Hierarchical Behavior Organization*

Hans Utz[1], Gerhard Kraetzschmar[2], Gerd Mayer[1], Günther Palm[1]
[1]*Neuroinformatics, University of Ulm, 89069 Ulm, Germany* {hutz,gmayer,palm}@neuro.informatik.uni-ulm.de
[2]*Fraunhofer AIS, Schloss Birlinghoven, 53754 Sankt Augustin, Germany* gerhard.kraetzschmar@ais.fraunhofer.de

***Abstract*— In most behavior-based approaches, implementing a broad set of different behavioral skills and coordinating them to achieve coherent complex behavior is an error-prone and very tedious task. Concepts for organizing reactive behavior in a hierarchical manner are rarely found in behavior-based approaches, and there is no widely accepted approach for creating such behavior hierarchies. Most applications of behavior-based concepts use only few behaviors and do not seem to scale well. Reuse of behaviors for different application scenarios or even on different robots is very rare, and the integration of behavior-based approaches with planning is unsolved. This paper discusses the design, implementation, and performance of a behavior framework that addresses some of these issues within the context of behavior-based and hybrid robot control architectures. The approach presents a step towards more systematic software engineering of behavior-based robot systems.**

I. Introduction and Motivation

Mobile robots acting in dynamic environments populated by humans and other robots must be able to react quickly to unexpected situations. Behavior-based approaches have been suggested and successfully used to implement reactive robot behavior for almost 20 years now [2], [4], [12]. Nevertheless, implementing a broad set of different behavioral skills and coordinating them to achieve coherent complex behavior is still an error-prone and very tedious task. To more than a few, behavior programming therefore seems to be more of an art than a science.

Most robot programming environments provide little conceptual and tool support for behavior engineering and have been designed without paying much attention to modern software techniques like object-oriented abstraction, multi-threaded programming, or event-based communication, all of which are by now standards in industrial-strength systems and applications programming. Furthermore, few behavior-based approaches provide concepts for organizing reactive behavior in a hierarchical manner, and there is no widely accepted methodology for creating such hierarchies in a systematic way.

This paper presents the *behavior, action pattern, policy* (BAP) framework for specifying hierarchical, event-driven, behavior-based control systems. The framework allows to adopt and integrate many well-known behavior-based approaches, such as those based on subsumption [4], fuzzy control [15], [16], or potential fields [2], but was designed to incorporate concepts of modern software technology, like an event-driven control model. The BAP framework was designed to ease the integration with planning-based methods and to foster the use of learning algorithms. To enable reuse and generalization of behavior libraries in and for different scenarios and robot platforms it is integrated into the MIRO middleware [18]. It is used in very different scenarios, ranging from office delivery tasks to robot soccer and on different commercial platforms as well as on several custom-built robots.

*The work described in this paper was part of the project *Adaptivity and Learning in Teams of Cooperating Mobile Robots* supported by DFG SPP-1125.

II. Design Issues

Designing a set of behaviors producing a particular desired system behavior often proves to be very tricky. The behavior engineer must consider numerous issues related to system integration and interaction with the other system components. The design and implementation of behavior-based systems often appears to have little structure and seems to lack methodology. Programming complex behavior-based robot applications could be significantly improved by developing methods that allow for more systematic development, support prevalent software engineering desirabilities like modularity, functional abstraction, reuse, etc., and reduce development time and effort by providing libraries, middleware systems, and appropriate tools for design, implementation, and evaluation of partial or complete behavior-based systems.

a) Reactivity by supporting concurrent behavior execution: A basic notion of all original behavior-based approaches is their intrinsic concurrency [2], [4]. This allows the programmer to factor out detection and handling of potential failure situations in separate behaviors and to retain a concise formulation of the actual task. The concurrent execution of all behaviors ensures automatic surveillance of potential failure situations. However, some more recent behavior-based approaches have traded concurrent behavior execution in favor of easier behavior sequencing. This puts the burden to ensure timely evaluation of failure conditions back on the programmer. Therefore, we consider concurrent behavior execution as indispensible.

b) Flexibility by allowing different arbitration mechanisms: If more than one behavior is active and producing inputs for motor control, then the behavior outputs are potentially in conflict and must be arbitrated. Different behavior-based approaches, like subsumption, potential field methods, and fuzzy behaviors, mainly differ in what type of output the behaviors produce and what methods are used to arbitrate these outputs. A general-purpose behavior architecture should therefore support the use of multiple arbitration mechanisms.

c) Taskability by supporting behavior sequencing: A weakness of early behavior-based approaches is the difficulty to cope with a wide set of different tasks. Murphy introduces the notion of taskability for describing how easy it is to switch between different tasks [13]. Taskability is facilitated by executing at any time only the minimal set of behaviors necessary to produce the desired system functionality. Upon arisal of certain situations, the system switches to another behavior set. Thus, temporal sequencing of behavior sets is a highly desirable mechanism to provide easy taskability of the robot.

d) Functional abstraction by providing behavior parameterization: In practice, there are many situations where behaviors in different behavior sets are quite similar and differ only in a few parameter settings, like settings for translational and rotational velocities and accelerations, and safety distances to be observed. Thus, there is considerable potential for code reuse, if the behavior architecture supports functional abstraction and concepts for parameterization.

e) Modularity by hierarchical policy specification: Simple temporal sequencing of behavior sets is often insufficient, if the application domain and the task set gets more complex. Therefore, concepts for hierarchically building policies, which implement contingent plans and have proper entry and exit semantics, would greatly simplify the overall structure of the behavior system and increase modularity.

III. The BAP Framework

The acronym BAP stands for ***b**ehaviors*, ***a**ction patterns*, *and* ***p**olicies*, which capture the main concepts of our framework. The next two paragraph introduce the main concepts informally, before we provide more formal definitions further on.

We assume a robot with multiple sensors, which are controlled by sensory processes that read out or interface to the sensors and deliver their data into an observation space. Interpretation, fusion, and integration of sensor data is performed by perceptual processes, which represent their results in a space of state variables. The robot's effectors are controlled via low-level motory processes, the inputs of which make up motor space. Observation space, state space, and motor space together constitute data space, i.e. the set of variables constituting the information available to the robot at any time instant and the basis for any decision making. A special type of perceptual processes are guards, which observe particular conditions on data space, e.g. that an object is visible, or detect state changes, e.g. that an previously visible object has been lost or that a goal state has been reached. Guards signal events, which can be viewed as logical predicates satisfied in a specific situation represented in data space. In the simplest case, events can be represented by binary logical variables; in a more elaborate situation, an event may be represented by a predicate with one or more arguments bound to terms, which can be used to communicate parameters.

Behaviors are mappings from data space to motor space. If several behaviors run concurrently, an arbiter deals with their potentially conflicting outputs. Like guards, behaviors may also signal events. A set of concurrently running behaviors together with a set of guards and an arbiter make up an action pattern, which can be viewed as a primitive, coherent action with safeguards. By combining a set of action patterns together with an event-based transition relation we get a policy, which defines temporally coherent sequences of action patterns. The transition relation allows for the specification of quite complex control structures, including sequences, conditionals, and loops. Policies can take the place of action patterns in policy definitions, thereby permitting hierarchical specifications of policies. The temporal extent of action coherence increases, the higher we move up in policy hierarchy.

A. Data Spaces

Within the BAP framework, behaviors select actions based on information contained in an internal representation called sensory space or data space, which can be divided into the two distinct subspaces observation space and state space. Observation space is the n_o-dimensional Cartesian product $\mathcal{O} = \otimes_i^{n_o} O_i$ over all available n_o sensorial dimensions $\mathbb{O} = \{O_i | 1 \leq i \leq n_o\}$. State space is the n_s-dimensional Cartesian product $\mathcal{S} = \otimes_i^{n_s} S_i$ over n_s state dimensions $\mathbb{S} = \{S_i | 1 \leq i \leq n_s\}$. Data space is the n_d-dimensional Cartesian product $\mathcal{D} = \otimes_i^{n_d} D_i$ over n_d data space dimensions $\mathbb{D} = \{D_i | 1 \leq i \leq n_d\}$, where $n_d = n_o + n_s$ and $\mathbb{D} = \mathbb{O} \otimes \mathbb{S}$. Motory space is the n_m-dimensional Cartesian product $\mathcal{M} = \otimes_i^{n_m} M_i$ over all available n_m motory or effectory dimensions $\mathbb{M} = \{M_i | 1 \leq i \leq n_m\}$. Parameter space is the n_p-dimensional Cartesian product $\mathcal{P} = \otimes_i^{n_p} P_i$ over n_p parameter dimensions $\mathbb{P} = \{P_i | 1 \leq i \leq n_p\}$. The domains for any of the dimensions in the previously defined spaces may be finite or infinite, and discrete- or continuous-valued. Finally, event space is a finite set $\mathcal{E} = \{e_i | 1 \leq i \leq n_e\}$ of n_e discrete events.

B. Behaviors

A *behavior* is a mapping from data space to motory space and events (see Fig. 1), formally denoted by $b : \mathcal{D}(b) \mapsto \mathcal{M}(b) \times \mathcal{E}(b)$, where $\mathcal{D}(b)$ is some d_b-dimensional subspace of $\mathcal{D}$, $\mathcal{M}(b)$ is some m_b-dimensional subspace of $\mathcal{M}$, and $\mathcal{E}(b)$ is a finite subset of $\mathcal{E}$. The set of available behaviors in any particular implementation is referred to as $\mathcal{B} = \{b_1, \ldots, b_m\}$.

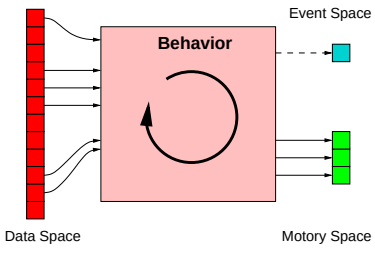

Fig. 1. Structure of a behavior.

Usually, behaviors are parameterized mappings, which we refer to as *behavior schemata*. A set of n available behavior schemata in an actual implementation is denoted by $\mathcal{BS} = \{BS_1, \ldots, BS_n\}$. Behavior schemata must be instantiated to produce actual behaviors that can be used in action patterns. Instantiation requires providing parameter settings.

We do not make any assumption about the kind of mapping a behavior implements nor about how a behavior is actually implemented. Our definition allows to capture most of the known behavior implementation methods, including augmented finite state automata [4], potential field methods [2], and fuzzy behaviors [15], [16].

C. Arbiters

Concurrent execution of behaviors was identified as one of the design requirements. When a set $\{b_1, \ldots, b_n\}$ of behaviors is executed concurrently, their outputs may be in conflict. This is obviously but not exclusively the case, if they share dimensions in motory space. In order to resolve such conflicts and to ensure some coherence in the output delivered to motory space, an *arbiter* process (Fig. 2) is defined, which takes the outputs of all behaviors and maps them to motory space: $f : \otimes_{i=1}^{n} \mathcal{M}(b_i) \mapsto \cup_{i=1}^{n} \mathcal{M}(b_i)$. An arbiter can be viewed as a fusion process, which reduces the dimensionality of the output space of all concurrent behaviors from $\sum_{i=1}^{n} \|\mathcal{M}(b_i)\|$ to $\| \cup_{i=1}^{n} \mathcal{M}(b_i)\|$.

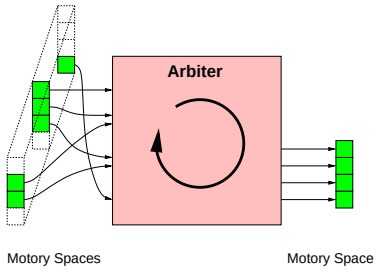

Fig. 2. Structure of an arbiter.

Similar to behaviors above, arbiters may be parameterized, and we can define a set $\mathcal{FS}$ of available arbitration schemata FS and instantiation of arbitration schemata $inst_F$ in an analogical fashion. The set of available arbiters is denoted by $\mathcal{F}$.

As for behaviors, our definition of arbiters does not constrain the behavior designer to a particular arbitration method. Some well-known arbitration methods are static or dynamic priorities, subsumption, superposition of potential fields [2], or fuzzy inference and defuzzification [16]. However, dependencies exist between the used arbitration scheme and the types of behaviors usable with it; e.g. when using fuzzy arbitration the behavior outputs are usually required to be fuzzy variables.

D. Guards

Guards can be viewed as a special class of behaviors which only detect certain conditions on data space and generate *events*, but do not produce any output for motory space. This is formally denoted by $g : \mathcal{D}(g) \mapsto \mathcal{E}(g)$, where $\mathcal{D}(g)$ is some d_g-dimensional subspace of $\mathcal{D}$ and $\mathcal{E}(g)$ is a subset of $\mathcal{E}$. The set of available guards is denoted by $\mathcal{G}$.

Just as behaviors, guards can be parameterized mappings (guard schemata) and we can define a set $\mathcal{GS}$ of available guard schemata GS and instantiation $inst_G$ of guard schemata in an analogous way.

E. External Events

Events may also be generated by processes outside of the behavior engine, e.g. by sensor reading processes, motor control processes, or cognitive processes. This is useful for interfacing the behavior architecture with other system components and for overall system integration. Formally, we denote these events by a set $\mathcal{E}_{\triangleright} \subseteq \mathcal{E}$. These events are treated just like events generated inside the behavior architecture.

F. Action Patterns

An *action pattern* consists of a set of behaviors, a set of guards, and an arbiter, all of which are concurrently executed. An action pattern is formally denoted by a triple $a = \langle B, G, f \rangle$, where $B = \{b_1, \ldots, b_m\} \subseteq \mathcal{B}$, $G = \{g_1, \ldots, g_n\} \subseteq \mathcal{G}$, and $f \in \mathcal{F}$, i.e. $a = \langle B, G, f \rangle \in 2^{\mathcal{B}} \times 2^{\mathcal{G}} \times \mathcal{F}$. When necessary, we distinguish action patterns and their components by indexing them appropriately, as e.g. in a_1 and B_{a_1}.

An action pattern represents a controller for the autonomous mobile system (see Figure 3). Its behaviors, guards, and the arbiter form the emergent control loop. Action patterns allow for monitoring and control of many different sensory and effectory capabilities in a structured and modular way and ensure timely responses (reactivity) via concurrent execution of its constituents.

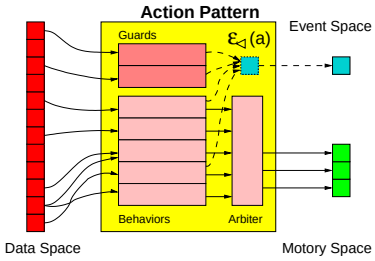

Fig. 3. Structure of an action pattern.

For the execution of different tasks the capability of executing different action patterns is necessary. The set of available action patterns is denoted by $\mathcal{A} = \{a_i | 1 \leq i \leq k\}$. At any time, *only a single action pattern is active*. Behaviors, guards, and arbiters are called active, if they belong to the active action pattern. Thus, the action pattern is the central concept for capturing concurrent execution of behavior-producing functionality (and thereby taking care of the reactivity requirement), while task sequencing will require to switch between action patterns.

The sets of behavior and guard schemata, from which the sets of behaviors and guards of two different action patterns are instantiated, may be the same, partially overlapping, or disjunct. The arbiter can also be the same, a different instance of the same arbitration schema, or a completely different one. Within the scope of this paper, a single arbiter always jointly arbitrates all motory outputs. However, the framework could be quite easily extended to allow for multiple arbiters, which might e.g. arbitrate distinct subsets of motory space dimensions.

Of further importance is the set of events potentially signalled by the behaviors and guards of an action pattern, also called internal events, which we denote by $\mathcal{E}_\triangleleft(a) = \bigcup_{i=1}^{m} \mathcal{E}(b_i) \cup \bigcup_{j=1}^{n} \mathcal{E}(g_j)$. The set of events possibly signalled while an action pattern is active consists of the events signalled either internally or externally: $\mathcal{E}(a) = \mathcal{E}_\triangleleft(a) \cup \mathcal{E}_\triangleright$. Whenever an (internal or external) event is signalled, the execution of an action pattern is terminated.

G. Transitions

After defining the basic concepts for producing behaviors, arbitrating conflicting behavior output, generating events for termination conditions, and the basic activity unit for concurrent execution, we can now tackle task sequencing. Assume we have some set $\mathcal{A}$ of available action patterns, for each of which we know the set $\mathcal{E}(a)$ of possibly signalled events.

A *transition* specifies the successive action pattern, if an active action pattern is terminated by a specific event, and is formally denoted by a triple $t = \langle a_i, e_j, a_k \rangle$ with $a_i, a_k \in A$ and $e_j \in \mathcal{E}(a_i)$. a_i is called the source, a_k the target of a transition. The event e_j should be one that is either produced internally by the action pattern a_i or an external event; otherwise a transition will never be performed.

For sets $T = \{t \mid t = \langle a_i, e_j, a_k \rangle\}$ of transitions a consistency criterion can be defined: $\forall t, \widehat{t} \in T : [(a_i = \widehat{a_i} \wedge e_j = \widehat{e_j}) \rightarrow a_k = \widehat{a_k}]$. I.e., the successive action pattern is uniquely defined. Loops may be defined, i.e. transition $t = \langle a_i, e_j, a_i \rangle$ may have the same action pattern as source and target. This is useful to re-initialize all behaviors and guards of an action pattern upon occurrence of a specific event. Insofar, as a direct loop masks an event and enforces continuation with the currently active, but resetted action pattern.

A consistent set T of transitions uniquely defines a family of partial discrete functions $ev_{a_i} : \mathcal{E}(a) \mapsto \mathcal{A}$ with $ev_{a_i}(e_j) = a_k$ iff $\langle a_i, e_j, a_k \rangle \in T$. Each of the functions ev_{a_i} specifies a local event handler for a single action pattern a_i. If an event handler function ev_{a_i} is total, i.e. if $dom(ev_{a_i}) = \mathcal{E}(a_i)$, the function is said to *fully cover* the event set of a_i. In this case, an appropriate reaction is defined for each event possibly generated while a_i is active. This criterion is often too constraining, because the behavior designer may not know of all externally created events. A slightly less constraining criterion is the following: If an event handler function is defined such that its domain covers at least all internally generated events, i.e. $dom(ev_{a_i}) \supseteq \mathcal{E}_\triangleleft(a_i)$, the function is said to *locally cover* the event set of a_i.

For further reference, we define for a given set A of action patterns and a given set $\mathcal{E}_\triangleright$ of external events the *set of possible transitions over A and $\mathcal{E}_\triangleright$* as follows: $\mathcal{T}(A, \mathcal{E}_\triangleright) = \{\langle a_i, e_j, a_k \rangle \mid a_i, a_k \in A \wedge e_j \in \mathcal{E}(a_i)\}$.

H. Transition Patterns

In addition to transitions between specific action patterns, we define a generic form of transitions, called *transition patterns*: $t^* = \langle *, e_j, a_k \rangle$ with $a_k \in \mathcal{A}$ and $e_j \in \mathcal{E}$. Such a transition pattern actually represents a whole set $\widehat{T} = \{\langle a_i, e_j, a_k \rangle | a_i \in A\}$ of transitions for some set A of action patterns, which is appropriately determined by the context in which the transition pattern is specified (see further below). A transition pattern is very useful, if within some context the occurrence of a particular event should always cause the execution of the same action pattern. For example, events indicating a "*reset*" or "*emergency stop*" situation would certainly qualify for that. In the RoboCup domain, referee calls are good examples for using transition patterns.

Similar to transitions, a consistency criterion can be defined for a set $T^* = \{t^* \mid t^* = \langle *, e_j, a_k \rangle\}$ of transition patterns: $\forall t^*, \widehat{t^*} \in T^* : [(e_j = \widehat{e_j}) \rightarrow a_k = \widehat{a_k}]$. This constraint allows for each event only a single transition pattern and therefore ensures that the reaction to the occurrence of an event is uniquely defined. A consistent set T^* of transition patterns uniquely defines a partial discrete function $ev_* : \mathcal{E} \mapsto \mathcal{A}$ with $ev_*(e_j) = a_k$ iff $\langle *, e_j, a_k \rangle \in T^*$. This function specifies an event handler for all transition patterns of a consistent set T^* of transition patterns.

Like for transitions, we define for a given set A of action patterns and a given set $\mathcal{E}_\triangleright$ of external events the *set of possible transition patterns over A and $\mathcal{E}_\triangleright$*: $\mathcal{T}^*(A, \mathcal{E}_\triangleright) = \{\langle *, e_j, a_k \rangle \mid a_k \in A \wedge e_j \in \mathcal{E}(a_i)\}$.

I. Flat Policies

The combination of action patterns and transitions allows us to specify a higher-level robot controller, which does not only handle a particular task like an action pattern, but complex task sequences. Such a higher-level robot controller is called a *flat policy* and formally specified by a quadruple $p = \langle A, T, T^*, a_0 \rangle$, where $A \subseteq \mathcal{A}$ is a set of action patterns (instances), $a_0 \in A$ is a uniquely determined start pattern, $T \subseteq \mathcal{T}(A, \mathcal{E}_\triangleright)$ is a consistent set of transitions over A, and $T^* \subseteq \mathcal{T}^*(A, \mathcal{E}_\triangleright)$ is a consistent set of transition patterns over A. Informally speaking, a flat policy consists of a set of action patterns, each of which is associated with a local event handler function taking care of transitions with the action pattern as source, together with a policy-level event handler for transition patterns, and a dedicated start pattern, which is executed upon activation of the policy. The targets of both transitions and transition patterns must be elements of the flat policy's set of action patterns.

The flat policy in Figure 4 illustrates the concepts and shows how simple task sequencing, conditional execution of contingent activities, and iteration can be achieved.

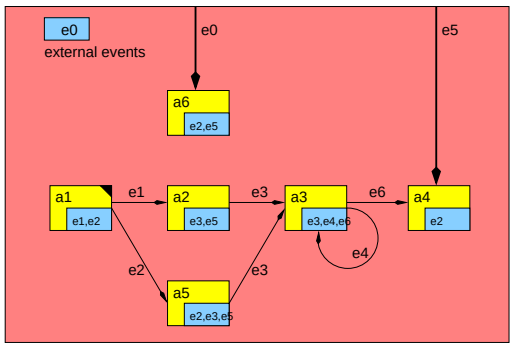

Fig. 4. Example of a policy with sequencing, conditional execution, and iteration of tasks. The start pattern a_1 is marked with a black triangle. Internal event sets are illustrated by the blue boxes in action patterns. Transition patterns are drawn as thicker arrows from the policy border to one of its components.

Although we require both the set of transitions and the set of transition patterns to be consistent, this does not necessarily hold for their union. The definitions of transitions and transition patterns allow for a situation where $\exists t \in T, \widehat{t} \in T^* : e_j = \widehat{e_j}$, possibly with $a_k \neq \widehat{a_k}$, i.e. there is both a transition and a transition pattern specifying a reaction to a specific event. This problem is solved by the following conflict resolution rule: *Transitions take precedence over transition patterns.* More formally, there is a partial order $\prec$ defined over $T \cup T^*$ as follows: $\forall t \in T, \widehat{t} \in T^* : t \prec t^*$ iff $e_j = \widehat{e_j}$.

The enclosing (flat) policy defines the appropriate context for transition patterns as already referred to previously. In fact, within a flat policy p, each transition pattern $\langle *, e_j, a_k \rangle$ is equivalent to a *set* of transitions $\{\langle a_i, e_j, a_k \rangle \mid a_i \in A\}$, i.e. we could replace a transition pattern by a set of transitions, one from each of the policy's action patterns as source and all of which sharing the event and the target, unless there is already a different transition specified for the source and the event. Thus, a policy p is equivalent with a policy $\widetilde{p}$, formally $p = \langle A, T, T^*, a_0 \rangle \equiv \langle A, \widetilde{T}, \varnothing, a_0 \rangle = \widetilde{p}$, where $\widetilde{T} = T \cup \{\langle a_i, e_j, a_k \rangle \mid a_i \in A \wedge \langle *, e_j, a_k \rangle \in T^* \wedge \forall a_l \in A : \langle a_i, e_j, a_l \rangle \notin T\}$.

Flat policies are terminated, if an event occurs that is not covered either by a transition with the currently active action pattern as source or by a transition pattern. Formally, the set of events terminating a policy, also called *unhandled events of a policy*, is determined by $\mathcal{E}(p) = (\bigcup_{i=1}^{n} (\mathcal{E}(a_i) \setminus dom(ev_{a_i}))) \setminus dom(ev_*)$. Note that the conflict resolution rule reappears here in form of parentheses ensuring the correct precedence of set operations.

If for any event possibly occurring during execution of a policy p there is always an appropriate transition specified, i.e. if $\mathcal{E}(p) = \varnothing$, the policy will never be terminated. In this case, we say that the policy is *closed*. This can be achieved for example by ensuring that each event handler function ev_{a_i} associated with an action pattern a_i locally covers the event set of the action pattern, and that the policy-level event handler function ev_* handles (at least) all external events. In the less constrained case, where the policy handles at least all locally generated events, i.e. $\mathcal{E}(p) \subseteq \mathcal{E}_\triangleright$, the policy is said to be *locally closed*.

For further reference, the set of available flat policies over $\mathcal{A}$ is denoted by $\mathcal{P}_f = \{p = \langle A, T, T^*, a_0 \rangle \mid A \subseteq \mathcal{A} \wedge T \subseteq \mathcal{T}(A, \mathcal{E}_\triangleright) \wedge T^* \subseteq \mathcal{T}^*(A, \mathcal{E}_\triangleright) \wedge a_0 \in A\}$.

J. Hierarchical Policies

The concepts introduced so far cover all design requirements except for modularity by hierarchical policy specification, which we will now introduce. The basic idea is to also allow (sub)policies in the place of action patterns in the definition of policies. In order to do that, a hierarchy of *activity units* $\mathcal{U}_i$ is specified. Level 0 activity units are just the available action patterns: $\mathcal{U}_0 := \mathcal{A}$. Level 1 activity units extend $\mathcal{U}_0$ by the available flat policies defined above: $\mathcal{U}_1 := \mathcal{U}_0 \cup \mathcal{P}_f$. Level n activity units are recursively constructed by adding level n policies (defined below) to the level $n-1$ activity units: $\mathcal{U}_n := \mathcal{U}_{n-1} \cup \mathcal{P}_{n-1}$. Sets of possible transitions and transition patterns are now defined equivalently over sets of activity units: $\mathcal{T}_i(U_i, \mathcal{E}_\triangleright) = \{\langle u_i, e_j, u_k \rangle \mid u_i, u_k \in U, e_j \in \mathcal{E}(u_i)\}$ and $\mathcal{T}_i^*(U_i, \mathcal{E}_\triangleright) = \{\langle *, e_j, u_k \rangle \mid u_k \in U, e_j \in \mathcal{E}(u_i)\}$. With these definitions, it is now straightforward to define hierarchical policies: $\mathcal{P}_0 := \mathcal{P}_f$ and $\mathcal{P}_n := \{\langle U, T, T^*, u_0 \rangle \mid U \subseteq \mathcal{U}_n \wedge T \subseteq \mathcal{T}_n(U, \mathcal{E}_\triangleright) \wedge T^* \subseteq \mathcal{T}_n^*(U, \mathcal{E}_\triangleright) \wedge u_0 \in U\}$. Hierarchical policies are illustrated in Figure 5. Note, that

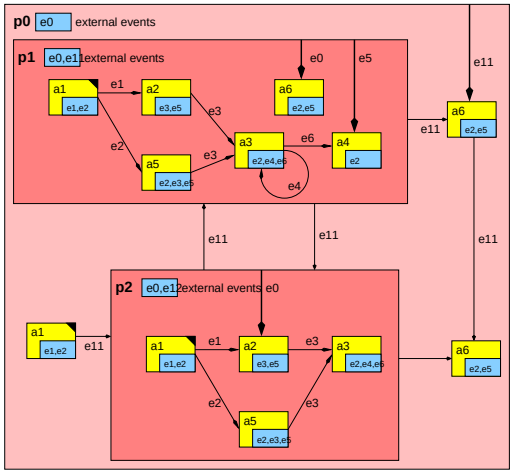

Fig. 5. Example of a hierarchical policy.

both action patterns and policies can be embbeded multiple times in different higher-level policies, thereby facilitating code reuse. However, both direct and indirect loops and recursion are excluded, i.e. the hierarchy of policies is indeed well-defined.

IV. Uses of the BAP Framework

The BAP framework can be used and applied in various ways. Often, behavior-based robotic systems are developed in a *bottom-up* manner:

1) The developer can start with developing several primitive behaviors, each of which would concern only a particular aspect of a more complex action, e.g. following a wall, reacting to obstacles appearing in the robot's path, etc. As pointed out, different behavior-based concepts and theories could be applied in implementing these primitive behaviors.
2) The next step would combine several behaviors and add an arbiter to build an action pattern, which presents a coherent representation of some higher-level action. By limiting concurrent execution to the behaviors, guards, and the arbiter of an action pattern, the conceptually and intellectually challenging task of understanding and controlling concurrent process execution is kept to a reasonable level and can still be handled by the programmer.
3) By adding guards the programmer can formalize conditions for terminating an action pattern or situations that are better handled by another action patterns or require the attention of some higher cognitive level.
4) If not developed in a generic matter right away, the programmer should then review the behaviors, guards, and arbiters developed so far and find suitable functional abstractions. Thereby, we get a set of behavior/guard/arbiter schemata, which can be instantiated and appropriately parameterized whenever needed in an action pattern.
5) Once a set of action patterns is developed, their sequencing can be defined by adding transitions between them, thereby yielding a flat policy.
6) As policies are treated just like action patterns, it is easy to add more and more functionality in a stepwise manner, by organizing action patterns and policies in a hierarchical manner.

An alternative approach to apply the BAP framework is to use it in a *top-down* manner: Much like a hierarchical task planner [5], we can successively divide a complex task into smaller subproblems (policies) and define the transition conditions between them, until we have reached a suitably specific level of action descriptions, which we can directly implement via action patterns. This approach suggests that it is also possible to use the BAP framework in combination with reactive plan languages like RPL [3]. RPL plans could be transformed into BAP policies almost automatically, and can then be conveniently executed even on distributed or networked robot systems and robot teams.

V. Implementation and Tool Support

The architecture discussed above was implemented as part of MIRO, our Middleware for Robots [18], as an extensible C++ white box framework. MIRO is a CORBA-based middleware architecture for autonomous mobile robots. It provides generalized sensor and actuator interfaces for various robot platforms as well as higher-level frameworks for robotic applications.

A. Control Loop Evaluation

A central concept of the framework is that behaviors run asynchronously in parallel. The BAP framework inverts the control flow by defining a behavior base class with virtual methods (OO callback hooks), for starting, stopping and reinitialization, as well as for calculating the output for a single iteration of the control loop of a behavior. The behavior engine runs its own control loop and calls the different behaviors instances as specified in the configuration of the action pattern. Different concurrency models exist that can be used simultaneously by different behaviors. They can either be preemptively multitasked or triggered by timers. In the later case, each behavior can define its own pace, at which the evaluation of the control loop is triggered. Additionally, behaviors can be triggered by events. This allows behavior control loops that are driven by the occurrence of sensory events, like the delivery of an infrared distance sensor measurement, or the activation of a bumper. As a consequence, the different behaviors do not have to run altogether at the same pace, but can choose their own, adequate evaluation rhythm.

B. Policy Configuration

The configuration of behaviors, arbiters, action patterns, transitions and subpolicies into a policy is a substantial engineering effort in itself. The implemented framework does not force such configurations to be hard wired within the source code, but enables configurations (in particular parameterizations) to be separated out into a configuration file using an XML-based grammar. This allows for fast and easy reconfiguration of a policy without recompilation, which is especially useful during development.

C. Dynamic Reconfiguration

Static configurations of a robot's actuatory capabilities are sufficient, as long as all of the robots actions can be defined a priori. But skill learning or the interaction with symbolic AI planning make it desirable to overcome the limitations of static configurations.

In BAP, there are two levels of dynamic reconfiguration of a policy. The first level is the reconfiguration of the parameter sets of behaviors and arbiters. The second is addition and deletion of behaviors, action patterns, subpolicies, transitions and transition patterns. The first level of reconfiguration is fairly straightforward from the implementations perspective. Behaviors and arbiters are already defined to work on different parameter sets within different action patterns. Therefore, the change of a parameter set in a currently active action pattern is equivalent to a transition to the same action pattern with the new parameter set. The second level of reconfiguration is conceptionally well-defined, but includes subtle locking issues to prevent actions from e.g. deleting a currently executed action pattern.

D. Tool Support

An environment for behavior programming should also provide tools to support and assist with tasks, which are

time-consuming, error-prone or of boring, mechanical nature. A good toolkit can contribute to the concrete solution of a problem as much as a well-founded theory for backing up the solution approach. The BAP framework therefore provides infrastructure, that facilitates the development, debugging and maintenance of behaviors, action patterns, and policies.

A central tool for development support, the framework offers the *Policy Editor* a GUI-based editor for the generation of static policies. It supports graphical editing of the action pattern graph as well as generic, type-safe editing of the behavior parameters for each action pattern. New behaviour schematas can be easily added to the policy editor, by specifying their name, their parameters and their events in an XML-based description language. The descriptions are also used for automatic code generation, enforcing consistency between the editor an the actual implementation. Such specifications of behaviours also allow to derive the set of potentially signalled events of action patterns $\mathcal{E}_{\triangleleft}(a)$ and (sub-)policies $\mathcal{E}_{\triangleleft}(p)$. Along with the specification of the set of external events $\mathcal{E}_{\triangleright}$, this allows the editor to ensure the consistency of transition sets and to track the global and local closedness of policies and subpolicies.

Operation and debugging of policies with multiple robots in a distributed environment is supported by various means. The behavior engine provides an interface for remote initialization, starting and stopping of policies that is used by the policy editor as well as by remote control panels. So different parameter sets or entirely new policies can be sent (i.e. from the *Policy Editor*) to a running robot without even stopping its control program.

VI. Application

A. The BAP software framework

The BAP framework is used successfully in a robot indoor navigation office scenario as well as in the highly reactive robotic soccer scenario of the middle-size league of RoboCup. The basic BAP framework is in charge since the late 90s. A theoretically motivated differentiation between internal and external events was introduced in 2002. As a proof of concept, generalized behaviors were written for all platforms currently supported by the middleware MIRO, which include the B21 platform by RWI, the Pioneer series by ActivMedia and two generations of custom-built soccer robots.

The flexibility of the BAP framework integrated with the middleware architecture is best shown by the fact, that it enabled us to play robot soccer with a soccer team of robots of two structurally different robot platforms, using the identical set of behaviors and policies, that only differed in the parameterization of the behaviors.

B. Specifying Complex Tasks with BAP

The power, expressiveness and simplicity of the BAP method for specifying complex task for autonomous mobile robots is best illustrated by a real world example. The most sophisticated use case for the BAP framework so far is its application in the RoboCup environments. In the 2003 version of our soccer code, a flat policy describing the complete behavior of a field player consisted of 13 behaviors which were used by 20 action patterns. An action pattern is typically composed of 4 to 5 behaviors. The policy graph was connected by 70 transitions.

As the interaction of the robots with the referee is very limited at the moment, and the strongly reactive nature of the scenario imposes little need for other external event sources like higher level reasoning systems, the number of external transition messages is limited to 4 within this policy: Kickoff, Opponent Kickoff, Stop and Formation. These are handled by transition patterns.

Although a graph of 20 nodes and 70 edges seems not to be very impressive, it is large enough so that it becomes very difficult to fully grasp its behavioral logic and to maintain and extend it. The flat policy can be naturally structured into 7 different subpolicies, that represent the different tasks of the flat policy. An 8th policy is actually a second instance of one of the 7 policies reached from a different context. The hierarchical grouping also allows to define a previous transition message as a transition pattern, as it always has a uniform successor pattern in the different task contexts. As the hierarchically structured policy has the same semantics as the flat one, the number of concrete action patterns and transitions is actually the same. Nevertheless the savings in the specification are obvious. Three action patterns could be saved by the reuse of one single subpolicy. And various transitions were either replaced by transition patterns or could be better structured by grouping them within subpolicies.

VII. Related Work

The work discussed in this paper combines the theory of hybrid automata [6] with the behavior-based decomposition of control loops, which is then extended by the introduction of recursive structuring. Reactive execution, task decomposition and sequencing is addressed by reactive control systems as well as by robot control languages.

A. Reactive Control Systems

The reactive layer of a robot control system is most often based upon a behavior oriented control regime. However, structuring reactive control is usually delegated to the next higher level, so that few conceptual solutions exist on this layer. The classical subsumption architecture [4] organizes reactive behavior in levels of competence. It uses a priority-based arbitration scheme, as higher levels of competence can override the output of lower levels. Aside of that there is little structural support, especially for behavior sequencing. On the contrary, the basic idea is that lower levels stay active all the time.

A very interesting approach in organizing behavior sets into different modes of control is taken by the Dual Dynamics approach [7]. Behavior sequencing modeled as a continuous system based on bifurcation points in activation dynamics. However, each elementary behavior has to be

aware of all the available activation variables in order to respond correctly in all modes.

The Extensible Agent Behavior Specification Language XABSL [10] models finite state automatons but uses decision trees for hierarchical structuring. The overall behavior is specified in a directed acyclic graph, with actually forms the decision tree. In contrast to BAP every leaf node consists of only one behavior.

B. Robot Control Languages

Robot control languages are designed to bridge the gap between the deliberative layer and the reactive execution engine. They handle sequencing and parallel execution at the task level. But they are usually designed from a top down perspective and have little conceptual connection to the behavior level. For instance two robot control languages PRS-lite [14] and Colbert [9], that are available for the Saphira architecture [8] have no apparent link to the behavior framework [16] of the system (which is based on fuzzy control theory), even though they usually are needed to activate and deactivate its behaviors for different tasks.

The Task Definition Language (TDL) [17], is designed as an extension to C++, that is compiled down to plain C++ and supported by a runtime library. Its main constructs are a set of task identifiers (Goal, Command, Monitor...) that are prefixes for a C++ global function, the spawn keyword for starting parallel subtasks and a set of constraints for synchronization of spawned subtasks. Exception management is used for expressing failure, using similar semantics as the event and binding semantics in BAP.

C. Structured Behavior-Based Control

The architecture most similar to MIRO is probably MissionLab [1], an end-user oriented robot task-level control software, which is designed for mission specifications for in the military domain. It builds upon the Societal Agent theory and allows the composition of higher-level agents by combining a set of (atomic) agents with a coordinaton operator. The reactive behavior of robots can be specified by the use of a configuration description language (CDL), for which a graphical frontend exists [11].

Coordination operators can select and fuse outputs from the agents and allow for temporal sequencing by implementing a FSA. Events are only handled by a coordination operator. Termination conditions like successful completion or failure of a subtask need therefore be checked on the next higher level and are not part of the agent itself. MissionLab uses different compiler backends to translate the specified control program for different robots.

VIII. CONCLUSION

In our BAP framework (Behavior, Action pattern, Policy) we evaluate methods to overcome some intrinsic scalability issues of behavior-based robot control architectures. The key issue described within this paper is how behaviors can be organized in a hierarchical way. This allows for complexity reduction, for reuse of action patterns or complex sequences of actions, either within a single control system or for different scenarios.

The implementation of the BAP framework is used for robot controllers in different autonomous mobile robot scenarios such as office robots and RoboCup soccer robots, as well as for robotics classes. It consists of a class framework for behaviors and arbiters, generalized basic behaviors as well as arbiters, GUI-based modeling and debugging tools, as well as auto code generation tool for strictly typed parameter handling. The framework is capable of handling, managing and evaluating complex behavior-based control systems and proved itself in very different scenarios on very different robot platforms.

REFERENCES

[1] R. Arkin, T. Collins, and Y. Endo. Tactical mobile robot mission specification and execution, 1999.

[2] R. C. Arkin. Towards the unification of navigational planning and reactive control. In *Working Notes of the AAAI Spring Symposium on Robot Navigation*, March 1989.

[3] M. Beetz. *Concurrent Reactive Plans: Anticipating and Forestalling Execution Failures*, volume LNAI 1772 of *Lecture Notes in Artificial Intelligence*. Springer Publishers, 2000.

[4] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1), March 1986.

[5] K. Erol, D. Nau, and J. Hendler. UMCP: a sound and complete planning procedure for hierarchical task network planning. In *Proceeding of AIPS-94*, Chicago, IL, 1994.

[6] T. Henzinger. The theory of hybrid automata. In M. Inan and R. Kurshan, editors, *Verification of Digital and Hybrid Systems*, NATO ASI Series F: Computer and Systems Sciences 170, pages 265–292. Springer-Verlag, 2000.

[7] H. Jaeger and T. Christaller. Dual dynamics: designing behavior systems for autonomous robots. In *The Sixth International Symposium on Artificial Life and Robotics (AROB 6th ´01)*, 1997.

[8] K. Knolige and K. Myers. The saphira architecture for autonomous mobile robots. In *Artificial Intelligent and Mobile Robots*, chapter 9, pages 211–242. AAAI Press, 1998.

[9] K. Konolige. Colbert: A language for reactive control in saphira. In G. Brewka, C. Habel, and B. Nebel, editors, *Advances in Artificial Intelligence*, volume 1303 of *Lecture Notes in Computer Science*. Springer, 1997.

[10] M. Lötzsch, J. Bach, H.-D. Burkhard, and M. Jüngel. Designing agent behavior with the extensible agent behavior specification language XABSL. In *RoboCup 2004: Robot Soccer World Cup VII*, LNCS. Springer, 2004. to appear.

[11] D. C. MacKenzie, R. C. Arkin, and J. M. Cameron. Multiagent mission specification and execution. *Autonomous Robots*, 4(1):29–52, March 1997.

[12] M. J. Mataric. Behaviour-based control: Examples from navigation, learning, and group behavior. *Journal of Experimental and Theoretical Artificial Intelligence*, 9, 1997.

[13] R. R. Murphy. *An Introduction to AI Robotics*. MIT Press, November 2000.

[14] K. Myers. A procedural knowledge approach to task-level control. In *Third International Conference on AI Planning System*. AAAI Press, 1996.

[15] A. Saffiotti, K. Konolige, and E. H. Ruspini. A multivalued logic approach to integrating planning and control. *Artificial Intelligence*, February 1995.

[16] A. Saffiotti, E. Ruspini, and K. Konolige. Blending reactivity and goal-directedness in a fuzzy controller. In *Neural Nets and Fuzzy Control*, San Francisco, 1993. IEEE.

[17] R. Simmons and D. Apfelbaum. A task description language for robot control. In *International Conference on Intelligent Robots and Systems*, 1998.

[18] H. Utz, S. Sablatnög, S. Enderle, and G. K. Kraetzschmar. Miro – middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, 18(4):493–497, August 2002.