figure 2: Screenshot of the graphical user interface of the SVW.

3.1 Scene Description

The scene description contains all information necessary to represent the configuration of the environment: position, orientation of objects and workers (agents), as well as their relations. It defines a hierarchical scene

graph in which each node contains an object, a reference to its parent and the local transformation relative to the parent's space.

An object is defined by four mandatory attributes: id, name, group, and path to the geometry file. The attribute group is used to form groups of objects with similar properties (for example storage, transport, etc.). The SVW framework contains no function to create geometry. Therefore, there is a need to import desired objects into the software. For this, a connection to Autodesk® Factory Design Utilities was created, which allows the import of objects from their asset library. Along with the geometry description, further properties such as weight, bounding box as well as assembly and grab points are imported. Furthermore, for objects such as tables or workbenches, areas can be defined, where objects can be placed or assembled. Once these objects are integrated to the SVW they are part of its library and thus can be reused in future scenes.

The agent is a special type of object with additional attributes, e.g. constitution, strength and functional reach ranges. There are predefined agent types, that can be customized in the GUI.

3.2 Task Description

The task description module holds all defined tasks and tracks which of them are already solved and which are still open. The task description is stored in an XML file. Every task is listed as a <task> element. It contains the type of task, the involved objects as a parent-child relationship, as well as the target position and orientation relative to the parent. Further constraints can be specified, e.g. the chronological order of the objectives.

Within the software, there are two basic tasks for the interaction with objects: transport/alignment and assembly/disassembly. Both task types could either be specified within the GUI or imported as part of the task description. A list of all available task types is shown in Tab. 1.

<i>name</i>	<i>description</i>
Transport	Move object from position A to B
Alignment	Change object orientation without moving
Assembly	Attach an scene object to another scene object
Disassembly	Detach an scene object from other scene object
WalkTo	Walk to a given location and optionally replay a captured motion

Table 1: Objectivs defined in the SVW framework.

In order to specify a transport task within the GUI, the user has to select the scene object, and then must click at the target position. A further and more general approach is to declare the object and the target position.

In a similar way, assembly tasks can be specified. For the interactive definition in the GUI, the assembly points of the scene objects to be joined need to be clicked. If the desired construction is more complex, the Autodesk® Inventor® interface should be used. The supported assembly joints not only contain the component's relative position but also define possible types of degree of freedom, e.g. rigid, rotational, slider, cylindrical, planar and ball.

The following example shows the description of three transport tasks implemented in the scene shown in Figure 2. This example will be used throughout the paper to demonstrate the planning capabilities of the SVW.

```
<sequence>
  <task type="transport" taskId="1" objId="4" objP="5" pos="300 800 786" quat="0.7 0 0 0.7"/>
```

```

<task type="transport" taskId="2" objId="6" objP="7" pos="300 300 786" quat="0.7 0 0 0.7"/>
<task type="transport" taskId="3" objId="10" objP="8" pos="300 800 786" quat="0.7 0 0 0.7"/>
</sequence>

```

The planning module translates the tasks first into a set of objective, and then generates a sequence of actions to achieve these. All possible actions types (which are combined with specific parameters to form a concrete action) are listed in Tab. 2. Note that in the following only walk, grabTH and release will be

<i>name</i>	<i>description</i>
walk	walk to a position
carry	carry an object to a position
grabTH	grab an object with both hands
grabLH	grab an object with the left hand
grabRH	grab an object with the right hand
release	release an item at a given position
assemble	assemble two items
separate	disassemble two items
push	push an object
relocate	tilt or rotate an object
idle	play a prepared action sequence

Table 2: Possible action types within the SVW framework. An action consists of an action type and several paramters to specify the exact execution.

considered as all other actions (e.g. assemble) are variations of these with similar internal representations, but maybe additional parameters (e.g. assembling part A onto part B is just releasing part A onto part B while holding the necessary tools for assembly in the other hand). An internal differentiation between these action types is still necessary for the motion generation, less so for the actions sequence planning.

4 OPTIMIZATION BASED PROCESS PLANNING

In this section we describe how the SVW automatically computes a solution to a given planning problem. Here we only consider the generation of a discrete action sequence, rather than the keyframe animation, which is generated from the actions by subsequent calls to the path planning and motion modules as described in Section 3. We start with the formalization of the problem as a state space search.

We first give a general overview: the planning process consists of *cycles*, each of which returns a sequence of actions that transforms the initial state into a terminal state corresponding to the tasks (represented by the terminal objectives, see Section 4.2). Several cycles are performed to improve the quality of the results. In each cycle, the action sequences are generated iteratively, one action at a time. To determine an appropriate next action, each iteration performs the following three steps – objective pre-selection, action pre-selection, and action selection.

Above iteration is repeated until a terminal state is reached (i.e. all terminal objectives are satisfied).

4.1 Formulation as a state space search

The planning problem of the SVW is modeled as a *state space search*. The general state space search problem can be formulated as a 6-tuple $(S, i, T, A_s, \delta_s, c_s)$, which consists of the following components:

- the *state space* S , i.e. the set of all possible configurations of the environment.
- an *initial state* $i \in S$.
- the set of all *terminal states* $T \subseteq S$. Note, that there may be several states that satisfy the terminal objectives of the planning problem (compare Section 4.2).
- to each state $s \in S$, a set of possible *actions* A_s .
- to each state $s \in S$ a *transfer function* $\delta_s : A_s \rightarrow S$, which to each action $a \in A_s$ assigns the *successor state* s' that emerges from s when applying a . We denote $\delta_s(a) = s'$ in the short form $s \xrightarrow{a} s'$.
- to each state $s \in S$, a *cost function* $c_s : A_s \rightarrow \mathbb{R}_+$, which to each action $a \in A_s$ assigns a non-negative real number, which represents the costs of performing the action a in state s .

A *feasible solution* of the search is a sequence of zero or more actions (a_0, \dots, a_{n-1}) , so that

$$i = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n \in T.$$

The *total costs* of a solution are given by $c(a_0, \dots, a_{n-1}) := \sum_{t=0}^{n-1} c(s_t, a_t)$. We are looking for an *optimal solution*, i.e. a feasible solution that minimizes the costs.

4.2 O-spaces and objective trees

Our planning algorithm uses the concept of *objective spaces* (O-spaces) $O \subseteq S$ to model the informal use of the term objective. Intuitively, an objective is a requirement on the configuration of the environment, and the O-space O represents the set of all those states in which this requirement is satisfied. We will use the term objective (and the associated requirement) interchangeably with its formalization as an O-space. If $s \in O$, we say that s *satisfies* O , or that O is satisfied in s . If there is some action $a \in A_s$ with $\delta_s(a) \in O$, then we say that O can be satisfied from s (or that O is *satisfiable* in s). The state s is then called a *predecessor state* of O . The set of all those predecessor states is denoted by $P(O)$ – the *predecessor space* of O .

We construct our objectives (and corresponding O-spaces) from the tasks given by the SVW. To distinguish these initially constructed objectives from those which are added for sub-problems, we call the former ones *terminal objectives*. Our general strategy to achieve the terminal objectives is to recursively generate predecessor spaces and to represent these in terms of simpler objectives. More precisely, given an O-space O and its predecessor space $P(O)$, we use a representation $O_1 \cap \dots \cap O_m \subseteq P(O)$. The O-spaces O_1, \dots, O_m are called *preconditions* of O and are determined using the knowledge base described in Section 4.7. Given a set of preconditions of O , we can test whether O is satisfiable, simply by checking whether all preconditions are satisfied. Note that if O is always satisfiable, we can formalize this as $P(O) = S$.

Given above representation of the predecessor space of some O-space O as the intersection of further O-spaces, it seems natural to organize O-spaces in a tree structure. An *objective tree* is a rooted tree in which the nodes are O-spaces, and the children of a node are its preconditions. The roots of these trees will represent the terminal objectives – they are not preconditions of other objectives. The goal of a single cycle is therefore to satisfy the roots of all objective trees. If an O-space in the tree is not satisfied yet, we can satisfy it by first satisfying all its children, and then applying an appropriate action.

In the objective pre-selection step (see Section 4.3), we will use the objective trees to efficiently conclude on a set of relevant objectives. In the action pre-selection step (see Section 4.4), we will use the tree to identify invalid actions (actions which undo certain already satisfied objectives). More precisely, each node in

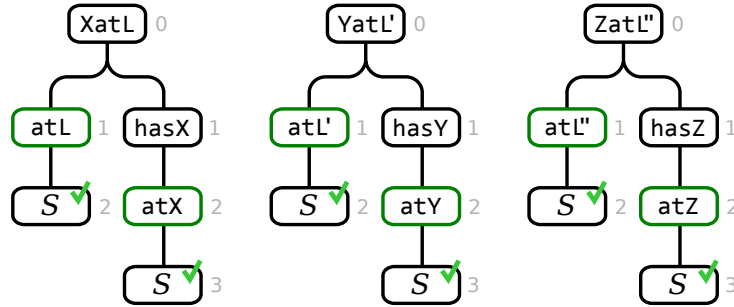


Figure 3: The three objective trees generated at the beginning of a cycle for the problem of Section 3.2. A gray number indicates the iteration of the tree updating loop, in which this node was added. A zero means that this node was added during initialization of the trees. The check-marks indicate that these O-spaces are satisfied in the initial state, hence the updating process stopped there. A green boundary marks the satisfiable O-spaces, those which will be suggested by the objective pre-selection.

an objective tree is either *locked* or *unlocked* (initially, all nodes unlocked). Locked nodes are those which are not allowed to become unsatisfied again (how these are chosen is explained in Section 4.7). Formally, an action is considered *invalid*, if its successor state does not satisfy all locked objectives (with an exception explained in Section 4.4). In this way, we can avoid undoing achieved goals, and hence avoid getting stuck in loops. By systematically locking/unlocking objectives, it is possible to narrow down the set of possible actions, and therewith minimize the effect of combinatorial explosion on the state space search.

To discuss the example from Section 3.2, a set of only four objectives is sufficient. These are also sufficient to model a wide range of transportation tasks in the SVW:

- XatL: Object X is located at location L .
- atL: The agent is at location L .
- atX: The agent is at the location of object X .
- hasX: The agent carries X .

Additionally it is useful to consider S as an O-space. It can be used as a precondition of O-spaces that are always satisfiable. Transport tasks will be interpreted as XatL objectives with appropriate object X and location L . Walk-to tasks will be interpreted as atL objectives with appropriate location L .

4.3 Updating objective trees and objective pre-selection

The objective trees are initialized at the beginning of a cycle, before the first iteration. For each terminal objective we create an objective tree \mathcal{T}_i with the associated O-space as a root.

At the beginning of each iteration, we need to update the objective trees: for each unsatisfied O-space O that is a leaf of an objective tree, we generate a set of preconditions of O and add them as children of O to the tree. We repeat this iteratively, until all leaves are satisfied. The objective trees generated in the example of Section 3.2 are shown in Figure 3.

The task of the *objective pre-selection* is to generate a list of O-spaces which should be considered as goals in this iterations. These O-spaces are called *active*, and are those which are satisfiable, but not yet satisfied. The list of all active O-spaces is the result of the objective pre-selection step.

4.4 Action pre-selection

The task of the *action pre-selection* is to take the list of active O-spaces, and generate a list of *valid* actions, more precisely, a list of pairs (O, a) , where O is a node representing an active O-space, and a is an action for achieving O . A pair (O, a) is considered as valid, if the successor state $\delta_s(a)$ (s being the current state) satisfies all locked objectives, *except* any locked descendants of O . The descendants of O can be ignored, because they were added to the objective trees with the purpose to satisfy O , the goal we are trying to reach by a anyway. The list of all valid pairs is the result of the action pre-selection step.

4.5 Action selection

The task of the *action selection* is to choose a single pair (O^*, a^*) from the list provided by the action pre-selection. Since the action selection is implemented in a world-agnostic manner, the nature of the elements of the list is ignored in this step. We therefore abbreviate a pair (O, a) by an abstract action α .

The action selection is the appropriate part of the planning to use the reinforcement learning. However, for the sake of a compact description, we instead consider a simple (but already useful) implementation of the action selection by always choosing a random action α from the list. Since the action pre-selection only produces target-aimed actions, we can never enter a loop, independent of the actual choice process.

By the actual action selection process, we obtain a pair $\alpha^* = (O^*, a^*)$. This pair is the result of the action selection step. The following steps finalize an iteration of the planning algorithm:

- The action a^* is sent to the environment and applied to the current state $s \in S$. This results in the new state $\delta_s(a^*)$.
- The O-space O^* is locked.
- The objective tree is pruned at O^* , i.e. all descendants of O^* are removed (this includes unlocking any locked descendants).

Note, that while there may be other objectives becoming satisfied when applying a^* , we only lock O^* . This models the fact that we intentionally chose this objective to be important, so that we do not want it to become unsatisfied again. Locking other objectives might prevent us from reaching a terminal state.

4.6 The planning algorithm

The planning algorithm is guaranteed to terminate after a finite number of iterations. This is evident from the fact, that the locked O-spaces are successively moving closer to the roots of their respective objective tree: in each iteration, the chosen O-space O^* is locked. This lock will not be removed (and consequently the objective will never become unsatisfied) until an ancestor of O^* is locked and O^* is deleted.

The pseudo code of a single cycle of the planning algorithm is given in the following:

```

( $O_1^t, \dots, O_n^t$ ) = generateTerminalObjectives(getComplexTasks());
initializeObjectiveTrees( $O_1^t, \dots, O_n^t$ );
while a terminal objective  $O_i^t$  was not achieved do
    updateObjectiveTrees();
    // objective pre-selection + action pre-selection
    L = { };
    for each active O-space  $O$  do
        for each action  $a$  with  $\delta_s(a) \in O$  do
            if isValid( $O, a$ ) then
                | L = L  $\cup$  { ( $O, a$ ) }
            end
        end
    end
    if L is empty then
        | break;
        | // reached a dead end. End cycle
    end
    // action selection
    ( $O^*, a^*$ ) = makeDecision(L);
    lock( $O^*$ );
    pruneObjectiveTreeAt( $O^*$ );
    apply( $a^*$ );
end

```

4.7 Notes on the implementation

The objective and action pre-selection steps depend on world knowledge. This world knowledge is provided by the implementation of the objectives. The algorithm is described as operating on abstract objectives, which are modeled as sets of states. Set operations are not feasible for an actual implementation. Instead, we use an efficient and easily reproducible implementation of the concept of objectives, which allows us to

- determine, whether a state s achieves O .
- provides appropriate preconditions O_1, \dots, O_m of O , as well as actions to reach O from $P(O)$.
- determine whether an action a is valid for O , i.e. leaves O satisfied.

An objective is uniquely determined by an implementation for only these three queries. The first query is mostly trivial to implement, as it only checks a certain property of the environment. Table 3 contains descriptions how the other two queries are implemented for the four fundamental objective types used in the example. Note that in the example of Section 3.2, all objects must be grabbed two-handed, hence any grab action is of the type grabTH.

5 COMPUTATIONAL RESULTS

To illustrate the applicability of our planning approach, we consider two example. As mentioned in 4.5, we use a random action selection instead of the learning approach for a self-contained description and an easier understanding of the results. Although random choice is one of the simplest action selection strategies, which on its own is not guaranteed to terminate (e.g. we can loop on repeatedly picking up and releasing a box),

<i>objective</i>	<i>decomposition of $P(O)$</i>	<i>invalid actions</i>
xatL	Can be satisfied by a “release X at L ” action. This action is meaningful if hasX and atL are satisfied: $P(O) \supseteq \text{atL} \cap \text{hasX}$	Grab actions that involve X
hasX	Can be satisfied by a “grab X ” action. This action is meaningful if atX is satisfied: $P(O) \supseteq \text{atX}$	Release actions that involve X
atL	Can be satisfied by a “walk to L ” action. This action is always meaningful, hence $P(O) = S$.	Walk actions
atX	Can be satisfied by a “walk to location of X ” action. This action is always meaningful, hence $P(O) = S$.	Walk actions

Table 3: Overview of the implementation of four fundamental objectives.

the used action pre-selection techniques suffice to ensure termination after a small number of steps. We show this by directly comparing action generation with and without the action pre-selection.

For the evaluation we used a further optimization to limit the run time and memory consumption. In each cycle, the algorithm explores a random path in the state tree. In order to further minimize the number of branches in each state, we decided to delete states which are no longer needed. A state deletion will happen in one of the following cases:

- The planning algorithm prevents loops by two techniques: 1. only considering target-aimed actions, 2. excluding actions that undo former achievements. It can happen that these restrictions exclude all actions, hence that the current state is a dead end in the current search path. More general, any non-terminal state without successors can be considered dead. Dead states are deleted in order to be not visited again.
- If a state is reached with more costs than the lowest costs known so far for reaching a terminal state, then this state is deleted to be not considered again.

Note that the deletion of a state will reduce the number of branches of the parent state. If enough successor states are deleted, the state will have no branches, is considered dead and will be deleted itself.

Both exemplary planning procedures were executed on a single core of an Intel(R) Core(TM) i7-6700HQ CPU (2.60 GHz).

5.1 Example: Four tables, three boxes

The scene is given in Figure 4 with the example tasks from Section 3.2. This configuration is chosen to have a non-trivial optimal solution. A simple greedy strategy would always choose the cheapest action, which is to transport the closest box one table to the right. However, the overall best solution is to walk to the left and start with the box on the left table. This is, because then any box ends up where the next one can be picked up.

Initially we ran 10,000 cycles (in under a second) *without* an action pre-selection (i.e. all logically valid actions are considered). This, for example, allowed the agent to walk to tables, and subsequently leave them again without doing anything there. This resulted in state spaces in the order of 1,300,000 visited states (mostly all are remaining since dead ends cannot be detected without pre-selection) and not a single encounter of an optimal solution. The first encounter of the optimal solution was not before the 1,000,000th iteration.

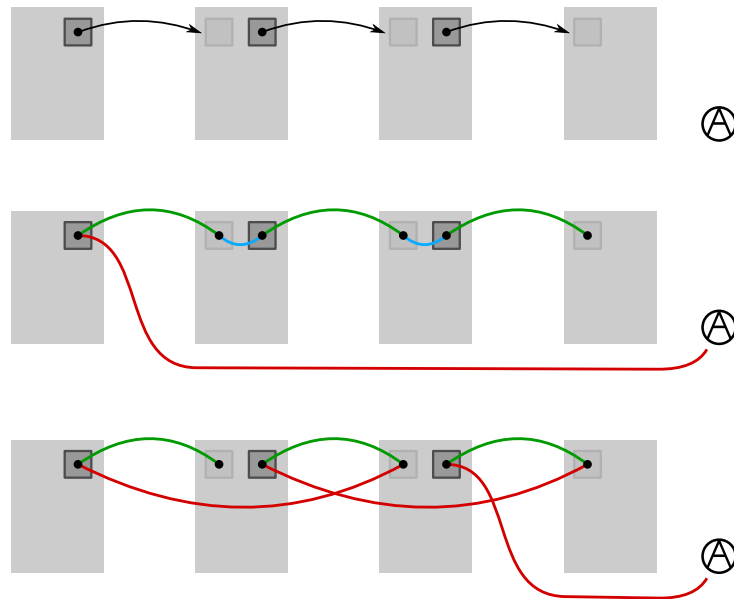


Figure 4: Schematic representation of the scene from example 1, with the example task from Section 3.2. The scene contains four tables with three boxes, each of which should be moved one table to the right. The agent starts at the far right. The middle figure shows the optimal path of the agent. The bottom figure shows the greedy solution, which is far from optimal. A red curve indicates that the agent is walking without moving any box closer to its location. A green curve indicates that the agent is productive by carrying a box to its target location. A blue line does not represent a relocation of the agent, it is just there to present the path of the agent as a continuous curve. A blue line indicates that the connected places could be reached by the agent from a single location and no intermediate walk action is necessary.

Running the planning procedure *with* pre-selection resulted in an optimal solution within 30 cycles with a high probability. This is because the state space in average only contains 60 states and branch deletion resulted in only 11 surviving states (an initial state plus one states per step in the optimal solution; compare Fig. 4).

Running 10,000 cycles with pre-selection (in under a second), the percentage of cycles resulting in dead ends drops to zero rapidly after all dead ends are removed. This example demonstrates how the state space can be trimmed effectively. Here, the size of the state space was reduced by an order of 10^5 compared to a run without pre-selection.

Figure 5 shows some initial iterations for a single objective tree from a cycle resulting in an optimal actions sequence.

5.2 Example: Eight tables, twelve boxes

For the second example we considered a more complex transport scene with eight tables and twelve boxes, and without any *obvious* optimal solution (see Fig. 6).

Running the procedure without pre-selection only gave highly sub-optimal solutions because the state space without reduction is far too large. Even with pre-selection and 10,000 cycles run, we visited more than 13,000,000 states with still 10,000,000 of them remaining alive. During the search, approximately 1,600,000 states were dropped because they were dead, but still only about 50% of all paths lead into dead ends. Another

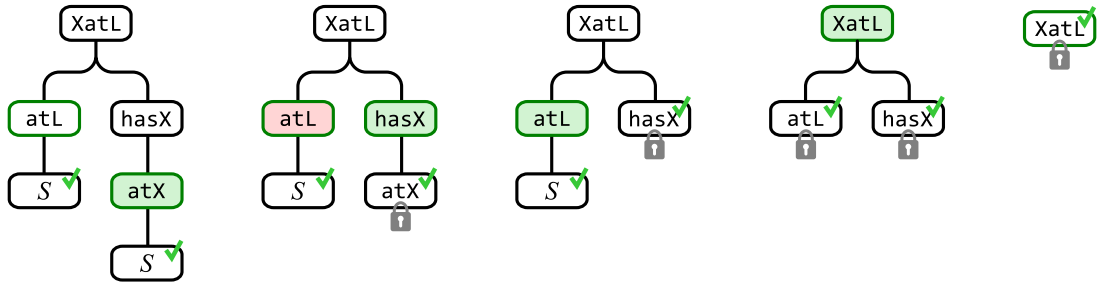


Figure 5: Four iterations in a cycle that results in an optimal action sequence for the example from Section 3.2. The evolution of only one of the three objective trees is shown in the image. The green highlighted objective represents the objective that was chosen by the action selection (the green boundary again indicates active objectives). The red highlighted objectives are blocked from becoming satisfied, because they contradict a currently locked objective (indicated by a gray lock symbol). Note that in the second and fourth step, the action selection had no real choice, because there is only a single remaining active and valid objective.

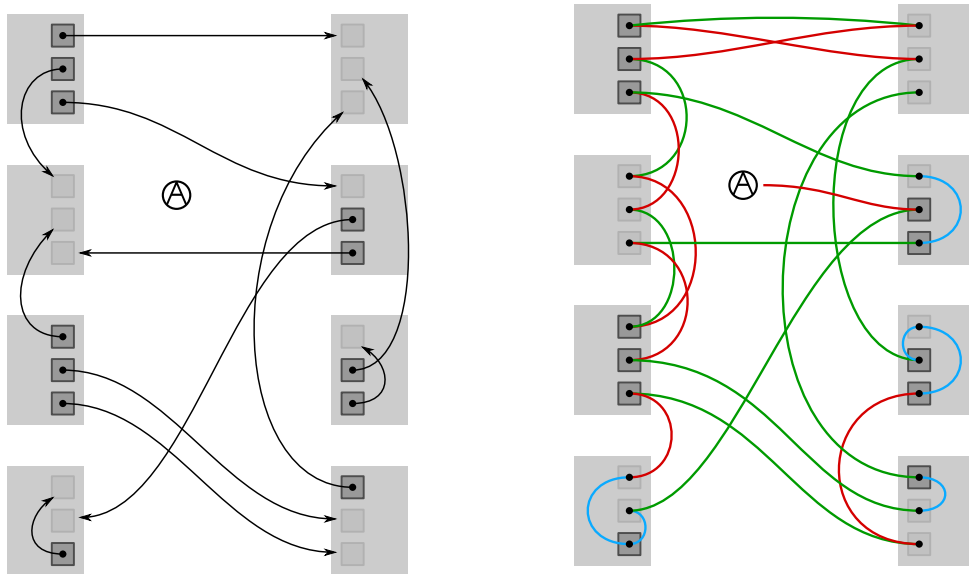


Figure 6: Schematic representation of the scene and task from example 2. The right figure represents the best solution found by the planning algorithm. The colors are to be understood as explained in the caption of Fig. 4.

1,000,000 states were removed because of sub-optimal costs. Running these 10,000 cycles took under a second. The best solution found is shown in Fig. 6. This solution is probably optimal since a second run of the planning algorithm for over a minute (and 500,000 cycles) resulted in no better solution. This example demonstrates the application of the pre-selection in a case which is intractable otherwise.

6 CONCLUSIONS

For given descriptions of transport or assembly tasks the “Smart Virtual Worker” computes action sequences to fulfill these tasks. An optimization procedure is used to find solutions that balance the requirements of efficiency and ergonomics according to the specifications of MTM and RULA. Since these scores can only be computed as we traverse the state space, world agnostic learning methods must be used to compute the solution. In this paper we present a compact and easy extendable implementation of a world-knowledge based action pre-selection mechanism to enhance the performance of such strategies. To demonstrate the usefulness of the method we demonstrate that even with a complete random action selection our method is capable of solving non-trivial planning problems by heavily reducing the size of the search space.

ORCID

Martin Winter  <http://orcid.org/0000-0003-0422-7774>

Thomas Kronfeld  <http://orcid.org/0000-0002-6399-8352>

Guido Brunnett  <http://orcid.org/0000-0002-8224-015X>

REFERENCES

- [1] Choi, S.; Kim, B.H.; Do Noh, S.: A diagnosis and evaluation method for strategic planning and systematic design of a virtual factory in smart manufacturing systems. *International Journal of Precision Engineering and Manufacturing*, 16(6), 1107–1115, 2015. ISSN 2234-7593. <http://doi.org/10.1007/s12541-015-0143-9>.
- [2] Devedzic, V.: Knowledge modeling—state of the art. *Integrated Computer-Aided Engineering*, 8(3), 257–281, 2001.
- [3] Kronfeld, T.; Fankhänel, J.; Brunnett, G.: Representation of motion spaces using spline functions and fourier series. In *Proceedings of the MMCS 2012, LNCS (Lecture Notes in Computer Science)*, 265–282, 2014. http://doi.org/10.1007/978-3-642-54382-1_16.
- [4] Kunz, A.; Zank, M.; Fjeld, M.; Nescher, T.: Real Walking in Virtual Environments for Factory Planning and Evaluation. *Procedia CIRP*, 44, 257–262, 2016. ISSN 22128271. <http://doi.org/10.1016/j.procir.2016.02.086>.
- [5] Lu, Y.; Li, Q.; Zhou, Z.; Deng, Y.: Ontology-based knowledge modeling for automated construction safety checking. *Safety science*, 79, 11–18, 2015.
- [6] Maynard, H.; Stegemerten, G.; Schwab, J.: *Methods-time measurement*. In *Proc. of the Sixth Intl. Conf. on Epigenetic Robotics*. McGraw-Hill, New York, 1948.
- [7] Mcatamney, L.; Corlett, E.N.: Rula: a survey method for the investigation of work-related upper limb disorder. *Applied Ergonomics*, 24(2), 91–93, 1993. [http://doi.org/10.1016/0003-6870\(93\)90080-S](http://doi.org/10.1016/0003-6870(93)90080-S).
- [8] Papudesi, V.; Huber, M.: Learning behaviorally grounded state representations for reinforcement learning agents. In *Proc. of the Sixth Intl. Conf. on Epigenetic Robotics*, 2006.
- [9] Petnga, L.; Austin, M.: An ontological framework for knowledge modeling and decision support in cyber-physical systems. *Advanced Engineering Informatics*, 30(1), 77–94, 2016.

- [10] Polášek, P.; Bureš, M.; Šimon, M.: Comparison of digital tools for ergonomics in practice. *Procedia Engineering*, 100(January), 1277–1285, 2015. ISSN 18777058. <http://doi.org/10.1016/j.proeng.2015.01.494>.
- [11] Spanner-Ulmer, B.; Mühlstedt, J.: Digitale menschmodelle als werkzeuge virtueller ergonomie. *Industrie-Management*, 26(4), 69–72, 2009.
- [12] Sundin, A.; Örtengren, R.: Digital Human Modeling for CAE Applications. In G. Salvendy, ed., *Handbook of Human Factors and Ergonomics*, chap. 39, 1053–1078. John Wiley & Sons, Inc, 3 ed., 2006.
- [13] Truschzinski, M.; Dinkelbach, H.; Muller, N.; Ohler, P.; Hamker, F.; Protzel, P.: Deducing human emotions by robots: Computing basic non-verbal expressions of performed actions during a work task. In 2014 IEEE International Symposium on Intelligent Control, ISIC 2014, 1342–1347, 2014.
- [14] Wischniewski, S.: *Digitale Ergonomie 2025. Trends und Strategien zur Gestaltung gebrauchstauglicher Produkte und sicherer, gesunder und wettbewerbsfähiger sozio-technischer Arbeitssysteme*. Bundesanstalt für Arbeitsschutz und Arbeitsmedizin, Dortmund, 2013.
- [15] Yang, X.; Deines, E.; Lauer, C.; Aurich, J.C.: A Human-centered Virtual Factory. In 2011 International Conference on Management Science and Industrial Engineering (MSIE), 1138–1142. IEEE, Harbin, China, 2011. ISBN 9781424483853. <http://doi.org/10.1109/MSIE.2011.5707619>.
- [16] Zhang, S.; Boukamp, F.; Teizer, J.: Ontology-based semantic modeling of construction safety knowledge: Towards automated safety planning for job hazard analysis (jha). *Automation in Construction*, 52, 29–41, 2015.