

TPL-R: A High-Level Task Programming Language for Robots

Rockwon Kim*, Seung-Ik Lee*, Sung Hoon Kim*, Wan Chul Yoon**

*ETRI, 161 Gajeong-Ro, Yuseong-gu, Daejeon, Korea

**KAIST, 335 Gwahak-Ro, Yuseong-gu, Daejeon, Korea

{rwkim, the_silee, saint}@etri.re.kr, wcyoon@kaist.ac.kr

Abstract— Many service robots have emerged recently and have been used in real environments. The robots usually cooperate with people to perform specific jobs with unpredictable situations. Moreover, users may want to change already installed behaviors inside the robots. To address this problem, we present a high-level task programming language for robots, TPL-R, based on state machine that is formal model and visualized easily. Graphical view of a TPL-R task enhances user's understanding of the task. TPL-R supports the definition of user's viewpoint of sensors and actuators called *model*. TPL-R interpreter supports the mechanism to bind user-defined models with their implementation. The implementation of user-defined model is not a part of TPL-R task. This separation enables users only focus on high-level control structure of a robot task. To prevent state-explosion and make a simple state machine, we enriched the expressive power of TPL-R with diverse constructs. Without considering native programming language but with graphical diagram, users can program robot tasks.

Keywords— Robot task, control system, task description

I. INTRODUCTION

Controlling robot tasks is a main challenge for robotics. Necessary elements include control architectures, world model representations, task descriptions, and task executions. In general, robot task specification is tightly coupled with robot control mechanisms, which are largely logic-based control [6, 7], behavior-based action selection [5], or a hybrid of the two [8]. Therefore, designing robot tasks without an *a priori* understanding of the robot control mechanism is nontrivial.

Contemporary robot applications extend to many fields of service, including fire fighting, cleaning, and housekeeping, etc. Robots that perform specific services are called service robots. In real world environments (outside of the laboratory), such service robots commonly face unpredictable situations while performing their tasks. Moreover, service robot users often require the service procedure to be changed or updated. To cope with such problems, robot task descriptions or

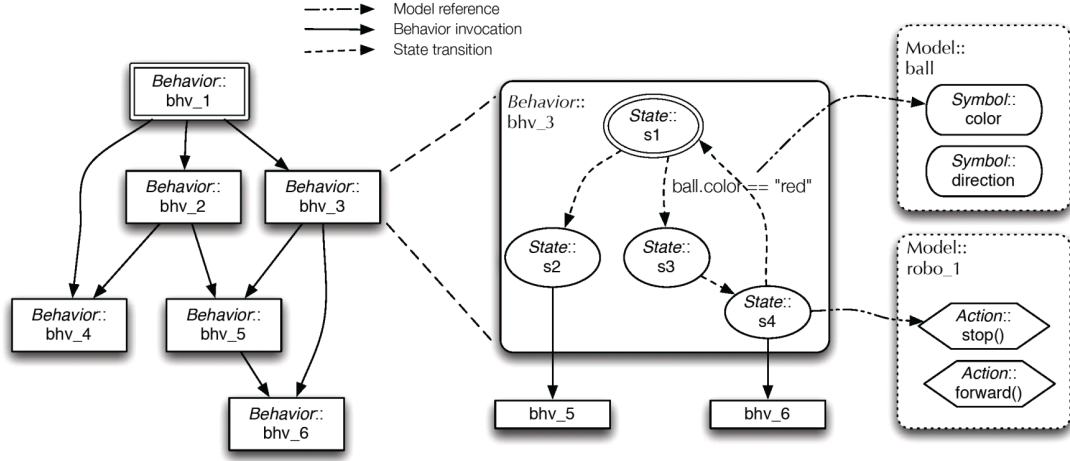
languages should be easy enough such that end users, who typically do not have significant knowledge of robot control mechanisms, can use for describing robot tasks.

A number of robot task description languages have been proposed: Task Definition Language (TDL) [1] and Executive Support Language (ESL) [2] are extensions of C++ and Lisp, respectively; GOLOG [3] is logic-based task programming language; and Extensible Agent Behavior Specification Language (XABSL) [4] is based on a hierarchical the finite state machine (FSM). C/Lisp-like task languages are generally not suited for service domain users, as they are typically not familiar the language syntax and programming structures. In Logic-based systems it is very hard to symbolize control sequences and contingency reaction plans. In XABSL, several FSMs, called *options*, can be hierarchically combined to allow multiple sub-goals and task decompositions. The FSM is simple formal model and it can be visualized as a direct graph; therefore, domain users can design a robot task graphically. However, when we describe a large or complex task using FSMs, the number of graph nodes, called states, is increased. This phenomenon called as the state-explosion problem. This makes it difficult to specify and understand the FSM because of the large number of states and transitions. XABSL also causes state explosion on describing complex tasks.

In this paper, we propose a high-level Task Programming Language for Robots (TPL-R) based on a hierarchical FSM similar to XABSL. Unlike XABSL, we define a new and rich syntax that provide more expressiveness and prevents state-explosion.

II. OVERVIEW OF TPL-R

TPL-R is based on hierarchical FSM and it is composed of three parts: model, behavior, and task. A *behavior* is represented by a FSM composed of a finite number of states and transitions. A *task* has a finite number of behaviors that are hierarchically composed. As shown (a) in Figure 1, the hierarchies of behaviors are a directed acyclic graph. To represent hierarchy among behaviors, TPL-R supports the *behavior-transition* on which a state in a behavior can invoke other sub-behaviors.



(a) An example of the hierarchy of behaviors

(b) The internal state machine of a behavior

Figure 1. An example of the structure of a task using TPL-R

Handling sensor data in a task makes the task complicated with large parts of data processing for sensors. To interact easily with the sensors and actuators in a robot, TPL-R supports the definition of world model simply called *model*. Models reflect the logical and user's viewpoint of sensors and actuators installed in a robot. In models, we can define every *symbols*, *variables*, *functions*, and *actions* that are used in behaviors. Behaviors refer to models while running tasks as shown (b) in Figure 1. To get/set logical sensor value, we have to define a model and then declare symbols or functions in the model. To manipulate robot actuators such as servo motor, speaker and display device, we also define a model for actions and then declare actions in the model. An action is also a logical view of robot actuators. For example, in case of wheel-based robot, *TurnRight* action is the combination of left wheel move forward and right wheel move backward with same speed.

III. TPL-R CONSTRUCTS

In this section, we formally define TPL-R constructs that are introduced in the previous section.

Definition1: A task description by TPL-R is a tuple $\langle M, B, ta \rangle$ where M is a finite set of models, B is a finite set of behaviors, and TA is a task.

Definition2: A model m is a tuple $\langle mid, ME \rangle$ where mid is a model name, ME is a finite set of the model elements. me is a model element. $me \in v \cup sy \cup f \cup a$ where v is an element of a finite set of variables, sy is an element of a finite set of symbols, f is an element of a finite set of functions, and a is an element of a finite set of actions that are functions for actuating a robot.

Definition3: A Task ta is a tuple $\langle tid, con, des, sb \rangle$ where tid is a task name, con is construct block, des is destruct block, and sb is a start behavior such that $sb \in B$

Definition4: A behavior b is a tuple $\langle bid, S, is, gs, cs, P \rangle$, where: bid is a model name, S is a finite set of states, is is an initial state of a behavior, gs is a goal state, cs is a current state such that $cs \in S$, and P is a finite set of input parameters of a behavior.

A behavior is a FSM. TPL-R supports input parameters in behavior. Like function input parameters are used as local variables.

Definition5: A state s is a tuple $\langle sid, T, sa^\downarrow, sa^\bullet, sa^\uparrow, st, subB \rangle$ where: sid is a state name, T is a finite set of transitions, $subB$ is a list of names of currently called sub-behaviors in T , $sa^\downarrow$ is an action block executed at entry point into a state, $sa^\bullet$ is an action block executed while holding a state, $sa^\uparrow$ is an action block at exit point from a state, and st is a integer constant representing state holding time from start of a state.

In XABSL, the hierarchy of a FSM is represented by calling sub-options in a state. To call sub-option needs one state because we can call only an action or an option in a state. In TPL-R, the hierarchy is built by behavior-transitions in the transition block. The behavior-transition reduces the number of states in a FSM comparing to XABSL.

To make control structure delicate, a state in a behavior has three kinds of action blocks: *entry*, *stay* and *exit* block. Because entry and exit block have their own operational semantics, we can reduce the number of states that is required to represent entry and exit situation.

Definition6: A transition is a tuple $t = \langle tlb, b_s, cs_{bs}, cond, tb \rangle$ where tlb is a label of a transition. $cond$ is a condition for a transition, cs_{bs} is a state name having a transition, b_s is a behavior name having the state cs_{bs} , and tb is an transition block executed when $cond$ is true.

Definition7: A condition is $cond = (x_1 \text{ op_r } x_2) \cup \{\text{'true'}, \text{'false'}\}$ where x_1, x_2 are either variable, symbol, function, constant or $cond$, and $op_r \in \{<, >, \leq, \geq, \neq, =\}$ with $Type(x_1) = Type(x_2)$. $Type$ maps its attribute to one of the types, *string*, *bool*, *integer*, *float* or *double* such that $Type(x_1 \text{ op_r } x_2) = bool$. In case of $Type(x_1)=bool$, op_r and x_2 may be empty.

Definition8: An *Expression* is $expr = x_1 \text{ op_c } x_2$ where x_1, x_2 are either variable, symbol, function, constant or $expr$ and $op_c \in \{+, -, /, \times, =\}$ with $Type(x_1) = Type(x_2)$. op_c and x_2 may be empty.

Definition9: A evaluation function is defined as $eval: cond \cup expr \rightarrow \{\text{Text}, \text{Enum}, \text{B}, \text{Z}, \text{R}\}$ where B is boolean value, Z is integer, and R is real number.

Definition10: An *action block* is a finite set of action block statements. An action block statement is $ab_stmt \in (compound_stmt \cup expr \cup action_stmt)$. A *transition block* is a finite set of transition block statements. A transition block statement is $tb_stmt \in (compound_stmt \cup expr \cup trans_stmt)$ where $compound_stmt \in (cond_stmt \cup parallel_stmt \cup sequential_stmt)$ and $trans_stmt \in (callbvh_stmt \cup movetostate_stmt \cup gotostay_stmt)$.

In this paper, we do not introduce the syntax of statements in detail, but we only show the sorts of statements.

IV. OPERATIONAL SEMANTICS OF TPL-R

We define the operational semantics of TPL-R for executing a task. To execute *hFSM* we first find out proper behavior and state from rooted directed acyclic graph by using transition information.

Definition11: A transition function is defined as follows:
 $trans: tb_{tlb,bs,cs} \rightarrow target$, $target \in \{b_t, s_t, sa^*, \perp\}$ where t is a transition, b_t is a target behavior that is an element of B , s_t is a target state that is an element of S in b_s , sa^* is stay action block in s_s and $\perp$ denote the value "do nothing".

$$trans(tb_{tlb,bs,cs}) = \begin{cases} b_t, & \text{if } eval(cond_{tlb,bs,cs}) \wedge callbvh_stmt \in tb_{tlb,bs,cs} \\ s_t, & \text{if } eval(cond_{tlb,bs,cs}) \wedge movetostate_stmt \in tb_{tlb,bs,cs} \\ sa^*, & \text{if } eval(cond_{tlb,bs,cs}) \wedge gotostay_stmt \in tb_{tlb,bs,cs} \\ \perp, & \text{if } eval(cond_{tlb,bs,cs}) \wedge trans_stmt \notin tb_{tlb,bs,cs} \end{cases}$$

$callbvh_stmt$ is used for calling a sub-behaviors in B , $movetostate_stmt$ represents state transition from cs_{bs} to another state s_t in b_s , and $gotostay_stmt$ represents execution of sa^* in cs_{bs} .

Definition12: A *state selection* is a sequence of function $trans(tb_{tlb,bi,csbi})$ where $i \geq 0$ and $b_0=sb$:

$trans(tb_{tlb,b0,csb0}), trans(tb_{tlb,b1,csb1}), trans(tb_{tlb,b2,csb2}), \dots, trans(tb_{tlb,bn-1,csbn-1}), trans(tb_{tlb,bn,csbn})$ such that $trans(tb_{tlb,bk,csbk}) = b_{k+1}$, $b_{k+1} \in B$, $k+1 \leq n-1$ and $trans(tb_{tlb,bn,csbn}) = s_n$, $s_n \in S$ in b_n .

Definition13: A *state tree* is a directed acyclic graph of nodes composed of source behavior, source state and target behaviors: $(b_0, cs_{b0}, [b_1, b_2]), (b_1, cs_{b1}, [b_3]), (b_2, cs_{b2}, [b_4]), \dots, (b_n, cs_{bn}, [])$ where $b_i \in B$, $b_0=sb$ and $i \geq 0$. Leaf nodes of a *state tree* do not have target behaviors.

If the number of target behaviors are greater than two, this means that multiple behaviors is executed concurrently. A transition block in a state can have *parallel_stmts* in its *cond_stmts*.

A *state tree* is memory of *state_selection* for a task execution. To traverse and update a *state tree* has three operations: $newNode(b_t)$, $deleteNode(b_s)$, $updateNode(b_s, s_t)$. The operation $newNode$ generates a new *state tree* node $(b_t, cs_{bt}, [])$ where $cs_{bt} = is$ in b_t . The operation $deleteNode(b_s)$ removes the *state tree* node having b_s as its source behavior. The operation $updateNode(b_s, s_t)$ assigns $cs_{bs} = s_t$ and deletes existing target behaviors in the *state tree* node having b_s as its source behavior.

Definition14: To select a state, we start *state selection* from the root node of a state tree. We check every steps of the result of function $trans: tb_{tlb,bi,csbi} \rightarrow target$ until we reach a state. The *status* of a state tree node is changed after performing function $trans$:

$$status(b_i, cs_{bi}) = \begin{cases} transBhv, & \text{if } target \in B \wedge target \notin t-bhvs \\ stateHold, & \text{if } target \in B \wedge target \in t-bhvs \\ transState, & \text{if } target \in S \text{ in } b_i \text{ and } cs_{bi} \neq target \\ selectState, & \text{if } target = sa^* \text{ in } cs_{bi} \end{cases}$$

Definition15: According to the status of a state tree we perform proper *state tree operation*:

$$stree_op = \begin{cases} newNode(b_{target}), & \text{if } status(b_i, cs_{bi}) = transBhv \\ trans(tb_{tlb,b_{target},cs_{b_{target}}}), & \text{if } status(b_i, cs_{bi}) = stateHold \\ updateNode(b_i, target), & \text{if } status(b_i, cs_{bi}) = transState \end{cases}$$

Definition16: An *action block execution* is to execute one of action blocks: entry, stay and exit block in a state.

$$ae = \begin{cases} exe(cs_{b_{target}}, sa^*), & \text{if } status(b_i, cs_{bi}) = transBhv \wedge stree_op = newNode(b_{target}) \\ trans(tb_{tlb,b_{target},cs_{b_{target}}}), & \text{if } status(b_i, cs_{bi}) = stateHold \\ exe(cs_{bi}, sa^*) \wedge exe(cs_{b_{target}}, sa^*), & \text{if } status(b_i, cs_{bi}) = transState \wedge stree_op = updateNode(b_i, target) \\ exe(cs_{bi}, sa^*), & \text{if } status(b_i, cs_{bi}) = selectState \end{cases}$$

The function exe performs every statements in an action block procedurally.

Definition17: A *task execution* is iterations of an *action block execution*. One action block execution is one iteration of a task execution. If the start behavior sb of a task reaches goal state gs , the execution of task is stopped.

V. A TASK EXAMPLE USING TPL-R

We shall illustrate a simple robot task *Helloworld* using TPL-R. The task is for the robot to say “hello” if it recognizes a user while wandering around, as well as avoiding obstacles. Figure2 is graphical representation of *Helloworld* task.

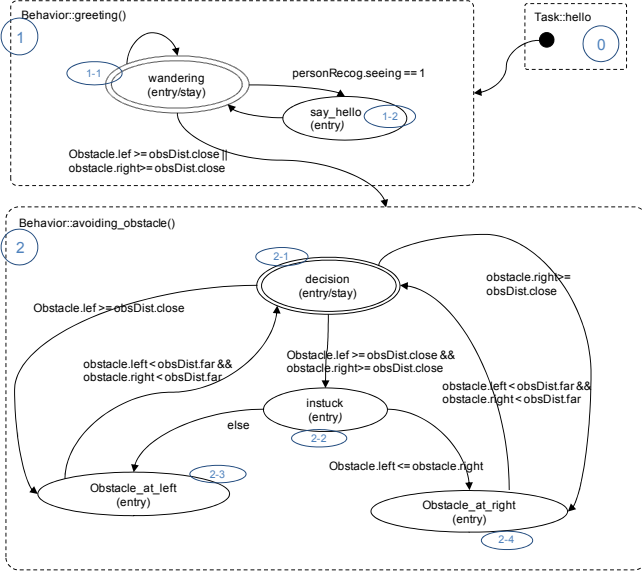


Figure 2. A task diagram for Helloworld example

For the *Helloworld* task, we defined three models: *Obstacle*, *PersonRecog*, and *Robot*. (a) of Figure 3 shows the TPL-R description of these models. The TPL-R description of the task *Helloworld* is presented in (b), (c) and (d) of Figure 2. As depicted in Figure 3, we defined two behaviors for the task *Helloworld*: *Greeting* and *AvoidingObstacle*. The behavior *Greeting* has two states: *Wandering* and *Say_hello*. The state *Wandering* checks for obstacles and users. If obstacles are detected, the behavior *Greeting* invokes the behavior *AvoidingObstacle*, which has four states: *Decision*, *Instuck*, *Obstacle_at_left*, and *Obstacle_at_right*. In state *Decision*, the robot first stops and detects obstacles and their direction. Once the obstacles position is confirmed, the state *Decision* transits to one of the other three states depending on obstacle's direction. If users are recognized while holding the state *Wandering* in the behavior *greeting*, the state *Wandering* transits to the state *Say_hello*. In this state, the action *speak* is executed only once at entry point of the state. In the next task cycle, the state *Say_hello* is moved back to the state *Wandering*.

```
enum obsDist{ //distance from obstacle
  none, far, close, bumped
}
model obstacle{
  in int left;
  in int right;
}
/* model for recognition of a person */
model personRecog{
  in bool seeing; //notification of recognition of a person
  in int id; //user id
  in int location; //current location
  in float accuracy; //accuracy
}
```

```
string getPersonName(int id); //function for getting the person's name
}
/* model for actuating the robot*/
model robot{
  action void turnLeft();
  action void turnRight();
  action void forward();
  action void backward();
  action void stop();
  action void tts(string msg);
}
```

(a) Definitions of world models for Helloworld example

```
#include "hello_world.behavior"
#include "hello_world.model"

/* Correspond to label [0] in Figure 2 */
task Helloworld{
  constructor{
    // initialize variable, symbol, function, action in Model
    robot.tts(msg="Welcometo hello world");
  }
  destructor{
    // executed when task is stopped
  }
  iterate{
    call greeting();
  }
}
```

(b) Description of the task for Helloworld example

```
/* Correspond to label [1] in Figure 2 */
#include "hello_world.model"
behavior greeting(){
  /* Correspond to label [1-1] in Figure 2 */
  initial state wandering{
    transition{
      if(obstacle.left >= obsDist.close ||
        obstacle.right >= obsDist.close)
        call avoiding_obstacle();
      else if(personRecog.seeing ==true)
        moveto say_hello;
      else
        goto stay;
    }
    stay{
      robot.forward();
    }
  }

  /* Correspond to label [1-2] in Figure 2 */
  state say_hello{
    entry{
      robot.tts(msg="Hello~~");
    }
    transition{
      if(true)
        moveto wandering;
    }
  }
}
```

(c) Description of the behavior greeting() for Helloworld example

```
/* Correspond to label [2] in Figure 2*/
behavior avoiding_obstacle(){

  /* Correspond to label [2-1] in Figure 2 */
  initial state decision{
    entry{
      robot.stop();
    }
    transition{
      if (obstacle.left >= obsDist.close &&
        obstacle.right >= obsDist.close)
        moveto instuck;
      else if (obstacle.left >= obsDist.close)
        moveto obstacle_at_left;
      else if (obstacle.right >= obsDist.close)

```

```

    moveto obstacle_at_right;
  else
    goto stay;
  }
  stay {
    robot.stop();
  }
}

/* Correspond to label [2-2] in Figure 2 */
state instuck{
  entry{
    robot.backward();
  }
  transition{
    if (obstacle.left <= obstacle.right)
      moveto obstacle_at_right;
    else
      moveto obstacle_at_left;
  }
}

/* Correspond to label [2-3] in Figure 2 */
state obstacle_at_left{
  entry{
    robot.turnRight();
  }
  transition{
    if (obstacle.left < obsDist.far && obstacle.right < obsDist.far)
      moveto decision;
    else
      moveto obstacle_at_left;
  }
}

/* Correspond to label [2-4] in Figure 2 */
state obstacle_at_right{
  entry{
    robot.turnLeft();
  }
  transition{
    if (obstacle.left < obsDist.far && obstacle.right < obsDist.far)
      moveto decision;
    else
      moveto obstacle_at_right;
  }
}

```

(d) Description of the behavior *avoiding_obstacle()* for Helloworld example

Figure 3. Descriptions of Helloworld task

Now we explain the execution of *Helloworld* in Figure 3 by operational semantics of TPL-R. (b) of Figure 3 shows a task description. The iteration block contains start behavior *greeting* of the task *Helloworld*. The start behavior is the root of a state tree. The root of this task is (*greeting*, *wandering*, []). If the robot detects a person then the function trans returns *say_hello* as its target. The root of the state tree is changed to (*greeting*, *say_hello*, []) and the status of the state tree is *transState*. On *transState* we execute exit action block in the state *wandering* and then execute entry action block in the state *say_hello*. At next iteration the root node is changed back to (*greeting*, *wandering*, []) by the trans function and the status of the state tree is *transState*. As there is no exit block in *say_hello* and entry block in *wandering*, we do not perform exe function. At next iteration if the robot do not detect a person or obstacles then the trans function return stay the action block in *wandering*. At next iteration if the robot detects obstacles then trans function returns *avoiding_obstacle*. The state tree has two nodes (*greeting*, *wandering*, [*avoidingobstacle*]), (*avoidingobstacle*, *decision*, []) and executes entry action block in the state *decision*. At

next iteration if robot detect a person then the root state tree node is changed to (*greeting*, *say_hello*, []) and the node (*avoidingobstacle*, *decision*, []) is deleted. As the state wandering does not have exit action block, the robot directly executes entry action block of the state *say_hello*.

The root behavior *greeting* does not have goal state, a robot executes Helloworld task infinitely.

VI. BINDING SYMBOL IN MODEL TO APIs

A TPL-R task is developed through two phases. The first phase is the description of a task with behaviors and models. The second phase is the implementation of the models that are declared during the first phase.

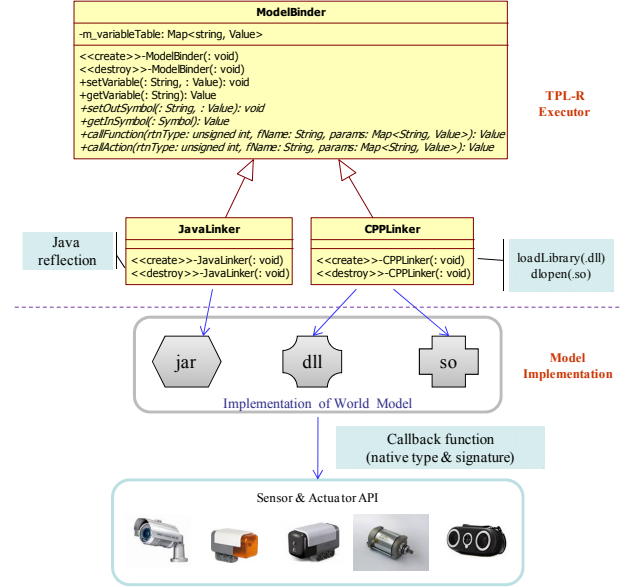


Figure 4. Classes for Binding symbol to APIs

As shown in Figure 4, models are must be implemented as dynamic library like *dll* (dynamic linkage library) or *so* (shared object). As Java is natively supports dynamic linking, we only need jar package. Using dynamic link libraries enables the TPL-R executor to bind symbol/function into other APIs with its name. Model implementation classes that are dynamically linked must have callback functions calling device APIs or other useful libraries.

VII. DISCUSSION

To design robot tasks, we have to understand the control architecture, programming language or logic system adopted in robots. For domain users who are not experts in robotics, understanding such background knowledge for describing a robot task is nontrivial. TPL-R is the combination of hierarchical task control and FSM that has a simple formal model and simpler syntax.

To increase usability, TPL-R separates user's viewpoint of world models from their implementation. Task designers do not need to care for primitive APIs for sensors and actuators because models can be defined appropriately according to the problems in hand and dynamic libraries are responsible for the model and real-world binding. This distinction enables

designers only focus on high-level task control structure. For this characteristic, a TPL-R interpreter have to support dynamic binding mechanism that provides symbol or function values from dynamic libraries correspond to user-defined models and this is helpful for trial and error approach. In addition, a TPL-R interpreter can automatically generate the dynamic library template for models, and then users can only write the callback functions for device APIs in the template files.

As TPL-R is a high-level task control language and its execution is based on dynamic linkage, it is not suitable to use for low-level reactive control that requires rigorous time constraints. TPL-R cannot handle continuous problems directly from low-level symbols or functions because TPL-R is based discrete control model and do not support powerful numerical operators and data types. Therefore, continuous modeling method like handling robot manipulators should be abstract as high-level actions and symbols in models.

Although we permitted concurrent execution of behaviors and actions, we do not cope with the resource conflict problem yet. Resources, such as wheel, arm and speaker, are accessed from model implementation. To prevent the resource conflicts, TPL-R interpreter has to know which action uses which resource. For accurate concurrent action control, TPL-R interpreter has to provide timing mechanism too. We also plan to extend the concurrency to multi-task execution.

ACKNOWLEDGMENT

This work was supported by the Industrial Foundation Technology Development Program of MKE/KEIT[2008-S-030-02, Development of OPROS(Open Platform for Robotics Service) Technology]

REFERENCES

- [1] Simmons, R. and Apfelbaum, D., "A task Description Language for Robot Control", Proc. Of the Conference on Intelligent Robotics and Systems, Vancouver, Canada, October 1998.
- [2] Gat, E., "ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents", Proc. of the AAAI Fall Symposium on Plan Execution, 1996.
- [3] Levesque, H., R. Reiter, Y. Lespérance, F. Lin, and R. Scherl 1997. "GOLOG: A logic programming language for dynamic domains". *Journal of Logic Programming*, 31, 59-84, 1997
- [4] Martin Loetzsch, Max Risler, Matthias Jungel. "XABSL- A Pragmatic Approach to Behavior Engineering", *IEEE/RSJ International Conference on (2006)*, pp. 5124-5129.
- [5] R.A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal on Robotics and Automation*, Vol.2, No.1,1986.
- [6] Fikes, R. E. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189-208.
- [7] Yves Lespérance, Hector J. Levesque, F. Lin, Daniel Marcu, Raymond Reiter, and Richard B. Scherl. A logical approach to high-level robot programming – a progress report. In Benjamin Kuipers, editor, *Control of the Physical World by Intelligent Agents*, Papers from the 1994 AAAI Fall Symposium, pages 109-119, New Orleans, LA, November 1994.
- [8] M^uller, J. P. and Pischel, M. (1994). Modelling interacting agents in dynamic environments. In *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI-94)*, pages 709-713, Amsterdam, The Netherlands.