Defence Research and Development Canada | Recherche et développement pour la défense Canada

DEFENCE DÉFENSE

Architectural support for robotics

Software prototyping and testing

D.J. Mackay, G.S. Broten and S. Monckton
DRDC Suffield

Defence R&D Canada

Technical Memorandum

DRDC Suffield TM 2010-271

December 2010

Canada

Architectural support for robotics

Software prototyping and testing

D. J. Mackay, G. S. Broten, and S. Monckton
DRDC Suffield

Defence R&D Canada – Suffield

Technical Memorandum

DRDC Suffield TM 2010-271

December 2010

Principal Author

Original signed by D.J. Mackay

D. J. Mackay

Approved by

Original signed by D.M. Hanna

D. M. Hanna
Head/AISS

Approved for release by

Original signed by R. Clewley

R. Clewley
Head/Document Review Panel

Abstract

The development of robotics software is difficult. The sheer size and complexity of these software systems, the interaction of robots with their environment including the potential need for real-time operation, and the lack of accepted standards all contribute to this situation. Due to the diverse nature of robot systems, contributions from multiple researchers with disparate fields of expertise are required and that in itself further complicates software development.

At Defence R&D Canada Suffield, we have developed domain-specific middleware based upon the Miro framework, that uses the ACE/TAO implementation of CORBA. This middleware provides infrastructure services enabling seamless communication between software components. Using this framework enables researchers to focus their attention at a level of abstraction close to the problem domain, for the most part ignoring the underlying details that enable a system to work.

Over the past six years, researchers in Autonomous Intelligent Systems Section have developed numerous software components for an heterogeneous set of robotic platforms. A key conclusion arising from this endeavour was the recognition that components and middleware, although crucial enablers, do not of themselves fully address the difficulties associated with software reuse. Software reuse can only be realized if both the component interfaces and the underlying data structures are stable. And, although software reuse simplifies the process of adapting existing software to other robotic platforms, it doesn't obviate the need for tools and techniques that assist in prototyping and debugging. Our middleware provides for prototyping and debugging by allowing a system to seamlessly switch between real world data, logged data and data provided by a simulation environment. This paper describes our implementation, provides examples that illustrate its usage, and discusses the benefits accrued.

Résumé

Développer des logiciels en robotique est une entreprise ardue. Cela découle entre autres de l'immense taille et l'extrême complexité des systèmes, des interactions entre les robots et leur environnement (notamment des besoins potentiels d'opération en temps réel) et de l'absence de normes reconnues. La nature multiforme des systèmes robotiques, qui exigent l'intervention de nombreux chercheurs de divers domaines d'expertise, rend le développement d'autant plus complexe.

RDDC Suffield a élaboré à partir du cadre Miro un intergiciel propre à ce domaine fondé sur la mise en œuvre ACE/TAO de CORBA. Cet intergiciel prend en charge des services d'infrastructure qui permettent à divers éléments logiciels de communiquer entre eux. À l'aide de ce cadre, les chercheurs peuvent centrer leur attention à un niveau d'abstraction se rapprochant du domaine d’exploitation des robots, en se souciant peu des détails de mise en œuvre sous-jacente qui rendent possible le fonctionnement des systèmes.

Les chercheurs de la Section des systèmes intelligents autonomes ont depuis six ans élaboré bon nombre d'éléments logiciels destinés à des plates-formes robotiques hétérogènes. Une conclusion importante se dégage de ces efforts : les habilitants essentiels que sont intergiciels et éléments logiciels n’aplanissent pourtant pas en soi les difficultés entourant la réutilisation des logiciels, car cette réutilisation n’est possible que si les interfaces matérielles et les structures de données sous-jacentes restent stables. La réutilisation des logiciels simplifie certes l’adaptation des logiciels à une autre plate-forme robotique, mais elle n’élimine pas pour autant le besoin de recourir aux outils et techniques de prototypage et de débogage. Notre intergiciel simplifie le prototypage et le débogage, car il permet de passer facilement d’une source de données à une autre, qu’il s’agisse de données d’entrée réelles, préenregistrées ou provenant d’une simulation. Le présent exposé décrit l’intergiciel mis en œuvre, illustre comment l’utiliser à l’aide d’exemples, et en décrit les avantages.

Executive summary

Architectural support for robotics: software prototyping and testing

D. J. Mackay, G. S. Broten, S. Monckton; DRDC Suffield TM 2010-271; Defence R&D Canada – Suffield; December 2010.

Background: The development of robotics software is difficult. The sheer size and complexity of these software systems, the interaction of robots with their environment, including the potential need f or real-time operation, and the lack of accepted standards all contribute to this situation. The diverse nature of robot systems necessitates contributions from multiple researchers with disparate fields of expertise and that in itself f urther complicates software development.

Over the past six years, researchers DRDC Suffield have developed numerous software components for diverse robotic platforms. In the parlance of Domain Engineering, a couple of enduring business themes were identified over the course of this work: first, the need for infrastructure services enabling communication between software modules, and second, the need for a seamless integration of software components into a simulation environment for prototyping and debugging. This paper describes the component-based software architecture that has evolved and elucidates the benefits arising from its use.

Results: To address the need for infrastructure services, a component-based framework based upon the Miro framework, using the ACE/TAO implementation of CORBA was developed. Adoption of this communications middleware allows development under the component paradigm to proceed close to the abstractions of the problem domain. The developer need only be concerned with content of intermodule communication and not the underlying mechanisms provided that allow this communication to take place. With the addition of an interface to Gazebo, DRDC's middleware allows components to seamlessly switch between real world data, logged data, and data provided by the simulation environment

Significance: The adoption of middleware to encapsulate the enduring business themes identified during the development of DRDC's Architecture for Autonomy has resulted in systems that are arguably easier to prototype, debug, and maintain than would otherwise have been possible.

Sommaire

Architectural support for robotics: software prototyping and testing

D. J. Mackay, G. S. Broten, S. Monckton ; DRDC Suffield TM 2010-271 ; R & D pour la défense Canada – Suffield ; décembre 2010.

Contexte: Développer des logiciels en robotique est une entreprise ardue. Cela découle entre autres de l'immense taille et l'extrême complexité des systèmes, des interactions entre les robots et leur environnement (notamment des besoins potentiels d'opération en temps réel) et de l'absence de normes reconnues. La nature multiforme des systèmes robotiques exige l'intervention de nombreux chercheurs de divers domaines d'expertise, ce qui rend le développement d'autant plus complexe.

Les chercheurs de la SSIA ont depuis six ans élaboré bon nombre d'éléments logiciels destinés à des plates-formes robotiques hétérogènes. Dans le jargon du « génie des domaines » (Domain Engineering), deux thèmes opérationnels (business themes) se dégagent de ces travaux : d'une part, le besoin de services d'infrastructure qui permettent la communication entre les modules, et d'autre part, la nécessité d'une intégration souple à un environnement simulé aux fins de prototypage et de débogage. Le présent exposé décrit l'architecture logicielle fondée sur les composantes que nous avons développée, et décrit les avantages de l'utiliser.

Résultats: Afin de fournir des services d'infrastructure, nous avons élaboré à partir du cadriciel Miro [9] un cadriciel fondé sur des composantes propre au domaine robotique et fondé sur la mise en œuvre ACE/TAO de CORBA. Utiliser cet intergiciel de communication permet, dans le développement fondé sur les composants, de se rapprocher sensiblement du niveau d'abstraction correspondant au domaine d'exploitation des robots. Le développeur n'a qu'à se soucier du contenu des communications entre les modules; les mécanismes de communication eux-mêmes sont gérés automatiquement. Avec l'ajout d'une interface avec Gazebo, l'intergiciel de RDDC permet aussi de passer facilement d'une source de données à une autre, qu'il s'agisse de données d'entrée réelles, préenregistrées ou provenant d'une simulation.

Portée: Adopter un intergiciel afin de mettre en place les thèmes opérationnels se dégageant du développement de l'Architecture pour l'autonomie de RDDC a incontestablement permis de simplifier le prototypage, le débogage et la maintenance des systèmes robotiques.

Table of contents

List of figures

List of tables

This page intentionally left blank.

1 Introduction

The development of robotics software is difficult. Numerous factors contribute to this situation. The sheer size and complexity of these software systems makes their design and implementation difficult. The interaction of robots with their environment and the potential need for real-time operation also complicates robotics software development. The situation is further exacerbated by a lack of accepted standards. Due to the diverse nature of robot systems, contributions from multiple researchers with disparate fields of expertise are required and that in itself further complicates software development. Domain Engineering (DE), with its focus on creating reusable software, offers an engineering approach to deal with these issues [1, 2]. Using stability analysis, the *enduring business themes* (EBT), corresponding to the change invariant aspects of the environment in which the software will operate, can be identified [3, 4]. This approach, when applied to robotic systems, allows for the reuse and adaptation of specialized knowledge in the areas of architectures and software components [5]. Thus, DE promises an evolution in software production from a one-off craftsman approach to a mass-production approach based upon prebuilt standard parts and assemblies [6].

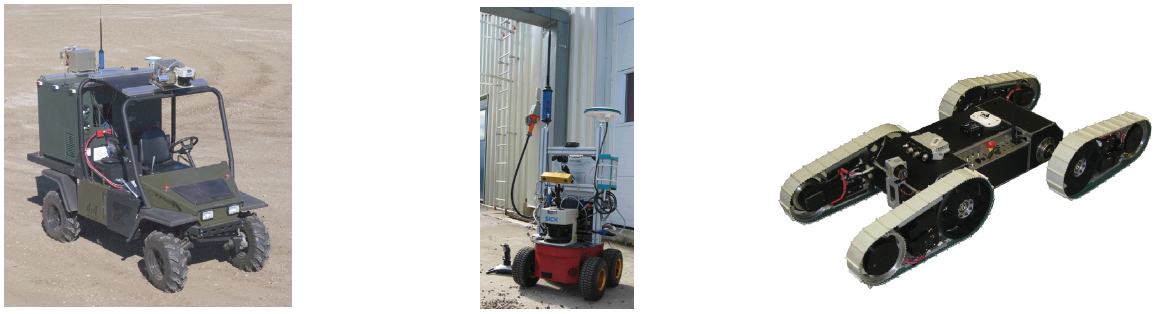
Defence R&D Canada's (DRDC) mandate covers both research and development; thus it has interests that span the gamut from basic research in robotics to the deployment of operational systems. This broad view is a key factor driving our exploration of DE as a means to develop reusable software components. Before commencing the development of robotic capabilities, DRDC undertook an investigation that identified a number of aspects of robotic systems (or EBTs in the parlance of DE) that were deemed crucial to a successful program [7]. These aspects of robotics are invariant to systems changes, such as the targeted platform, environment, or application. The first EBT addressed was the requirement for infrastructure services that allow components to seamlessly exchange information via com-munications middleware. The approach taken to address this requirement is described in [8] and is based upon the Miro framework [9], using the ACE/TAO [10] implementation of CORBA [11]. Adoption of this communications middleware allows development under the component paradigm to proceed close to the abstractions of the problem domain. The developer need only be concerned with content of inter-module communication and not the underlying mechanisms that allow the communication to occur.

Over the past six years, researchers in AISS have developed numerous software components for a heterogeneous set of robotic platforms. This extensive usage has given us an in-depth appreciation of the strengths and weaknesses associated with the component paradigm. A key conclusion arising from this endeavour was the recognition that components and middleware, although crucial enablers, aren't a panacea for the difficulties inherent in software reuse [12]. Using components and middleware provides an easy to use and robust mecha-nism for intermodule data exchange, however software reuse in general is not possible unless both the component interfaces and the underlying data structures are stable.

The second EBT DRDC identified relates to debugging and simulation services. Although software reuse simplifies the process of adapting existing software to other robotic platforms, it does not obviate the need for tools and techniques that assist in prototyping and debugging. Given the complexity associated with robotics software, a seamless integration into a simulation environment is essential. Using DE terminology, Brugali and coworkers [13] identify these issues as: “the need of abstract models to cope with hardware and software heterogeneity” and “the need of development techniques to enable a seamless transition from prototype testing and debugging to real system implementation and exploitation”. DRDC has developed an approach that allows the robotics software to seamlessly switch between real world data, logged data, and data provided by a simulation environment. This paper describes our implementation, provides examples that illustrate its usage, and discusses the benefits accrued. Section 2 provides some background for our heterogeneous fleet of robotic platforms. Section 3 details the implementation and highlights the importance of stable interfaces and data structures. The crucial role that data logging plays in debugging is discussed in Section 4. Section 5 examines the importance of simulation in testing of prototype software capabilities. Finally, a summation is presented in Section 6.

2 Background

Over the past 25 years DRDC has developed numerous robotic vehicles, including teleoperated and autonomous versions. Descendants of these early implementations are currently in service and development is still occurring. Unfortunately these legacy systems, effectively being one-offs, were not designed with software reuse in mind; they have little capability to systematically log data, and don't integrate with any type of simulator. Given their implementation, it would be difficult to incorporate any of these features and thus it is unlikely they will ever be supported.



(a) Raptor UGV (b) Pioneer 3AT (c) STRV

Figure 1: DRDC Robot platforms.

With the resurgence of an autonomy program in AISS, the issues mentioned above were recognized as major impediments to an evolutionary development, testing, and deployment of autonomous robots. Software reuse was recognized as an issue facing all developers of robotics software. DRDC researchers believed that a framework that encouraged component development, built upon capable middleware supporting network transparency and distributed computing, could provide a means of address most of these issues. This approach would lead to flexible, extensible, scalable, and portable software with well defined interfaces which would naturally support logging and could easily interface with robotics simulators.

After reviewing possible options [14], the open source Miro framework [15], built upon ACE/TAO middleware and tailored for robotic implementations, was selected as a robotics framework. Numerous robotic capabilities have been developed, subsequently, under this framework, targeting a variety of platforms. Figure 1 illustrates the heterogeneous nature of DRDC's current fleet of robotic platforms including the Raptor Unmanned Ground Vehicle (UGV), an indoor Pioneer 3-AT, and the Shape Shifting Tracked Vehicle (STRV).

3 Implementation

3.1 Software framework

The Miro framework, building upon CORBA capabilities, allows communication under either a publish/subscribe or a query/response paradigm. The Naming Service facilitates data exchanges by providing phone directory-like functionality.

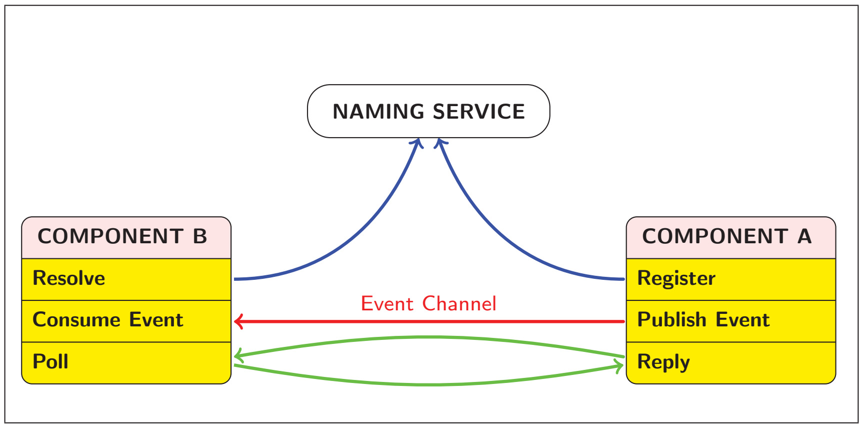


Figure 2: Miro derived components.

Components created under the Miro framework inherit these capabilities, as shown in Figure 2. All DRDC components register their interfaces with the Naming Service, thus en-abling query/response data exchanges. Additionally, each component resolves the event channel reference and registers/resolves names. With the exception of a few graphical in-terfaces, all DRDC components rely exclusively on the publish/subscribe paradigm via an event channel.

3.2 Modelserver

To support this software framework, DRDC developed a geometry database server, Modelserver, to provide device positions in vehicle coordinates. Modelserver derives these positions from Body and Assembly XML files that describe component bodies and binding constraints, respectively. Every device is hand measured for significant frames (e.g., for a camera: mounting points and image planes) in a local device coordinate system and stored in a Body XML file. With all vehicles and sensor devices surveyed in this manner, new assemblies can be quickly created or modified within a single, relatively simple Assembly XML file. Further, any changes to a single sensor geometry are automatically propagated through all assemblies.

Modelserver converts these XML files into three data types:

- a body, $\mathcal{B}$,
- a body frame, $\mathcal{F}$, and
- a constraint, $\mathcal{C}$,

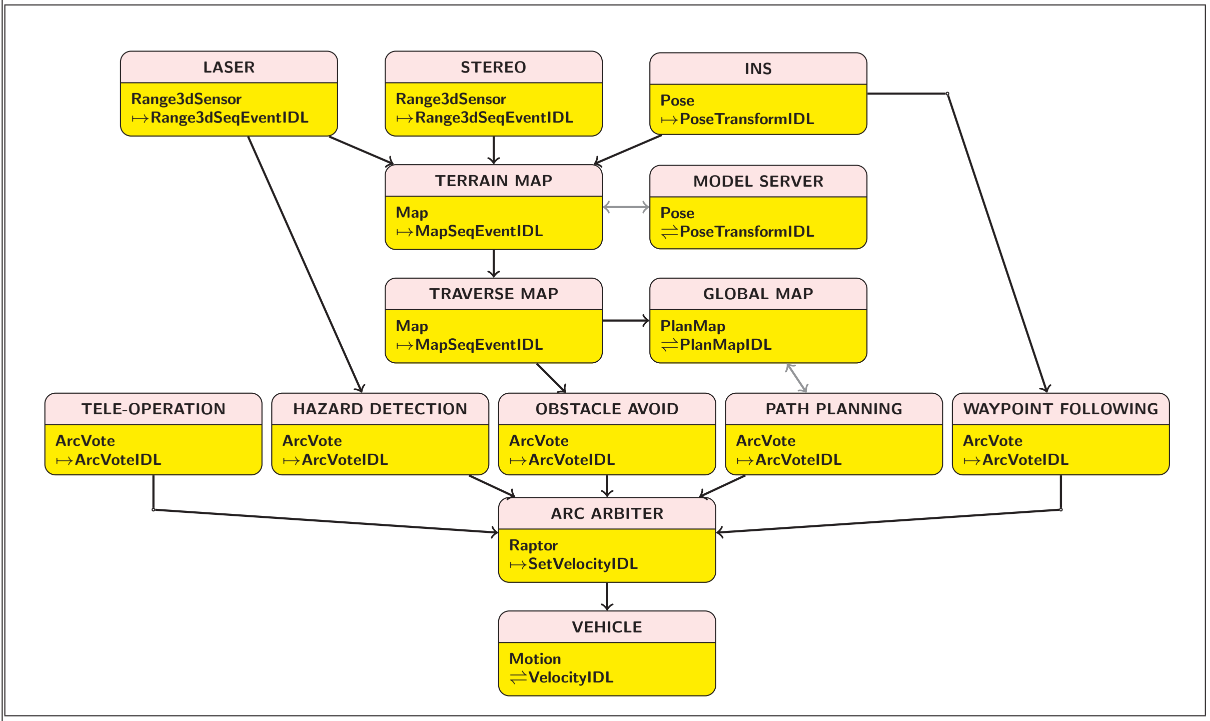


Figure 3: Current components, with data structures and information flows.

accessible through a body list, $\mathbf{L}_{\mathcal{B}} = [\mathcal{B}_i \mid i = 1 \mathinner{..} m]$, a constraint list, $\mathbf{L}_{\mathcal{C}} = [\mathcal{C}_i \mid i = 1 \mathinner{..} q]$, and a directed graph model, $\mathcal{M}$. The relationship between these types appears in Figure 4. A body consists of a tuple containing a unique string name identifier, s_b, and a list of body frames, $\mathbf{L}_{\mathcal{F}} = [\mathcal{F}_j \mid j = 1 \mathinner{..} n]$, e.g., the ith body is $\mathcal{B}_i = \langle s_b, \mathbf{L}_{\mathcal{F},i} \rangle$. “Raptor” [1] or “Flea” [2] are examples body strings.

A body frame consists of a tuple containing a unique string identifier, s_f, a homogeneous transform from the origin of the ith body to the jth frame, ${}^{i}A_j$, and pointers to the parent body, $\mathcal{B}_i$, and a constraint, $\mathcal{C}$. The jth frame of ith body is $\mathcal{B}_i : \mathcal{F}_j = \langle s_f, \mathcal{B}_i, \mathcal{C}, {}^{i}A_j \rangle$. A significant frame on the `Raptor` body is the `FrontBumperCenter` (or `Raptor:FrontBumperCenter`) and on the `Flea` body, `ImagePlane` (or `Flea:ImagePlane`).

Constraints bind distinct body:frame pairs through a simple time invariant homogeneous transform, T_k. A constraint consists of a tuple containing a unique string identifier, s_c, pointers to the body frames, $\mathcal{F}_{\mathbf{from}}$ and $\mathcal{F}_{\mathbf{to}}$, joined by the constraint, and the homogeneous transform encapsulating the constraint. The constraint captures the *direction* of its transformation through the pointers to the *from* and *to* bodyframes, $\mathcal{F}_{\mathbf{from}}$ and $\mathcal{F}_{\mathbf{to}}$, respectively. The kth constraint is $\mathcal{C}_k = \langle s_c, \mathcal{F}_{\mathbf{from}}, \mathcal{F}_{\mathbf{to}}, T_k \rangle : \mathcal{F}_{\mathbf{from}} \neq \mathcal{F}_{\mathbf{to}}$. Binding `Raptor:TopCenterRail` to `Flea:BottomCenter` would thus represent a mounting constraint of the Flea Body to the Raptor.

1. The Raptor is an ATV from Koyker Manufacturing Company modified by DRDC for autonomous operation.
2. The Flea$^{\text{TM}}$ is an IEEE-1394 FireWire camera from Point Grey Research.

$$\begin{tikzcd}
 & \mathcal{B}_1 : \mathcal{F}_3 & & \mathcal{B}_2 : \mathcal{F}_3 & \\
 & \mathcal{B}_1 : \mathcal{F}_2 & & \mathcal{B}_2 : \mathcal{F}_2 & \\
\mathcal{B}_1 \arrow[uur, bend left] \arrow[ur, bend left] \arrow[dr, bend right] & & & & \mathcal{B}_2 \arrow[uul, bend right] \arrow[ul, bend right] \arrow[dl, bend left] \\
 & \mathcal{B}_1 : \mathcal{F}_1 \arrow[uurr, dashed, bend left, "\mathcal{C}_{12}"] & & \mathcal{B}_2 : \mathcal{F}_1 &
\end{tikzcd}$$

Figure 4: The symbolic relationship between bodies $\mathcal{B}_1$ and $\mathcal{B}_2$, their frames, $\mathcal{F}_j$, and a constraint, $\mathcal{C}_{12}$.

The model $\mathcal{M}$, consisting of $\mathbf{L}_{\mathcal{B}}$ constrained by $\mathbf{L}_{\mathcal{C}}$, resembles a cyclic directed graph of $\mathcal{B}_i : \mathcal{F}_j$ vertices with ${}^{i}A_j$ and T_k transformation connectors. The following simple rules govern the construction of $\mathcal{M}$:

1. There must be $m \geq 2$ bodies. Clearly, the minimum assembly is composed of two bodies.
2. A body must have $n \geq 1$ bodyframes.
3. $\forall \mathcal{C}_k \in \mathbf{L}_{\mathcal{C}}$, $\mathcal{F}_{\mathbf{from}}$ and $\mathcal{F}_{\mathbf{to}}$ must exist. Constraints can exist only between real body frames.
4. $\forall \mathcal{B}_i : \mathcal{F}_j$, if $\mathcal{F}_j$ is constrained, only one constraint may exist. In this prototype, a bodyframe cannot be constrained to more than one other bodyframe.

Modelserver is designed to provide geometry services to client processes. In general, consumers of geometry need to know the location of one frame with respect to another. In Modelserver's terms, the server must provide the pose for a *frame of interest* (FOI) in the coordinates of some *frame of reference* (FOR).

The server performs a simple depth-first search, recursively constructing a transformation between FoR and FoI vertices. Starting at the $\mathcal{B}_{FoR} : \mathcal{F}_{FoR}$ vertex, the search compares $\mathcal{B}_i : \mathcal{F}_j$ in $\mathbf{L}_{\mathcal{F},i}$ against the $\mathcal{B}_{FoI} : \mathcal{F}_{FoI}$. Finding no match, the search will examine the next frame unless the frame is constrained. In this case, it pushes $\mathcal{B}_i : \mathcal{F}_j$ onto a *path* stack, "crosses" the constraint, and examines the attached frame. If a $\mathcal{B}_i : \mathcal{F}_j$ matches any in *path*, a cycle exists and the search examines the next branch, popping *path* as necessary. The search continues recursively until $\mathcal{B}_{FoI} : \mathcal{F}_{FoI}$ is found. The search then unwinds the recursion, building the transform product, ${}^{\mathcal{F}_{FoR}}T_{\mathcal{F}_{FoI}}$, according to the directed graph.

The CORBA poll interface to Modelserver in Figure 5 allows clients to interrogate the model for all available bodies through `getBodyList()`; framelists for any body through `getBodyFrameList()`; and transformations between any two body/bodyframe nodes through `getTransformation()`. For example, the transform between the Raptor `FrontBumperCenter` and the Fleas `ImagePlane` becomes a simple call:

```
typedef sequence<string> StringSequenceIDL;
interface ModelServer
{
  PoseTransformIDL getTransformation(in string FOI, in string FOR);
  StringSequenceIDL getBodyList ();
  StringSequenceIDL getBodyFrameList (in string BodyName);
};
```

Figure 5: A summarized view of the The Platform CORBA

Object. `PoseTransformIDL T = theModel->getTransformation( "Flea:ImagePlane", "Raptor:FrontBumperCenter" );`

By using relatively simple XML files for individual bodies and a single constraint assembly file, Modelserver encapsulates the complexity of geometric model generation, maintenance, search, and transform algebra. In so doing, the server encourages symbolic frames and, in turn, platform-agnostic algorithms that can be safely applied to a variety of vehicles with little difficulty.

3.3 DRDC components and stable interfaces

Under Miro, DRDC researchers created a set of components that collaboratively imple-ment autonomous capabilities. Figure 3 illustrates the current component configuration: the interfaces, their data structures and information flows that have evolved from DRDC's experiences [12]. As can be seen, only a small number of interfaces are defined and required. Table 1 lists the most used and stable interfaces, each of which are described in the sections that follow.

Table 1: Stable data representations.

Representation	Data Storage	Payload
Range3dSeqIDL	Sequence	4-tuple: (X, Y, Z, R)
PoseTransformIDL	Array	Homogeneous transformation and covariance matrix
MapSeqEventIDL	Sequence	A $2\,\mathrm{D}$ grid with a variable number of data planes
VoteIDL	Array	5-tuple: veto, curvature, max. speed, vote, and confidence for each arc
VelocityIDL	Array	Desired steering angle and speed

3.3.1 Range interface

Ranging devices are heterogeneous in nature, relying on a variety of sensing modalities. These include laser ranging, laser triangulation, stereo vision, sonar and radar. Even within a given class, such as laser rangefinders, data densities and formats vary significantly. DRDC has defined a single format that is applicable to all such ranging devices. All range data is encoded within a variable length container, a CORBA sequence, as a 4-tuple representing the position, (X, Y, Z), and the range. Each device's driver converts its raw sensor data to this format before publishing the data as an event. In this manner, all range consumers are able to receive and process range data arising from any ranging device and a stable interface is maintained. The following ranging devices support this interface:

- SICK LMS200 laser rangefinder,
- Velodyne HD laser rangefinder,
- Hokuyo URG laser rangefinder,
- Point Grey Digiclops trinocular stereo vision,
- Point Grey BumbleBee 2 stereo vision,
- Point Grey BumbleBee XB3 stereo vision, and,
- Simulated laser rangefinder under Gazebo.

3.3.2 Pose interface

The pose is represented as a $[4 \times 4]$ homogeneous transform and its corresponding $[3 \times 3]$ attitude covariance matrix. This data representation, though perhaps not a standard, is based upon data that is available from all GPS/INS devices. Numerous GPS/INS and IMU devices support the pose interface:

- Consumer Garmin GPS,
- Novatel Sokkia GPS,
- Novatel GNSS/INS SPAN[3] with an integrated Honeywell IMU,
- Microstrain 3DMG IMU,
- Crossbow RGA300 IMU, and,
- Simulated pose under Gazebo.

3.3.3 Map interface

Maps are a means of representing the world in a format that is amenable to robot systems. The MapSeqEventIDL, based upon a CORBA sequence, allows for grid based representation of dimensions, (j, k), with i separate planes of data. This flexible interface can encode a majority of DRDC defined maps including $2^1/_2$ D terrain maps, $2^1/_2$ D inferred geometry maps, and occupancy grid-like traversability maps.

3.3.4 Arc vote interface

All modules that contribute to driving the vehicle, including

- Tele-operation,
- Hazard Detection,
- Obstacle Avoidance,
- Path Planning, and
- Waypoint Following,

emit Vote events. These events consist of an array of candidate arcs for the vehicle to follow; each arc contains its curvature, a maximum forward speed, a vote, a confidence in that vote, and a Boolean veto. The ArcArbiter subscribes to all Vote events and combines them to produce a commanded steering angle and desired speed.

3. Synchronous Position, Attitude, and Navigation

3.3.5 Vehicle command interface

The ArcArbiter is the only module that sends commands directly to the vehicle. These commands consist of the desired steering angle and forward speed. Currently, the following vehicles are compatible with this interface:

- the Pioneer 3-AT,
- the Raptor UGV,
- their simulated analogues in GazeboClient, and
- the Multi-Agent Tactical Sentry (MATS) vehicle.

3.4 Performance vs. re-usability

CORBA sequences are an elegant means to enhance re-usability. They are flexible, network transparent, and run-time defined; hence, they are well suited for scalable applications. They do, however, consume more resources than their static counterparts. This includes a larger memory footprint and slower data access times. DRDC has extensively investigated the trade-offs associated with sequence based structures and found that for small to medium sized structures, up to approximately 500 Kbytes, the performance penalty is small so the net benefits are significant [16]. For larger structures, such as imagery consisting of several megabytes of data per frame, the use of sequences incurs a noticeable performance penalty. Simply publishing such a large sequence can require 100 to 300 ms, depending on the size of the image. In instances like this, reverting to a static data structure reduces publication times by an order of magnitude or more. The drawback of this approach is reduced flexibility. A static definition is not run-time defined; hence, it will necessitate a code recompilation. Additionally, depending on the approach taken, it is possible to lose network transparency. This can occur when the static implementation is simply a container for bytes. In this case, if the endianness of the sending and receiving computers differs, then the data upon reception will not be valid.

Currently the DRDC implementation runs on a dual, quad-core computer (2.2 GHz), and all communications are local. Extensive investigation into time delays introduced by event publication has shown that this implementation is sufficient under soft real-time requirements [17]. Although some applications are computationally intensive, the current DRDC autonomy suite does not stress this computer. Given that core parallelization it expected to ramp up, with 8 core processors available in the near future, computational limits are not expected to be confronted any time soon.

3.5 Inherent support for testing and debugging

Support for prototype testing and system debugging must be included, as part of the architectural design, in order to ensure seamless transitions between the various developmental phases. The popular Player/Stage project [18] has shown the effectiveness of such an approach with its inherent logging and playback capabilities [19]. Unfortunately Player/Stage, as a device server, doesn't provide the developer with either the middleware or framework to develop autonomous capabilities, nor does it directly address software stability issues.

The Boss vehicle, winner of the DARPA Urban Challenge, also incorporated data logging and playback facilities [20]. Although these capabilities were extensively used, the custom C++ application framework upon which they were developed doesn't seem to have been disseminated to the robotics community at large.

The DRDC approach, although influenced by Player's virtual sensors from logged data, provides a more unified approach. It builds upon two key factors:

- Stable interfaces and data structures, as described in Section 3.3.
- Miro's use of CORBA notification services to provide publish/subscribe capabilities.

The publish/subscribe paradigm decouples data producers and consumers. The primary means of intercomponent communication are *events* that encapsulate data and are transported via CORBA notification services. If the enclosed data structures represent the stable, unchanging aspects of the process, then alternate data producers and consumers can easily be substituted. This level of interchangeability enables a seamless transition between prototype testing, debugging and real system operation. Figure 6 graphically illustrates this concept. Whether the system is operating on the real platform, with real sensor data, or the

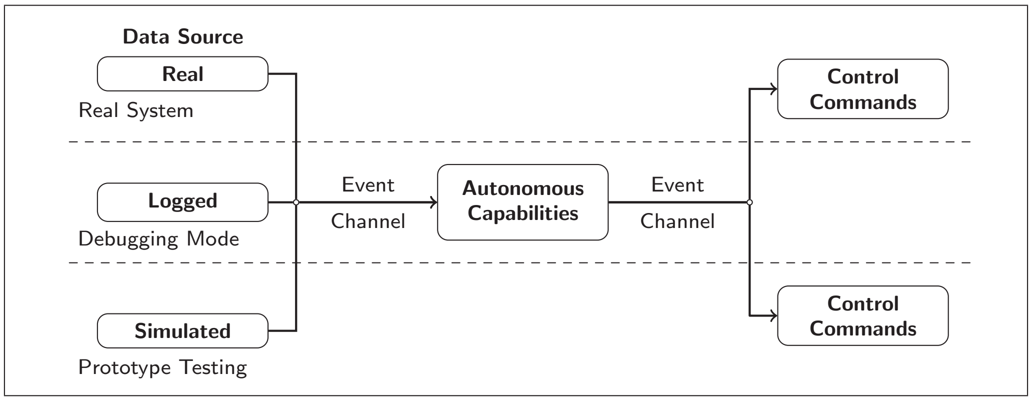


Figure 6: Seamless transitions between modes.

developer is debugging using logged data, or prototypical algorithms are under test with simulated data, the identical code, implementing autonomous capabilities, is in use. The ease with which the developer can transition between the various modes greatly simplifies the development process and hence, is a major boost to productivity.

3.5.1 Specific range example

Using range data as a concrete example to illustrate these mode transitions, consider the three sources of range data, shown in Figure 7. The consumer is unable to distinguish between the various sources of the Range3dSeqEventIDL event, as the event delivered via the Event channel is generic in nature. This allows the developer to seamlessly transition between prototype testing using simulated data, debugging with logged data, and real system operations on the physical platform.

Although the example shown in Figure 7 only highlights the Range3dSeqEventIDL structure,

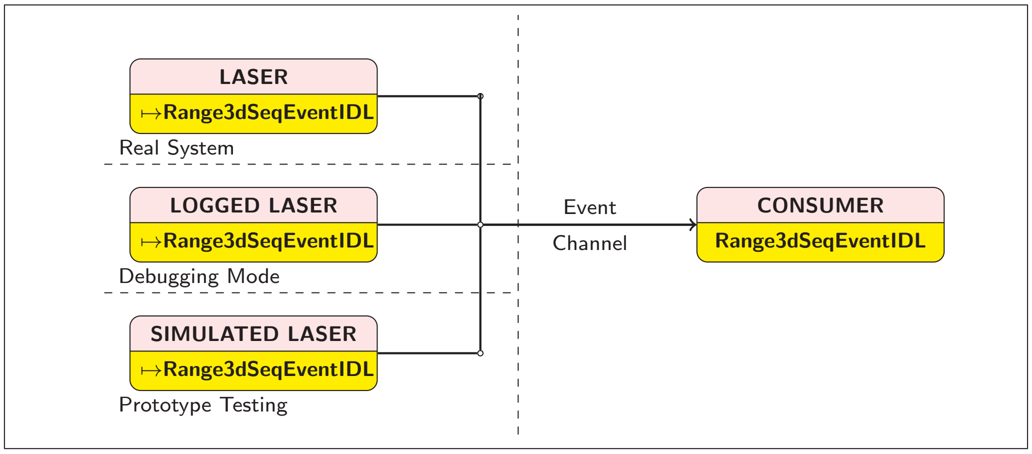


Figure 7: Mode transitions for range data.

the same principles and conclusions apply to any event-based data structure transmitted via the event channel.

3.5.2 Transitioning to the real world

As mentioned earlier in Section 3.5, our implementation requires only a **single** code instance. This is possible because of a hybrid implementation [21], which splits the non real-time from the real-time. As shown in Figure 8, lower level controls, with real-time requirements are implemented under a real-time operating system. For the Raptor UGV this corresponds to an MPC555 running RTEMS [4], which controls the actuators and electronics required for motion. Higher level capabilities, often referred to as *intelligence*, require only soft real-time

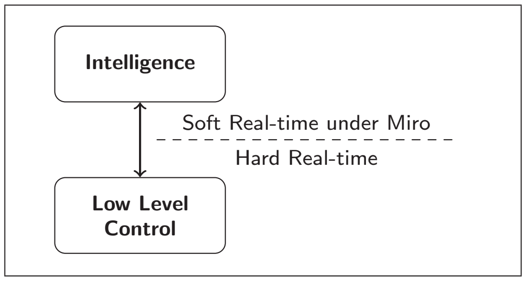


Figure 8: Hybrid implementation.

performance and only these capabilities operate within the Miro framework. Soft real-time suffices because of the nature of the environment. Uncertainties in pose, especially the orientation [22], sensor mount locations, sensor data such as range, time of data acquisition, and data processing into appropriate representations, all contribute to a world representation that is far from exact. Additionally, the Raptor's physical response to commands introduces more variability. When commanded to stop, the interaction between the wheels and terrain defines the stopping distance and this relationship has a fair degree of uncertainty [17].

4. Real-Time Executive for Multiprocessor Systems; an open source real-time operating system.

As an ensemble these real world details preclude hard real-time operation. To ensure safe Raptor UGV operations the vehicle speed is selected with a sufficiently large safety margin such that an emergency stop is guaranteed to avoid a collision with an obstacle and vehicle stability is assured.

As this approach requires only one code instance, issues such as resource management and concurrency can be addressed before real world testing. The publish/subscribe implementation employs a single event channel as shown in Figures 2 and 6. By routing all data through a single event channel all events become sequential; hence, under all modes of operation, events are published/received one at a time. This approach has advantages for debugging and logging data, but introduces a single point of failure that can effectively cripple the entire system [5].

4 Debugging using logged data

Debugging code is arguably the most arduous task that a software developer faces. Given the difficulties associated with this process it is crucial that the developer has access to the best tools available.

Although instances exist when debugging must occur on the real, physical platform, such instances are, fortunately, rare. Debugging on the physical platform is tedious, as not only requires the proper operation of numerous physical devices required, but in addition the developer has only marginal control over the environment. Thus, it is difficult to both repeat experi-ments and to control the rates at which sensor data is received. Finally, as a general rule, only one developer can exercise the physical platform at any given time.

4.1 Logging

The DRDC implementation, using the capabilities of the Miro framework, elegantly addresses the debugging issue. As described in Section 3.1, all DRDC developed components inherit publish/subscribe capabilities, allowing producers to anonymously publish data, making it available to subscribing consumers. This event-based paradigm can be viewed as a method to distribute raw sensor data; thus, the stream of events can be interpreted as a complete trace of the system state [23]. By subscribing to desired event streams, it is easy to log the data to file. Miro provides this generic logging capability though a facility named LogNotify. It should be stressed that only events can be logged in this manner. Intercomponent communication using the query/response paradigm cannot be captured by LogNotify. Figure 9 shows the data logging process in the context of DRDC's implementation. The specification of the events to be logged occurs within a human readable configuration XML file.

5. Within DRDC's experience, this particular failure has rarely, if ever, occurred.

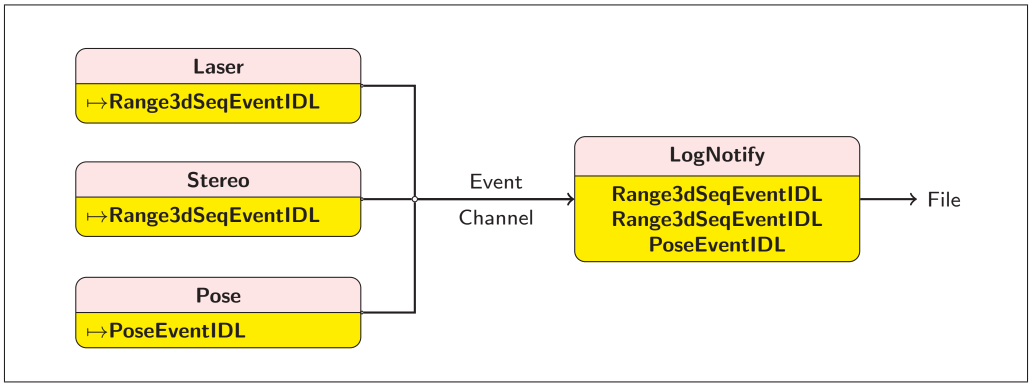


Figure 9: Logging data using LogNofity.

4.2 Data playback

The Miro LogPlayer allows logged data to be injected onto the event channel, in place of newly acquired sensory data. Thus, the system under test can repeatedly encounter the identical stream of input events. This ability to play back logged data is a great help in the debugging process. The LogPlayer supports additional features:

- The data play back rates can be varied from slower than to faster than real-time,
- The single event mode allows events to be published one at a time, and
- Event streams can be enabled and disabled.

The LogPlayer interface is shown Figure 10. The dial specifies the play back rate and the buttons control the play back process. Typically, during the debugging process, the user

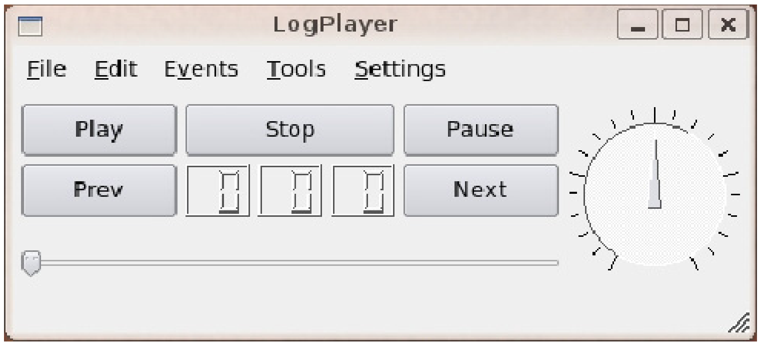


Figure 10: The Miro LogPlayer.

repeatedly plays back data while observing debugging outputs. Alternatively, while a debugger such as gdb is running, the user injects a single event onto the event channel and uses the debugger's capabilities to investigate the code's operation. Using the event timestamp, the LogPlayer recreates not only the original data, but the timing associated with the various event streams.

5 Prototyping with simulations

Prototyping in simulation and data logging offer similar benefits for autonomous systems development. In a simulated environment, developers can rapidly trace and modify the operation of the system under test. By ensuring identical initial conditions between repeated tests, simulation permits the rapid exploration of tuning parameters for the system under test, arguably with greater precision than with real hardware where the repetition of initial conditions is problematic. Further, re-initialization after faults or exceptions is easy in simulation and has none of the damaging consequences that can arise in real hardware if corrective action is not taken quickly. Significantly, in contrast to using logged data, mechanical simulation combined with simulated sensing allows autonomous behaviours to be exercised. For example, employing a path planner and an obstacle avoidance behaviour, the vehicle can actually explore a simulated world, moving within it to achieve a goal.

5.1 Simulation packages

While kinematic/dynamic simulators are relatively common mechanical design tools (e.g. Working Model [24]) and a component of gaming toolkits (e.g. Unreal Tournament [25]), dynamic simulations rarely include support for simulated sensors such as LIDAR and their particular characteristics. Of many capable robot modellers available (such as USAR-Sim [26]), two major systems have emerged to dominate the robotics community: the Player/Stage/Gazebo system and Microsoft's Robotics Studio [27]. Of these, only the Gazebo system is open source and thus both neutral to languages, operating systems, and control architectures and, more interestingly, mutable into additional roles beyond simulation alone.

Gazebo [28] is a multi-robot simulator for outdoor environments, a 3 D version of Stage, the original 2 D simulator developed under DARPA contracts. Gazebo can simulate robots and their sensors along with other objects in a three dimensional world. It uses the Open Dynamics Engine (ODE) [29] to generate both realistic sensor data and physically plausible interactions between objects. Bodies are modelled as point masses only, with extended shape used to calculate contact forces; visual meshes and textures are used for *eye candy*. As a consequence of simple bodies being point masses, in Gazebo, only composite bodies have moments of inertia.

Given that Gazebo met DRDC's requirements, was freely available and open source, it was the natural choice as a simulator.

5.2 Using Gazebo at DRDC

Gazebo obtains information about the world and the simulated Raptor vehicle from XML descriptions contained in a world file. In addition to the world description, these XML fields contain the physical layout, steering geometry, and sensor positions of the simulated Raptor, chosen to mimic those of the real vehicle. Joints in a model that can be controlled are defined using *controller* fields like the `controller:steering_position2d`, shown in Figure 11, that is used to drive the simulated Raptor. Likewise, a sensor, used to extract data from the simulation, is also defined using a *controller* field like the `controller:sicklms200_laser`.

The Gazebo simulator communicates with a client via shared memory. The *interface* tags in the world file determine the layout of the shared memory block. By default, the Gazebo simulator makes some data available through the unnamed *SimulationIface*. Gazebo's internal time and control of its execution is available via this interface. Additionally, one can connect to the *SimulationIface* and request by name the position and orientation of any body in the world. Further data is made available through additional named interfaces shown in Figure 1, such as the `interface:position` (named `position_iface_0`) tag in the `controller:steering_position2d` field and the `interface:laser` (named `laser_iface_0`) tag in the `controller:sicklms200_laser` field specified in the XML fragments. The client connects to these named interfaces and can then read from and/or write to the associated sections of the shared memory block.

5.2.1 Gazebo Miro interface

The GazeboService component, shown in Figure 12, is a Gazebo simulator client. It is a consumer of VelocityIDL events and a producer of PoseTransformIDL and Range3dSeqEventIDL events. Data from the VelocityIDL events are used to set the vehicle's commanded forward speed and steering angle by writing to the named *PositionIface*, `position_iface_0`. GazeboService populates the PoseTransformIDL events with data obtained by requesting the current position and orientation of the IMU on the simulated Raptor from the *SimulationIface*. The Range3dSeqEventIDL events are populated with data obtained from the named *LaserIface*, `laser_iface_0`, interface. When GazeboService is substituted for the Vehicle component and the Laser, Stereo, and INS sensor components, the remaining components are *unaware* of the change. The traffic on the event channel is identical in content, frequency, and relative timing to that occurring in the real hardware case.

5.2.2 Gazebo Modelserver compatibility

As depicted in Figure 12, Modelserver [30] provides vehicle and payload geometries to clients. In simulation, Modelserver complements Gazebo by using a hierarchical database of component frames joined through simple constraints and interrogated for *static* structural relationships. Gazebo, with a less flexible database, fully models mechanisms using a powerful computational engine to maintain and interrogate a *dynamic* model of the vehicle and components. As a temporary measure, Modelserver was modified to generate a Gazebo vehicle model, greatly simplifying an often frustrating exercise for complex vehicles. Currently, Modelserver can not build a complete Gazebo vehicle model; manual editing is still necessary. Modelserver also doesn't have any knowledge of sensor geometries and capabilities. It is necessary to manually check, for instance, that the maximum range of the laser rangefinders and the cameras' fields of view in the Gazebo world agree with those in the real world. This type of consistency checking should be made entirely automatic. In the long term, it is likely that Gazebo or one of its descendants may be enlisted as a vehicle control modeller, providing Miro control systems with a built-in capacity for geometric self-modelling.

5.2.3 Benefits of simulation

The use of the Gazebo simulator has proven to be invaluable in debugging the TerrainMap and TraverseMap components. As a source of noise-free data for debugging, LogNotify was used to capture GazeboService's PoseTransformIDL and Range3dSeqEventIDL events while the simulated Raptor moved within a simple blocks world (i.e., plane walls perpendicular to the ground) in Gazebo. Subsequently, that event stream was played back with the LogPlayer and those events captured by the TerrainMap and TraverseMap components. The relative simplicity of the data derived from the blocks world made it very easy to visually identify problems in the generation of terrain and traversability data. Having first used real world data to develop the TerrainMap and TraverseMap components, we were somewhat surprised to discover that errors remained in the generation of terrain and traversability data that only became apparent when using data from the Gazebo simulations. Additionally, sensor data with added Gaussian noise allowed for the systematic investigation and verification of a variance weighted technique to optimally fuse sensor readings. Verifying this performance with real data was not feasible due to the lack of repeatability, but simulated data made this investigation possible.

Using the Gazebo simulator has also made it possible to exercise autonomous behaviours, like the PathPlanner and ObstacleAvoidance components, on the desktop, a much quicker and easier undertaking than working with the real hardware. Since the autonomous behaviours actually drive the vehicle, any change in these components can affect the path taken by the vehicle in moving to a goal and hence any data acquired by the vehicle's sensors while following that path. This obviously precludes the use of logged sensor data in the development of these components. Thus, the entire collection of components, shown in Figure 12, must be running to exercise the autonomous behaviour components. Although it is possible to exercise these behaviours using real hardware outdoors [6], during the development and testing of these components, doing so in simulation is much easier and more convenient. Since the content and timing of events on the event channel are identical in the simulated and real world cases, the overall behaviour of the system should be similar in both cases. Differences will exist, obviously, due to modelling *errors* resulting from the simplicity of the simulated environment in comparison with the real world, a failure to capture the dynamics of the vehicle [7], and the idealized performance of the sensors.

6. Obviously, this is the ultimate goal of the entire exercise.
7. For instance, engine induced vehicle vibration doesn't exist in Gazebo.

```
<model:physical name="simpleCar_model">
  <controller:steering_position2d name="a␣car">
    <wheel>
      <joint>left_front_wheel_hinge</joint>
      <type>steer</type>
      <torque>1000</torque>
      <steerTorque>1000</steerTorque>
    </wheel>
    <wheel>
      <joint>right_front_wheel_hinge</joint>
      <type>steer</type>
      <torque>1000</torque>
      <steerTorque>1000</steerTorque>
    </wheel>
    <wheel>
      <joint>left_rear_wheel_hinge</joint>
      <type>drive</type>
      <torque>1000</torque>
    </wheel>
    <wheel>
      <joint>right_rear_wheel_hinge</joint>
      <type>drive</type>
      <torque>1000</torque>
    </wheel>
    <wheelSeparation>0.9</wheelSeparation>
    <wheelDiameter>0.8</wheelDiameter>
    <steerPD>10 0</steerPD>
    <steerMaxAngle>50</steerMaxAngle>
    <interface:position name="position_iface_0"/>
  </controller:steering_position2d>
  .
  .
  .
  <include embedded="true">
    <xi:include href="models/simplecar.model" />
  </include>
</model:physical>

<sensor:ray name="laser_0">
  <rayCount>361</rayCount>
  <rangeCount>361</rangeCount>
  <origin>0.05 0.0 0</origin>

  <minAngle>-90</minAngle>
  <maxAngle>90</maxAngle>

  <minRange>0.6</minRange>
  <maxRange>15</maxRange>
  <resRange>0.05</resRange>
  <displayRays>lines</displayRays>

  <controller:sicklms200_laser name="laser_controller_0">
    <updateRate>37.6</updateRate>
    <interface:laser name="laser_iface_0"/>
  </controller:sicklms200_laser>
</sensor:ray>
```

Figure 11: Gazebo XML world file

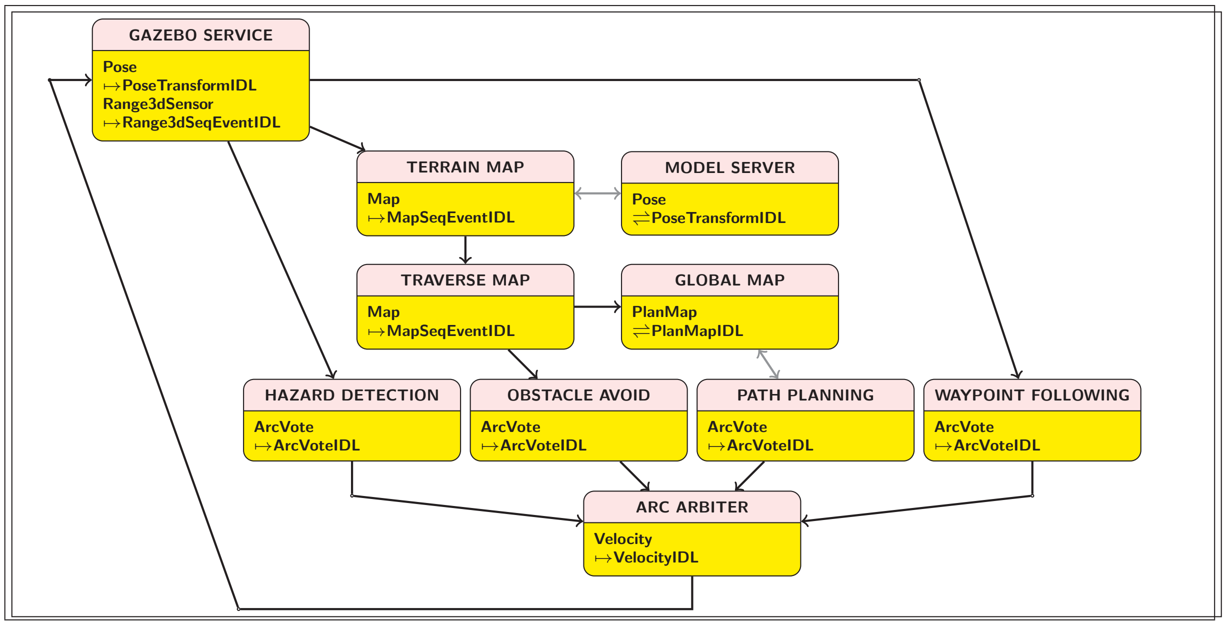


Figure 12: The gazebo service substituted for the Raptor and sensor components.

5.3 Example: Navigating a simulated world

By way of a concrete example illustrating the use of Gazebo, the simulated Raptor vehicle, shown in Figure 13, was driven through a blocks world while logging pose and laser data. The wheelbase, track, and steering angle limits along with the positioning of the sensors on

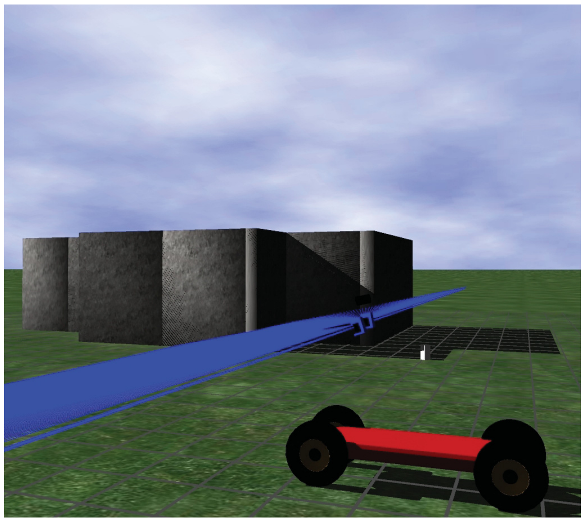


Figure 13: The simulated Raptor vehicle.

the vehicle mimic the real Raptor vehicle. The blue fans represent the range and angular extent of the laser rangefinders; offset on either side of the vehicle centreline, the one on the right looks down at 12° below the horizontal and the one on the left at 20°. The blocks world is intended to mimic the layout of buildings on an area of the DRDC range for which logged data already exists. The upper terrain map in Figure 14 has been generated with data from the real vehicle, travelling from right to left, the lower one was built with data logged from the Gazebo simulation. The colours represent the height of the terrain; green is neutral, red is below grade, and blue is above grade. The blue dot in the cross hairs denotes the location of the vehicle. The building near the top of the frame is a trailer with a platform and stairs out front. The Gazebo world uses two blocks as seen in Figure 13 to simulate this building. There are obvious differences between the two terrain maps. Clearly, the scale of the buildings is different. Also, the real world is not entirely flat whereas the Gazebo world, with the exception of the blocks, is completely flat. Nonetheless, apart from the source of the data, the generation of both maps was identical; the LogPlayer, ModelServer, TerrainService, and QtMap[8] components were used in each case.

8. Used for displaying the growing map.

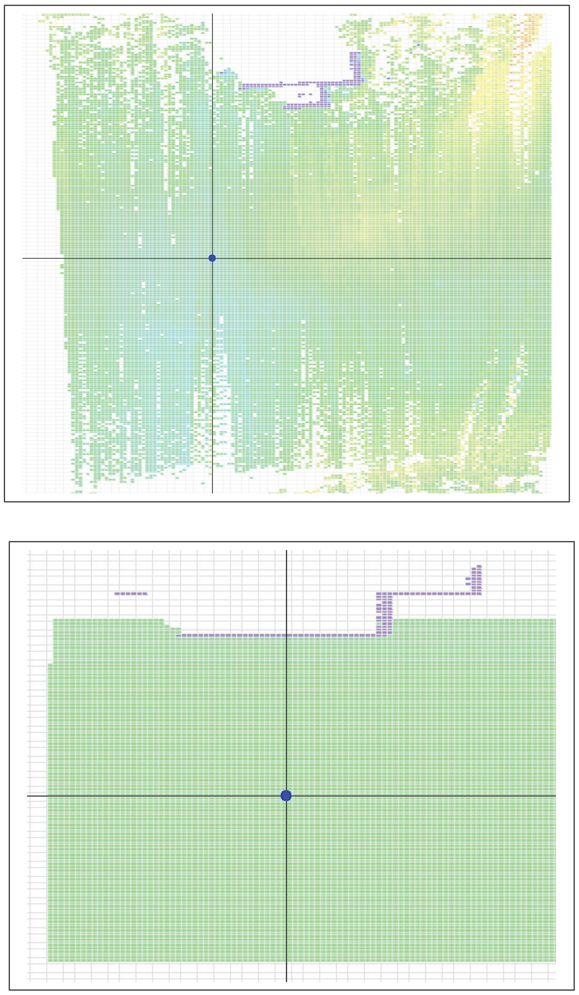


Figure 14: Comparison of a terrain map derived from real world data and gazebo simulation.

6 Summary and conclusions

The development of robotics software is difficult. We have shown that the adoption of a component-based framework built upon Miro and the use of the Gazebo 3 D simulator have resulted in systems that are arguably easier to prototype, debug, and maintain than would otherwise have been possible. The reasons for this are twofold:

1. The use of a component-based framework allows development to proceed at a level close to the abstractions of the problem domain. The developer need only be concerned with the content of intermodule communication and not the underlying mechanisms provided by the framework that allow the communication to take place. In addition, stable component interfaces further ease the burden of development, debugging, and maintenance.
2. Execution of the system in simulation, in the real world, or using logged data from either source proceeds with the indentical code instance. The timing and the type of the content in an event stream is identical regardless of its source, be it the real world, the LogPlayer, or the GazeboService. Thus, the software modules comprising a robotic system built upon Miro are *unaware* of the source of the events that they receive or, put another way, the execution of the system is unaffected by the source of the event stream.

References

[1] Neighbors, J.M. (1989), Software Reusability, Volume 1: Concepts and Models, ACM Press Frontier, Ch. Draco: A Method for Engineering Reusable Software Systems, pp. 295–319, Addison-Wesley.

[2] O'Connor, J., Mansour, C., Turner-Harris, J., and Campbell, Jr., G.H. (1994), Reuse in Command-and-Control Systems, *Software, IEEE*, 11(5), 70–79.

[3] Fayad, F. and Altman, A. (2001), An Introduction to Software Stability, *Communications of the ACM*, 44(45), 95–98.

[4] Coplien, J., Hoffman, D., and Weiss, D. (1998), Commonality and variability in software engineering, *Software, IEEE*, 15(6), 37–45.

[5] Brugali, D. and Salvaneschi, P. (2006), Stable Aspects in Robot Software Development, *International Journal of Advanced Robotic Systems*, 3, 1.

[6] Neighbors, J. M. (1992), The Evolution from Software Components to Domain Analysis, *International Journal of Soflware Engineering and Knowledge Engineering*, 2(3), 325–354.

[7] Broten, G., Monckton, S., Giesbrecht, J., Verret, S., Collier, J., and Digney, B. (2004), Towards Distributed Intelligence - A high level definition, (DRDC Suffield TR 2004-287) Defence R&D Canada – Suffield.

[8] Broten, G., Monckton, S., Giesbrecht, J., and Collier, J. (2006), Software Sysetms for Robotics, An Applied Research Perspective, *International Journal of Advanced Robotic Systems*, Volume 3, 1(2005-204), 11–17.

[9] Utz, H., Sablatnog, S., Enderle, S., and Kraetzschmar, G. (2002), Miro - Middleware for Mobile Robot Applications, *IEEE Transactions on Robotics and Automation*, 18, 493–497.

[10] (2003), TAO Developer's Guide, Oci tao version 1.3a ed, Vol. 1 and 2, 12140 Woodcrest Executive Drive, Suite 250, St. Louis, MO, 63141: Object Computing Inc.

[11] Henning, M. and Vinoski, S. (1999), Advanced CORBA Programming with C++, Addison-Wesley.

[12] Broten, G., Mackay, D., S., Monckton, and Collier, J (2009), The Robotics Experience: Beyond Components and Middleware, *IEEE Robotics and Automation Magazine*, 16(2008-227), 46–54.

[13] Brugali, D., Agah, A., MacDonald, B., Nesnas, I., and Smart, W. (2007), Engineering for Experimental Robotics, Ch. Trends in Robot Software Domain Engineering, pp. 3–8, Springer Tracts in Advanced Robotics.

[14] Broten, G., Monckton, S., Giesbrecht, J., and Collier, J. (2006), Software Engineering for Experimental Robotics, Number 2005-227, Drdc suffield sl Towards Framework-Based UxV Software Systems, An Applied Research Perspective, p. 34, Springer Tracts in Advanced Robotics.

[15] Utz, H. (2009), Miro Framework. http://www.informatik.uni-ulm.de/neuro/index.php?id=301&L=0. Accessed.

[16] Broten, G. (2007), Enhancing Software Modularity and Extensibility: A Case for Using Generic Data Representations, In *Proceedings of the 2007 IEEE International Conference on Robotics and Automation*, Vol. 2007-019, pp. 299–304, Roma, Italy.

[17] Broten, G., Mackay, D., and Desgagnes, R. (2008), Middleware for Robotics: Applications on Real-time Systems, In Brugali, D., (Ed.), *Third International Workshop on Software Development and Integration in Robotics2008 IEEE International Conference on Robotics and Automation*, Number 2007-088, p. 6, 2008 IEEE International Conference on Robotics and Automation.

[18] Gerkey, Brian, Vaughan, Richard T., and Howard, Andrew (2003), The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems, *Proceedings of the 11th International Conference on Advanced Robotics*, pp. 317–323.

[19] Vaughan, R. and Gerkey, B. (2007), Software Engineering for Experimental Robotics, Ch. Reusable Robot Software and the Player/Stage Project, pp. 267–289, Springer Tracts in Advanced Robotics.

[20] Baker, C.R. and Dolan, J.M. (2009), Street smarts for boss, *Robotics & Automation Magazine, IEEE*, 16(1), 78–87.

[21] Pont, F., Kolski, S., and Siegwart, R. (2005), Applications of a Real-Time Software Framework for Complex Mechatronic Systems, In *Proceedings of IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, Monterey, CA, USA.

[22] Broten, G. and Collier, J. (2006), Continuous Motion, Outdoor, 2 1/2D Grid Map Generation using an Inexpensive Nodding 2-D Laser Rangefinder, In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, Number 2006-061, pp. 4240–4245, Orlando, Fl.

[23] Department of Computer Science, University of Ulm (2005), Miro Manual, 0.9.4 ed, University of Ulm.

[24] Working Model 2D (online), Design Simulation Technologies, Inc. 43311 Joy Road, #237 Canton, MI 48187, `http://www.design-simulation.com/WM2D/description.php` (Access Date: May 25, 2009).

[25] Manojlovich, J., Prasithsangaree, P., Hughes, S., Chen, Jinlin, and Lewis, M. (2003), UTSAF: a multi-agent-based framework for supporting military-based distributed interactive simulations in 3D virtual environments, Vol. 1, pp. 960–968 Vol.1.

[26] Carpin, S., Lewis, M., Wang, Jijun, Balakirsky, S., and Scrapper, C. (2007), USARSim: a robot simulator for research and education, pp. 1400–1405.

[27] Jackson, J. (2007), Microsoft robotics studio: A technical introduction, *Robotics & Automation Magazine, IEEE*, 14(4), 82–87.

[28] Koenig, N. and Howard, A. (2004), Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator, pp. 2149–2154.

[29] Smith, Russell (2004), Open Dynamics Engine v0.5 Users Guide.

[30] Monckton, S., Vincent, I., and Broten, G. (2005), A Prototype Vehicle Geometry Server: Design and development of the ModelServer CORBA Service, (DRDC Suffield TR 2005-240) Defence R&D Canada – Suffield, Medicine Hat, Alberta.

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.) Defence R&D Canada – Suffield Box 4000, Station Main, Medicine Hat, Alberta, Canada T1A 8K6	2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.) UNCLASSIFIED **(NON-CONTROLLED GOODS)** **DMC A** **REVIEWED: GCEC JUNE 2010**

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

Architectural Support for Robotics: Software Prototyping and Testing

4. AUTHORS (Last name, followed by initials – ranks, titles, etc. not to be used.)

Mackay, D.J.; Broten, G.S.; Monckton, S.

5. DATE OF PUBLICATION (Month and year of publication of document.) December 2010	6a. NO. OF PAGES (Total containing information. Include Annexes, Appendices, etc.) 36	6b. NO. OF REFS (Total cited in document.) 30

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Suffield
Box 4000, Station Main, Medicine Hat, Alberta, Canada T1A 8K6

9a. PROJECT NO. (The applicable research and development project number under which the document was written. Please specify whether project or grant.)	9b. GRANT OR CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.) DRDC Suffield TM 2010-271	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

(X) Unlimited distribution
(   ) Defence departments and defence contractors; further distribution only as approved
(   ) Defence departments and Canadian defence contractors; further distribution only as approved
(   ) Government departments and agencies; further distribution only as approved
(   ) Defence departments; further distribution only as approved
(   ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11)) is possible, a wider announcement audience may be selected.)

Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

The development of robotics software is difficult. The sheer size and complexity of these software systems, the interaction of robots with their environment including the potential need for real-time operation, and the lack of accepted standards all contribute to this situation. Due to the diverse nature of robot systems, contributions from multiple researchers with disparate fields of expertise are required and that in itself further complicates software development.

At Defence R&D Canada Suffield, we have developed domain-specific middleware [8] based upon the Miro framework [9], that uses the ACE/TAO [10] implementation of CORBA [11]. This middleware provides infrastructure services enabling seamless communication between software components. Using this framework enables researchers to focus their attention at a level of abstraction close to the problem domain, for the most part ignoring the underlying details that enable a system to work.

Over the past six years, researchers in Autonomous Intelligent Systems Section have developed numerous software components for an heterogeneous set of robotic platforms. A key conclusion arising from this endeavour was the recognition that components and middleware, although crucial enablers, do not of themselves fully address the difficulties associated with software reuse [12]. Software reuse can only be realized if both the component interfaces and the underlying data structures are stable. And, although software reuse simplifies the process of adapting existing software to other robotic platforms, it doesn't obviate the need for tools and techniques that assist in prototyping and debugging. Our middleware provides for prototyping and debugging by allowing a system to seamlessly switch between real world data, logged data and data provided by a simulation environment. This paper describes our implementation, provides examples that illustrate its usage, and discusses the benefits accrued.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

software architecture
robotics
simulation