2007 IEEE International Conference on
Robotics and Automation
Roma, Italy, 10-14 April 2007

Enhancing Software Modularity and Extensibility: A Case for using Generic Data Representations

Gregory Broten
Defence Research and Development Canada – Suffield
{*greg.broten*}@*drdc-rddc.gc.ca*

***Abstract*— Portable, modular and extensible software allows robotics researchers to pool their resources by sharing algorithms, thus advancing research in the field of robotics at a faster rate than is possible under a non-collaborative model. The development and use of frameworks and middleware, allowing researchers to encapsulate robotic capabilities within a component structure, has traditionally been the focus of robotics software engineering research. Although components greatly enhance the software mechanism's portability, modularity and extensibility, they do not directly address the algorithmic issues confronting developers of robotics software. Software algorithms, implementing specific robotic capabilities, require input data and produce output results. As a rule, these input/output data representations are closely tied to a given algorithmic implementation and hence impose limitations on modularity and extensibility. This paper investigates the use of generic data representations to enhance software modularity and extensibility. Experiments, conducted on the DRDC Raptor Unmanned Ground Vehicle, compared the performance of algorithms based upon both generic and algorithm specific data representations. This research has determined that the performance penalty, resulting from generic data representations usage, is manageable by robotic platforms using current off-the-shelf computing platforms.**

I. Introduction

The *holy grail* of software engineering is portable, modular and extensible "software", which can be easily and seamlessly transported to, and integrated into, new applications. Software researchers have made significant progress towards this goal through a component based approach[1] using frameworks and middleware [1], [2], [3], [4], [5], [6]. Components are the key idea behind this approach and Szyperski defines components as "binary units of independent production, acquisition, and deployment that interact to form a functioning system" [7]. Thus, a component is an independent entity that is capable of executing without requiring the services of a complete system and is often considered to be a separate process that runs under its own workspace. Through the use of components, a system can be decoupled into its constituent elements. This decoupling results in portable and modular software that exhibits *plug-and-play* characteristics.

Frameworks define standard design patterns that lead to common or generic component implementations [8]. These design patterns are based upon generic communications patterns, such as send, query or push, through which components share objects [9]. Middleware provides standard mechanisms for moving the data encapsulated by an object between components and this is usually accomplished in a network transparent and platform independent manner.

Thus, the combination of components, frameworks, and middleware allows researchers to create a component repository where they can share their algorithms. Under ideal circumstances, a researcher could *download* available components from such a repository and easily integrate them onto his given robotic platform. Such a process would eliminate the need to re-implement existing and commonly used algorithms. This process of making algorithms transparently accessible to other researchers requires more than common component mechanisms, it also requires data abstractions that implement generic representations.

This paper investigates generic data representation for robotic systems. Section II of this paper introduces the idea of generic data representations. Section III describes the generic data representations used on DRDC's Raptor unmanned ground vehicle (UGV). The performance results using generic data representations are presented in Section IV. Finally, our conclusions are presented in Section V.

II. Generic Data Representations

A. Motivation

The growing acceptance and adoption of components, frameworks, and middleware is a major step towards realizing unfettered software modularity and extensibility. These three concepts implement the software mechanisms that theoretically allow researchers to easily share algorithms, but algorithmic modularity and extensibility requires more than this. It requires that algorithmic implementations are independent from both the input and output data structures.

B. Software Development Process

All algorithms require input data and the results of algorithms are also stored in data structures. Unfettered modularity and extensibility demands that algorithmic implementations be independent from these input and data structures. Figure 1 shows the standard flow of data, from sensing through to processing, in a robotic system. The data flows in this diagram show an obvious implementation strategy, which starts at the sensing devices and proceeds towards the application.

Such an implementation plan initially focuses on the sensing device's requirements. The requirements of the application/algorithm are considered after the device driver software

[1]Often referred to as Component Based Software Engineering (CBSE).

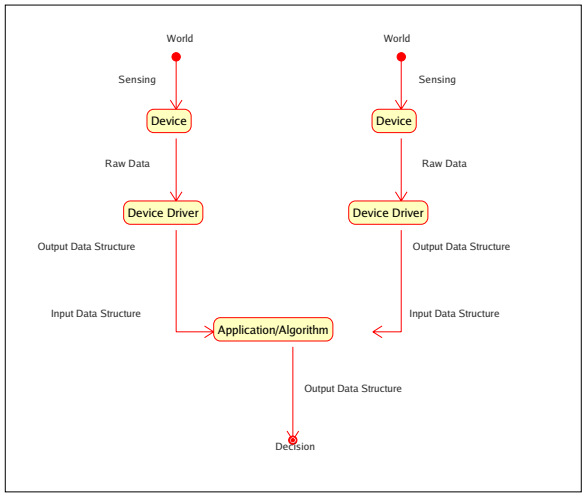

Fig. 1. Standard Data Flow.

has been completed. This approach immediately leads researchers down the path towards "algorithmic dependence".

C. Algorithmic Dependence

Historically there has been an unfortunate tendency for robotics researchers to propagate the raw sensor data structures, from the device driver, towards their application/algorithmic implementation. To illustrate this issue, it is instructive to look at VFH algorithm [10], [11] from the Player open source project [5], [12]. The VFH algorithm input data can come from one of two sources, a horizontally mounted laser rangefinder or a ranging sonar. Player provides a driver for the ubiquitous SICK LMS 200 laser rangefinder. Thus, the SICK laser is commonly used by VFH [2]. Shown below is a partial listing of the structure Player uses to encode the range data returned from a laser:

```
1 /* The max # of laser range values */
2 #define PLAYER_LASER_MAX_SAMPLES  1024

3 /* The basic laser data packet.  */
4 typedef struct player_laser_data
5 {
6 /* Start & end angles for the scan */
7 float min_angle;

8 /* Number of range readings.  */
9 uint32_t ranges_count;

10 /* Range readings [m]. */
11 float ranges[PLAYER_LASER_MAX_SAMPLE];
12 } player_laser_data_t;
```

As can be readily seen on line 2, the PLAYER_LASER_MAX_SAMPLES variable defines the maximum number of laser samples. Although Player allocates 1024 elements for data, the SICK laser usually returns either 181 or 361 data points. Thus, the *ranges_count* variable, found on line 9, stores the actual number of laser ranges received. On line 11, the *ranges* variable reserves a static array of floats for the laser data.

The VFH application uses the PLAYER_LASER_MAX_SAMPLES definition to create an extended internal representation that includes both range and bearing information. As shown on line 3 in the code below, this static allocation is a 2-D array of doubles.

```
1 // Laser range and bearing values
2 int laser_count;
3 double
  laser_ranges[PLAYER_LASER_MAX_SAMPLES][2];
```

The VFH algorithm uses the PLAYER_LASER_MAX_SAMPLES definition to initialize the *laser_ranges* array, as shown below on lines 1 and 2. Unfortunately, VFH is not consistent, as shown in lines 6 through 14 of the code where an algorithmic calculation is performed, since it uses the *magic number* of 181 as a loop counter.

```
1 for(i=0; i<PLAYER_LASER_MAX_SAMPLES; i++)
2   this->laser_ranges[i][0] = -1;

3 // vfh seems to be very oriented around
4 // 180 d scans so interpolate to get 180
5 // b += 90.0;
6 for(i = 0; i < 181; i++)
7 {
8 unsigned int index = rint(i/db);
9 assert(index>=0&&index<data.ranges_count);
10 this->laser_ranges[i*2][0] =
11   data.ranges[index] * 1e3;
12 //this->laser_ranges[i*2][1] = index;
13 // b += db;
14 }
```

The most important conclusion illuminated by these VFH code snippets is not the coding style, but the fact that the VFH algorithm is intimately tied to the original SICK laser data representation.

D. Algorithmic Independence

Modern programming languages, such as C++ using the Standard C++ Template (STL) library, define dynamic arrays such as vectors. Although dynamic vectors are less efficient[3] than their static cousins, they provide the runtime flexibility that is essential for modular and extensible software. The following code snippet, representing the VFH code rewritten in C++, illustrates the flexibility of C++ vectors. On line 2 is the *laser_ranges* variable, which is defined as a vector of doubles. The code on line 3 loops using the size of the received data, while line 4 both allocates storage and initializes the *laser_ranges* variable. Lines 5 through 9 perform the same algorithmic calculation, but on data that in now stored in a STL vector.

```
1 // Laser range
2 vector<double> laser_ranges;
```

[2]Player encodes the laser using polar co-ordinates where the angle between consecutive ranges in implicitly defined by the array size and laser's FOV.

[3]Both computationally and in memory requirements.

```
3 for(i=0; i < data.ranges.size()*2; i++)
4  laser_ranges.push_back(-1);

5 for(i = 0; i < laser_ranges.size(); i++)
6 {
7 unsigned int index = rint(i/db);
8 laser_ranges[i*2]=data.ranges[index]*1e3;
9 }
```

Dynamic vectors are allocated while an application is executing and, hence, do not require compile time definitions and static allocations. Additionally, they encapsulate vector related meta information such as the vector size. The above code, with no compile time dependencies, shows this runtime flexibility. These properties make dynamic arrays ideal for implementing modular and extensible software, assuming the performance penalty is manageable.

III. DRDC's Generic Representations

A. The Miro Framework at DRDC

DRDC adapted and extended Miro (MIddleware for RObots) [4], [13] as the framework for its robotics software development. Miro is a distributed, object oriented framework for mobile robot control. It reduces software development times and costs by providing data structures, functions, communications protocols, and synchronization mechanisms specific to robots. The Miro framework is built upon the TAO/ACE middleware combinations where TAO (The ACE ORB) [14] is a CORBA (Common Object Request Broker Architecture) [15] implementation using ACE (Adaptive Communications Environment) [16]. The TAO/ACE combination provides component interfaces, network transparency, and platform independence.

Although Miro was originally developed for soccer robots, its flexible framework was easily adapted to the outdoor, unmanned vehicle environment.

B. Representing Generic Range Data

CORBA supports dynamic arrays through its sequence construct. Although the operational aspects of a CORBA sequence are similar to a C++ vector, a CORBA sequence benefits from network transparency where as a C++ vector does not [4]. Thus, both Miro and DRDC extensively use the CORBA sequence as constructs within CORBA objects.

1) Miro Range Representations: Miro defines dynamic range representations using both 1 and 2 dimensional CORBA sequences. These constructs are shown in the following code snippet.

```
//! A vector of sensor readings.
typedef sequence<long> RangeGroup;
//! A vector of sensor groups.
typedef sequence<RangeGroup> RangeScanIDL;
```

The *RangeGroupIDL* represents range data as a dynamic, 1-D array of long integers, and the *RangeScanIDL* implements a variable length 2-D representation. Using these two constructs, Miro represents range data in a flexible manner, but these construct do not lead to unfettered modularity and extensibility since *significance* is still attached to the entry order of the array [5].

2) DRDC 3-D Range Representations: Although the Miro range representations are runtime dynamic, they do not provide sufficient abstraction to implement transparently modular and extensible software. To rectify this deficiency, DRDC created a 3-D range construct that natively supports code modularity and extensibility. Under this approach, the device driver decomposes the range into its 3-D constituents; namely, (x,y,z) positions. This generic range representation is devoid of any underlying assumptions and is applicable to all ranging devices whether they be laser rangefinders, triangulation rangefinders, stereo vision, or sonars.

DRDC implemented two types of 3-D range representations. The first representation, shown in the following code snippet, used static, predefined arrays.

```
//! Optimized 3D Range Sensor.
const long POSE_COLS = 4;
const long POSE_ROWS = 4;
const long LASER_NUM =  361;
const long PLANES_NUM = POSE_COLS;
//! An array of  3d point groups.
typedef double
  Range3dLaserIDL[LASER_NUM][PLANES_NUM];
```

This static implementation uses an array of 2 dimensions to store the (x,y,z) position and the scaler[6]. It sacrifices modularity/extensibility for performance and serves as a basis for comparison with a second DRDC representation. The second DRDC implementation, using CORBA sequences, creates a runtime dynamic 3-D range representation. This implementation, shown below, uses the Miro *RangeScanIDL* 2 dimensional sequence to represent range data.

```
//! The full scan of all sensor groups.
RangeScanIDL range3d;
```

The positional information is again encoded as (x,y,z, scale).

IV. Experimental Results

From a theoretical perspective, generic data structures offer numerous advantages, as was detailed in Section II, but is it practical to implement such representations using currently available computing platforms? DRDC performed experiments that quantified the performance of various data representations and the results of these experiments are given in the following sections.

A. Setup

For these experiments, DRDC used its Raptor UGV as shown in Figure 2. The Raptor contains a complete suite of applications that allow the vehicle to autonomously operate in low complexity, outdoor environments. Figure 3 is a flow

[4]This means C++ vectors and similar STL constructs cannot be embedded in CORBA objects.

[5]A line scanner, like the SICK laser, encodes angular information using the entry order in the array.

[6]With the scaler, the position representation is homogeneous transform compliant.

diagram that shows the various services running on the Raptor and the data flows between these services.

Fig. 2. Raptor Unmanned Ground Vehicle. Each Raptor used one or more roof mounted SICK lasers and stereo cameras; Differential GPS and IMU; and wireless mesh networking routers.

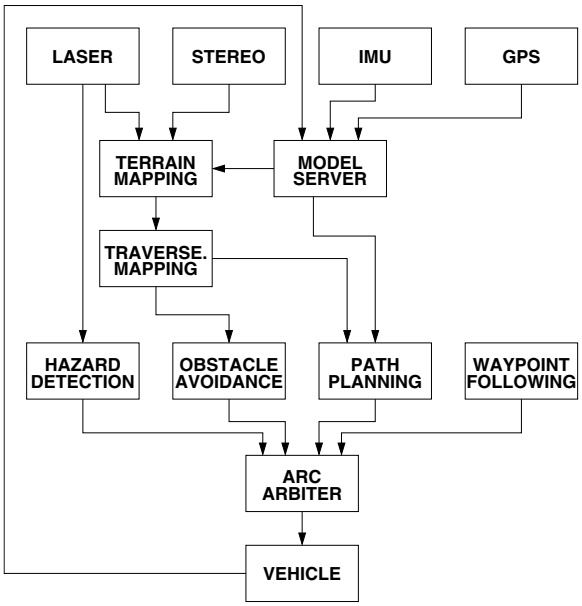

Fig. 3. Raptor Architecture Flow Diagram. Each box represents a service implementing a specific capability.

To facilitate the investigation of data representations, the Raptor was used to log the real world data. The logged data consisted of raw range data from a SICK laser rangefinder, GPS position, and IMU orientation. Using the Miro Logplayer, this data could easily be played back into the modules that comprise the Raptor system. Given the generic data representations focus of this research, the range data was logged in two formats:

- Range data stored as 3-D points in the static *Range3dLaserIDL* structure, and
- 3-D range data stored in the dynamic *RangeScanIDL* structure.

The experiments were then performed on a Dell Optiplex GX280 computer running at 3.2 GHz with 1 GB of RAM. The Miro LogPlayer was used to replay the logged data and the system's performance was investigated.

B. CORBA Event Publication Performance

DRDC's laser device driver converts the raw SICK laser data representation into a 3-D representation. The DRDC world representation module requires a 3-D representation, thus this conversion does not impose an extra burden, but this representation consumes three times more memory than the native laser structure and this size difference does impose a performance penalty. Under the DRDC architecture [8], the range data is published as an event under the CORBA notification service and an experiment was performed to quantify the extra time consumed by the larger generic 3-D range representation. The results, representing 18,000 CORBA publications, are shown in Table I.

Range Representation	Array Size	Time	Std.
Native Array	361×1 doubles	340 μs	37 μs
3-D Range Array	361×4 doubles	371 μs	27 μs

TABLE I
TIME REQUIRED TO PUBLISH RANGE EVENTS

As can be seen in Table I, the 3-D range event consumes a mere 31 more μs than the native array. Thus, it can be concluded that the range representation memory requirements have a minimal impact on the overall performance.

C. Generic vs. Specific Representations

Transparent modularity and extensibility requires generic data representations that abstract the underlying data by using runtime dynamic data structures. DRDC performed experiments that quantified a specific application's performance using both static and dynamic data structures. To conduct this comparison, DRDC investigated the performance of the two range data representations detailed in Section III-B.2.

The laser device driver first converts the raw range data to a 3-D representation. Then, before this range data can be used by other applications, it first must be transformed from its Laser Beam frame into the Map frame. Figure 4 shows the transformation sequence that must occur before the range data can be integrated into a world representation by DRDC's map application.

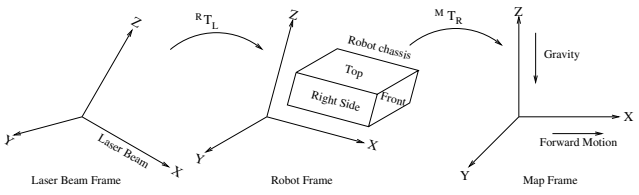

Fig. 4. Co-ordinate Systems and Transformations

This transformation between frames is computing intensive and must be performed each time range data is received from the SICK laser. The frame transformations require the multiplication of a 4×4 matrix by a 4×361 matrix at the laser update rate of 26.6 ms. For these experiments, the time required to complete these transformations was measured and statistics were gathered for a sample of 18,000 range data representations. Additionally, these experiments investigated the transformation times using two different matrix library implementations.

1) Performance using ATLAS Libraries: The 3-D range data transformation, from the Laser Beam frame to the Map frame, is accomplished by multiplying the raw 3-D data by the ${}^{R}T_{L} \times {}^{M}T_{R}$ homogeneous transform. Matrices represent all the entities used in this process. This experiment measured the time required to perform this matrix algebra using the ATLAS [17] *cblas_dgemm* function. Three separate ATLAS based experiments were conducted:

- Static arrays using standard C/C++ array indexing,
- Dynamic sequences using standard CORBA sequence indexing, and
- Dynamic sequences using CORBA pointers and auto-incrementing.

CORBA sequences, using the standard [][] index operators, are known to be less efficient than using pointers to access the sequence data[7]; thus, both CORBA approaches were tested.

Representation	Description	Access	Time	Std.
Range3dLaserIDL	Array 361×4	Indices	64 μs	11 us
RangeScanIDL	CORBA Sequences	Indices	98 μs	18 μs
RangeScanIDL	CORBA Sequences	Pointers	68 μs	12 μs

TABLE II
TIME TO PERFORM RANGE TRANSFORMATION USING ATLAS

Table II shows that all data representations produce fast transformations, which require little execution time. In comparison, an indexed CORBA sequence consumes $\simeq 53\%$ more time than the static *Range3dLaserIDL* array. A CORBA sequence, accessed using CORBA pointers, is a more efficient implementation and required only $\simeq 6\%$ more time than the static array. Using CORBA pointers, along with standard C auto-incrementing, was significantly more efficient than using standard CORBA indices on sequences. The switch from standard CORBA sequence indexing to CORBA sequence pointers resulted in a $\simeq 31\%$ decrease in execution time. Given the ATLAS *cblas_dgemm* function requires a fixed execution time, the extra overhead accrued by the *RangeScanIDL* CORBA sequence implementation(s) can be attributed to copying the range data out of the CORBA sequence into the array structure that is suitable for the ATLAS *cblas_dgemm* function.

2) Performance using BOOST Libraries: The same experiments were conducted using BOOST matrix math [19]. Although ATLAS is a very capable implementation, its roots in FORTAN and C result in complex function calls. The code snippet, given below, shows that the ATLAS *cblas_dgemm* requires 14 parameters, all of which must be correctly specified.

```
cblas_dgemm( CblasRowMajor,
             CblasNoTrans,
             CblasNoTrans,
             Miro::POSE_COLS,
             Miro::LASER_NUM,
             Miro::POSE_COLS,
             alpha,
             (const double *)pose,
             Miro::POSE_ROWS,
             (const double *)range3d,
             Miro::POSE_COLS,
             beta,
             (double *)&trans_range,
             Miro::LASER_NUM);
```

In comparison to ATLAS matrix math, the BOOST implementation is very simple. The equivalent BOOST implementation reads as one would write the mathematical equation and is shown below.

```
HTproduct = prod( pose, transform )
```

Table III shows the time required to complete the range transformations using three separate BOOST implementations.

Representation	Description	Access	Time	Std.
Range3dLaserIDL	Array 361×4	Indices	368 μs	51 us
RangeScanIDL	CORBA Sequences	Indices	527 μs	62 us
RangeScanIDL	CORBA Sequences	Pointers	509 μs	60 us

TABLE III
TIME TO PERFORM RANGE TRANSFORMATION USING BOOST

Table III shows that using BOOST libraries imposes a significant performance penalty when compared to the execution times yielded using the ATLAS libraries. The most efficient ATLAS implementation is approximately 6 times faster than the most efficient BOOST implementation. Even the slowest ATLAS implementation was significantly faster than the best BOOST performer. As was observed under the ATLAS experiments, a static array yielded the best BOOST performance, followed by CORBA pointers on sequences, with sequences using standard CORBA indexing being the least efficient. The performance difference between the best and worst implementation using BOOST libraries was $\simeq 43\%$, which can be attributed to BOOST's optimized static array multiplication method.

3) Discussion: Although the percentage performance difference between a static array implementation and a sequence based implementation is significant under ATLAS, the absolute time difference is relatively insignificant. Specifically, the transformation of DRDC's generic range 3-D representation, using a static array, required 64 μs whereas the implementation using a CORBA sequence required between 68 μs and 98 μs to complete the transformation. The percentage difference between the two techniques is, at worst, approximately 50%

[7]The CORBA ::get_buffer function [18] implements this functionality and is especially useful for large blocks of data.

but the overall time is only 34 μs. Given the overall time budget for this application is 26.6 ms, this 34 μs penalty represents only $\simeq 0.1\%$ of the total available time.

V. Conclusions

Portable, modular and extensible software is a necessary prerequisite for the wide scale dissemination of robotics algorithms. Using component based software engineering, robotics researchers have made strides toward this goal, but CBSE addresses only the software mechanisms aspects of this problem. A component's algorithmic implementation, providing specific functionality, must also be portable, modular, and extensible. This paper introduces the notion of generic data representations as a means of decoupling an algorithm from its input/output data structures and, hence, more modular and extensible algorithmic implementations.

Runtime dynamic data structures, such a C++ STL vectors or CORBA sequences, are key tools in decoupling an algorithm from its input/output data representations. Historically, the real-time requirements of robotic systems and the lack of available, portable computing power has precluded the use of such dynamic structures. Experiments were performed that investigated the performance issues associated with runtime dynamic data structures. These experiments determined that, although dynamic data structures incurred significant performance penalty when compared to its static brethren, the overall magnitude of the performance cost was manageable.

Additionally, this paper investigated using BOOST mathematics libraries to simplify and clarify algorithmic implementations. Although BOOST's performance was significantly slower than the ATLAS libraries, its performance did not preclude its usage.

DRDC's experience using components highlighted the need for generic data representations and experiments have shown that currently available computer systems have the processing power to support such abstract data representations. Given the impressive performance results achieved using dynamic data structures for range data, this approach was subsequently migrated to DRDC's terrain map implementation. Whereas the range data sets were relatively small, the terrain map data sets are large, representing approximately 150,000 C++ double values. The original map implementation was based upon a static array and the publication of a map event required $\simeq 4.9 \pm 0.57$ ms. The publication of the new sequence based map event was only marginally slower, requiring $\simeq 5.9 \pm 0.41$ ms. This corresponds to a 20% time penalty and does not adversely effect the overall performance of the system.

The use of generic data representations has allowed DRDC to implement flexible components that are configured at runtime. This allows researchers to easily modify the Raptor UGV's configuration, thus simplifying the vehicle's overall operation. These data representations have also improved the software's portability and extensibility by reducing its reliance on "hard coded" variables that limit an application's performance. Modularity has been improved since individual components both produce and expect to receive runtime dynamic data structures.

References

[1] C. Cote, D. Letourneau, F. M. J.-M. Valin, Y. Brosseau, C. Raievsky, M. Lemay, and V. Tran, "Code reusability tools for programming mobile robots," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[2] *CLARAty: An Architecture for Reusable Robotic Software*. Orlando, Florida: SPIE Aerosense Conference, April 2003.

[3] A. Brooks, T. Kaupp, A. Makarenko, A. Oreback, and S. Williams, "Towards component-based robotics," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, August 2005.

[4] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar, "Miro - middleware for mobile robot applications," *IEEE Transactions on Robotics and Automation*, June 2002.

[5] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," *Proceedings of the 11th International Conference on Advanced Robotics*, pp. 317–323, 2003.

[6] M. M. N. Roy and S. Thrun, "Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit," in *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems*, 2003. [Online]. Available: http://robots.stanford.edu/papers/Montemerlo03b.html

[7] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Reading, MA: Addison-Wesley, 1998.

[8] G. Broten, S. Monckton, J. Giesbrecht, and J. Collier, "Software sysetms for robotics, an applied research perspective," *International Journal of Advanced Robotic Systems*, vol. Volune 3, 1, no. 2005-204, pp. 11–17, March 2006.

[9] C. Schelegel, "Communications patterns as key towards component-based robotics," *International Journal of Advanced Robotic Systems*, vol. Vol. 3, No. 1, pp. 49–54, 2006.

[10] I. Ulrich and J. Borenstein, "Vfh*: Local obstacle avoidance with look-ahead verification," in *IEEE International Conference on Robotics and Automation*, San Fransico, CA, 2000, pp. 2505–2511.

[11] ——, "Vfh+: Reliable obstacle avoidance for fast mobile robots," in *IEEE International Conference on Robotics and Automation*, Leuven, Belgium, 1998, pp. 1572–1577.

[12] R. T. Vaughan, B. Gerkey, and A. Howard, "On device abstractions for portable, reusable robot code," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2121 – 2427, October 2003.

[13] S. Enderle, H. Utz, S. Sablatnog, S. Simon, G. Kraetzschmar, and G. Palm, "Miro - middleware for autonomous mobile robots," *International Federation of Automatic Control*, 2001.

[14] *TAO Developer's Guide*, Oci tao version 1.3a ed. 12140 Woodcrest Executive Drive, Suite 250, St. Louis, MO, 63141: Object Computing Inc., 2003, vol. 1 and 2.

[15] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.

[16] S. Huston, J. Johnson, and U. Syyid, *The ACE Programmer's Guide*. Addison-Wesley, 2004.

[17] "Automatically tuned linear algebra software (atlas)," http://math-atlas.sourceforge.net/, August 2006.

[18] F. Bolton, *Pure CORBA: A code intensive premium reference*. SAMS, 2002.

[19] "Boost c++ libraries," http://boost.sourceforge.net/, August 2006.