Modeling Robot Behavior with CCL

Konrad Kułakowski and Tomasz Szmuc

Department of Applied Computer Science,
AGH University of Science and Technology
Al. Mickiewicza 30,
30-059 Cracow, Poland
{konrad.kulakowski,tomasz.szmuc}@agh.edu.pl

Abstract. This paper presents the use of a *Concurrent Communicating Lists (CCL)* library in robot behavior modeling. *CCL* provides several software components, which allow the model to be built, simulated and formally verified. Due to the integration with the *Robust* library the *CCL* models can be deployed and executed on the actual hardware platforms. Besides the modeling robot behavior, the work also addresses the problem of modeling a robots environment.

The *CCL* models can be verified either formally or by simulation. Since the use of formal methods is always associated with the state explosion problem, the work provides practical guidelines on how to deal with this problem using *CCL*.

1 Introduction

In recent years, increased interest in the design and building of robots has been visible. Robots have become accessible to a wide audience. The ease and availability of even sophisticated robotics platforms encourages researchers to seek new, efficient methods of modeling of control software for such constructions. One of them can be *Concurrent Communicating List (CCL)* - the *Clojure* language library supporting executable modeling of concurrent and distributed systems. It allows users to write a control program in a special *lisp-like CCL* notation, run it step by step in a simulation mode, perform their formal verification or execute them like a regular computer program.

The first two sections of this paper contain a brief outline of *AI* robotic architectures and, on this basis, tries to draw a map of various approaches to the modeling of AI robot software. Section 3 summarizes the *CCL* library. Section 4 presents a simple control algorithm allowing the robot to move and sense. Section 5 discusses *CCL* in model simulation and formal analysis. Finally, Section 6 includes a work summary and presents the plan for future research and development.

2 Robotics Models and Architectures

The architecture design in mobile AI robotics tries to follow the three intelligent control architectural styles [2]:

I. Noda et al. (Eds.): SIMPAR 2012, LNAI 7628, pp. 40–51, 2012.

- Hierarchical Planning and Control Architecture
- Reactive/Behavior Based Control Architecture
- Hybrid Architecture

One of the most influential representatives of the first approach is *"A Reference Model Architecture of Intelligent Control"* proposed by *J. S. Albus* and *A. M. Meystel* later on implemented as *4D/RCS* [1]. The Albus model provides several levels of control nodes, where each of them is able to sense the environment, judge the situation on a certain level of granularity and generate behavior. The nodes higher in the hierarchy take strategic decisions and perform actions on the higher level of abstraction, whilst the nodes lower in the hierarchy have the shorter time perspective and perform simpler actions. All the nodes maintain the data base (world model) storing important facts about the environment.

Another architectonic style is determined by the famous *Subsumption Architecture* proposed by Brooks [5]. Following the principle *"The world is its own best model"* it focuses on immediate data sensing and behavior generation rather than spending time on the possibly resource-consuming: sense, process the knowledge, and execute the plan processing loop. Due to the relative simplicity and intuitiveness of model creation, the reactive approach resulted in a number of works on the various frameworks and notations supporting behavior modeling and analysis [19,21,18,8,17].

The third, the hybrid approach tries to take benefits from both hierarchical planning and a reactive approach. Its supporters argue that the previous two approaches in fact do not exclude each other but rather try to perceive the same phenomenon from two different perspectives. They observe that sometimes intelligent constructions need to behave in a reactive manner, and at other times to perform careful knowledge-based hierarchical planning. Example of this approach is *AuRA* [14].

Among the papers that focus on modeling system behavior, there is an important group of works that use formal methods. Using formalisms allows the system behavior to be specified more clearly and efficiently, and opens the possibility of using formal techniques for validation and verification of the model. An example of such an approach is *Behavior Language* [6]. This is directly derived from the widely recognized *Subsumption Architecture* [5]. Its syntax is based on the *AFSM (Augmented Finite State Machines)* description language, which allows the model to be compiled and deployed on different hardware platforms such as *Motorola 68000* or *Hitachi 6301*. Other robot behavior specification languages using the state machine concept are *COLBERT* [10] and *XABSL* [19]. The first of them, supporting the *SAPHIRA* platform, is designed for modeling behavior of individual robots, whilst *XABSL* tries to address the problem of behavior specification for multi-robot systems. In addition, *process algebras* [4] or *behavior trees* are represented as formalisms for modeling robots behavior [9,20]. *Petri Net Plans (PNP)* [22] proposes *Petri nets* layer as an actual model specification language, and then offers possibility of formal model verification. Although formal methods are often used for modeling behavior, the reactive systems that use them also take benefits from hierarchical approach. An example of

solution, which tries to use both reactive and hierarchical methods is *RS (Robot Schemas)* [15].

CCL notation is derived from process algebras and primarily focuses on behavior modeling. It defines operators and actions (as with the process algebras) which are used later to create more complex expressions forming a model specification. There are two types of communication, internal, between two different processes within the model, and external, between the model and the rest of the system (e.g. *world model*). Such a distinction provides modularization, since once module can be completely external to the other module. *CCL*, due to its close relationship with process algebras, gives the possibility to perform formal verification of the model. Some operations, such as *deadlock* finding or *bisimulation* checking, are supported directly by the *CCL* library. Others, such as temporal formula verification, are supported by exporting the model into the *CADP* tool [7]. *CCL* is executable, which means that all the models can be freely simulated and executed. For the purpose of simulation, the external environment can be modeled using the *CCL* simulation environment *CCL Sim* [12].

3 CCL Library at a Glance

3.1 CCL Notation

The *CCL* syntax is modeled on the process algebras, such as *CCS* [16], and the *Clojure* and *Lisp* language. For this reason all the *CCL* expressions are in the form of lists, and they are built up from the operations, which are close in meaning to what can be found in algebraic notations. The *CCL* model consists of lists denoting processes, communication channels between them, and primitives – *Clojure* functions, which are called by the processes during the execution of a model. The basic notion introduced by the *CCL* notation is the *nlist* expression denoting the sequence of operations to execute. The *nlist* expressions are executed within the *CCL* processes launched as the part of the concurrent composition clause. Processes can be anonymous or named. A brief *CCL* syntax summary can be found in *Table* 1[1]. The *CCL* processes communicate via blocking queues. The use of the synchronization queue mechanism is possible through the set of queue access methods, such as: q-get, q-put, q-peek, q-try-put, q-size and q-capacity. The q-get and q-put functions add and remove elements from the queue. These functions can be blocking or non-blocking depending on the adopted strategy and the number of elements in the queue. The next two functions q-peek and q-try-put behave like q-get and q-put but they do not wait. When they fail the *nil* value is returned. The last two functions do not change the state of the queue. They return the number of actual elements in the queue (method: q-size) and return the maximal possible size of the queue (method: q-capacity). Depending on the adopted policy and the length of the queue, the processes attempting to read from or write into the queue can be blocked for a while or returns immediately.

[1] The more comprehensive syntax reference with examples can be found at `www.kulakowski.org/ccl`

Table 1. *CCL* - Syntax summary

Construction	Description
(defn Foo [] body-expr) (reg-as-prim Foo)	Defines the *Clojure* function *Foo* and registers it as a *CCL* primitive.
def-nlist Boo (exp_1 exp_2 ... exp_N))	Defines the *nlist Boo* executing its nlist-body i.e. the list of subsequent expressions: exp_1, $exp_2, \ldots, exp_N$.
(def-nlist (Roo :y) ((exp_1 :y) ... (exp_k :y))	Defines the *nlist* named *Roo* with the initial parameter :y. The expressions in *Roo's* nlist-body can freely use the parameter :y.
((:x (Foo)) (Roo (+ :x :y)))	Defines the local *nlist* variable :x and initializes it to the value returned by *Foo*, then starts execution of *Roo* with the input value set to the sum of :x and :y.
(? (cond_1) (nlist_1) ... (cond_N) (nlist_N))	The conditional choice operator allows the definition of the nlist-expression to be executed next depending on their condition expressions, i.e. if cond_k is the first true expression on the left then the nlist_k expression is to be executed.
(?? X_1 (nlist_1) X_2 (nlist_2) ... X_N (nlist_N))	Within the random choice statement, nlists are picked for further execution randomly. The chance of being selected for nlist_k is given as: $x_k / \Sigma(x_1,\ldots,x_k)$
(| Moo :moo Goo :goo)	As a result of execution of this expression two *CCL* processes have been launched, where the first process labeled *:moo* will execute the *nlist Moo*, whilst the second process *:goo* will execute the *nlist Goo*.

Due to the blocking property and the maximal number of elements in the queue (it is assumed that a queue can be zero-length or non zero-length) there are eight possible types of synchronization queue. All of them have been summarized in Table 2. There are five columns, where *type* means the type id of a synchronization queue, *cap.* comes from the maximal capacity of the queue, *read* and *write* determines whether the operations read and write are blocking and non-blocking. Since these parameters affect the meaning of queuing methods, the fifth table column contains a brief function semantics summary. Synchronization queues are used for modeling communication between different processes within the model. Communication between the external environment and the model is implemented by primitives call (Table 1). In such a case all the technicalities of a communication channel are hidden and it is assumed that the function call returns a correct result as soon as possible.

Table 2. Synchronization queues - functions meaning

Type	Cap.	Read	Write	Functions meaning
1	0	n-b	n-b	The 0-length queue is always empty. Thus, all the operations except q-size and q-capacity, are ineffective.
2	0	b	n-b	Since at the given point of time the queue is empty (there is no space to store the element for any non-zero period of time) the operations q-peek and q-try-put are ineffective. The function q-get always blocks and waits for the counterpart q-put. The function q-put always adds the element to the queue. If there is no waiting q-get on the other side the inserted element is lost.
3	0	n-b	b	As for type 2 operations, q-peek and q-try-put are ineffective. The function q-put always blocks and waits for the counterpart q-get. The function q-get removes the element from the queue. If there is no waiting q-put on the other side the returned element is *nil*.
4	0	b	b	As for type 2 operations, q-peek and q-try-put are ineffective. The function q-put always blocks and waits for the counterpart q-get, and reversely the function q-get always blocks and waits for the counterpart q-put. When both functions meet each other q-get returns the element inserted by q-put.
5	$k > 0$	n-b	n-b	The functions q-peek and q-try-put are ineffective, since they work as q-put and q-get. The function q-get is successful if the queue is non-empty, q-put when the queue is not full.
6	$k > 0$	b	n-b	The function q-try-put is ineffective. The function q-get blocks until the queue is empty. The function q-put fails immediately when the queue is full.
7	$k > 0$	n-b	b	The function q-peek is ineffective. The function q-get fails immediately when the queue is empty. The function q-put blocks as long as the queue is full.
8	$k > 0$	b	b	The function q-get blocks as long as the queue is empty, and similarly the function q-put blocks as long as the queue is full.

CCL notation provides an *externalization* mechanism which allows the synchronization queue to be wrapped within the primitives call, so that the explicit communication link between two processes becomes external to the model. This leads to a decrease in model complexity[2] as regards the number of inter-process synchronizations, and finally may result in splitting one model into several submodels. Thus, the externalization mechanism introduces modularity, so that one

[2] Of course, at the expense of model accuracy.

model can be independently modeled and analyzed from the others. This property seems to be especially useful when different parts of the model are loosely coupled as, for example, a sub-model of a robot and the sub-model of its environment. The *CCL* library[3] provides a few *APIs* allowing the model of a system to be created, executed or simulated, and formal analysis of a model to be carried out. In addition, the *CCL* software bundle contains *CCL Sim* [12], an interactive model development environment facilitating step-by-step tracking of the model and building various mockups helping simulation of an external environment model.

3.2 CCL Software Setup

One of the key component of the *CCL* software setup is the *Robust* platform. This was originally conceived as a simple *Mindstorm NXT Java* library moving *CPU* intensive processing to a *PC* platform and providing a robust and efficient *PC-NXT* communication link. With time, *Robust* gained new components allowing the creation of control programs running on another robotic platform *Hexor II* [13], and the *cljRobust API* [11] interfacing *Robust* with the *Clojure* programming language. In this way *cljRobust API* functions can be declared as *CCL* primitives, then the models written in *CCL* notation can actually control the mobile robots supported by the *Robust* library. Such a tool-chain involves a few additional, not explicitly mentioned yet, software components. In the case of the *NXT* computer LeJOS - the embedded version of *Java* for *Mindstorms NXT* is required. Hexor II comes with its own operating system and proprietary control libraries. The *Robust* library as well as *CCL* are run under the control of a Java Virtual Machine. The same applies to the *Clojure* language library which binds the *Robust* platform and *CCL APIs* together. When working with *CCL* models, choosing one of a few professional *Clojure* developer environments[4] is worth considering.

4 Modeling Robot Behavior - Study Case

One of the basic *CCL* constructions is primitive. From the system modeling perspective a primitive is like an indivisible action, which can take some parameters from the model and return the computed value. Implementation details behind the primitive call are not important except for the fact that primitives should be interruptible, i.e. as functions executed within the JVM threads they should be able to safely break ongoing operations when the interruption request is raised. Since the primitive is able to transmit the values to and from the model, it can be used for implementing communication between the robot model and the robot model's environment. Due to the externalization mechanism, there is no

[3] The *CCL* library binaries, manual and examples are available at `www.kulakowski.org/ccl`

[4] There are, for instance: Eclipse with counterclockwise plugin and Net Beans with enclojure plugin.

need to fix the model boundaries at the very beginning, and the designer is able to decide later on where the robot model stops and where the model of the environment starts. Let us consider a simple reactive robot with one touch sensor (bumper) sending a short stimulus when the robot hits the obstacle. Implementing that with *CCL* and *Robust* requires definition of the synchronization queue touch-event-source (Listing: 1, line: 1) and definition of the *Clojure* function touch-handler (Listing: 1, line: 2) being an event listener hooked up in *Robust API*. When the *bumper* hits the obstacle, touch-handler puts an element into the touch-event-source queue (Listing: 1, line: 4).

```
1 (def-queue touch-event-source :size 1 :rb)
2 (defn touch-handler [value]
3   (if (= value 1)
4     (agh.ccl.nlists/q-put :touch-event-source 1)))
```

Listing 1: Communication between the robot model and its environment - executable version

In response to the appearance of an element in the queue the robot should retreat a little bit the same way it came and then choose the other direction. That simple behavior can be easily specified using standard *CCL* constructions.

```
5 (def-nlist ExplorationRobot
6     ( (rb-system-startup)
7     (rb-touch-async-handler touch-handler)
8     (rb-move-forward (rnd 200 400) 200)
9     (| GoAhead :goAhead CollisionDetector :colDetector)))
10 (def-nlist GoAhead
11     ( (! AvoidObstacle)
12     (rb-move-forward (rnd 200 400) 200)
13     (rb-move-wait-for-new-move)
14     GoAhead))
15 (def-nlist AvoidObstacle
16     ((rb-move-forward 200 -100) ; move backward
17     (rb-move-inplace-turn (rnd -120 120) 100)
18     (rb-move-forward (rnd 200 400) 200)
19     GoAhead))
20 (def-nlist CollisionDetector
21     ( (q-get touch-event-source)
22     (rb-move-stop-now)
23     (-> :goAhead)
24     CollisionDetector ))
```

Listing 2: Random exploration. CCL/Robust executable behavior specification

The first *nlist* expression ExplorationRobot (Listing: 2, line: 5) calls the mandatory *Robust* initialization function rb-system-startup (line: 6), registers the touch-handler, puts on the execution queue the one *move forward* command, and then launches two threads :goAhead and :colDetector. They start executing

correspondingly the GoAhead (line: 10) and CollisionDetector (line: 20) expressions. The first action of the GoAhead expression is to register (operator !) an interruption handler AvoidObstacle (Listing: 2, line: 11). Thus, when the interruption request has been raised, the :goAhead thread immediately starts processing the AvoidObstacle expression. Next GoAhead follows the processing loop: queues one straightforward move with a randomly chosen length between 200 and 400 millimeters and speed 200 (line: 12), waits until the currently executed move ends (line: 13), and starts execution from the beginning (line: 14). In the case of the robot's bumper hitting into an obstacle, the *CCL* process :goAhead is interrupted and the fallback procedure is executed. In such a case, the AvoidObstacle *nlist* expression is executed (Listing: 2, lines: 15 - 19), i.e. after withdrawal of the robot 100 units back (line: 16), the new random direction is chosen (line: 17), and the construction continues moving ahead (line: 18). `CollisionDetector` (Listing: 2, lines: 20-24) is the last expression in the *Random Exploration* example. It is designed as a collision listener, which in the case of collision immediately stops the whole construction (line: 22) and interrupts the :goAhead process execution (line: 23).

5 Model Simulation and Formal Verification

Although the model as presented on Listings 1 and 2 is fully executable[5] its simulation and formal analysis require the introduction of several additional enhancements. For the purpose of simulation, due to the lack of the *Robust* library, all the functions referring to the external environment provided by *Robust* need to be replaced by mockups or modeled in *CCL* as sub-models. The *CCL* library supports simulation experiments by providing the additional GUI application *CCL Sim [12]*, together with a *ccl-sim-utils* API allowing for creation of primitives controlled remotely from within the *CCL Sim*. Hence, every action made by the model upon the external environment can be logged, and every sensor reading request can be manually handled in *CCL Sim*. An example of mockup implementation of (Listing: 3) first waits a random amount of time (no longer than 200 milliseconds, and not shorter than 100 milliseconds), then inserts a log entry, which shows up in the *CCL Sim* application's dashboard.

```
25 (defn rb-move-inplace-turn [x y]
26   (do (wait 100 200)
27       (ccl-sim-model-log-writer "rb-move-inplace-turn" x y)))
```

Listing 3: An example cljRobust API mockup implementation

Although *CCL Sim* can capture all the I/O communication between the model and its environment, it does not allow the external world to be modeled. Its functionality is limited to receiving data from the model and sending the manually chosen or automatically pre-specified values back to the model. Such a solution

[5] The whole model code, together with a short movie showing the model execution can be found at:
`http://www.kulakowski.org/ccl/`

works very well when the model need to be debugged, but it is less useful in the case of the long-term simulation runs. In the second case, the external environment needs to be modeled as a separate sub-model TouchHandler. In the considered example TouchHandler sends (Listing: 4) in a loop an interrupt request (Listing: 4, line: 29) then waits a random amount of time (line: 30) and starts its execution once again (line: 31).

```
28 (def-nlist TouchHandler (
29     (q-put touch-event-source 1)
30     (wait 300 700)
31     TouchHandler))
32 (def-nlist TouchWorld (| TouchHandler ExplorationRobot))
```

Listing 4: An example cljRobust API "dummy" implementation

Now the model (expression *TouchWorld*) is self-contained in the sense that there is no explicit synchronization queue leading outside the model. Thus, it is possible to generate a graph where nodes represent states of the model and arcs between them possible state transitions, and then perform their formal analysis. For formal reasons it is convenient to call such a graph as a labeled transition system *(LTS)* [3]. The *LTS* for such a simple model has 261 states and 707 transitions[6] - in this approach the state in this formalization is represented by the set of states of synchronization queues and the set of states of all the processes. In the adopted approach the values of variables are not taken into account during the *LTS* construction, thus the state of a synchronization queue is reduced to its length, and the every deterministic choice *?* is reduced to its non-deterministic counterpart *??* (Table 1). The change of state is determined both by the primitive call and the operator evaluation. Thus, the transitions can take the labels of both primitives and operators (Fig. 1).

Using *CCL* shell its easy to check that the *TouchWorld* model is deadlock free, compare the *LTS* with another *LTS* in terms of the weak and strong bisimulation [4] or export it into *CADP* [7] and check other temporal properties like *safety* [3]. Sometimes it is convenient to analyze only a part of the model. In such a case all the synchronization queues leading outside the sub-model of interest need to be wrapped into primitives. Thus, in the case of TouchWorld, to be able to to separately analyze ExplorationRobot and TouchHandler, the queue touch-event-source needs to be externalized. For this purpose the queue definition gets a new flag :ext (Listing: 5, line: 33), and both queue ends need to be accessed through the wrappers (lines: 34 - 35). After replacing the queue operations q-get and q-put by their wrapper counterparts the model is ready to be analyzed locally (Listing: 5).

The touch-event-source externalization reduces *LTS* almost ten times to 35 states and 62 transitions (Fig. 1). Of course, the reduced *LTS* loses the information related to states of the sub-model mimicking the robots environment, thus it is impossible to automatically prove that every state of the environment

[6] Obtained by calling the *CCL* shell command (gen-lts WorldModel).

```
(def-queue touch-event-source :size 1 :rb :ext)
(defn-queue-wrapper bumper-stat-out touch-event-source :get)
(defn-queue-wrapper bumper-stat-in touch-event-source :put)
(reg-as-prim bumper-stat-in bumper-stat-out)
(def-nlist TouchHandler ((bumper-stat-in :touch-event-source 1)
    (wait 300 700) TouchHandler))
(def-nlist CollisionDetector (
    (bumper-status-out)
    ...
```

Listing 5: An example cljRobust API "dummy" implementation

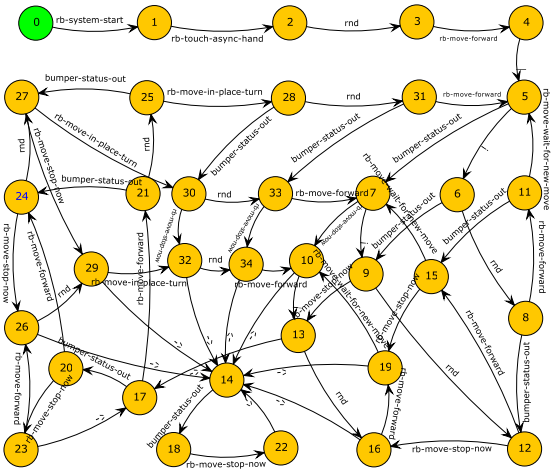

Fig. 1. LTS Graph for ExplorationRobot after touch-event-source externalization

is covered by the appropriate behavior of a robot, but still some important and sound model properties can be proved. That is because, when wrapping the q-get function into the bumper-stat-out primitive, the implicit assumption was made that the bumper-stat-out function behaves like any other function i.e. when called it returns some value in a finite amount of time. Since the robot moving straight forward all the time should sooner or later hit the obstacle, it seems to accurately reflect the actual behavior of the robot under control of the ExplorationRobot algorithm. It is also consistent with the behavior modeled by the TouchHandler expression. Thus, marking communication between TouchHandler and ExplorationRobot as external (thus not considering it during ExplorationRobot sub-model analysis) does not limit verifiability of other temporal model properties, such as deadlock freedom or liveness. Moreover, the reduced *LTS* is small enough to be browsed and analyzed manually. Of course, there is no one golden rule for deciding when the communication channel can be hidden by externalization. In general, it is assumed that the states of models on both ends of the synchronization queue are loosely coupled, hence the omission of one sub-model

will have little effect on other sub-models. Of course, by excluding the synchronization queue out of the model a designer runs the risk of losing something important, so eventually he has to decide whether this will affect the property he wants to examine.

6 Summary and Future Work

In this paper the new *CCL (Communicating Concurrent Lists)* notation and its application to robot behavior modeling has been presented. The proposed notation is supported by the *CCL* library, which offers several software components allowing for model building, model execution, model simulation and debugging and formal analysis and verification of the model. The *CCL* library integrates well with *cljRobust* and the *Robust* library, thus all the models created and verified in the *CCL* notation can be easily executed on the actual hardware platforms.

The article also tackles the hard problem of modeling the boundary between the model of robot behavior and the external environment. The *CCL* library addresses the problem by providing the *CCL Sim* simulation environment, which can imitate the outer environment, and allowing users to write the model of the external world directly in the *CCL* notation. In the latter case, sometimes it makes sense to separately analyze the model of robot behavior and the model of the surroundings. *CCL* facilitates such analysis by providing externalization - an effective syntactic mechanism supporting sub-model separation.

Although the *CCL* library is ready to download and use, still a lot of problems need to be addressed. Since, initially, the *CCL* was designed as a set of *Clojure* macros rather than a regular modeling language, the syntax error information is difficult to understand for end-users. Thankfully, work on the new *CCL* parser is already underway. At the moment the *CCL* library supports only a limited number of predefined formal methods itself. Thus, the project will also try to provide methods which allow for easy construction of any temporal formula.

Acknowledgment. This research is partially supported by AGH University of Science and Technology, contract no.: 11.11.120.859.

References

1. Albus, J.S., et al.: 4D/RCS: A Reference Model Architecture For Unmanned Vehicle Systems Version 2.0. Technical report, NIST Interagency (2002)
2. Arkin, R.C.: Intelligent Control of Robot Mobility, ch. 16. Wiley (2007)
3. Baier, C., Katoen, J.: Principles of model checking. The MIT Press, Cambridge (2008)
4. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): Handbook of Process Algebra. North-Holland (2001)
5. Brooks, R.A.: A robust layered control system for a mobile robot. IEEE J. Robot. and Auto. 2(3), 14–23 (1986)

6. Brooks, R.A.: The behavior language; user's guide. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory (1990)
7. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
8. Groves, W., Collins, J., Gini, M.: Visualization and analysis methods for comparing agent behavior in TAC SCM. In: AAMAS 2009: The 8th International Conference on Autonomous Agents and Multiagent Systems (May 2009)
9. Hardey, K., Mattis, M., Goadrich, M., Corapcioglu, E., Jadud, M.: Exploring and Evolving Process-oriented Control for Real and Virtual Fire Fighting Robots. In: Proceedings of Genetic and Evolutionary Computation Conference (2012)
10. Konolige, K.: COLBERT: A Language for Reactive Control in Sapphira. In: Brewka, G., Habel, C., Nebel, B. (eds.) KI 1997. LNCS, vol. 1303, pp. 31–52. Springer, Heidelberg (1997)
11. Kułakowski, K.: cljRobust - Clojure Programming API for Lego Mindstorms NXT. In: Jędrzejowicz, P., Nguyen, N.T., Howlet, R.J., Jain, L.C. (eds.) KES-AMSTA 2010, Part II. LNCS, vol. 6071, pp. 52–61. Springer, Heidelberg (2010)
12. Kułakowski, K.: CCL Sim, the simulation environment for concurrent systems. In: Proceedings of Dependability and Complex Systems, DepCoS (2012)
13. Kułakowski, K., Matyasik, P.: RobustHX - The Robust Middleware Library for Hexor Robots. In: Ando, N., Balakirsky, S., Hemker, T., Reggiani, M., von Stryk, O. (eds.) SIMPAR 2010. LNCS, vol. 6472, pp. 241–250. Springer, Heidelberg (2010)
14. Long, L., Hanford, S., Janrathitikarn, O.: A review of intelligent systems software for autonomous vehicles. In: IEEE Symposium on Computational Intelligence in Security and Defense Applications, CISDA (2007)
15. Lyons, D.M., Arbib, M.A.: A formal model of computation for sensory-based robotics. IEEE Transactions on Robotics and Automation 5(3), 280–293 (1989)
16. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
17. Nalepa, G.J., Biesiada, B.: Declarative Design of Control Logic for Mindstorms NXT with XTT2 Method. In: Jędrzejowicz, P., Nguyen, N.T., Hoang, K. (eds.) ICCCI 2011, Part II. LNCS, vol. 6923, pp. 150–159. Springer, Heidelberg (2011)
18. Rai, L., Kook, J., Hong, J.: Non-Deterministic Behavior Modeling Framework for Embedded Real-Time Systems Operating in Uncertain Environments. Journal of Information Science and Engineering 26(1), 83–96 (2010)
19. Risler, M., von Stryk, O.: Formal Behavior Specification of Multi-Robot Systems Using Hierarchical State Machines in XABSL. In: Workshop on Formal Models and Methods for Multi-Robot Systems, pp. 1–7 (August 2008)
20. Xiao, W., Liu, T., Baltes, J.: An intuitive and flexible architecture for intelligent mobile robots. In: The Second International Conference on Autonomous Robots and Agents (ICARA), Palmerston North, pp. 52–57 (2004)
21. Zhang, Q., Zhang, Y.-F., Qin, S.-Y.: Modeling and analysis for obstacle avoidance of a behavior-based robot with objected oriented methods. Journal of Computers 4(4), 295–302 (2009)
22. Ziparo, V.A., Locchi, L., Nardi, D., Palamara, P.F., Costelha, H.: Petri net plans: a formal model for representation and execution of multi-robot plans. In: AAMAS 2008: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (May 2008)