Published in final edited form as:

Ind Rob. 2016 ; 43(5): 495–502. doi:10.1108/IR-01-2016-0037.

The Canonical Robot Command Language (CRCL)

Frederick M. Proctor^{*,†}, Stephen B. Balakirsky[‡], Zeid Kootbally[§], Thomas R. Kramer[¶], Craig I. Schlenoff^{*}, and William P. Shackleford^{*}

^{*}National Institute of Standards and Technology [‡]Georgia Institute of Technology [§]The University of Maryland [¶]The Catholic University of America

Abstract

Industrial robots can perform motion with sub-millimeter repeatability when programmed using the teach-and-playback method. While effective, this method requires significant up-front time, tying up the robot and a person during the teaching phase. Off-line programming can be used to generate robot programs, but the accuracy of this method is poor unless supplemented with good calibration to remove systematic errors, feed-forward models to anticipate robot response to loads, and sensing to compensate for unmodeled errors. These increase the complexity and up-front cost of the system, but the payback in the reduction of recurring teach programming time can be worth the effort. This payback especially benefits small-batch, short-turnaround applications typical of small-to-medium enterprises, who need the agility afforded by off-line application development to be competitive against low-cost manual labor. To fully benefit from this agile application tasking model, a common representation of tasks should be used that is understood by all of the resources required for the job: robots, tooling, sensors, and people. This paper describes an information model, the Canonical Robot Command Language (CRCL), which provides a high-level description of robot tasks and associated control and status information.

1 INTRODUCTION

Manufacturing growth has been sustained by continued productivity increases due to automation. Extending this growth will require applying automation to areas that have been difficult to automate, notably those that require a high degree of human dexterity and ability to react to unforeseen circumstances. Robots are promising, but their traditional program-by-teaching model takes time and occupies the robot, and has kept them out of low-volume, short-turnaround jobs. Off-line programming of robots is possible, similar to the computer-aided manufacturing (CAM) method widely used for machine tools. However, the poor accuracy of robots compared with machine tools limits them to jobs with undemanding tolerance requirements, or requires additional methods such as calibration, modeling, and external sensing to improve their accuracy. These methods increase the up-front cost of a robotic system, but advances in sensing technologies (especially laser scanning) and cost reduction have brought them into the range of even small-to-medium enterprises. Indeed, a survey by Modern Machine Shop shows a more than doubling in the use of machine tending

[†] frederick.proctor@nist.gov.

robots over the period 2010 to 2014 (Korn, 2015). An article in the same issue (Albert, 2015) reported on the integration of a robot into a high-precision turbine blade machining and polishing application, highlighting the use of sensing to maintain tight part tolerances. The increased use of sensing to adapt robots to more demanding environments is expected to continue, with visual perception one of four trends noted in the The Robot Report (Tobe, 2015).

As the market broadens for robots and associated sensors and tooling, there is the potential for a proliferation of proprietary interfaces that link together components from single vendor sources but make it difficult to integrate across a wider vendor base¹. While some stability has been achieved through standardization in physical connections and device networking, software incompatibilities exist at the application layer. The situation in robotics is similar to that in the machine tool industry, where a proliferation of numerical control (NC) code dialects made it difficult to automate the programming of these machines in a uniform manner. A higher-level machining task language was developed, Automatically Programmed Tool (APT) (Ross, 2015), which enabled machine tool programs to be generated by CAM systems independently from a particular machine tool. These APT cutter location (CL) programs could be later converted to machine-specific NC code through a post-processing step when a particular machine was selected for use. APT's successors include ISO 10303-238, known as STEP-NC (International Organization for Standardization, 2007; Xu *et al.*, 2006), which standardizes the programming interface to machine tools using modern information modeling techniques.

Current industry practice for robots is to rely on vendor-supplied tools for programming using teach pendants or programming environments provided with the equipment. Portability of programs between different vendors is not possible. Third-party off-line programming systems are available that support common industrial robots using translators for each robot type. Some portability is afforded by these systems, at least among those robots supported by the third party. This has led to a multiplicity of point-to-point translators, often referred to as the N-squared Problem, where the large number of third-party applications multiplied by the large number of robot types results in a very large number of translators. Standards can reduce this burden to a manageable level, where each third party and robot vendor implements to the single standard. This also decreases the likelihood that a particular point-to-point solution would be unavailable, a sadly common experience.

Standards for programming robots have been developed and have met with varying degrees of success. ISO 10562, which specified an intermediate code and associated lists of data that could be converted to robot programs, was published in 1995 but withdrawn in 2004. JIS B 8439 is a Japanese standard, Standard Language for Industrial Manipulators (SLIM), published in 1992 and reaffirmed in 2012, which has some support from Japanese robot vendors. Perhaps the most popular standard for programming robotic equipment is ISO/IEC

¹Certain commercial equipment, instruments, or materials are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

61131, targeted for programmable logic controllers (PLC), with ladder logic and structured text formats that are widely used to connect and synchronize actions between robots and other equipment in a work cell. ISO/IEC 61131 is not a robot-specific language, however, and does not support the definition of robot trajectories.

Open-source robot software development environments have emerged as popular alternatives to a standard robot programming language. These environments use conventional computer programming languages like C++ for application development, rather than defining their own programming language, and provide useful libraries for communication and robot control that serve as a basis for further application development. One example is the open robot control software framework, OROCOS (Bruyninckx, 2001; Open Robot Control Project, 2013), developed by the European Robotic Network EURON. OROCOS is comprised of core robot control functionality and toolkits for kinematics, dynamics, and real-time control. Another example is the Robot Operating System (ROS) (Quigley *et al.*, 2009; Open Source Robotics Foundation, 2016), developed by Stanford University with funding from technology incubator Willow Garage, and now maintained by the Open Source Robotics Foundation. ROS provides core facilities for defining messages and services and establishing distributed networked communication among components of a complete robotic system. Contributors are encouraged to provide packages back to the ROS community, ranging from device drivers for sensors to complete applications. ROS is also integrated with a 3D simulator, Gazebo, that provides a physics engine in order to enable realistic simulation of a robot and its interaction with the environment.

Taking another step further away from full programming languages, robot messaging specifications focus on standardizing the format and content of messages between robotic system components and applications. The implementation of the components that send and respond to the messages can be done in any programming language, or in hardware, or even by a person. One example is the Joint Architecture for Unmanned Systems (JAUS), which defines a message set for unmanned land, sea, and air vehicles (Clark, 2005). Originally sponsored by the U.S. Department of Defense, JAUS is standardized through the Society of Automotive Engineers in their SAE AS4 series. An open-source implementation, OpenJAUS (OpenJAUS, 2013), is available but is not part of the standard. In the manufacturing domain, the Association for Manufacturing Technology sponsored a project at the University of California Berkeley for an Internet-based standard for communicating with manufacturing equipment, called MTConnect (Edstrom, 2013). MTConnect is built on a standard web framework, using the hypertext transfer protocol (HTTP) and the extensible markup language (XML) to exchange the properties and operating condition of machine tools and other manufacturing automation equipment. MTConnect-compliant agents run on the equipment, and client applications provide visualization and other useful capabilities that work with any machine. Open-source implementations are available for generic agents and some useful clients, but are not part of the standard.

In the research described herein, the authors took the messaging specification approach similar to JAUS and MTConnect, and developed the Canonical Robot Command Language (CRCL). The work focused on the domain of industrial robotics as a special case of an overall ontology for automation (IEEE, 2015) defined by the Institute for Electronics and

Electrical Engineers (IEEE). The industrial robot ontology was chosen as the first test case to validate the concept of a core ontology with domain-specific extensions. Extending this suite of ontologies can be done in two ways. If an entirely new domain is needed, an ontology similar to CRCL can be developed that uses the core concepts but supplements it with the messages relevant to the new domain. If new messages for industrial robots are needed, the existing schemas can be extended with the new messages. Existing systems would remain compatible with the revised schemas, providing that the entities in the original schema were not modified. Indeed, this was done many times during the development and validation of CRCL in the authors' systems. As new messages were developed and added to the schemas, care was taken to maintain compatibility with existing software, while new software used the new messages.

The remainder of this paper describes CRCL and its implementation and validation at the authors' institutions. Section 2 provides a detailed description of CRCL, including how it is represented and some of its relevant constructs. Section 3 describes the implementation of CRCL on two different robots to show its portability among different robotic platforms. Section 4 describes software tools that helped with the implementation and validation efforts. Section 5 concludes the paper.

2 CRCL

The Canonical Robot Command Language provides message content and syntax for a low-level robot command-status protocol. Command message content includes:

- setting robot-independent parameters (units, speeds, accelerations, tolerances)
- setting robot-dependent parameters
- performing Cartesian (robot-independent) motions
- performing joint-level (robot-dependent) motions
- operating an end effector, e.g., gripper or dexterous hand
- dwelling for a specified time
- setting hardware-specific end effector parameters
- changing end effectors
- configuring status reports
- getting a status report immediately
- showing a message to a human user of the robot
- starting and stopping a session
- running a robot program in some other language
- stopping motion (with various degrees of urgency)

The message content was developed by analyzing the documentation for commercial robot controllers (e.g., VAL-II and Karel), and a similar document that analyzed numerical control

(NC) code dialects for machine tools to develop a set of canonical machining commands (Proctor & Kramer, 1997).

CRCL is intended for use with devices typically described as industrial robots and for other positionable automated devices such as automated guided vehicles (AGVs). An AGV with a robotic arm attached may be regarded as a single robot responding to a single stream of CRCL commands or as two robots (the vehicle platform and the robotic arm) responding to two separate streams of CRCL commands. The mapping between CRCL commands and status messages depends on the underlying implementation. A robot vendor could choose to provide a CRCL interface directly. If the robot vendor provides a software development kit (SDK), an end user or third party could implement the mappings themselves. The authors have used this approach, as described in Section 3. An implementation using ROS has also been developed, leveraging the ROS mappings that have already been developed.

The meaning of CRCL is specified by the ontologies noted in Section 1, and codified as information models written in XML Schema Definition Language (XSDL). The meaning is captured through explicit representation of the attributes that pertain to concepts, limitations on what information can be contained within those attributes, and relationships to other concepts within the schema. CRCL messages follow a simple protocol that governs when messages are valid, and how they can be associated with status. Communication takes place within a “session” that begins with an initialization message, and concludes with a termination message. Each command in a session is associated with a unique serial number, and information on its progress and result are provided in status messages that indicate the serial number. General status items, such as the current position, are independent of commands. The information in status messages, and the frequency at which they are sent, can be tailored by the application during a session to make efficient use of communication bandwidth.

CRCL is divided into five separate information models. Three of these models specify messages: messages to the robot containing a single command, messages to the robot containing an entire program, and status messages from the robot back to the sender of commands. Two other models define the overall structure and common elements of CRCL command messages, and underlying data types such as poses.

Instances of CRCL messages are eXtensible Markup Language (XML) sequences of plain text characters conforming to one of the three message models. The character sequences may be contained in strings or in files. Compression of large XML messages can be performed using Efficient XML Interchange (EXI) (World Wide Web Consortium, 2009a) if CRCL is sent directly and communication bandwidth is limited.

Cartesian motion commands specify six-degree-of-freedom motions. The point specified in these commands is called the controlled point and is located at the tip of the end effector (or final joint of the robot if the robot has no end effector in place). The orientation is specified by giving the X and Z directions of that coordinate system in the coordinate system of the robot.

Continuous Cartesian motion along a path through waypoints with varying speeds and accelerations in segments of the path is supported by the *MoveThroughToType* command. This command also allows requiring straight line motion between waypoints or allowing the path between waypoints to be determined by the system carrying out the command. In addition, rotational motion may be required (or not required) to be coordinated with translational motion in each segment of the path. Cartesian motion from the current pose of the robot to a single new pose is supported by the *MoveToType* command, which has the same choices of straight line motion and motion coordination as the *MoveThroughToType* command. A *MoveScrewType* command supports motion required to attach a nut to a bolt or to drive a screw.

Joint motion is supported by the *ActuateJointsType* command. It provides an *ActuateJoint* element for each joint that is to be moved. Each *ActuateJoint* element has *JointNumber*, *JointPosition* (translational or rotational), and *JointDetails* elements. The *JointDetails* specify either force (for a translational joint) or torque (for a rotational joint) with an optional rate of change (analogous to acceleration), or optional speed and acceleration for the joint.

Speeds and accelerations may be set either as absolute values, such as 1.5 meters per second, or as relative values, which are fractions of the maximum value possible for the robot, such as 0.7.

Status messages may be returned regularly (at 100 Hz, for example), only when requested by a report status command, or under any other scheme devised by the builder of a robot system. Regular reporting is expected to be the most common use.

A CRCL status message consists of one to many parts:

- the (required) status of the most recent command that was received
- zero to many robot joint status reports
- an optional pose that is the pose of the controlled point and coordinate system
- an optional end effector status report

The command status:

- echoes the ID of the command for which status is being reported
- gives the ID of the status report
- gives the status of the command (one of *Done*, *Error*, or *Working*).

The joint status reports in a status message must be in the order specified by the most recent *Configure-JointReports* command.

3 IMPLEMENTATION

The authors have validated the CRCL specification in two projects, one at the Georgia Tech Research Institute (GTRI), and one at the National Institute of Standards and Technology (NIST).

The architecture of these systems is shown in Figure 1. An ontology for a particular application domain such as assembly is developed and represented in a formal information modeling language such as the Web Ontology Language (OWL) (World Wide Web Consortium, 2009b). The formal descriptions in OWL are converted into a particular pair of problem- and domain-files in the Planning Domain Definition Language (PDDL) (Ghallab *et al.*, 1998) that inform a domain-independent planning application.

A resource ontology supplements that of the application domain, and includes robot information such as range of motion, payload limits, speed, accuracy, and information about tooling, fixturing, and other resources that could be used.

A manufacturing activity begins with a request to fulfill an order. The request is submitted to the planning application, which queries the current state of the world and the capabilities of the available resources from which it generates a plan for the order in PDDL. Once the plan is generated, an executor application translates each PDDL action into a series of CRCL commands, and sends each to the robot controller. The executor application continually tracks the execution status of each CRCL command, ensuring that the preconditions and postconditions of the PDDL commands are met as the statements are processed. If failures are detected, e.g., a missing tool or dropped part, the executor aborts the PDDL plan and calls for replanning. In simple cases, this could be the initiation of a pre-planned PDDL activity to recover and continue. In more complex cases, complete replanning may need to be done, using the current state of the system and a possibly revised goal state.

The current state of the world is maintained in a world model (WM) database which is updated by a sensor processing (SP) system, which continually estimates the state of the environment, synthesizes higher-level information about the state of the world by fusing lower-level sensor information, and updates the WM, which could be implemented in a distributed fashion.

The following sections detail the implementations at the authors' institutions. Although both GTRI and NIST implemented CRCL directly, passing plain text XML messages during application execution, this heavyweight XML parsing may not be suitable for embedded systems. For these systems, CRCL can serve as the normative reference for the behavior of equivalent commands and status, with harmonization documents making it clear how they are mapped.

3.1 GTRI Implementation

GTRI has implemented CRCL on a Fanuc LR Mate 200iD as part of its assembly workcell. As shown in Figure 2, the cell is currently being used to assemble mixed kits of gears from uniform trays of parts. The overall system is designed to be agile and flexible; it is able to build kits directly from knowledge stored in the system's ontology, and is able to detect and recover from assembly errors. The system follows the architecture shown in Figure 1 and incorporates a PDDL planner to compose a general plan for the construction of the kit based on knowledge from our formal world model. More information on the overall planning and execution system may be found in (Balakirsky, 2015). The PDDL vocabulary of the system is custom-designed for each application. Since a domain independent planning system is

used as part of the overall workcell, no changes to the planning system software are necessary to change domains.

Each PDDL action contains a high-level command to the robot. For example, the PDDL command of *pickpart* commands the system to pick a part out of a parts tray. This action is considered as “unbound” since the type of part and the ID of the tray are specified as parameters, rather than the identity of the specific part to be picked. The exact part to pick and its location are determined at run time through the execution of actions specified by the PDDL command decomposition.

At run time, knowledge from the world model, application ontology, and resource ontology is combined in order to decompose each PDDL command into one or more CRCL actions. The *pickpart* command, for example, is decomposed into several CRCL movement and end effector commands. Each of these commands is directly encoded as a *CRCLCommandInstance* and transmitted to the robot. The GTRI system uses a TCP/IP socket connection for the interface, and the plain text representation of the XML is directly transmitted.

The robot control computer is running a Visual Basic program that receives CRCL commands from the socket and reports status back. Position control is accomplished by translating the CRCL commands to parameters for native Karel-language code that is run on the robot. Depending on the end effector being used (which is auto-detected), either Karel routines running on the robot or vendor-provided end effector libraries running on the control computer are used.

The process of kit building does not require a high-speed planner-to-robot control loop. It has been found that it is possible to generate the XML in the executor, transmit it to the robot, parse the XML, and execute the commands so that the robot is in continuous motion. Command flow is controlled through CRCL status messages that are sent only when requested. The GTRI system is able to close the command request-status receipt loop at 55 Hz.

3.2 NIST Implementation

NIST has implemented CRCL on both real and simulated systems. These implementations followed the architecture shown in Figure 1. They differ only in the implementation of the *Robot Controller*.

3.2.1 Real Equipment—CRCL interfaces were developed for two resources in a robotic kitting application: a Kuka LWR 4 seven-axis robot arm, and a Robotiq three-fingered dexterous hand. A single program served as the CRCL focal point for both the robot and dexterous hand, reading commands for each, converting them into the native communication formats, monitoring execution of each, and reporting CRCL status messages. Thus, the *Robot Controller* in Figure 1 was actually two controllers, one for the robot and one for the hand. Communication was done via plain text XML over network socket connections, where the focal application served one port for each. From the *Executor* point of view, these were two controllers, with a client connection to each.

The CRCL commands were generated by the *Executor* from a higher-level input format in PDDL. The PDDL plan file was generated automatically when requests to build kits of parts were submitted as a *Manufacturing request* through an *Operator interface*, in this case a mobile tablet device, as shown in Figure 1. Specific robot poses were computed based on the sensed locations of objects by the *Sensor Processing* system, maintained in a *World Model* MySQL database. Cartesian move requests were made using CRCL, and were converted to native robot commands. The path planning and inverse kinematics calculations were carried out by the robot controller. Likewise, end effector open and close commands were sent as needed to the dexterous hand controller, which managed the details of controlling finger positions that corresponded to open and closed positions.

3.2.2 Simulation—Simulations of the Kuka LWR 4 and Robotiq dexterous hand were also undertaken using both the Gazebo (The Open Source Robotics Foundation, 2016) and V-REP (Rohmer *et al.*, 2013) physics-based simulators. These support several real-time physics engines that provide realistic simulations of effects such as friction and inertia. Models for the robot and hand were obtained and configured to run within these simulators. For Gazebo, built-in capabilities to communicate with ROS were used, with ROS's *MoveIt!* package providing motion control and performing inverse kinematics calculations that convert Cartesian-space move requests to joint motion. A CRCL-to-MoveIt application translated CRCL move requests to messages on MoveIt's input topic, following ROS's publish-subscribe model. The image on the left of Figure 3 shows a Gazebo simulation of the robot and hand prior to acquiring a peg.

For V-REP, CRCL integration was achieved using a plug-in that implemented the ROS-Industrial Simple Message (ROS, 2014) protocol. Simple Message is a lightweight binary protocol suitable for implementation on embedded controllers that provides joint-level control, and is integrated with several ROS packages for planning and executing trajectories. The authors extended Simple Message to support Cartesian control, and wrote a proxy application that converts CRCL command and status messages to and from Simple Message. The image on the right of Figure 3 shows a V-REP simulation of the robot and hand in a kitting application.

4 SOFTWARE TOOLS

The CRCL applications described in Section 3 were written in the C++, Visual Basic, and Python programming languages. Since the CRCL specification does not define the syntax of language bindings, common libraries or modules supporting bindings were developed to handle the reading and writing of XML messages into and out of classes for each message type. These author-developed tools provide their own syntax, which will in general be different from the syntax that other tools or developers would use when implementing the CRCL specification. Thus, source code portability between different applications will not be achieved unless the developers agree on using these same tools. The source code for these tools is publicly available from the ROS-Industrial Consortium (Southwest Research Institute & ROS-Industrial Consortium Americas, 2014).

4.1 CRCL Applications with C++

C++ code and YACC-Lex parsers have been developed at NIST for dealing with CRCL command messages and status messages in XML format. The messages are sent to or from a low-level controller for a robot or an end effector. The code and parsers can be used as follows: 1) When a status message comes through, a call to the status parser populates a tree of C++ class instances. One can go down the tree to retrieve all the information in the status message. 2) To send a command to a robot or to an end effector, a CRCL command is first built as a tree of C++ classes and then packaged into a message before being sent through a message system.

The C++ code and parsers for CRCL were generated automatically by a generator built at NIST (Kramer *et al.*, 2015). The generator reads an XML schema that models a particular domain (CRCL commands and CRCL status in this case) and writes software for a parser that reads and writes XML instance files or strings conforming to the schema. For each schema file, a C++ header file defining classes for the domain, a C++ code file implementing the classes, and a C++ code file with a main program are generated. YACC and Lex files for a parser that populates the C++ classes are also generated.

4.2 CRCL Applications with Java

Java language support is provided through a project that is split into modules, so that client applications can depend on only the parts that are necessary for that application. These modules include a base module with JAXB (Java Architecture for XML Binding) annotated classes generated by the xjc utility that correspond directly to the XML types in the schema; a utilities module with network, math, and conversion utilities; a user interface module that supports the development of Swing-based desktop graphical user interfaces; a Webapp module with a Vaadin-based web application alternative user interface; and a FANUC robot server module built with Com4J that provides an additional way of controlling the testbed's FANUC robot through CRCL. The utilities module was used to develop examples for CRCL clients and servers in Java as well as C++, Matlab, Python, and Javascript.

For testing, a CRCL simulator was developed that responds to command messages and replies with a stream of status messages that plausibly behave like a real robot and end effector. For example, in response to a *MoveToType* command for the robot to move to a target pose, the simulator progresses the simulated robot pose according to the established speed, continually updating CRCL status messages. Error conditions can be triggered if desired to test the error handling of the sending application. A counterpart CRCL graphical user interface developed using the Swing user interface module allows any CRCL message to be sent to either a robot or end effector, and the viewing of their current status. The CRCL graphical user interface can also connect to real devices for interactive testing, while the CRCL simulator can stand in for physical devices that may be unavailable, and allow the testing of other parts of a full application. The same graphical user interface that connects to the simulation server has also been tested with the testbed's FANUC robot.

4.3 CRCL Applications with Python

For Python, a module was manually written to handle XML reading and writing, and serialization and deserialization using TCP/IP sockets. The XML hierarchy from generic types down to actual messages was preserved, and the standard Python methods for string output and class construction were implemented to convert between class instances and streams of plain text XML. Generic tools for generating Python module code from XML schemas are available, such as Python XML Schema Bindings (PyXB), but manual writing of the module was preferred to reduce the dependencies on external Python modules.

5 CONCLUSION

Agile tasking of robotic systems is possible using off-line programming supplemented with calibration, modeling, and sensing. To extend the benefit of reduced application development time throughout a manufacturing enterprise, a common information model for tasks is necessary. The Canonical Robot Command Language is an information model developed by the authors to demonstrate the concept. Formalized using XML Schema, including substantial semantic explanations of the required behavior of commands and their associated status reporting, CRCL can be implemented directly via XML messages, or as the normative reference for other protocols that may be better suited for embedded systems. Implementations at two facilities have shown the benefits of this approach. Software tools for reading and writing CRCL messages are available from the ROS-Industrial Consortium. Standardization of CRCL through the IEEE Robotics and Automation Society is underway. Commercial implementations will be necessary for adoption by end users, and are being pursued through the ROS-Industrial Consortium. While CRCL is sufficient for many assembly and kitting tasks, extensions will likely be needed in order to handle a wider variety of robot applications.

References

- Albert, Mark. Adaptive Machining Makes a Robot More Productive. *Modern Machine Shop*. 2015; 8:78–82.
- Balakirsky S. Ontology Based Action Planning and Verification for Agile Manufacturing. *Robotics and Computer-Integrated Manufacturing*. 2015; 33(1):21–28.
- Bruyninckx H. Open robot control software: the OROCOS project. *Proceedings of the IEEE International Conference on Robotics and Automation*. 2001:2523–2528.
- Clark M. JAUS Compliant Systems Offers Interoperability across Multiple and Diverse Robot Platforms. *Proc AUVSI Unmanned Systems North America Conference*. 2005
- Edstrom, Dave. MTConnect: To measure is to know. 2013 *Virtual Photons Electrons*.
- Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., Wilkins, D. PDDL – The Planning Domain Definition Language. AIPS-98 Planning Competition Committee, Yale Center for Computational Vision and Control; 1998. Tech. rept. CVC TR-98-003/DCS TR-1165(10)
- IEEE. 1872–2015: Ontologies for Robotics and Automation. 2015
- International Organization for Standardization. ISO 10303-238:2007, Industrial automation systems and integration - Product data representation and exchange - Part 238: Application protocol: Application interpreted model for computerized numerical controllers. 2007
- Korn, Derek. Best Practices of Top U.S. Shops. *Modern Machine Shop*. 2015; 8:70–76.

- Kramer, Thomas, Marks, Benjamin, Schlenoff, Craig, Balakirsky, Stephen, Kootbally, Zeid, Pietromartire, Anthony. Software Tools for XML to OWL Translation. National Institute of Standards and Technology; Gaithersburg, MD: 2015. NISTIR 8068(6)
- Open Robot Control Project. OROCOS. 2013
- Open Source Robotics Foundation. Robot Operating System. 2016
- OpenJAUS. OpenJAUS, A Free Open Source implementation of the Joint Architecture for Unmanned Systems. 2013
- Proctor, Frederick M., Kramer, Thomas R. Canonical machining commands. National Institute of Standards and Technology; Gaithersburg, MD: 1997. NISTIR 5970
- Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J. ROS: an open-source Robot Operating System. IEEE International Conference on Robotics and Automation: Workshop on open source software. 2009
- Rohmer E, Singh SPN, Freese M. V-REP: a Versatile and Scalable Robot Simulation Framework. Proc of The International Conference on Intelligent Robots and Systems (IROS). 2013
- ROS. Simple Message. 2014
- Ross, Douglas T. Origins of the APT language for automatically programmed tools. ACM SIGPLAN Notices - Special issue: History of programming languages conference. 2015; 13(8):61–99.
- Southwest Research Institute, & ROS-Industrial Consortium Americas. ROS-Industrial. 2014
- The Open Source Robotics Foundation. Gazebo: Robot Simulation Made Easy. 2016
- Tobe, Frank. What's Hot in Robotics? Four Trends to Watch. The Robot Report. 2015:6.
- World Wide Web Consortium. EXI Primer. 2009a
- World Wide Web Consortium. OWL 2 Web Ontology Language Structural Specification and Functional Syntax. 2009b. <http://www.w3.org/TR/owl2-syntax/>
- Xu X, Klemm P, Proctor F, Suh SH. STEP Compliant Process Planning and Manufacturing. International Journal of Computer Integrated Manufacturing. 2006; 19(9):491–494.

Figure 1.
The architecture of a complete robot kitting application, with the use of CRCL shown.

Figure 2.

Georgia Tech's kitting cell. The cell is able to pick parts from uniform parts trays (red) and place them in a mixed kit (blue). No pre-programming is performed, and all knowledge is contained in the knowledge ontology. Control is operated in a loop closed by a vision system.

Figure 3. Simulations of a robot and dexterous hand. Gazebo (left) is simulating the moving of a peg. V-REP (right) is simulating a kitting application.