

# Structured Reactive Controllers and Transformational Planning for Manufacturing

Thomas Rühr

Dejan Pangercic  
Intelligent Autonomous Systems Group  
Department of Informatics  
Technische Universität München  
{ruehr,pangercic,beetz}@cs.tum.edu

Michael Beetz

## Abstract

*While current manufacturing systems are built to avoid uncertainty, the increase of setup reconfiguration frequency and ever higher numbers of variants produced on the same systems motivate the exploration of new approaches to manufacturing and manufacturing control. In this paper we discuss the application of plan based controllers and transformational planning in manufacturing. Autonomous control techniques are used to create flexible, robust and adaptive behaviour in the artificial intelligence community for about 20 years. In order to show the applicability and adequacy of plan based control in manufacturing, we exemplify the approach in a Flexible Manufacturing System (FMS). The experiments show that significant performance boosts are possible through transformational planning.*

## 1. Introduction

Manufacturing is evolving towards individualized production processes where Flexible Manufacturing Systems (FMS) set up themselves for the individual production tasks and might even reconfigure themselves to perform the tasks efficiently. However, dealing automatically with a large variety of production tasks will come at a price. Because process plans can not be completely specified in advance and the hardware setup will have to be modified on the fly, the process control systems can not reach the high degree of reliability that mass production systems reach. As a consequence these future manufacturing systems must be capable of detecting failures during an execution, diagnose them, recover from them locally when possible and change process plans on the fly to avoid failures and exploit opportunities.

Autonomous robot control has been facing these kinds of problems for more than thirty years and there has been a wealth of tools developed in this field that aim at reliable and efficient task execution when facing uncertainties and execution failures. To this end, service robots flex-

ibly interleave their tasks, exploit opportunities, quickly plan their courses of action, and, if necessary, revise their intended activities. The robots' controllers execute *plan-based control*, that is, they explicitly manage the robots' beliefs and current goals and revise their action plans accordingly.

In recent years, autonomous robots, including XAVIER [21], MARTHA [1], RHINO [4], MINERVA [2], and REMOTE AGENT [15], have shown impressive performance in longterm demonstrations. In NASA's Deep Space program, for example, an autonomous spacecraft controller, called the Remote Agent, has autonomously performed a scientific experiment in space. At Carnegie Mellon University XAVIER, another autonomous mobile robot, has navigated through an office environment for more than a year, allowing people to issue navigation commands and monitor their execution via the Internet. MINERVA [24] acted for thirteen days as a museum tourguide in the Smithsonian Museum, and led several thousand people through an exhibition.

These autonomous robots have in common that they perform plan-based control in order to achieve better problem solving competence. In the plan-based approach robots generate control actions by maintaining and executing a plan that is effective and has a high expected utility with respect to the robots' current goals and beliefs. Plans are robot control programs that a robot can not only execute but also reason about and manipulate [11]. Thus a plan-based controller is able to manage and adapt the robot's intended course of action – the plan – while executing it and can thereby better achieve complex and changing tasks. The plans used for autonomous robot control are often *reactive plans*, that is they specify how the robots are to respond in terms of low-level control actions to continually arriving sensory data in order to accomplish their objectives. The use of plans, as depicted in Figure 1, enables these robots to flexibly interleave complex and interacting tasks, exploit opportunities, quickly plan their courses of action, and, if necessary, revise their intended activities.

In this paper, we propose to apply the plan-based con-

trol techniques developed in the autonomous robotic domain to the control of next generation manufacturing systems.

To the best of our knowledge, there are no comparable systems to ours as a whole. However, there are frameworks which hold up with sub-components of our system. In the plan representation domain, the XFRM [12] represents an implicit design practice as opposed to our explicit approach implemented by RPL (section 5). PRS [7], TDL [20] and Architecture for Autonomy [1] focus on the design and implementation of plan execution languages that incorporate failure monitoring and recovery as well as concurrent execution. Williams et al. [25] developed the Reactive Model-based Programming Language (RMPL) to represent reactive plans, however the plans are not explicit and transparent. One of the best-known reactive planning languages is RAP [5] which is very robust and can also monitor the desired and actual outcome of the given plan. Classical (non-reactive) planning, represented by PDDL [13] for instance, can account for the uncertainty in planning, however, it lacks the necessary control structures, failure monitoring, and parameter representations. The adequate related work to TRANER (section 6) encompasses Hacker [23, 22] which also, like TRANER, aims at learning plan libraries by debugging the flaws of default plans. Its downside is that it only works in the idealized block-world domain. Further transformational planners are CHEF [6] and Gordious [19] which mainly differ from TRANER in that they only reason about plans that are sequential as oppose to the concurrent ones. Additionally, they try to produce the correct plans while TRANER adapts plans to specific environments and corrects failures during execution.

We will start by detailing the principles of plan-based control, comparing *classical* planning to *hierarchical* planning. We then describe our testbed system - Cognitive Factory - by giving an overview over its hardware and simulation setup as well as the autonomous robot we introduced to the factory in the simulated environment. In section 5 we give an overview of the language features of RPL, addressing the issues of manufacturing and assembly planning. In sections 6 and 7 respectively, a description of our recent additions to the RPL language is given - namely a Transformational Planner (TRANER) and a Robot Learning Language (RoLL). A section on empirical results follows, showing how *plan transformation* improves the plan performance on a distinct example. We then conclude the paper with a discussion of future research directions.

## 2. Principles of Plan-based Control

The building blocks of plan-based control are the representation of plans, the execution of plans, various forms of automatic learning, and reasoning about plans, including plan generation and transformation, and teleological, causal, and temporal reasoning. Before we dive deeper

into the concept of plan-based control, let us first get an intuition how the classical AI planning techniques work.

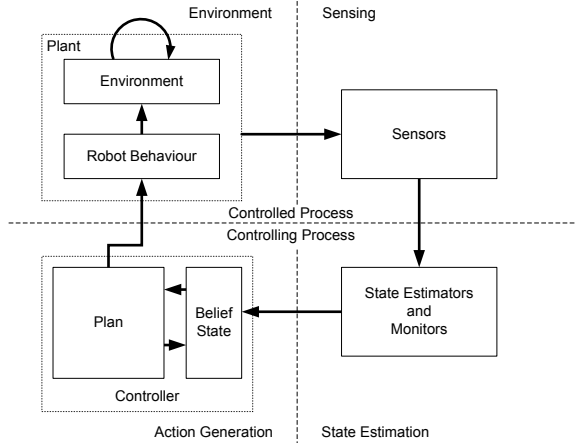


Figure 1. Block diagram of our dynamic system model for plan-based control.

### 2.1. Classical Planning

Most of these techniques are based on the problem-space hypothesis [16]: they assume that problems can be adequately stated using a state space and a set of discrete and atomic actions that transform current states to successor states. A solution is an action sequence that transforms a situation described by a given initial state into another state that satisfies the given goal. Planning usually proceeds by finding operations that have the desired effects, and by making the preconditions of these into subgoals. Plan generation is the key inferential task in this problem-solving paradigm.

The representational means are therefore primarily designed to simplify plan generation. Problem-space plans are typically used in layered architectures [3], which run planning and execution at different levels of abstraction and time scales. The models in these approaches are too abstract for predicting all consequences of the decisions made during planning. The planning processes on the other side can not exploit the control structures provided by the lower layer. Therefore they lack appropriate means for specifying flexible and reliable behaviour and plans can only provide guidelines for task achievement.

### 2.2. SRCs and Hierarchical Planning

Structured Reactive Controllers (SRCs) are collections of *concurrent control routines* that specify routine activities and can adapt themselves to non-standard situations by means of planning.

In contrast to classical planning, where actions are assembled in order to attain goals, hierarchical planning is about reducing high-level tasks down to primitive tasks. The goal is usually specified as a high-level task to be performed rather than as a set of conditions that are to be attained. During planning, the high-level tasks are expanded

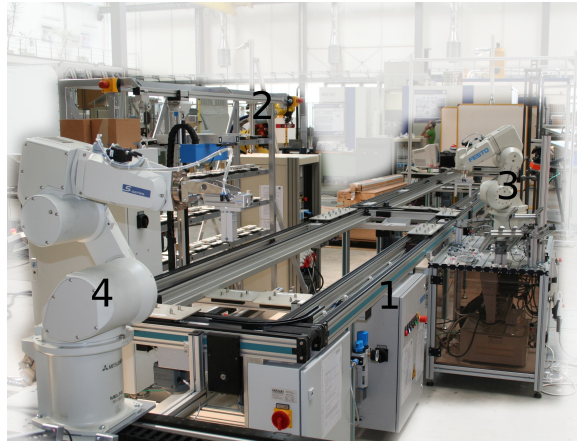
to networks of the lower level tasks that accomplish the high-level task. The methods describing these expansions are called "goal routines" in our system. They map the high-level task to a number of sequential or parallel, partially ordered set of the tasks, until the plan consists only of the primitive tasks that can be directly interpreted and control the actions of the hardware or the simulation setup. This whole workflow is depicted in Figure 4.

Plans in plan-based control have two roles. They are both executable prescriptions that can be interpreted by the system to accomplish its jobs and syntactic objects that can be synthesized and revised by the planner to meet the aspired criterion of utility. Contrary to the problem-space oriented planning approaches, the plan-based approach distinguishes between three different types of inference tasks: the inference tasks that enable the competent execution of given plans, the ones that allow for learning plans and other aspects of plan-based control, and the reasoning tasks. The latter comprise the generation and assessment of alternative plans, monitoring of the execution of a plan, and failure recovery.

The inference tasks are all performed on a common data structure: the plan. Consequently, the key design issues of plan-based control techniques are representational and inferential adequacy and inferential and acquisitional efficiency as key criteria for designing domain knowledge representations [18]. Transferring these notions to plan-based control, we consider the representational adequacy of plan representations to be their ability to specify the necessary control patterns and the intentions of the controlled system. The inferential adequacy is the ability to infer information necessary for dynamically managing, adjusting, and adapting the intended plan during its execution. Inferential efficiency is concerned with the time resources that are required for plan management. Finally, acquisitional efficiency is the degree to which it supports the acquisition of new plan schemata and planning knowledge. To perform the necessary reasoning tasks the plan management mechanisms must be equipped with inference techniques to infer the purpose of subplans, find subplans with a particular purpose, automatically generate a plan that can achieve some goal, determine flaws in the behavior that are caused by subplans, and estimate how good the behavior caused by a subplan is with respect to the overall system's utility model. Pollack and Horty [17] point out that maintaining an appropriate and working plan requires the system to perform various kinds of plan management operations including plan generation, plan elaboration, commitment management, environment monitoring, model- and diagnosis based plan repair, and plan failure prediction.

### 3. Hardware and simulation environment

We are working on the FMS (Festo iCIM 3000) installed by Festo Didactic as displayed in Figure 2. It comprises of three stations interconnected by a conveyor trans-



**Figure 2. Hardware Setup with conveyor(1), storage(2), assembly(3) and mill and turn filler (4) (machines not shown)**

port system equipped with a changeable number of pallet carriers. The transport can only move pallets, which in turn are configured specifically for carrying different product parts. The stations are an automatic storage, a milling and turning station and an assembly station. Each station has a stopper position on the transport system. The mill-and-turn station is equipped with a robot arm acting as a loader mounted on a linear drive (7th axis) while the assembly station uses a fixed robot. Both, the assembly and the mill-and-turn stations are equipped with pallet buffers and feeders. The storage has a capacity of 40 pallet carriers.

The Gazebo-based simulation [9] we use for plan evaluation is set up to mimic the main features of the earlier mentioned real FMS by Festo. We implement the simulation in house since we were expecting difficulties reaching the required functionality with a commercially available simulation software. The problematic issues we identified were the *faster* than realtime performance, the not trivial integration of our B21 autonomous robot and the limited extensibility of new sensors, machines and such. The milling and the turning of parts is not simulated physically, since machining is currently out of our research scope. Gazebo is a part of the Open Source project Player/Stage/Gazebo and represents the 3D simulation part of the package. The physical simulation is based on ODE (Open Dynamics Engine) and the visualisation is based on OpenGL (respectively the OGRE3D engine in the latest Gazebo version). Player provides the package with all the needed communication and hardware abstraction capabilities. Its data exchange is based on the TCP/IP protocol and is used to implement the so-called drivers, which serve as abstract interfaces to the hardware devices and their respective simulation models. By implementing a driver with the same interface to the control code (in our case the RPL language) for both, the simulation and the real hardware, it is possible to switch between them without changing the high level control code. We currently use a self programmed inverse kinematic model implemented

as the player driver for simulation.

#### 4. Integration of our autonomous robotic assistant

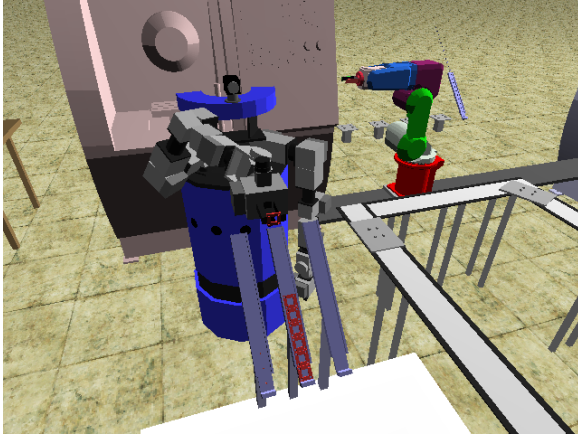


Figure 3. B21 robotic assistant filling a part feeder

We integrated an autonomous robotic assistant into the scenario (See Figure 3, blue robot). It models the real B21 robot equipped with powercube arms that we currently use in other projects. Apart from being very flexible as the assistant, it also serves as a mockup for a cooperating human in the simulation scenario. The robot is able to navigate in the factory environment, fill feeders with parts picked up from other locations and pick up and place parts on the pallets at a conveyor belt stopper. The robot is equipped with cameras and a laser scanner mounted on one of its arm, so that it can generate 3d point clouds of the environment by pivoting the scanner during operation. While we currently use the point clouds for navigation only, it is planned to use rich sensors like the 3d laser scanner for state estimation of the whole factory - such as recognizing when a carrier is stuck on the conveyor belt. We plan to use the 3d scanner to scan machined and assembled parts for quality assurance as a next step.

#### 5. The Reactive Plan Language

The Reactive Plan Language (RPL) was designed by Drew McDermott [10] to allow for the creation of reactive plans for robotic agents. Since it is implemented as a *macro extension* of LISP, RPL plans look very similar to LISP code, and LISP constructs can be used within the plans. RPL itself is a strictly procedural language, unlike LISP. Through the extensions done by our chair, the dynamic typing of LISP is fully supported in RPL. Memory management with garbage collection is done by the LISP interpreter. The basic control concept of RPL mimics the task scheduling loop of operating systems. Parallel tasks are handled so that every active task gets a chance to run. The hierarchy of the active tasks that RPL maintains can be accessed from within the plans. Tasks can observe the

execution state of other tasks and can inspect their local variable bindings. One task could for example explicitly wait for another one to start or finish. Another key feature of RPL is the *modelling of failures and success*. Their handling is implicit in many control structures, such as *'par'*, *'try-all'* or *'pursue'*, which all allow parallel execution of multiple tasks with different treatment of failures.

##### 5.1. Synchronisation

In manufacturing and assembly scenarios, different tasks often need to allocate the same resources, such as a specific robot, transport pallet or buffer space. The resources are therefore modelled with so-called 'valves' which work in a similar manner as mutexes in other programming languages. With 'multi-valves' we added an option for arbitrary access to multiple resources at the same time. A whole set of resources can be requested at once, avoiding deadlock situations where resource dependencies of different tasks intersect. Anonymous requests are an additional advantage, allowing the tasks to request for example a given number of storage places without preference for specific ones but only for a specific content (like  $2 \times \text{'empty' places plus } 2 \times \text{'part no. 101'}$ ).

##### 5.2. Plan exchange during runtime

RPL based manufacturing plans are *exchangeable during runtime*. This is an important precondition since we want to optimize the plans online and exchange them whenever we come up with a better one. We avoid to reset the whole factory when a better plan becomes available, but rather exchange the plans while the production continues. Exchanging plans comes down to just shutting down the current plan and starting the new version. The process of shutting down a task during its execution is called 'evaporation' in RPL. The new plan directly starts to accomplish the given goals from the current state of the system and produced parts. It allocates resources and materials from where they are at the moment, using the integrated belief system with information on the location and availability of parts. It does not matter whether a certain part is at the automatic storage, on the conveyor or already in the buffer of the assembly station, since the plan arbitration modules just pick up the right routines to handle any given state. The hierarchical structure of the plans takes care of the peculiarities since all the plans are formulated in an "achieve goal X" manner. If goal X is already achieved, the respective task terminates, if not, it will take the steps necessary to accomplish its goal. For example, if a requested product is already in the storage and free for allocation by a top level order, the task is done. If not, the plan will check whether an unallocated part is available at the transport or in the buffers. Should no part be found, the systems searches for the preliminary parts in order to carry out their assembly, or even starts the production of the preliminaries and then uses these to build the product.

If a certain task is not modelled in all its detail and therefore an unmodelled system state would occur during

the suspension of a plan, the execution of the subtasks can be protected from evaporation using a scheme similar to a critical section called "evap-protect" in RPL. An example of this technique is the filling of bolts into the bolt carrier. Assume that we do not maintain different part ids for each possible state of the bolt carrier (empty, filled with one .. four bolts), but only ids for the empty and the completely filled bolt carrier. The processes of filling the carrier or removing the bolts during assembly would then need to be protected from evaporation. If the plan gets evaporated, the machine continues to fill the carrier and leaves a defined state after finishing. A problem connected to the plan evaporation is the configuration of low level controllers and hardware. After a task was evaporated, we often have to reconfigure the machine controllers in order to remove all traces of the process that was running before. It often makes sense to "evaporation-protect" a lot of the routines and to model the process in a way that assures that consistent states are reached often instead of trying to recover from the complex left-over states during the evaporation. Similarly, it is often advisable to wait with the start of the new plan until the previous plan's evaporation protections are done and the system is in a defined state. A possible problem occurring when one does not wait for the evaporation protection is the request of an additional part to be produced, since its predecessors are not represented in the belief state anymore (or are even consumed already), but the produced part is not yet registered in it either. We are currently working on an improved solution that would then only freeze the top level orders that were responsible for the currently still running evaporation protected activity, allowing all other orders to proceed without delay after a plan exchange.

### 5.3. Fluents

The belief state is maintained by so-called fluents. Differing from normal variables, fluents have a built-in observer feature and can be combined to a fluent network. With constructs such as (*wait-for fluent*), a plan can wait for a fluent to become true, the statement (*whenever fluent body*) will execute 'body' whenever the fluent becomes true, in an endless loop. The composition of the fluent networks is achieved by combining fluents with a function or operator such as (*or fluent-a fluent-b*) or (*>fluent-a fluent-b*). The resulting fluents' values are re-calculated when one of the underlying fluents changes. All operators and most basic functions are defined for fluents of basic data types.

### 5.4. Designators

Designators are used whenever an object in the real world is referenced by the plan. It is often unpractical or even impossible to address all objects with unique identifiers such as part numbers. In scenarios with numerous similar or identical items, like boxes of screws, we prefer to point out the objects that should be used by the plan by the description of their properties. The following code il-

lustrates this:

(*some entity*

(*type screw*) (*status unused*) (*diameter 5mm*)).

The designator concept does just this. After one of the similar items has been picked, the plan instantiates the designator, adding more information like the coordinates of the specific screw to the description. The object is added as a reference to the symbolical designator description.

Designators can also be used for parameters. Instead of pre-programming default values, the parameters to be set are described by functional aspects and instantiated at the execution time.

By providing a declarative structure of what conditions the objects and parameters must fulfill, the system is able to justify its choice in terms of abstract concepts like "I needed a screw with diameter 5mm".

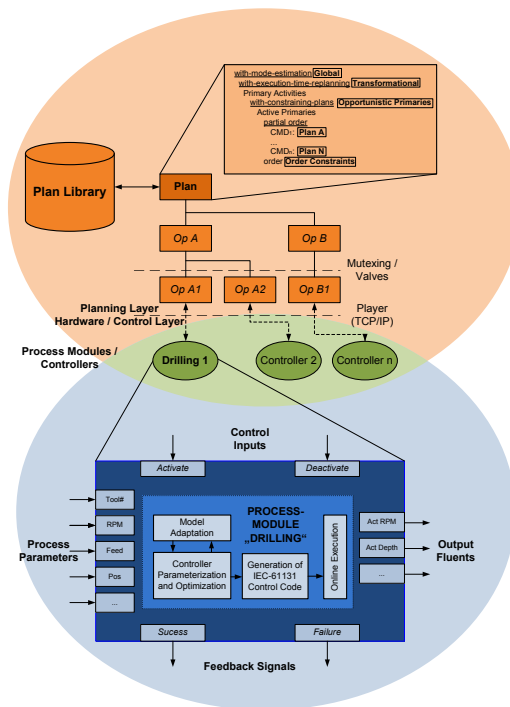


Figure 4. The high level hierarchical plan configures low level hardware controllers

## 6. TRANER

The paradigm of transformational planning assumes a (potentially suboptimal) plan is already available and optimizes it by revisions directly altering the structure of the plan. The promising idea here is to have a set of transformations reflecting the knowledge on how to optimize plans that are to be employed on a wide variety of different plans. We discriminate between "programming knowledge" that is useful independent of the domain, and domain specific knowledge that might be applicable only to the plans of a specific field. Just as the real programmer might start resolving the problem directly and then later

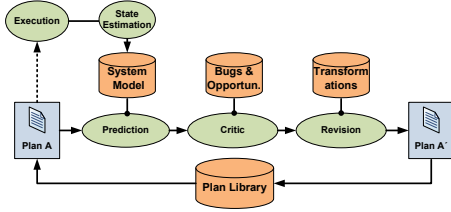


Figure 5. Plans are improved in a cycle of transformation, execution and critic

optimizes the solution, so does the transformational planner. Analogous to the real programmer, optimizations that were applied to programs in the past, shall be remembered and reused in the future. Regarding the domain specific knowledge, we aim at reaching a long-time learning. We expect that the analysis of the previous transformations and the execution of the resulting plans for other products that were made on the same machines in the past will allow to optimize future processes.

The elimination of superfluous storage operations is a good example of the domain specific plan transformation. Instead of bringing a processed part back to the storage before delivering it to the assembly station, the part can be transported directly from one station to the other without being stored intermediately.

It is convenient not to develop the process in such a detail manually, but leave this optimization to the transformational planner after the system has learned its utility.

The overall system comprises of a closed loop of plan transformations and performance criticism as depicted in the Figure 5. We currently use the simulation environment as a model for the evaluation of newly created plan variants. We currently set up the simulation to mimic the real system closely while our future research will explore possibilities to learn execution models that can be used for optimizations without manual modelling.

Transformation rules in TRANER [14] consist of a reasoning component and a rule body describing the syntactic operations on the plan. The reasoning part defines what kind of plans the rule is applicable to, and can consider conditions on the execution traces of previous executions of the input plan. An example is the observation of repeated rolling-in and rolling-out of parts into and from the automatic storage during the last executions. These kinds of conditions help to reduce the branching factor of the tree of automatically generated plans since not all transformation rules will be applied on all plans, but only the once that promise to fix a problem or exploit an opportunity. During the matching of the input plan to the input scheme, the plan structures are bound to variables of the transformation rule by pattern matching. Having checked the applicability of the transformation rule, the planner substitutes parts of the plan and changes its structure as described in the transformation rule. The new plan is then checked in simulation, tested and - after showing higher performance on the real system - set to be the default plan from that on.

## 7. RoLL

The Robot Learning Language (RoLL)[8] and its integration in RPL adds learning functionality to our plans. RoLL is built upon experience classes, learning problem classes and learning systems.

The experiences are directly linked to a formulation of *hierarchical hybrid automata*. These automata reflect the system states, their hierarchy and the transitions between them which are crucial about the learning problem while hiding superfluous complexity. By utilising them, we are able to define which data should be recorded in what manner and when. An example is the learning of parameters for the storage robot controller. The corresponding hybrid automaton models the states where the storage rolls a part in or out, and the data is recorded only when the system is in these states. We then collect data on the distance travelled, the time elapsed and so on. We can also define abstractions of the data, such as calculating the speed from the distance and the time, which will then be stored and made accessible to the learning system by ROLL.

The learning problem classes define how exactly the learned data should affect the execution. We have to define which learned parameter is going to affect which function. Whenever we learn a model of a routine instead of a single parameter, such as a time prediction, we have to define that too.

The definition of the learning system simply signals ROLL which kind of learning algorithm to use. RoLL currently supports decision tree learning with WEKA [26] and neural network learning with SNNS [28].

The main advantage of RoLL is in how easy it becomes to learn within the plan. Without bothering all the technical details, we can simply add learning within a few lines of code. The learning takes place during execution, and if we want so, we can directly use the results in the next program loop.

## 8. Empirical results

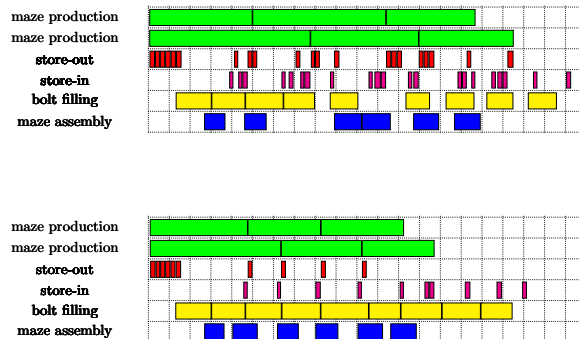


Figure 6. Activities during production of six mazes. Initial process (top) vs. Process after transformation (bottom). (1 grid division = 50 s).

We chose the assembly process of the *play maze* as developed on the real FMS system to exemplify our

approach, as defined for the CoTeSys<sup>1</sup> project Cog-MaSh [27] we are part of. The process consists of two main activities, preparation of a bolt carrier by filling it with 4 bolts at the so called "mill-and-turn station" and the final assembly of the base plate from storage with a cover from the "assembly station" feeder using the bolts picked from the bolt carrier. The bolt carriers are used since the space at the assembly station was insufficient for adding another feeder feeding the bolts directly. They are simple plate-like parts with 4 holes to carry bolts in.

The two subprocesses are implemented independently. The assembly station always waits until a filled bolt carrier is available. The filling of the carriers is run as a maintenance process, always filling all the available empty carriers regardless of the current order situation for play mazes. The initial plan is arranged to wait for a filled bolt carrier to show up in the storage. It will however already allocate the needed two pallet buffers at assembly station and bring the base plate there although the bolt carrier might not yet be available.

As explained in the TRANER section, a transformation was employed to reduce superfluous storage operations. The transformation considers all places of the plan in which a part is brought back to the storage as well as all the places where the task waits for a part to show up in the storage. An additional data structure is introduced serving as a black board for the exchange of parts between different tasks. The code is changed in a way that it announces all the parts currently on the way back to storage on the black board. The waiting tasks are changed so that they check the storage and the black board parallelly to come up with the part they are waiting for. If the part that was originally sent back to the storage is allocated by the waiting task before reaching the storage, the superfluous plan steps are eliminated and the part is sent directly to the final destination without being driven to the storage inbetween.

For evaluation, we arranged the experiments simulating the production of two and six play mazes respectively. Base plates and bolt carriers were located in the automatic storage at the start of the simulations. One of the bolt carriers was provided filled, so the first part could be instantly assembled while the following processes had to wait for newly filled bolt carriers. While the top level order finishes after all maze production tasks are done, the system continues to fill empty bolt carriers in preparation of future orders (see Figure 6).

The results from the experiments show (Table 1) a speedup of about 14% after the plan transformation. Videos of the experiment in simulation as well as the process on the real factory can be found at our website: <http://www9.cs.tum.edu/projects/cogmash/>.

Experiment	Time (s)	Average (s)	Speedup
Two parts	347.05	351.27	
	337.42		
	369.35		
Two parts, optimized	298.76	303.27	13.7 %
	302.21		
	308.85		
Six parts	808.75	817.72	
	814.97		
	835.46		
	811.69		
Six parts, optimized	705.13	697.58	14.7%
	689.04		
	714.04		
	682.09		

**Table 1. Comparison of original and optimized processes for the production of two and six play mazes measured in simulation time.**

## 9. Conclusions and Future work

We showed the possibility and convenience of simulating factory automation hardware physically in the Gazebo simulator. The simulation can be shared with other projects given its open source character. It serves as an essential tool for the development of processes and the underlying language RPL. In order to push the complexity of processes in simulation, we have, in addition to the above presented play maze, developed an abstract product consisting of Tetris-block like components which can be assembled in arbitrary configurations and changed by removal of subblocks in a simulated machining operation. We are currently developing the corresponding planning system that will be able to plan and execute machining and assembly for any arbitrary product fitting into Tetris scenario, taking into consideration the available materials as well as the capabilities of the machines and robots. The most important accomplishment presented in this paper is the transfer of the RPL-written SRCs from autonomous robotics scenarios towards the manufacturing domain. The paradigm of LISP as a programmable programming language as often claimed by LISP enthusiasts is to first develop a language within LISP that is best to represent the problem domain before starting to solve the problems itself. Accordingly, RPL as a language was developed to represent planning problems and their solutions in domains of high uncertainty and inherent error-proneness.

Since the applicability of RPL has been shown only in simulation, we are currently developing bindings to our hardware systems and plan to drive the hardware FMS via RPL in a few weeks. In the event of fabrication, the next step will be the implementation of a new production process for a miniature robotic arms which will have configurable degrees of freedom, orientation of axes and dimensions of links. The process will then also include joint

<sup>1</sup>The research reported in this paper is partly supported by the cluster of excellence CoTESYS (Cognition for Technical Systems), [www.cotesys.org](http://www.cotesys.org).

human-robot actions for tasks like the wiring of servos.

We believe that the Transformational Planning and the Robot Learning Language can be important tools to achieve long and short-term learning and optimization of process plans.

In order to push the system towards *cognitive capabilities*, we are currently integrating a knowledge data-base. We want to allow the system to reason about the right course of actions, taking the machine capabilities, the resources and past experience into consideration. The processes will be shifted from an imperative to a declarative programming style, step by step towards the autonomous planning of machining and assembly of processes for the new product variants solely based on a CAD-like description of the product. When a machining tool breaks, the system shall e.g. find another machine, perhaps of a different type or model, that is able to cut the same features out of the base material. When assembly robots are available, the process shall re-route part of the assembly to the unoccupied robots and thus, together with very flexible fixtures and grippers, the overall performance will be enhanced.

The system we envision will be able to expand its abilities autonomously, while at the same time the processes will be decoupled from the machine setup. The setup specific optimization will then be done by the manufacturing system. As a result, we will observe a profound simplification of production process development and maintenance.

## References

- [1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17(4), 1998.
- [2] M. Beetz, T. Arbuckle, M. Bennewitz, W. Burgard, A. Cremers, D. Fox, H. Grosskreutz, D. Hähnel, and D. Schulz. Integrated plan-based control of autonomous service robots in human environments. *IEEE Intelligent Systems*, 16(5):56–65, 2001.
- [3] P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1), 1997.
- [4] W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lake-meyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2), 2000.
- [5] J. Firby. Adaptive execution in complex dynamic worlds. Technical report 672, Yale University, Department of Computer Science, 1989.
- [6] K. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45(1):173–228, 1990.
- [7] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. In *Proceedings of DARPA Workshop on Innovative Approaches to Planning*, San Diego, CA, 1990.
- [8] A. Kirsch. *Integration of Programming and Learning in a Control Language for Autonomous Robots Performing Everyday Activities*. PhD thesis, Technische Universität München, 2008.
- [9] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, 3:2149–2154 vol.3, 28 Sept.–2 Oct. 2004.
- [10] D. McDermott. A Reactive Plan Language. Research Report YALEU/DCS/RR-864, Yale University, 1991.
- [11] D. McDermott. Robot planning. *AI Magazine*, 13(2):55–79, 1992.
- [12] D. McDermott. Transformational planning of reactive behavior. Research Report YALEU/DCS/RR-941, Yale University, 1992.
- [13] D. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55, Summer 2000.
- [14] A. Müller. *Transformational Planning for Autonomous Household Robots Using Libraries of Robust and Flexible Plans*. PhD thesis, Technische Universität München, 2008.
- [15] N. Muscettola, P. P. Nayak, B. Pell, and B. Williams. Remote Agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.
- [16] A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990.
- [17] M. Pollack and J. Horty. There’s more to life than making plans: Plan management in dynamic, multi-agent environments. *AI Magazine*, 20(4):71–84, 1999.
- [18] E. Rich and K. Knight. *Artificial Intelligence*. McGraw Hill, New York, 1991.
- [19] R. Simmons. A theory of debugging plans and interpretations. In *Proc. of AAAI-88*, pages 94–99, 1988.
- [20] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, Victoria, Canada, 1998.
- [21] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, and J. O’Sullivan. A modular architecture for office delivery robots. In *Proceedings of the First International Conference on Autonomous Agents*, pages 245–252, 1997.
- [22] G. Sussman. *A Computer Model of Skill Acquisition*, volume 1 of *Artificial Intelligence Series*. American Elsevier, New York, NY, 1977.
- [23] G. J. Sussman. *A computational model of skill acquisition*. PhD thesis, Massachusetts Institute of Technology, 1973.
- [24] S. Thrun, M. Beetz, M. Bennewitz, A. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Probabilistic algorithms and the interactive museum tour-guide robot Minerva. *International Journal of Robotics Research*, 2000.
- [25] B. C. Williams, M. Ingham, S. H. Chung, and P. H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, 9(1):212–237, January 2003.
- [26] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2<sup>nd</sup> edition, 2005.
- [27] M. F. Zäh, C. Lau, M. Wiesbeck, M. Ostgathe, and W. Vogl. Towards the cognitive factory. *Proceedings of the 2nd CARV*, 2007.
- [28] A. Zell. *SNNS User Manual*. University of Stuttgart and University of Tübingen, 1998.