Multi-Tasking of Competing Behaviors on a Robot Manipulator

Christian Groth and Dominik Henrich
Angewandte Informatik III
Robotik und eingebettete Systeme
Universität Bayreuth, D-95445 Bayreuth, Germany
E-Mail: {christian.groth | dominik.henrich}@uni-bayreuth.de

Abstract—Behavior-based robotic manipulators are very flexible since they can perform many different tasks without reprogramming. Unfortunately none of the existing approaches is able to interweave multiple manipulation tasks and execute them reliably at the same time, enabling e.g. an intuitive human-robot cooperation. To bridge this gap, we suggest a novel reactive behavior-based architecture. For this, we transfer the multitasking concept from modern computer operating systems to the robot and use a resource based approach to coordination and synchronization. With this, we are able to safely run multiple competing behaviors on a robot. Due to the design of the behaviors, new behaviors can easily be added to the system. The behaviors can be interrupted by other behaviors in order to react properly to a dynamic environment. Later, the interrupted behaviors are resumed in such a way that the system keeps a consistent state.

I. INTRODUCTION AND RELATED WORK

In order to introduce robots into the household domain or small and medium enterprises (SME) they must be able to perform a lot of different tasks without the need of reprogramming every time. We refer to a task as a definite piece of work in order to reach a certain goal. Robot manipulators are usually set up for one specific task, which is time consuming to change. On the other hand mobile robots can react differently with respect to varying situations because of the widespread behavior-based architecture. These can often perform a lot of tasks by combining different basic programs (behaviors), which run concurrently and cooperate or compete to generate an emergent system behavior. So the robot is able to react properly, according to the changing environmental conditions. Behavior-based systems have gained great success on mobile systems [1]. Even some commercial household and entertainment robots use a behavior-based approach or provide possibilities to do so ([2], [3]). Although behavior-based systems are not widespread among manipulating robots, there exist some approaches. They can be divided into two groups.

In the first group, many behaviors perform one specific manipulation task. This approach is as an action fusion of active behaviors. It is achieved if behaviors generate different constraints. Behaviors can be executed in parallel as long as they keep the constraints of other behaviors consistent. An example is the CBFM in [4], where a door is opened with different behaviors. Another example is [5], where various coworker scenarios are demonstrated, like holding a box or a light source while some constraints are kept.

In the second group, there is one behavior for each task or sub task. This is used in [12], where different manipulating behaviors on a mobile manipulator perform interactions with a human, like hugging or playing a game. In [13] a mobile manipulator moves around taking pictures of humans and looking for objects with a pan-tilt camera. Other approaches rely on the Subsumption Architecture [14], like Connell’s soda can retriever [15] or Edsingers two-armed manipulator [16]. The two-armed robot can grasp objects with human help, stack them inside each other, and deploy them. In [17] a behavior-based manipulator is used to write letters on a wall. Every behavior represents a letter or a part of a letter. Another approach applies so called policies, which are often used with Programming by Demonstration [18]. Here, observations are mapped to actions, i.e., the same observation will always generate the same action.

All of these systems have their drawbacks. Either there are very few resulting manipulation tasks, that can be performed. If more tasks can be performed, they can mostly not be interrupted and resumed. The only systems with interruptable tasks are non-manipulating, i.e. the behaviors do not change the environment. None of the known approaches addresses the problem of several manipulating behaviors, which can safely be interrupted and resumed, such that the behaviors may be completed consistently later on.

In this paper we present a novel approach, where a behavior-based manipulator performs various tasks. All behaviors can be interrupted, resumed, and completed safely. We achieve this by transferring the concept of Concurrent Sequential Processes (CSP) from the computing domain to robotics. Since we deal with manipulating behaviors, which can change the environment, we will have to coordinate their execution. We need to know when we can apply a behavior and which behaviors are more urgent to execute. We also need to know which behaviors can be executed in parallel, without undesired side effects. We apply scheduling techniques to resolve the execution sequence, and we use a resource based synchronization for parallel behavior execution. The system can easily be scaled by adding and deleting behaviors, even at run time.

In the remainder of this paper, we present the design of the behaviors, their execution, and coordination (Section II).
Then we describe the experiments and discuss their results (Section III). Finally, we conclude and discuss future work (Section IV).

II. BEHAVIOR-BASED MANIPULATION

A. Behavior Model

Our approach consists of various behaviors running in parallel on a robot manipulator and performing different tasks. According to [19], we concentrate on the reactive and sequencing layer. There is no planning layer, that estimates or evaluates the result of an action. We concentrate on a mechanism to switch between multiple behaviors on a robotic manipulator. The behaviors need to be in a consistent state, even if they are interrupted or resumed at a later time. Therefore, a behavior is represented by basic stimulus-response mechanism extended by an inner state.

The behaviors react according to stimulations from the environment and the inner behavior status. In every time step the robot determines the behavior, which fits best to the current environmental situation, sensed by observers. An observer is a system component, that senses the environment and provides information about it. Since the robot changes the environment by manipulating behaviors, which can be quite complex and can have temporal or causal dependencies, we need a memory, that is holding the state of a behavior. Each behavior is modeled by a Mealy Machine \( B = [S, C, A, \delta, \omega, s_0, F] \), where \( S \) denotes a set of states, including the initial state \( s_0 \) and the final states \( F \). \( A \) denotes a set of actions and \( C \) a set of conditions. The transition function \( \delta \) is given as \( \delta : S \times C \rightarrow S \) and the output function \( \omega \) is given as \( \omega : S \times C \rightarrow A \). For a behavior to change state from \( s_i \) to \( s_j \), the conditions \( c_k \) with \( k = 0 . . n \) at a transition \( T(s_i, s_j) \) have to match corresponding stimuli \( \gamma_k \in \Gamma \), which are provided by the observers. The states are associated with different information regarding the robot, the behavior execution, and the environment. Examples for conditions are certain poses reached by the robot or certain objects recognized by sensors. The matching function is shown in Equation 1. We define the first condition \( c_0 \) of a transition as the main condition and \( c_1..c_{\text{in}} \) as constraints. Usually the main condition refers to objects, that will be manipulated and the constraints to robotic components, which are necessary to enable the execution. We call the conditions \( c_0..c_{\text{in}} \), which are attached to the transition \( T(s_0, s_j) \) originating from the initial state \( s_0 \) primary conditions. The matching stimuli \( \gamma_{0k} \) for these conditions are called primary stimuli, since these stimuli are responsible for the behavior to start working. The observers generate the stimuli from the environment, which can then be used to match the transition conditions (see Figure 1). In our work, we identify a core set of three observers, which is needed for vision-based robotic manipulation. We use a robot observer and a gripper observer to observe the robot’s intrinsic properties and a camera observer to observe the extrinsic stimulation from the environment.

A robot observer provides information about the robot, thus emitting \( \gamma_R \). It holds information like the joint configurations, joint speeds, tool center point transformation, tool center point velocity etc.

A gripper observer emits \( \gamma_G \). It holds information like the configuration of the gripper, velocities of the fingers or jaws, etc.

The camera observer detects the objects, which are segmented from the background. Each of these detected objects represents a stimulus \( \gamma_c = [\text{pos}, \text{intr}] \), where pos is the object’s extrinsic features, like position and orientation in world coordinates and intr are the intrinsic features like object type, shape, dominant color, etc.

We define a condition \( c \in C \) as \( c = [f_1, f_u] \). The parameters \( f_u = f_{u_0} . . f_{u_n} \) and \( f_1 = f_{1_0} . . f_{1_n} \) set the upper and lower bounds of the features. Therefore we can define a simple matching function

\[
\text{match}(c, \gamma) = \begin{cases} 
\text{true}, & \text{if } \forall i : f_{1_i} < \gamma_i < f_{u_i} \\
\text{false}, & \text{else.} 
\end{cases}
\]

(1)

In the case of camera stimuli, this is tested for every currently sensed object, which is a stimulus \( \gamma_c \). Of course, other matching functions are possible. If the condition is matched by a stimulus, the corresponding action of the transition is executed.

The action \( a \in A \) is a function of the executing elements, their current configuration and the corresponding stimulus of the main condition. It can be a low level action, like the opening and closing of the gripper, a robot movement, or a high-level action, like a trajectory or the grasping of an object. The actions are separated into three classes according to [20]. There are absolute actions, actions relative to the current robot-pose, and actions relative to an object. After the action is completed, the current state is changed. To enable the achievement of different goals, several competing behaviors are executed on the robot. Hereby a complete action is emitted from every behavior at each transition. Many of these behaviors have conflicting goals, which can not just be summed up. So we need a mechanism for consistent behavior execution.

B. Behavior Execution

To enable safe interrupt, change, and resume of the active behavior, even within the execution of an action,
we apply methods taken from modern operating systems.
These are well known concepts within the computing domain
and they are able to handle multiple processes on single-
core platforms. This is analogous to one robot executing
multiple behaviors. To do so, all behaviors are wrapped
into processes [21]. I.e. we look at the former defined
behavior as a program and execute one or more instances
of it as processes. The processes are managed by four
different priority lists, according to their process status:
ready, blocked, active, terminated.
Each process needs resources. The resources $R$ of the system
contain all stimuli $\Gamma$ and all executing components $\zeta$ of
the robot. The resources $R = \Gamma \cup \zeta$ can be divided into two
subsets $R = R_{np} \cup R_p$. Here, $R_p$ denotes all preemptive
resources and $R_{np}$ all non-preemptive. Preemptive resources
can be withdrawn from a process and can be assigned
to another process. Non-preemptive resources cannot be
withdrawn from a process, thus, a lock is kept on the
resource until process termination (see Figure 2). Preemptive
resources are usually the robot arm, the gripper, not manip-
ulated objects, and all kind of information stimuli, since we
can restore their current state. Non-preemptive resources are
all real-world objects, that are manipulated by the robot, so
a former state may not be restore-able. We define the set
of resources held by process $i$ as $H_i$ and the set of free
resources as $V$. Furthermore, there exists a set of resources
$N_{i,k}$, which is needed by process $i$ to execute the action
$a$ of it’s current transition with condition $c_k$. The set of
resources $N_{i,k}$ is a subset of $R$. It is defined by the matching
function of the main condition $c_0$ and the $n-1$ constraints
of the current transition. For every of this conditions $c_k$ with
$k = 0 \ldots n$ we define $H_{i,k} \subseteq H_i$ as the subset of $H_i$ through
$H_i \leftarrow \begin{align*}
\text{match}(c_k, \beta_i) & \rightarrow H_{i,k}.
\end{align*}
$. This means $H_{i,k}$ consists of all resources $\beta_j$ that satisfy the condition $c_k$ and are held by
process $i$. Further, we define analogous $V_k \subseteq V$ as the subset
of $V$ though $V \leftarrow \begin{align*}
\text{match}(c_k, \beta_j) & \rightarrow V_k.
\end{align*}
$, which consists of all free
resources that satisfy the condition $c_k$. Now we can define
$N_{i,k} = V_k \cup H_{i,k}$.
Since there are $n$ conditions for a transition, there may be
some identical conditions. Imagine an action that includes
two similar objects. Therefore we can group the identical
conditions of a transition. Let the transition have groups $G_g$
containing $m_g$ similar conditions $c_g$. Since all conditions in
a group are equal, any element $c_g$ of a group is representative
for the group. For every group $G_g$ let $H_{i,g}$ be the set of the
needed and already held matching resources. Let also be $V_g$
the needed matching and free resources. Using these sets
we can determine the status of a process.
A process is active, if the equation
\begin{equation}
|H_{i,g}| + |V_g|
\end{equation}
for every group $G_g$, that contain $m_g$ conditions of type $c_g$
is satisfied. This means, there are enough resources to satisfy
every condition of every group of the current transition.
The resources can either be already held ($H_{i,g}$) or still
available ($V_g$). Since all conditions are satisfied, the process

A process is ready, if the equation
\begin{equation}
|H_{i,g}| + |V_g| < \bigcup_{h \neq i} H_{h,g} \cap R_p
\end{equation}
for each group $G_g$, that contain $m_g$ conditions of type $c_g$
is satisfied. That means, we can match the conditions with
already held resources ($H_{i,g}$) and free resources ($V_g$), like
in Equation 2. But we would need to withdraw preemptive
resources $H_{h,g}$ from other processes.
A process is blocked, if neither Equation 2 nor Equation
3 can be satisfied.
A process is terminated if the current state $s_c$ of the
inherent behavior has reached the end state ($s_c = F$).
A scheduling algorithm can be used to choose which of the
ready processes is executed. This is discussed in Section II-E.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{process_synchronization}
\caption{Process synchronization by mutex locks on non-preemptive
resources}
\end{figure}

C. Behavior Synchronization

If an active process at time step $t - 1$ is also active
in time step $t$ the current action is kept on execution.
If the active process changes, we need to store the context
of this process and restore the new process context. This
means, we have to store the status of all resources $H_i$, that
are held by process $i$ in the process context. This is straightforward for all kind of information and intrinsic
resources, like the robot configuration, which can easily be
stored. Remember that non-preemptive resources hold locks.
So even if manipulated objects stay in the working area,
no other process can preempt these. Preemptive resources
may be withdrawn from the process. The preemption of
manipulating devices is more expensive since they must be
fully available for other processes. For example, if an object
is held within the gripper, it must be stored reliably, so it
can be restored later. Therefore special areas are provided,
where objects can be deployed. The corresponding deploy
position is also stored.

The context restoring of the process to executed next is
done in reverse order. First eventually deployed objects are
grabbed again and the last pose is restored. If there is an incomplete action, this is resumed and execution is continued. If resources have been withdrawn from the process, then resources matching the corresponding condition within the state machine are acquired again.

Releasing the resource lock on process termination yields to a steadily growing set of resources \( H_i \), held by process \( i \) while execution. The lock is kept that long because of consistency reasons. Usually most of the elements in a robotics application are preemptive. Exceptions are objects, that are manipulated by the robot. We only have to define once, which resources belong to which subset of \( R \) for every behavior.

To allow a temporal coordination of behaviors, we need to know, which behaviors were already applied to a resource. We also need a mechanism telling us, in which order we can apply behaviors to a resource. Therefore we extend the resource’s feature vector \( \gamma_c \). We store which behavior was already applied and how often it was applied. Now we can easily add already applied behaviors as required to a condition \( c \) and we can also define an upper limit how often a behavior can be applied to a resource. For this we define a vector \( X \) of the length of all known behaviors in the system. Whenever a behavior \( i \) acquires the resource, the counter at the corresponding position \( i \) is increased. The extended feature vector of a resource is \( \gamma_c \oplus X \). Analogous, we add lower and upper limits of applied behaviors to the condition limits \( f_l \) and \( f_u \) as defined in Section II-A.

D. Behavior creation

As already stated, there can be more multiple instances (processes) of the same type of behavior to handle different objects in the real world. But sometimes it is not useful to have multiple processes of the same type. For example, two search processes will probably not get a better result than have multiple processes of the same type. For example, two objects in the real world. But sometimes it is not useful to

D. Behavior creation

As already stated, there can be more multiple instances (processes) of the same type of behavior to handle different objects in the real world. But sometimes it is not useful to have multiple processes of the same type. For example, two search processes will probably not get a better result than have multiple processes of the same type. For example, two objects in the real world. But sometimes it is not useful to
III. EXPERIMENTAL RESULTS

To present the experimental results, we first describe the behaviors used and the experimental setup. Then we provide and discuss the experimental results.

A. Behaviors

We use a set of four behaviors for our experiments, which are adapted from everyday housework. In particular, we have chosen a kitchen-based environment, other applications scenarios will need other behaviors. We implemented the following behaviors:

Wipe

If an object of type sponge is present, it is picked up by the robot and it is used to clean the table.

Move(Object o, Destination d)

Different pick-and-place behaviors are available, where an object of type o is transported to it’s destination d.

Stir

If an object of type spoon is present, it is picked up by the robot. Afterwards, if an object of type bowl is present, the robot uses it to stir in the bowl.

Pour(Object o)

If an object of type o is present, it is picked up by the robot. If a bowl is present, it will pour the content of object o into the bowl.

The Wipe as well as the Stir behavior is representative for the use of an object, while the robot moves along a specific trajectory. The Move behavior is parametric in it’s conditions and it’s destination. We use this behavior, because many of the useful tasks a robot can do, are some sort of combined pick-and-place actions, like e.g. sorting objects. Finally there is the Pour behavior, which is also parametric in its object type, because not every object can be grasped and poured in the same way.

B. Test setup

Our experiments are performed on a Kuka LWR IV robot with seven degrees of freedom. The robot is mounted on a table, where all our experiments take place. A Microsoft Kinect is placed near the table to detect the objects. To analyze the images, the Point Cloud Library (PCL) is used. We perform a set of experiments with different settings (Table I). Experiment 1 shows a cooking scenario (Figure 3). A bowl and a spoon are initially provided. After some time, ingredients are added to the scene. The robot shall stir the content of the bowl until ingredients are present. Then it shall grasp and pour them into the bowl. Afterwards the robot shall continue stirring. Experiment 2 shows a cleaning scenario. When a sponge is present, the robot shall wipe the table with it. Whenever objects are placed on the table, the robot shall put them away into a deploy-area. New processes get an offset for the deployment position to avoid collisions. Experiment 3 equals to Experiment 1 with the exception that we use MHRN.

C. Results

The system performs well in the given situations. It reacts fast and correctly to all provided objects. The results of Experiment 1 are shown in Figure 4. On the left the process ID (PID) of the current active process is shown, where PID 0 refers to Stir and PID 1 and 2 are Pour behaviors. If no process is active the PID is set to -3. When a process context is stored the PID is -2, when a context is restored the PID is -1. On the right an overview of the number of processes sorted by their state is given. The amount of ready processes increases with every new recognized object, which matches a primary stimulus. A sponge, which is the primary stimulus of Stir, is present from the beginning. As shown in Figure 3 the robot grasps it. In the meantime a bowl is placed in the scene and the robot starts stirring. At time \( t = 55s \) two cups are added. The system reacts to them. The first blocked Pour process is set to ready because of the existing primary stimulus. Immediately after the first Pour process is not blocked anymore, a second process of the Pour behavior is created, which also becomes ready because of the second cup. Immediately after the second Pour - process is ready another one is created, which stays blocked to be able to react to future stimuli. Due to the higher priority of the Pour processes, the context of Stir is stored. I.e. the spoon is deployed back and the current position is stored. The Pour processes are executed. After both Pour processes are finished (\( t = 120s \)), the context of Stir is restored. I.e. the spoon is grasped again, the stored robot pose is restored, and Stir is resumed.

The results of the second experiment can be seen in Figure 5. PID -3, -2, and -1 have the same meaning as in Experiment 1. While Wipe is active, it is interrupted because of the higher

### TABLE I

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Behaviors</th>
<th>Priority</th>
<th>Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stir</td>
<td>1</td>
<td>HPF</td>
</tr>
<tr>
<td></td>
<td>Pour(cup)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Wipe</td>
<td>1</td>
<td>HPF</td>
</tr>
<tr>
<td>3</td>
<td>Move</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stir</td>
<td>3</td>
<td>MHRN</td>
</tr>
<tr>
<td></td>
<td>Pour(milk bag)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pour(cup)</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4. left: id of the active process (PID) over time \( t \) in Experiment 1; right: amount of ready processes(red line), blocked processes (green line) and terminated processes (blue line) over time \( t \) in Experiment 1
The prioritized Move process, which becomes ready. The process context of Wipe is stored and Move becomes active. After deployment of the cups, the Wipe context is loaded again and the process continues to clean the table at the point it was interrupted. Both experiments show, how versatile the behavior-based approach is. The robot can handle different situations with some very basic behaviors. Figure 6 shows the results of Experiment 3, which is again the cooking scenario but with the MHRN scheduling strategy. The negative PIDs keep the meaning of the former experiments. PID 0 is the Stir behavior, which is slightly modified. Instead of stirring \( n \) rounds, the robot stirs forever. PID 2 refers to the Pour(milk bag) behavior and PID 1 to the Pour(cup) behavior. At time \( t_0 = 25s \) the cup and the milk bag are added to the scene. The corresponding processes become ready and their priorities begin to rise. At \( t_2 = 45 \) and \( t_3 = 135s \) their priority exceeds the priority of Stir and the objects’ contents are poured into the bowl. This scheduling guarantees, that a process that has been suppressed for a long time, because of a lower initial priority, will finally manage to be executed. While our experiments take place in a cooking scenario, the scheduling can be applied to a lot of recurrent behaviors.

D. Experiment discussion

Although we use quite simple hand-written behaviors, they show the principle of the system. Just like on a computer, different tasks, which compete for the existing resources, can be executed on a robot. The behavior-based approach allows adding and removing of tasks, which is needed for future work. The process-based scheduling from operating systems helps to reach reasonable response times and allows consistent switching between the behaviors. The object based memory prevents behaviors from manipulating an object more often than desired.

The outcome of the experiments is quite predictable in the
beginning. And we think this is exactly what the user wants, when having a robot by his side. While many approaches with a behavior fusion mechanism have quite unpredictable outcome, this can be not helpful for the user or even dangerous. Nevertheless, some points are not exactly predictable yet. Looking at Experiment 1, two cups are placed in the scene. Since these are recognized at the same time, it is not predictable for the user, which cup will be poured first. More generally speaking, if two behaviors with the same priority are triggered at the same time, it is not transparent for the user which one will be executed first.

This also leads to the question, how the priorities should be set. In our experiments the priorities are set manually. So the priority scheduling strategy will perform as desired. When it comes to a scenario, where much more behaviors are involved, like the complete laying of a table, this can probably not be useful anymore.

An attempt adjust the priorities automatically, was done with the MHRN scheduling in Experiment 3. But we see, that an automatic setting of priorities has to be examined much more, to enable a fair scheduling. We cannot just take known scheduling algorithms from operating systems. Most of them assume negligible costs for context switching. But when we take a look at Table II, which shows the time consumption for context switching, we recognize this is different in robotic applications. Designing a good scheduling strategy, that allows fair behavior execution, fast response times and low flow times with automatic priority setting will be a key issue for future work. Maybe user interaction can be taken into account for this or statistics of former process runtimes. A decision making component could assist the scheduler, to choose the current best behavior or even to achieve a goal by combining behaviors.

When we take a closer look at Table II, we can recognize more problems in context switching. The time fraction \( t_p \) is the ratio of absolute time for context switching \( t_{abs} \), divided by the whole experiment’s time \( t_c \). In Experiment 1 and 2 \( t_c \) is the time between the first process being active and the last active process becoming terminated. In Experiment 3 we choose \( t_c = 225 \) because Stir will never terminate. Although the storing and restoring works good, it takes time. It obviously takes longer, when the objects are deployed far away from the current position of the tool center point. In our experiments we used the initial object position as the deploy position. But instead the robot should deploy the object to the nearest free position. In Experiment 2 we face the problem, that although the sponge is already lying on the table, the robot deploys it on it’s initial position.

When considering, that the stored tool center point position should be restored again or moving to the current action’s goal directly, one should take a look at Experiment 2. Here, it is crucial to continue the process at the last stored position. Otherwise, the table would not be cleaned completely. So if we do not know anything about the task, we will have to restore the position.

### IV. Conclusions

Most existing behavior-based manipulation systems rely on an action fusion mechanism. A task can only be accomplished, if all behaviors are well coordinated. In this paper a behavior-based robot manipulator was presented, which is able to execute multiple tasks. The system can interrupt and resume the execution of the tasks consistently. This is achieved by transferring well known concepts from the computer operating systems domain to the field of robotics. We introduce processes and scheduling to execute and coordinate the sequence of the behaviors on a robotic manipulator. We use process contexts to switch between behaviors consistently. Therefore the robot is able to do a multitasking of different manipulating behaviors in parallel. A resource based synchronization approach guarantees a consistent execution. The resource based synchronization approach can also be used with different task representations. The approach is not limited to reactive behaviors. Behaviors can also be high-level controller. The main contribution of this work is the ability of a robotic manipulator to consistently switch between multiple manipulating tasks.

The experiments show the flexibility and versatility of the approach. But they also show that further improvements may design better scheduling strategies, that address the demands of robotics applications. It should contain the automatic assignment of process priorities and optimize the context switching. It may also take into account statistics of already executed processes, since everyday tasks are usually recurrent.

The experiments also show, that powerful observers are

<table>
<thead>
<tr>
<th>Experiment</th>
<th># switches</th>
<th>( t_{abs} )</th>
<th>( t_{avg} )</th>
<th>( t_p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>27.58</td>
<td>13.79</td>
<td>16.0 %</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>28.02</td>
<td>14.01</td>
<td>12.4 %</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>60.08</td>
<td>15.02</td>
<td>28.9 %</td>
</tr>
</tbody>
</table>

**Table II**

**Time Consumption in Seconds for Context Switching with the Absolute Time \( t_{abs} \), the Average Time \( t_{avg} \) and the Fraction \( t_p \) of the Overall Experiment Duration.**

![Fig. 6](image.png)
needed. The better these observers are, the more powerful is our approach. If these observers are able to analyze objects, then the degree of similarity of the objects to the conditions within the state machine can also be used to influence the scheduling.

REFERENCES