Robot Sensor Language

Stephen Leake
National Bureau of Standards

Abstract

RSL (Robot Sensor Language) is a data-driven, semi-interpreted, hierarchical, user extensible, robot task description language. It provides four levels of task decomposition, with structures and syntax specialized for each level. The user can add commands for new sensors appropriate to the task at hand. The language is highly interactive, easing debugging and algorithm development. It may also be used as an interface to a task planning system.

Introduction

RSL is a response to the need for a robot language that supports user-designed sensors, along with hierarchical task decomposition and real-time execution. It is written in RCS, the NBS - developed Real-time Control System[1], and runs on 8086 based Multibus hardware.

RSL is a high-level language, specialized for sensor-interactive robot tasks. It is data-driven in the sense that all data relating to a particular task is separated from the control process that executes the task. This makes programming different tasks a matter of changing data files, rather than changing control code, leading to a much more reliable control system. Data describing a robot task is of two types: environmental data, such as object sizes and positions; and algorithms, which give information about what sequence of steps are needed to complete the task. RSL supports representation of both types of data in high-level source code. In order to speed execution, this high-level source code is first compiled into a linked list representation stored in common memory, which is then interpreted by control levels. Any piece of RSL source code can be edited and re-compiled at any time; the linked lists are updated, with garbage collection, and the result can be executed immediately. It is not necessary to re-compile an entire application to make small changes.

RSL supports hierarchical task decomposition. The high-level language is explicitly hierarchical, decomposing tasks into paths, path-points, and trajectories. The compiler and control levels also follow this hierarchy, and are highly modular. This structure makes it easy to add new sensors, or new functional capabilities at any level.

The rest of the paper is organized as follows: section 1 gives an introduction to RSL structures, and introduces a short example task used to illustrate the language. Section 2 discusses the implementation of the compiler and control levels. Section 3 gives a brief overview of some applications that have used RSL. Finally, section 4 concludes with a discussion of future work.

Commercial equipment is identified in this paper in order to adequately describe the systems under development. In no case does such identification imply recommendation by the National Bureau of Standards, nor does it imply that this equipment was necessarily the best for the purpose.

Author's current address: National Bureau of Standards, Bldg. 220, rm B127, Gaithersburg, MD 20899

Section 1 : RSL structures

Task decomposition

It is helpful to use a simple example to illustrate the task decomposition. Consider moving a box from a truck to a conveyor, using a forklift end-effector equipped with sonar range sensors and proximity sensors. The first level of task decomposition (called the task level), yields the following sequence of steps:

1) Move the fork to the vicinity of the truck.
2) Using long range sensors, find the approximate position of the box on the truck, then use short range sensors to align the fork tines with the box, and insert the tines under the box.
3) Lift the box clear of the truck.
4) Move to near the conveyor
5) Gently set the box on the conveyor. (The position of the conveyor is known to the robot controller, so no sensors are needed for this step.)
6) Remove the tines from under the box, and move clear.

The next level of decomposition is the path level. Each of the steps in the task level descriptions is, in fact, a path. Continuing the example, the path for step 2 decomposes into:

1) Scan across the truck, reading the long range sonars.
2) Goto 20 inches (as measured by sonar) in front of the closest point seen in the scan.
3) Move to the pickup side of the box.
4) Use several sonars to align with the floor of the truck, and the side of the box.
5) Insert the tines under the box, using proximity sensors to avoid hard collisions.

Each step in a path is called a path-point. The names path and path-point are derived from the notion of thru-points along a path used to go around obstacles, but the meaning here is generalized to include the use of sensors in various ways. A pre-planned path is a simple case, where no sensors are used.

The last level of decomposition is the trajectory level. Each path-point decomposes into one or more trajectories. For example, step 1 in the path above decomposes simply into a Cartesian straight-line trajectory, while step 4 involves several sensor-servoed rotations.

Each level of decomposition is now discussed in more detail, starting from the bottom.

Trajectory

A trajectory is an algorithm for commanding the position of the end-effector as a function of time. It may be calculated solely on the basis of a priori information (as in " goto the truck "), or incorporate sensor feedback (as in " align to the box "). The trajectory algorithms typically take parameters describing the goal pose and limits on the velocity and acceleration. Sensor-based algorithms will have more parameters describing how to use the sensor data. RSL provides four trajectory commands: Cartesian straight-line and joint interpolated for point-to-point motion, and two others for real-time Cartesian sensor servo[2]. The user may add other trajectory commands as needed.

Path-points

A path-point is an algorithm for a single motion of the end-effector, usually involving a sensor. For example, path-point number 2 above commands a motion towards a point, and uses the sonar to measure the distance remaining. When the distance drops to 20 inches, the motion is halted. This is a motion terminated by a sensor condition. On the other hand, the path-point that aligns the fork with the floor is sensor controlled: the sonars give the distance to the floor, and a rotation is computed that moves towards alignment. The orientation of the fork is servoed to the floor orientation.

There is typically a group of path-point commands for each type of sensor. For example, sonars are used in scan, range, and align path-points. Each path-point command takes parameters which identify the individual sensor to use, and give information on how to use the sensor data. For example, the align path-point command has parameters identifying two sonars, a rotation axis, and a goal orientation.

The base RSL system provides only the **goto** path-point command, which moves the end-effector to a given location, using no sensors. Users must add other path-points to use their sensors. Since the language was designed with user extensions in mind, this is easy to do.

Paths

A path is an algorithm for a simple task, such as moving between locations or grasping objects. The algorithm may be simply a path in space that guarantees no collisions, or it may involve sensors to help locate the object when its position is not accurately known. In the above example, the first path uses no sensors - it simply moves to the truck. The second path uses sensors to find the box. For simplicity, paths consist of a linear sequence of path-points. Any branching or looping must be done within a path-point, or at the task level.

Individual paths are identified by a path type and parameters. The parameters typically consist of named locations, objects, and tools that are involved in the task. The path type gives the intended purpose of the path, and is used in the task level decomposition. There are six path types provided by RSL; ***move-to, approach-pickup, depart-pickup, approach-release, depart-release,*** and ***named.*** The first five path types are used in the TRANSFER task, discussed below. They correspond to the paths in the example: step 1 is a ***move-to*** path, step 2 is ***approach-pickup,*** step 3 is ***depart-pickup,*** step 4 is ***move-to,*** step 5 is ***approach-release,*** and step 6 is ***depart-release.***

The sixth path type, ***named,*** is provided mainly for debugging; it provides a simple way to test small pieces of more complex paths. It also provides a way to program simple tasks that do not need a task level of decomposition.

The path parameters are handled in a way that allows a single path definition to be used for several related tasks. For example, the ***approach-pickup*** path that finds the box on the truck could also be used to find the box anywhere else, or it could find different sized boxes, since the location of the truck and the size of the box are path level parameters.

Tasks

For the task level, the term "task" is used in a specific way; RSL tasks are algorithms for high-level functions such as transferring pallets from a truck to a conveyor, or deburring machined parts.

All tasks are decomposed into a sequence of path types; the specific path to be executed is identified by the path type and the parameters of the current task. The user provides paths for each path type/parameter combination required.

The base RSL system provides two tasks; MOVE-TO and TRANSFER. MOVE-TO simply moves the robot to a goal location, by executing the ***move-to*** path that connects the current location to the goal. TRANSFER transfers objects from a source location to a goal location. This is the task used in the example. The sequence of paths for the TRANSFER task is;

TRANSFER *object, source location, goal location*
1) ***move-to** object , current location, source location*
2) ***approach-pickup** object , source location*
3) ***depart-pickup** object , source location*
4) ***move-to** object , source location, goal location*
5) ***approach-release** object ,goal location*
6) ***depart-release** object ,goal location*

This sequence decomposes the transfer task into six steps, each of which is programmed by the user as a path. The sequence is repeated if either the source or the destination is an array. Note that the user may treat each object / location combination differently, using different sensor based strategies, while maintaining the high-level **TRANSFER** task definition. For example, transferring a large box from a table would involve using sonar sensors to find the box, while transferring a small machined part from the same table would involve a vision sensor. The user would provide two different sets of paths, identified by the object type. The **TRANSFER** command would automatically select the appropriate path, based on the object type in the task parameters.

The user can add other tasks to RSL, to fit the user's application.

Environmental data

RSL provides ways of representing poses (position and orientation) of locations and objects. Locations can be defined in absolute world coordinates, or relative to a base location, using movetables. Movetables provide a convenient user syntax for specifying relative transforms; the transform is built up out of simple steps, consisting of a vector translation, or a rotation about a single axis. Some information about sizes of objects is also represented, for use by a gripping end-effector. Locations can be grouped into arrays, for use in palletizing operations.

Section 2 : Implementation

RSL is implemented using the NBS Real-time Control System (RCS). RCS is a micro-processor based system for real-time control applications. It runs on Multibus based 8086 / 8087 hardware. The operating system is based on FORTH, but has been significantly extended to support multiple processors, a mid-level high-speed compiled language, inter-processor communications, and common memory. It inherits from FORTH the user-friendly features of interactive execution and incremental compilation, making debugging a simple and easy process.

RCS is designed to support a hierarchical control structure, with all levels of the hierarchy executing in parallel, in a cyclic manner[6]. A system clock defines a cycle time, and each level executes its control process once each cycle. This cyclic execution means that each level in the hierarchy is executing the appropriate control algorithm at all times. This is contrasted with sequential execution in which a higher level routine calls a lower-level routine, and the higher level waits for the lower level to complete before it resumes executing. One advantage of cyclic execution is reaction time; since each level samples all inputs each cycle, the system can react to an external event, at any level, in one cycle. This could be done with interrupts in a sequential system, but it is hard to terminate the interrupt routine in a way that aborts the current routine cleanly, and even harder to predict all of the possible interactions.

For example, consider step 2 in the example task above. The Path-point level is monitoring the sonar sensors, and updating the command to the Trajectory level as often as possible. Meanwhile, the Trajectory level is controlling the fork's acceleration and velocity, to maintain smooth motion. The two levels execute simultaneously. In a sequential system, the Path-point level would have to read the sensor, issue a command to the Trajectory level, and then wait until the command was completed before reading the sensor again.

Another advantage of hierarchical design and cyclic execution is modularity; each level has well defined interfaces to sensors and other levels. As long as the interface design is met, any level can be modified independently of the others. Also, any or all levels can be single stepped while the others are running, to help in debugging.
RSL consists of a compiler and an interpreter (see figure 1). The interpreter consists of four control levels running on three processors; a fourth processor runs the compiler and acts as a system supervisor and user interface. No external development system is needed; all programming is done on the final application system. The RSL compiler compiles RSL source code describing locations, movetables, objects and paths into a linked list representation, which is stored in common memory. The control levels then access the common memory to retrieve the data as needed.

The four control levels in the interpreter correspond to the four levels of task decomposition. TASK, PATH, and PATH-POINT execute on one processor. The trajectory level is split into two modules, CARTESIAN and JOINT, which execute on separate processors, to achieve faster cycle rates. The Joint Servo level in figure 1 is assumed to be provided by the robot manufacturer.

The input to TASK is a user command. This command is decomposed according to the task definition, which results in a sequence of paths. One path at a time is commanded to PATH; as each path is completed, a new one is commanded.

PATH accepts path commands, retrieves all the path parameters from common memory, and decomposes the path into path-points, following the path definition compiled into common memory. It commands one path-point at a time to PATH-POINT, waiting for each to complete.

PATH-POINT accepts path-point commands from PATH, retrieves the path-point parameters from common memory, and executes the path-point algorithm. The path-point algorithm typically involves reading a sensor, calculating an object pose based on the sensor data, and updating the parameters in the current trajectory command.

CARTESIAN accepts trajectory commands from PATH-POINT, and retrieves the trajectory parameters from common memory. It executes the trajectory algorithm, and outputs a pose for the robot wrist, once every control cycle. JOINT accepts the wrist pose, converts it to joint coordinates, and commands it to the joint servos.

The cycle time is 28 milli-seconds for a Unimation PUMA 760 or 4000, and 24 milli-seconds for an American MERLIN. These times are determined by the servo rates in the two robots. Note that only the trajectory level must generate new output every cycle; the upper levels are free to take as much time as necessary to process sensor data, or decide on the next task decomposition step.

RSL is user-extensible in many ways. It is designed to allow addition of new trajectories, path-points, paths, and tasks. The user adds routines to the appropriate level of the interpreter to execute the control algorithm, and also adds routines to the compiler that compile the parameters for the algorithms into common memory. Note that the compiler is very simple compared to a typical computer language compiler; the syntax and structures used in RSL are very simple, so adding to the RSL compiler is straight-forward.

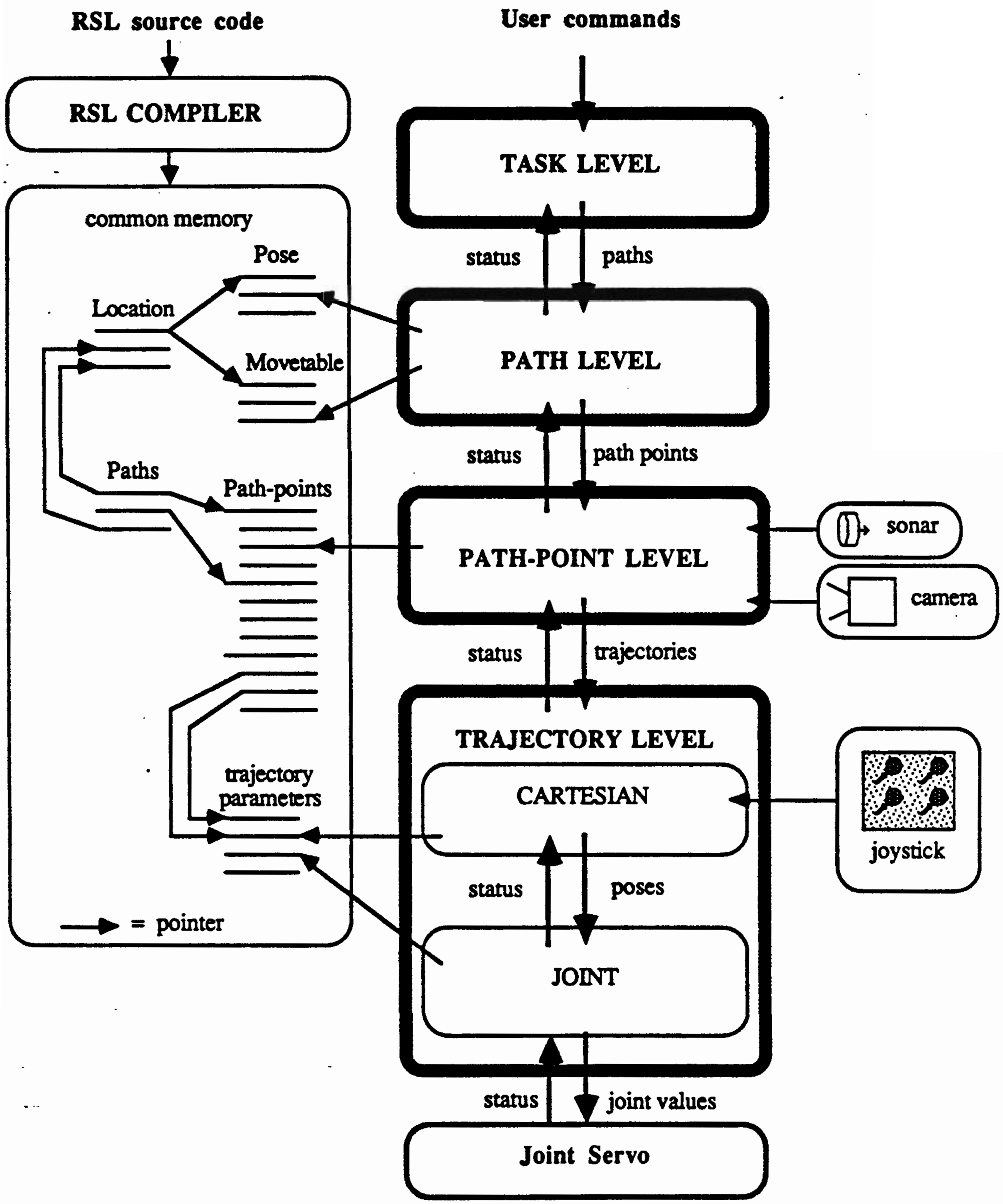

Figure 1. RSL control levels and common memory structures.

Section 3: current applications

RSL has been used successfully in three applications to date. The first is the Field Materiel Handling Robot system, which uses a fork with sensors (very much like the example above) to off-load boxes of ammunition (and other materiel) from trucks[3]. RSL was first used to program a mockup of this task on a PUMA 760, then the mockup was transferred to a UNIMATION 4000 robot. Many path-points were added to the base system, to handle all the sensors.

A second application is a cleaning and deburring workstation in the Automated Manufacturing Research Facility at NBS[4]. RSL is used to program a PUMA 760 to use air-powered deburring tools to deburr machined part edges. A separate workstation level (running on a SUN), encodes the part geometry into a path, with parameters indicating the tool to use, feed rates, etc. The path is down-loaded to the RSL system, where it is compiled and run. Path-points were added to control a vise, various deburring tools, and a tool quick-change. A deburring task and associated paths were also added. The task level accepts commands from the workstation level, and commands paths that deburr the parts, changing tools as required.

The third application is a satellite docking mockup[2]. The RSL system reads a solid-state camera, determines the position of a satellite (as indicated by four LEDs on the satellite), and drives the robot end-effector to dock with the satellite.

Section 4: future work

RSL does not support an explicit world model; all knowledge about how the world works, and in particular how sensors and end-effectors interact with objects, is implicit in the tasks and paths provided by the programmer. This makes it difficult to use more than one sensor at a time. A world model will make it possible to combine data from several sensors, by comparing the sensor readings with predictions, and servoing the model to the sensors[5].

A second area for future work is task representation. Currently, RSL uses SMACRO code to express the task level, and linear sequences of path-points to express the path level. (SMACRO is the computer programming language provided by RCS: it is similar to C, but less powerful). The path level needs to be more flexible, in particular to allow for error conditions. On the other hand, the task level should be more restrictive. The power provided by SMACRO (or C) code is deceptive; it is too easy to write code that works for a particular instance, but is not generic enough to be used for several tasks, or robust enough to work reliably. There should be a task description language that allows adequate flexibility, while guiding the programmer into writing code that works, and is generic and maintainable. The path structure provided by RSL is a first attempt at such a language.

Another reason for developing a good task description language is that it would make an excellent interface between a planning system and the control system. The deburring workstation application mentioned above has shown that this approach is worth pursuing. Having a more powerful task description language available will make it easier to incorporate task level planning for more complex tasks.

Acknowledgments

The author would like to thank Sandor Szabo and Karl Murphy for many useful ideas, and for using RSL in the applications. Tony Barbera and ML Fitzgerald designed and developed the first versions of RCS, and provided much support during the development of RSL.

References

1. The RCS Users Reference Manual. to be published.

2. Stephen Leake, " Cartesian Trajectory Algorithms for Real-time Sensor Servo ", to be published.

3. Harry G. McCain, Roger D. Kilmer, Sandor Szabo, Azizollah Abrishamian, " A Hierarchically Controlled Autonomous Robot For Heavy Payload Military Field Applications ". Intelligent Autonomous Systems, An International Conference, Amsterdam, The Netherlands, 8-11 December, 1986.

4. Harry G. McCain, Roger D. Kilmer, Karl N. Murphy, " Development of a Cleaning and Deburring Workstation for the AMRF", Proceedings of the Deburring & Surface Conditioning Conference, Sept 23-26, 1985, Chicago, Illinois.

5. Ernest W. Kent, James S. Albus, "Servoed world models as interfaces between robot control systems and sensory data", Robotica (1984) volume 2, pp 17-25.

6. A. J. Barbera, M.L. Fitzgerald, J.S. Albus, L.S. Haynes, " RCS: The NBS Real-Time Control System " , Proceedings of the Robots 8 Conference and Exposition, vol 2, pp 19-1 through 19-33, Detroit, Michigan, June, 1984.