TDM : A Software Framework for Elegant and Rapid Development of Autonomous Behaviors for Humanoid Robots

Vincent Berenz*, Fumihide Tanaka† and Kenji Suzuki†
Department of Intelligent Interaction Technologies
Graduate School of System and Information Engineering
*University of Tsukuba, Japan
†University of Tsukuba/JST, Japan
Email: vincent@ai.iit.tsukuba.ac.jp

Abstract—Through the use of module based software solutions, programming humanoid robots became simple in the sense that detailed knowledge of the underlying software and hardware layers became largely unnecessary. In this paper we argue that the current situation, while being satisfactory for most users, requires improvement for facing situations in which delivery of a complex autonomous behavior is part of the final target. In such case, implementing a dedicated behavior control architecture remains a complex task. In this paper we propose a behavior oriented software framework to be added above the existing modular architecture. This framework is based on centralized integration of sensory data, schematic representation of objects, resource management and intrinsic motivation. It supports code organization, favors code reuse and allows rapid obtention of behaviors that can be easily modified or extended. A version of the framework for Aldebaran Nao was developed and tested.

I. INTRODUCTION

A. Robots for non roboticists

As it requires integration of a broad range of diverse technologies and the implementation of reactivity loops between sensing and acting, programming a humanoid robot can be considered a complex task. To allow developers to access services of the robot without deep knowledge of the underlying hardware, modular software solution such as Sony’s OpenR[1], GenoM[2] or ROS[3] have been proposed. Features that facilitate robot programming even further: publication/subscription or service based communication protocols, specialized programming language such as URBI, graphical user interfaces (GUI) such as iRSP [Bonavision], Gostai Studio [Gostai], etc have also been implemented. Aside from notable differences, the global approach of available solutions can be roughly summarized as in Fig.1. If focusing on humanoid robots, one of the main benefits of such systems is that robot programming is no longer a task requiring a team of experienced roboticists. For example, the humanoid robot Nao [Aldebaran Robotic] is used as a tool by researchers whose background and interest are not robotics but for instance education[4] or psychology[5].

Such software platform, while being satisfactory for most users, requires to be extended with a behavior control architecture for facing situations in which delivery of a complex autonomous behavior is part of the final target. In this context, the term complexity refers to the variety of sensory information the robot must react to and the variety of tasks or actions the robot must perform autonomously based on equipped sensors. Complex behaviors require such architecture to be implemented above the module based architecture, in the area presented in gray in Fig. 1.

B. Complex behaviors

Table I gives examples of behaviors that could be implemented in a Nao robot. The complexity corresponding to Behavior 2 can be described: reactivity to detection of colored objects, human faces, petting actions must be enforced. Memory of existence and positions of colored objects, disposable boxes, human faces must also be implemented. Error recoveries of actions, especially when walking or picking objects, must be implemented. Treatment of sensory data for occasional false positive signals from the face detection or object detection modules is required. Interfacing with a reinforcement learning algorithm must be performed. It is reasonable to consider that other behavioral features, such as taking preventive action is case of low battery level, high

Fig. 1. Typical modular based robot architecture. Scripts can be written using programming languages or generated using a graphical interface. But if the desired behavior is complex, a supplementary behavior control architecture is required at this level.
C. Code elegance

As any programming project, it is reasonable to consider that the code written for behavior control should be elegant: well organized, reusable, easy to debug, to modify or to extend. For example, if Behavior 1 has been developed, it would seem reasonable to expect obtaining Behavior 2 through its simple extension. Similarly, it should be easy to obtain Behavior 2.bis through the modification of Behavior 2. But the modification of a behavior will require the difficult understanding of all its code and is likely to lead to unexpected debugging issues. If not elegant, the delivered code will be so difficult to understand and adapt by a third party programmer that starting the project from the beginning may even appear preferable.

D. Target

In this paper, we first analyse why code for complex behavior control is difficult to deliver in a rapid and elegant fashion. Then a solution in the form of a supplementary light software framework to be put on top of the existing module based architecture is proposed. Through its use, the level of programming skills required for delivering a complex behavior is decreased: for example experience in thread programming will no longer be a strict requirement. Furthermore, because it favors code reuse at a level higher than modules, it renders possible the delivery of complex behaviors in a few hours.

Usually categorized as deliberative, reactive, hybrid or behavior based, many behavior control architectures have been proposed [6][7]. But rarely implementation and adaptation to new behavior by other users has been their main focus. Their description as found in the literature, while providing insights to the experienced robotic community, will be of little help to the unexperienced developer required to deliver an autonomous behavior in a reasonable time frame. The purpose of the present work is not to merely describe how code should be organized: it is a software framework that is ready to use and can be easily readapted for new projects. It aims at lowering the level of technical knowledge required for behavior control implementation, making it a task accessible to a much broader community.

II. CODE COMPLEXITY

Typically the code corresponding to the implementation of a behavior control architecture will require the development of a collection of threads that exchange information. Depending on the chosen approach, the threads can correspond to modules or other construction of higher level. For generating actions suitable to the current situation and ensuring reactivity to various events, the code will contain a very large number of interrelated conditional loops that will activate or deactivate the threads and/or change their configuration. Difficulties can be listed:

- The design of the structure in terms of loops and threads to use as well as their interactions can be complex.
- As threads are running simultaneously, it is possible for them to send conflicting commands to actuators. The program must be designed to avoid such situations.
- Sensory data often require integration. This integration might be : 1) over time for dealing with noise and imprecisions and/or 2) over several sensors, for example sonars for distance and vision for color. Such integration renders code design even more arduous.
- Debugging is difficult : if the robot does not behave as expected, finding the source of the issue in the context of interrelated conditional loops can be wearisome.

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The robot must perform different actions: (1) detecting humans, (2) interacting with humans, (3) looking around for green objects on the floor. If green objects are detected, the robot must be able to go pick them and bring them to the human. Priority between search for object or interaction with human depends on how long the presence of human have been acknowledged and for how long the presence of green objects have not been checked. The robot must always respond positively to caring action from the human, such as petting, except if the robot is carrying an object, in which case petting should trigger the robot releasing the object. Furthermore when detecting the presence of a human for the first time in a while, the robot must looked nicely surprised and utter some greeting.</td>
</tr>
<tr>
<td>2</td>
<td>Same as (1), except that also blue and red objects can be detected and picked up. The robot must not only be able to bring the objects to the human partner, but also to bring them to a disposal box: the robot must also look for. The decision of bringing the object to the human or to the disposal box will depend on a reinforcement learning algorithm taking into account the color of the object currently hold and feedbacks of the human to the current action.</td>
</tr>
<tr>
<td>2.bis</td>
<td>Same as (2), but the choice of the robot is not to bring or dispose of the objects, but to dispose the objects in two separate disposal boxes.</td>
</tr>
</tbody>
</table>
Schematic representation of objects respond exclusively to push A. Integration of data

The difficulties listed in section II as follows: Integration of data to be manipulated. TDM uses these units to address the underlying modules. Programming a behavior using TDM is done through exclusive assembly of units of code in a process explicit in Fig. 3. All units of code can be created or reused from an existing library. The assembly can then be plugged in the system to obtain the desired behavior (Fig. 4). Different types of units of code have specific roles, such as representing data, evaluating a state, accessing the underlying modules or calculating a drive for action. To underline this rule of separation of concerns that has been enforced in its conception, the proposed framework has been named “Target, Drive and Means” (TDM). Table II lists all the types of units of code to be manipulated. TDM uses these units to address the difficulties listed in section II as follows:

A. Integration of data

Units of code never communicate directly one with another. Instead, they push or pull asynchronously units of data in a shared memory called Internal Model. Units of data correspond exclusively to schematic representation of objects.

Schemas are used as a configurable way of organizing data such as allowing centralized integration. Schemas pushed in the Internal Model contain the information on how they have to be compared and eventually integrated with already existing schemas: a schema consists of an array of properties. A property is a unit of code that once instanciated can contain data of any kind of format. As presented in Table II the property superclass requires from the user the implementation of two functions:

- similarity: evaluates the similarity between two instances of property.
- fusion: performs a customized integration of data by fusing two similar instances of property.

When a schema is pushed in the Internal Model, the later uses the similarity functions to compare and fuse it with existing schemas. This process is done with no consideration in regards to the units of code that pulled the schema. Table III gives an example of integration of schemas representing colored objects in the surroundings of the robot. In this example, an object is characterized by two properties: color (RGB code) and position (x and y relative to the robot). The units of code corresponding to the properties color and position are programmed:

- Two colors or two positions are similar if the distance between their components, (R,G,B) or (x,y) respectively, is small.
- Two colors are fused by averaging their (R,G,B) component, two positions are fused by keeping only the most recent one.

Using these functions, the Internal Model determines that the schema that is pushed at time t=27ms (schema 128) is similar to an existing schema (schema 32) and fuses the two schemas accordingly. In the contrary, schema 127 is considered corresponding to a newly detected object. In this particular case, these integration rules result in giving robustness to the system in regards to imprecision in color and position evaluation. This schema based approach has the following advantages:

- By programming of similarity and fusion functions, various integration methodologies can be implemented: using the value with the higher precision or with the most recent time stamp, averaging the values, maintaining a history of all past data for noise management, etc.
- The Internal Model can maintain different types of schemas having different integration rules.
- A property is a unit of code that can be reused. New types of schemas can be constructed by combining properties extracted from a library.
- This approach can be used for management of simple data, such as a boolean indicating a status, the temperature of a joint, a time stamp, the information the last time a human face was detected. But users can also associate customized properties such as creating and maintaining more complex constructions. As it has been underlined in [8], object schemas are convenient tools to organize perceptions and to implement the notion of

### Table II

<table>
<thead>
<tr>
<th>Unit</th>
<th>Role</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Unit of data representation</td>
<td>similarity()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fusion()</td>
</tr>
<tr>
<td>Schema</td>
<td>Array of properties, unit of data exchange</td>
<td>-</td>
</tr>
<tr>
<td>Functionality</td>
<td>Unit of behavior that directly access to the underlying modules</td>
<td>start()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stop()</td>
</tr>
<tr>
<td>Condition</td>
<td>Unit of conditional logic</td>
<td>areMet()</td>
</tr>
<tr>
<td>Action unit</td>
<td>Association of a functionality and an array of conditions</td>
<td>-</td>
</tr>
<tr>
<td>Sub-behavior</td>
<td>Reusable package of action units</td>
<td>-</td>
</tr>
<tr>
<td>Score calculator</td>
<td>Unit of drive</td>
<td>score()</td>
</tr>
</tbody>
</table>

### Table III

**Example of integration of data in the Internal Model**

<table>
<thead>
<tr>
<th>status/event</th>
<th>schema id</th>
<th>(R,G,B)</th>
<th>(x,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>status, t=0ms</td>
<td>schema 32</td>
<td>(120,50,254)</td>
<td>(0.4,1.2)</td>
</tr>
<tr>
<td>status, t=20ms</td>
<td>schema 127</td>
<td>(242,150,163)</td>
<td>(0.8,0.8)</td>
</tr>
<tr>
<td>status, t=27ms</td>
<td>schema 128</td>
<td>(116,58,246)</td>
<td>(0.3,1.3)</td>
</tr>
<tr>
<td>status, t=30ms</td>
<td>schema 32</td>
<td>(118,54,250)</td>
<td>(0.3,1.3)</td>
</tr>
<tr>
<td></td>
<td>schema 127</td>
<td>(242,150,163)</td>
<td>(0.8,0.8)</td>
</tr>
</tbody>
</table>

Considering the points above, if obtaining the desired behavior requires experience in terms of code design and thread management, obtaining the desired behavior through elegant code is even more challenging.

### III. Targets-Drives-Means Software Framework

We propose the implementation of a supplementary software framework placed above the existing modular architecture (Fig. 2). The fixed parts of the system consist of an internal model, an automatic resource management and a continuously running engine. Programming a behavior using TDM is done through exclusive assembly of units of code in a process explicit in Fig. 3. All units of code can be created or reused from an existing library. The assembly can then be plugged in the system to obtain the desired behavior (Fig. 4). Different types of units of code have specific roles, such as representing data, evaluating a state, accessing the underlying modules or calculating a drive for action. To underline this rule of separation of concerns that has been enforced in its conception, the proposed framework has been named “Target, Drive and Means” (TDM). Table II lists all the types of units of code to be manipulated. TDM uses these units to address the difficulties listed in section II as follows:

A. Integration of data

Units of code never communicate directly one with another. Instead, they push or pull asynchronously units of data in a shared memory called Internal Model. Units of data correspond exclusively to schematic representation of objects.

### Table II

**List of templates for reusable units of code**

<table>
<thead>
<tr>
<th>Unit</th>
<th>Role</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Unit of data representation</td>
<td>similarity()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fusion()</td>
</tr>
<tr>
<td>Schema</td>
<td>Array of properties, unit of data exchange</td>
<td>-</td>
</tr>
<tr>
<td>Functionality</td>
<td>Unit of behavior that directly access to the underlying modules</td>
<td>start()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stop()</td>
</tr>
<tr>
<td>Condition</td>
<td>Unit of conditional logic</td>
<td>areMet()</td>
</tr>
<tr>
<td>Action unit</td>
<td>Association of a functionality and an array of conditions</td>
<td>-</td>
</tr>
<tr>
<td>Sub-behavior</td>
<td>Reusable package of action units</td>
<td>-</td>
</tr>
<tr>
<td>Score calculator</td>
<td>Unit of drive</td>
<td>score()</td>
</tr>
</tbody>
</table>
Fig. 3. Programming a behavior is done through exclusive assembly of units of code. Association of functionalities with conditions that will dynamically test them for activation leads to a library of action units. Action units can be packaged into sub-behaviors, which can then be combined to create new behaviors. When doing so, each sub-behavior must be associated to a score calculator that will specify its priority for execution during run-time.

Fig. 4. The TDM software framework, currently running 5 sub-behaviors. Each sub-behavior is identified by a number between 1 and 5 and is represented by the functionalities (in green) and conditions (in pink) it encapsulates. Each sub-behavior is associated to a score calculator (in gray) identified by the same number. During run-time, the TDM engine continuously interrogates the score calculators, using the score they return to dynamically set the current priority of each sub-behavior for access to resources. On this figure sub-behavior 3 is currently having the highest priority.

In the implementation of TDM for Nao, the Internal Model uses kinematics to update dynamically all schema characterized by a position property. Finally, the pull/push paradigm that is supported by the Internal Model simplifies programming by removing all concerns for synchronization.

B. Thread management

The fundamental unit of code reuse that accesses the underlying modules is called a functionality. Functionalities are threads turned on and off by the framework and are the only threads to be manipulated by the user. TDM trivializes the process of thread development by providing a functionality abstract superclass that deals with all issues related with thread management and compatibility with the rest of the framework: when developing a new functionality, users are required to only implement start and stop functions (Table II). In these functions, schemas pulled asynchronously from the Internal Model can be dynamically used as configuration parameters. Functionalities share data by simple asynchronous push of schemas in the Internal Model.

C. Conditional loops management

Conditional loops are removed and replaced by units of code of type condition. By replacing conditional loops by units of conditional logic, problems related to management of interrelated conditional loops are avoided. When added to the currently developed behavior, a functionality has to be associated with an array of conditions, forming a reusable code unit of higher level referred to as action unit. With concerns for availability of resources (see Section III-D below), the system continuously runs the code corresponding to the conditions: if True is returned, the corresponding functionality is activated. If False is returned, the functionality is desactivated. Code implemented in conditions can rely on schemas pulled from the Internal Model.

D. Resource manager

When activated, functionalities corresponding to different action units have access to the underlying modules. Management of conflicts is implemented by simple use of a resource management system: functionalities must register to the resources they need before activation and a resource can be registered by only one functionality at a time. Functionalities not using the same set of resources can run simultaneously. Priority for accessing the resources is explained in the next section.

E. Open intrinsic motivation

Functionalities can be packaged in units of code of higher level referred to as sub-behavior. The order of the action units in the sub-behavior as decided by developers set their priority for resources. The final behavior of the robot is obtained through incremental addition of independently debugged sub-behaviors and sub-behaviors generators (see Section III-F). To dynamically set the priority for access to resources between running sub-behaviors while avoiding the use of conditional loops, a simple and open intrinsic motivation system is implemented. Developers are required to link each sub-behavior
to a unit of code referred to as score calculator, which use schemas pulled from the Internal Model to calculate the current relative importance of its related sub-behavior. As presented in Fig.4, when running, the system continuously interrogates all score calculators and dynamically re-orders corresponding sub-behaviors for access to resources.

F. Sub-behavior generators

Certain actions require specific schemas for execution. For example “walking and kick a ball” requires the robot to have detected one/several balls. To allow programming of behaviors adapted to such situation, developers are given the option to implement sub-behavior generators. Implementation of a sub-behavior generator is done in the same way a sub-behavior is, except that it is associated to a specific type of schema. During run-time, creation in the internal model of such specific schema will trigger the dynamic generation of a sub-behavior that will enter the competition for access to resources. In the example above, if the robot detected two balls, this feature will result in generation of two competing sub-behaviors each referring to a given ball.

G. TDM Engine

The engine runs continuously as follows: (1) all action units are sorted according to the current score returned by the score calculator of their corresponding sub-behavior as well as their relative position in it, (2) conditions of each action unit are then evaluated one by one:

- If conditions are met and resources are free, the start function of the corresponding functionality is called.
- If conditions are met and resources are not free, the stop function of functionalities of activated action units of lower priority that use required resources are called. As stop functions are required to be programmed such as ceasing the functionalities in a way safe for the integrity of the robot, the stopping process will take an undetermined number of cycles. As each functionality runs its own independent thread, the rest of the system can run normally in the meantime, continuing ordering cycles and calling start and stop function of functionalities using other resources.
- If conditions are not met, the stop function is called.

H. Programming with TDM

1) Auto-documentation for compatibility issues: Programming a behavior consists of assembling separately developed sub-behaviors. Some sub-behaviors will pull types of schemas that require to be pushed by other sub-behaviors. A documentation generator program that parses all the libraries and creates files in HTML format synthesizing such dependencies is provided. For example, a sub-behavior “walk to position” will be indicated as requiring schemas having a property “position” and a hyperlink will redirect to the list of all sub-behaviors and functionalities pushing such schemas. The user can then develop new sub-behaviors or reuse existing sub-behaviors accordingly.

2) Log files: At runtime, logs give the list of all schemas maintained in the Internal Model, the list of all running sub-behaviors and their related action units sorted according to their current priority score. The logs also indicate the activation status of each functionality and if not activated, they display which conditions for activation are not met or which required resource is currently not available. For example, if during an experiment the robot starts walking unexpectedly, the user can know instantly which functionality of which sub-behavior is responsible for the action.

IV. Experimental validation

As indicated in Fig. 3, programming using TDM consists in simple assembly of units of code. If suitable units of code are not proposed in the existing library, templates can be used to develop new units that will enrich the libraries. Code elegance is achieved in the sense that any given behavior can be modified or extended by simple replacement or addition of new units of code. This feature of TDM was validated as follows:

- TDM for Aldebaran Nao was developed, consisting in only around 1200 lines of code written in Python scripting languages.
- Three simple behaviors were successively developed and resulted in the creation of a library of code reuse.
- This library was then used to create a more complex behavior in a few hours.

A. Behavior 1: Human tracking

The following behavior was developed: the robot looks for and walks to humans. If its human partner leaves, the robot goes back to its original position and starts searching for humans again. The resulting sub-behaviors running at two different moments are presented in Fig.5.a and 5b. In Fig.5.a, the robot is standing at its initial position and is moving its head in the search for human faces. In Fig.5.b, the robot detected a human, walked to it and is currently executing precision steps to maintain its distance to it. Differences in functionalities activations, depending on conditions and resources availabilities, can be observed. The central components of this behavior are:

- A sub-behavior, “face detection and tracking”, commands head movements. A “face detection” functionality pushes schemas corresponding to detected human faces and containing two properties: time stamp and position relative to the robot. Rules for integration in the Internal Model are implemented in their fusion function: the full history of time stamps for the last minute is stored and incoming positions are averaged. A separate “face deletion” functionality continuously checks if the face is not seen at times it is expected to be in the field of vision. This functionality eventually sends a delete signal to the Internal Model. “Face tracking” and “random head movements” functionalities are associated with conditions based on the history of time stamps. They set the head
Fig. 5. Sub-behaviors of the human tracking behavior: (a) The robot did not detect any human, (b) the robot detected a human and walked to him/her. If a sub-behavior has been dynamically created by a sub-behavior generator, the corresponding type of schema is written below in italics. The current priority score is indicated in square brackets. For each functionality is indicated the resources it requires (in square brackets) and the list of conditions associated to it. Functionalties currently activated are in grey.

movement: looking at the position if the face has been frequently detected and random head movements of growing amplitude centered in direction of the known position if face detection has been unstable.

- A sub-behavior generator, “walk to target”, is implemented to set dynamically the walk parameter. As specified in Section III-F, the generator is set such as creation in the Internal Model of schemas relating to “face” and “initial robot position” will trigger dynamic creation of a sub-behavior: this results in the different number of sub-behaviors that can be observed in Fig.5.a and b. Once a sub-behavior created, its “walk” functionality, when activated, continuously pulls the face schema maintained in the Internal Model and uses its position property for dynamically setting the parameters of the walk. As indicated by the conditions to which it has been associated, the “precision step” functionality requests activation when the robot gets close to the targeted position.

- The “looking alive” sub-behavior is attributed a score calculator returning a low value and gives the robot a default behavior to execute when resources are not used.

While the code written in each unit of code is trivial, between 10 to 60 lines of code per unit, their association in TDM enforces a closed control loop for legs and head movements which is robust toward the unreliable face detection, imprecise when walking. When the human partner leaves, the “face deletion” action unit deletes the face schema in the Internal Model, resulting in the deletion of the corresponding sub-behavior: the robot walks back to its initial position as the corresponding sub-behavior becomes the one with the highest priority score.

B. Behavior 2: Red balls detection and picking

For this behavior, the robot must look for and pick up red balls. A sub-behavior “ball detection and tracking” was created based on the existing “face detection and tracking” sub-behavior. As indicated in grey in Fig. 6, only the action unit corresponding to face detection was replaced while the others were left as such. A “walk to and pick ball” sub-behavior was created through the addition of an action unit to the “walk to position” sub-behavior: a “picking” functionality that uses a look up table that generates suitable grasping movement for any position of the ball in front of the robot.

In both case, the other existing action units were reused without any adaptation: their functionalities and conditions pull the schema corresponding to the target of the sub-behavior and lead to reactions based on its time stamp and position properties. As they have no regard for the functionality that
pushed the schema or the other properties the schema might contain, these functionalities work similarly when targeting human faces, balls or any other schema with position and time stamp properties.

C. Behavior 3: Robot petting

In this behavior the robot reacts to petting action from a human partner. Functionalities developed for the “looking alive” sub-behavior were reused. As the ultra-sound sensors used for detecting the petting actions can be noisy, use of a schema containing a property integrating signals over time was used to achieve stability.

D. Complex behavior

The behavior is the following: The robot looks for humans. When detecting one, it walks to it and reacts positively to petting actions. If the human partner touches its head, the robot turns and looks for red balls. If a red ball is detected the robot picks it before bringing it back to its human partners. As a safety measure, if unexpected signals are detected from the ultra sound signals in other context than robot petting, the robot stops moving.

As presented in Fig. 6 the behavior was obtained almost exclusively through association of existing units of code. Only seven new units of code needed to be created, each of them containing less than 20 lines of code. Score calculators were created to dynamically set the priority of the two sub-behaviors using ultra-sound signals: petting animation is of higher priority only when the schema corresponding to human face indicates that the robot is facing a human partner. In any other situation, ultra sound signals activate functionalities stopping the robot.

Using existing libraries and facilities offered by TDM in terms of support for debug, this behavior was developed and debugged in only a couple of hours, showing the effective reusability of all the code that had been previously developed.

V. Future work

Future work will focus on evaluating the advantages of the user-oriented proposed approach in regard of other existing robot control architectures. The libraries of units of code that were created for the experiments will be extended in order to give the possibility to build behaviors more diverse than the ones described in this paper.

Among other project, TDM will be used for “Seeker”, a project of the University of Tsukuba Artificial Intelligence Laboratory targeting emotion based robot coaching. Its purpose is to test if distal electromyography (EMG) signals extracted from the side of the face can be used for natural interaction with robotic agents. The final required behavior for testing this approach will be similar to the ones described in Table I. Pilot experiments exploring the possibility to
interface TDM with a portable EMG signal interpreter device [9] and a heuristic reinforcement learning algorithm [10] were successfully performed. Results of these experiments in regard to the novel paradigm of emotionally assisted interaction between humans and robots are to be presented somewhere else.

VI. CONCLUSION

We proposed TDM, a behavior control software framework that provides developers with a predefined solution to the problem of code organization when programming autonomous behaviors for humanoid robots. Schematic representation of objects, elimination of conditional loops, support in thread management and open intrinsic motivation allow to program complex behavior through association of small and simple units of code. If a suitable library of units of code is provided, these complex behaviors can be obtained in very short time. In case new units of code are required, templates are provided to ease their development. Auto-generated documentation of libraries and usage of logs support the user in his projects.

Experimental results targeting Aldebaran Nao humanoid robot confirmed that usage of such library allows rapid obtainment of behaviors.

ACKNOWLEDGMENT

This study was partly supported by the Global COE Program on “Cybernics: fusion of human, machine, and information systems” and Funding Program for World-Leading Innovative R&D on Science and Technology (FIRST Program).

REFERENCES