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

A Prototype Vehicle Geometry Server

Design and Development of the ModelServer CORBA Service

S. Monckton, I. Vincent and G. Broten
DRDC Suffield

Technical Report
DRDC Suffield TR 2005-240
December 2005

Canada

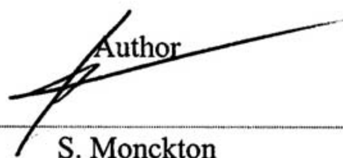
A Prototype Vehicle Geometry Server

Design and Development of the ModelServer CORBA Service

S. Monckton, I. Vincent and G. Broten
DRDC Suffield

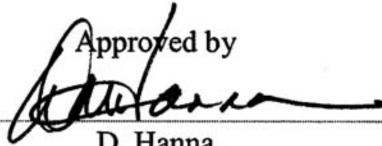
Defence R&D Canada – Suffield

Technical Report
DRDC Suffield TR 2005-240
December 2005



Author

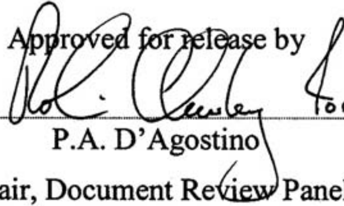
S. Monckton



Approved by

D. Hanna

Head, Autonomous Intelligent Systems Section



Approved for release by

P.A. D'Agostino

Chair, Document Review Panel

© Her Majesty the Queen as represented by the Minister of National Defence, 2005

© Sa majesté la reine, représentée par le ministre de la Défense nationale, 2005

Abstract

Defence R&D Canada is developing autonomy technologies for unmanned ground vehicles (UGV) and sensors (UGS); air (UAV) and subsea and surface (UUV and USV) vehicles – all operating together with minimal human oversight. To act autonomously, such devices must understand and change themselves and their surroundings through internal and external sensing and control. Autonomous systems must often model their physical structure as well as *localizing* or estimating their position in the world. Together, internal geometric models, localization, and range sensing produce an interpretation of the system's surroundings. For motion over and through complex environments, DRDC must further consider the use of dynamic models inside vehicle control loop to predict the necessary control forces for maneuver. These issues drove the adoption of modeling conventions compatible with systems that simulate mechanical dynamics including friction and closed kinematic chains. With sufficiently capable processing and high fidelity sensing, these systems can provide predictive control for even the most dynamic maneuvers. The first step, however, is to develop a regimented method of describing and storing system geometry. This report reviews the design of an elementary, prototype *geometric* model server and briefly describes the localization portion of the server.

Résumé

R&D pour la défense Canada met actuellement au point des technologies autonomes pour véhicules terrestres sans pilote (UGV) avec capteurs (UGS) dont des engins aériens sans pilote (UAV), des véhicules sous-marins sans équipage (UUV) ainsi que des véhicules de surface sans équipage (USV), tous opérant ensemble avec un minimum de surveillance humaine. Pour être en mesure d'agir de manière autonome, de tels appareils doivent comprendre leur milieu et effectuer des changements par rapport à eux-mêmes et à leur zone périphérique au moyen de capteurs et de contrôles à la fois internes et externes. Les systèmes autonomes doivent souvent aussi bien modéliser leur structure physique que localiser ou estimer leur position dans le monde. Les modèles géométriques internes, la localisation et la détection des distances, utilisés ensemble, produisent une interprétation de la zone périphérique du système. En ce qui concerne la motion au-dessus et au travers des environnements complexes, RDDC doit mieux tenir compte de l'utilisation des modèles dynamiques à l'intérieur de la boucle de commande du véhicule pour prédire les forces exercées sur les commandes qui sont nécessaires à la manœuvre. Ces problèmes ont amené à l'adoption des conventions de modélisation qui sont compatibles avec les systèmes simulant les dynamiques mécaniques y compris la friction et les chaînes cinématiques fermées. Avec des capacités suffisantes de traitement et de détection de haute fidélité, ces systèmes peuvent fournir des contrôles de prédiction même pour les manœuvres les plus dynamiques. La première étape consiste cependant à mettre au point une méthode uniformisée décrivant et conservant la géométrie des systèmes. Ce rapport examine le concept d'un modèle de prototype de serveur géométrique élémentaire et décrit brièvement la partie de la localisation effectuée par le serveur.

This page intentionally left blank.

Executive summary

A Prototype Vehicle Geometry Server

S. Monckton, I. Vincent, G. Broten; DRDC Suffield TR 2005-240; Defence R&D Canada – Suffield; December 2005.

Background: The long term objectives for Defence R&D Canada's autonomy program address disparate unmanned ground vehicle (UGV), unattended ground sensor (UGS), air (UAV), and subsea and surface (UUV and USV) vehicles operating together with minimal human oversight. To act autonomously, a robot must understand and change itself and its surroundings through internal and external sensing and control. Like fixed robots, autonomous vehicles must accurately model their physical structure. However, they must also *localize* or estimate their position in the world. Together, internal geometric models, localization, and range sensing produce an interpretation of the robot's surroundings. This report reviews the design of an elementary, prototype *geometric* model server and briefly describes the localization portion of the server.

On large vehicles, the displacement of sensors from each other and the vehicle's center of mass affects world representation accuracy. Sensors must be mutually calibrated to represent the world consistently within a single world model. During maneuvers, sensed displacements must agree despite physical displacement of sensors from one another. Accurate velocities and accelerations become important for both kinematic and dynamic modeling.

For motion over complex terrain, dynamic models should reside inside the vehicle control loop can to predict the necessary control forces for maneuver. This prospect and the requirements of the ALS project drove the adoption of modeling conventions compatible with systems that simulate mechanical dynamics including friction and closed kinematic chains. With sufficiently capable processing and high fidelity sensing, these systems can provide predictive control for even the most dynamic maneuvers. The first step, however, is to develop a regimented method of describing and storing vehicle geometry.

Principle Results: DRDC Suffield held vehicle trials over the summer and fall of 2005 exercising a converted Raptor ATV controlled by a distributed control network. Composed of distributed processes such as sensor drivers, device controllers, and mapping modules, the control system required the precise location or *pose* of devices and components in vehicle coordinates and the position of the vehicle in global coordinates. A single software service, *ModelServer*, provided CORBA-based local pose and global *localization* services to these client processes. In short, *ModelServer* provided the position and orientation representation of a *frame of interest* in the coordinates of a *frame of reference*.

Starting with an XML description of the vehicle components, *ModelServer* assembles the vehicle using three structures: a *Body*, a *BodyFrame*, and a *Constraint*. These three structures are stored within a *BodyList* and a *ConstraintList*. Together these structures form a directed graph *Model*. Given a desired *frame of interest* and *frame of reference*, *ModelServer* assembles the relative position and orientation between coordinate frames by searching this

graph and performing the necessary coordinate transformation computations. Furthermore, ModelServer used a Kalman Filter to localize the vehicle in global coordinates. Though not described in this report, the filter exploited GPS, IMU, and odometry to generate an optimal estimate of the vehicle position.

Significance of Results: In this module, DRDC Suffield developed a prototype netenabled system for internally storing, searching, manipulating, and communicating vehicle geometry to consuming processes. This work achieved two key results. First, a system for component surveys has been established that removes the necessity for large, expensive whole vehicle surveys. Secondly, the system established a starting point for extensions of vehicle modeling into control, such that multiple subsystems can simultaneously share an easily maintained geometric, kinematic or dynamic model. In this way, the same model may be exploited by vehicle controllers for world model building and dynamic path planning, or multiple vehicles for multivehicle coordination. In effect, ModelServer is DRDC's first step towards an integrated dynamic modeling environment for autonomous control.

Conclusions: Conclusions reached during the 2005 trials centred on usability of the XML database system and client polling services, and the accuracy and effectiveness of the localization filter.

Given a correctly formed XML geometry file, ModelServer repeatedly and successfully responded to polled arbitrary frame-to-frame tranformation queries through the CORBA interface. Though the client polling services were easy to use, assembling the XML database proved difficult and error prone for users unaccustomed to hierarchical body modeling systems such as ModelServer.

The localization system's event driven communication mechanics worked without error, though the localization filter itself proved difficult to tune to achieve a satisfactory result. Given odometry errors and unreliable GPS heading when stationary, the filter often became suboptimal. The team reached the conclusion that pure kalman filtering needs additional logic to correctly capture the vehicles global position and orientation – verifying the work of others in the field.

As a prototype, ModelServer successfully fulfilled the role of an elementary geometry database engine. However, in the long term, changes to this concept must include database editing/visualization tools, a dynamic modeling engine, and a distinct, expanded localization service.

Sommaire

A Prototype Vehicle Geometry Server

S. Monckton, I. Vincent, G. Broten; DRDC Suffield TR 2005-240; R & D pour la défense Canada – Suffield; décembre 2005.

Contexte : Les objectifs à long terme du programme d'autonomie de R & D pour la défense Canada traitent du problème des véhicules terrestres sans pilote (UGV) disparates, des capteurs terrestres (UGS) laissés sans surveillance, des engins aériens sans pilote (UAV), des véhicules sous-marins sans équipage (UUV) ainsi que des véhicules de surface sans équipage (USV), tous opérant ensemble avec un minimum de surveillance humaine. Pour être en mesure d'agir de manière autonome, de tels appareils doivent comprendre leur milieu et effectuer des changements par rapport à eux-mêmes et leur zone périphérique au moyen de capteurs et de contrôles à la fois internes et externes. Tout comme des robots fixes, les véhicules autonomes doivent modéliser leur structure physique avec précision. Ils doivent aussi cependant localiser ou estimer leur position dans le monde. Les modèles géométriques internes, la localisation et la détection des distances, utilisés ensemble, produisent une interprétation de la zone périphérique du robot. Ce rapport examine le concept d'un modèle de prototype de serveur géométrique élémentaire et décrit brièvement la partie de la localisation effectuée par le serveur.

Sur les gros véhicules, le déplacement des capteurs par rapport à chacun et par rapport au centre de gravité affecte l'exactitude de la représentation du monde réel. Les capteurs doivent être mutuellement calibrés pour représenter le monde réel de façon constante avec un seul modèle du monde réel. Durant les manœuvres, les déplacements captés doivent concorder malgré le déplacement physique des capteurs les uns par rapport aux autres. L'exactitude des vélocités et des accélérations devient très importante pour la modélisation à la fois cinématique et dynamique.

En ce qui concerne la motion sur des terrains complexes, les modèles dynamiques devraient résider à l'intérieur du réseau de la boucle de régulation du véhicule pour prédire les forces exercées sur les commandes nécessaires à la manœuvre. Cette perspective et les besoins du projet des Systèmes terrestres autonomes (ALS) ont amené à l'adoption des conventions de modélisation qui sont compatibles avec les systèmes simulant les dynamiques mécaniques y compris la friction et les chaînes cinématiques fermées. Avec des capacités suffisantes de traitement et de détection de haute fidélité, ces systèmes peuvent fournir des contrôles de prédiction même pour les manœuvres les plus dynamiques. La première étape consiste cependant à mettre au point une méthode uniforme décrivant et conservant la géométrie des systèmes.

Les résultats principaux : RDDC Suffield a effectué des essais sur véhicules durant l'été et l'automne 2005 sur un Raptor ATV contrôlé par un réseau de commande réparti. Composé de systèmes de régulation répartis tels que les capteurs conducteurs, les contrôleurs et les modules de cartographie, le système de contrôle exigeait la location ou la pose précise des appareils et des composants dans les modules de cartographie et la position du véhicule dans les coordonnées globales. Un seul service de logiciels, ModelServer, a fourni les services de pose locale et de localisation globale basés sur l'architecture CORBA à ces programmes

clients. En bref, ModelServer a fourni la représentation de la position et de l'orientation d'une trame d'intérêt dans les coordonnées d'une trame de référence.

En commençant avec une description XML des composants du véhicule, ModelServer assemble le véhicule en utilisant trois structures : une Carrosserie, une Ossature de carrosserie et une Contrainte. Ces trois structures sont mises en mémoire dans une Liste de carrosserie et dans une Liste de contraintes. Ces structures mises ensemble forment un Modèle de graphe orienté. Selon la trame d'intérêt et la trame de référence désirée, ModelServer assemble la position et l'orientation relatives entre les trames de coordonnées en faisant une recherche du graphe et en effectuant les calculs nécessaires à la transformation dans les coordonnées. De plus, ModelServer a utilisé un Filtre de Kalman pour localiser le véhicule avec des coordonnées globales. Bien qu'il ne soit pas décrit dans ce rapport, le filtre a exploité le GPS, l'UMI et l'odométrie pour générer des estimations optimales de la position du véhicule.

La portée des résultats : Dans ce module, RDDC Suffield a mis au point un prototype de système Internet pour mettre en mémoire, rechercher, manipuler et communiquer au niveau interne la géométrie du véhicule aux programmes clients. Ces travaux ont abouti à deux résultats clés. Un système pour le relèvement des composants a d'abord été établi et rend inutile les relèvements importants et coûteux de véhicules au complet. Le système a ensuite établi un point de départ pour des extensions de modélisation de véhicules dans les commandes, de telle manière que des sous-systèmes multiples peuvent facilement partager et maintenir des modèles géométriques, cinématiques ou dynamiques. De cette manière, le même modèle peut être exploité par les contrôles de véhicules pour la construction de modèles mondiaux et pour la planification de voies d'accès ou bien des véhicules multiples pour une coordination de véhicules multiples. En fait, ModelServer est la première étape de RDDC vers un environnement de modélisation intégrée et dynamique de contrôle autonome.

Conclusions : Durant les essais de 2005, on a abouti à des conclusions axées sur l'aptitude à l'emploi du système de base de données XML, des services d'appel sélectifs à la clientèle ainsi que sur la précision et l'efficacité du filtre de localisation.

Selon le dossier de géométrie formé avec précision par XML, ModelServer a répétitivement réussi à répondre aux requêtes arbitraires de transformation d'une trame à l'autre au moyen de l'interface CORBA. Les services d'appels sélectifs à la clientèle étaient faciles à utiliser mais il s'est avéré difficile d'assembler les bases de données XML. De plus, les utilisateurs non accoutumés aux systèmes de modélisation hiérarchiques de carrosserie, tels que ceux du ModelServer, étaient prédisposés à l'erreur.

La mécanique de la communication effectuée par le système de localisation a fonctionné sans erreur bien que le filtre de localisation lui-même se soit avéré difficile à mettre au point pour produire un résultat satisfaisant. Étant donné les erreurs d'odométrie et les tableaux peu fiables du GPS en position stationnaire, le filtre est devenu sous-optimal. L'équipe en a conclu que le filtrage kalman pur nécessitait une logique additionnelle pour capturer correctement la position et l'orientation globale des véhicules – en vérifiant les travaux des autres dans le domaine.

Comme prototype, le ModelServer a réussi à remplir le rôle d'un appareil élémentaire de bases de données géométriques. À long terme, il faudra cependant changer ce concept pour y inclure des outils d'édition et de visualisation de base de données, un moteur de modélisation dynamique ainsi qu'un service de localisation distinct et élargi.

Table of contents

Abstract . . . i

Résumé . . . i

Executive summary . . . iii

Sommaire . . . v

Table of contents . . . vii

List of figures . . . x

List of tables . . . xi

1 Introduction . . . 1

2 Why a Geometric Model? . . . 2

3 Geometric Model Design . . . 2

3.1 Pose . . . 2

3.2 The Single Rigid Body Geometric Model . . . 4

3.3 The Hierarchical Rigid Body Geometric Model . . . 4

4 DRDC's Geometry Server: ModelServer . . . 6

4.1 Pose Representation . . . 6

4.2 The Body . . . 7

4.2.1 The Body Object . . . 7

4.3 The BodyFrame . . . 8

4.3.1 The BodyFrame Object . . . 9

4.4 The Constraint . . . 10

4.4.1 The Constraint Object . . . 11

4.5 The Model . . . 11

4.6 The Search Engine: PlatformBase . . . 12

4.6.1 Searching for frames . . . 13

4.6.2 ModelServer search . . . 15
4.6.3 Building Transforms . . . 15
4.7 Localization . . . 15
4.8 ModelServer Communication . . . 16
4.8.1 Subscribe-Publish Server . . . 16
4.8.2 Typical ModelServer Client . . . 17
4.9 ModelServer IDLs . . . 17
4.9.1 Pose . . . 18
4.9.2 Platform . . . 18
5 Operation . . . 19
5.1 Proprioceptive Sensing . . . 19
5.2 Exteroceptive Sensing . . . 19
5.3 Miro Services . . . 20
5.4 ModelServer Operation . . . 20
5.4.1 Launch . . . 20
5.4.2 Steady State . . . 21
6 An Example: the DRDC Raptor . . . 21
6.1 Raptor Body . . . 21
6.2 Nodding SICK Lasers . . . 21
6.3 Digiclops Stereo Camera . . . 22
6.4 FLEA camera . . . 22
6.5 GPS and DGPS antenna body . . . 22
6.6 Microstrain 3DM-GX1 . . . 22
6.7 SpeedLan Wireless antenna body . . . 22

7 Discussion and Conclusion 23

7.1 Geometry Server (Poll Mode) 23

7.2 Localization Server (Event Driven Operation) 24

7.3 Conclusions and Future Work 25

References 31

Annex A: Position and Orientation 33

A.1 Orientation 33

A.2 Homogeneous Transform 33

Annex B: Geometry Parameter Files 35

B.1 Compile-Time Parameter Files 35

B.2 Parsing Run-Time Parameter Files 36

B.3 Run-Time Parameter Files 37

B.4 Nested Data Structures 37

Annex C: A Raptor XML Body Model 41

C.1 Structure 41

C.1.1 The Body XML entries 41

C.1.2 The Constraint XML entries 43

Annex D: AFA::namedObject Class Reference 45

Annex E: AFA::Body Class Reference 47

Annex F: AFA::Pose Class Reference 53

List of figures

Figure 1: A simple flat hierarchy 4

Figure 2: A hierarchical rigid body model. 5

Figure 3: The symbolic relationship between bodies $\mathcal{B}_1$ and $\mathcal{B}_2$, their frames, $\mathcal{F}_j$, and constraint, $\mathcal{C}_{12}$. 7

Figure 4: A summarized view of the The Pose Object 8

Figure 5: A summarized view of the The Body Object 9

Figure 6: A summarized view of the The BodyFrame Object 10

Figure 7: Inheritance Diagram for Body, BodyFrame, and Constraint objects . . . 11

Figure 8: A summarized view of the The Constraint Object 12

Figure 9: A Depth first search example superimposed in red on DRDC's model tree. With the digiclops imageplane as the frame of interest and the raptor origin as the frame of reference, this figure assumes the search order is bottom to top. 14

Figure 10: Publish-Subscribe Server Design Pattern 16

Figure 11: Client Design Pattern 17

Figure 12: A summarized view of the The Pose CORBA Object 18

Figure 13: A summarized view of the The Platform CORBA Object 19

Figure 14: The ALS Project Process diagram 20

Figure 15: World coordinate system on the Raptor body 23

Figure 16: Corners of the vehicle 24

Figure 17: Rail measurement points 25

Figure 18: Measurement points on the nodding SICK body 26

Figure 19: Digiclops measurement points 27

Figure 20: FLEAS measurement points 28

Figure 21: GPS and DGPS antenna measurement points 29

Figure 22: Microstrain IMU measurement point 29

Figure 23: Wireless antenna measurement points 30

Figure D.1:The Hierarchy for key geometry objects 45

This page intentionally left blank.

1 Introduction

With a long history of teleoperation research, Defence R&D Canada (DRDC) Autonomous Intelligent Systems Section (AISS) has embarked on autonomous systems development projects for the Canadian Forces. The first project, Autonomous Land Systems (ALS), sought to demonstrate basic autonomous multivehicle capabilities and establish the personnel base and technical foundations for future projects. The outcome of a 2 year development effort, this paper summarizes the design, implementation, and performance of a vehicle model database, ModelServer.

To react and make changes to the world, an unmanned vehicle must understand both itself and its surroundings. Like fixed robots, autonomous vehicles must accurately model their physical structure but they must also *localize* or estimate their position relative to the external world. By combining internal geometric models, external position estimates, and external range sensing, an unmanned vehicle can produce an interpretation of the world. This report discusses the infrastructure necessary to understand the vehicles internal geometry and external localization.

Assembling an accurate world representation depends on both internal and external position estimation working together seamlessly. Structural geometry and localization, though both geometric measurement problems, differ significantly in methodology. By precisely estimating position and motion estimates Internal modeling improves vehicle control by estimating geometry, kinematics, and dynamics of sensors and effectors in vehicle coordinates. Localization maintains an estimate of the vehicle coordinate system with respect to an external world coordinate, usually through world sensing and signal processing. To exploit sensed world features, feature locations with respect to the robot must be developed. This requires the precise position and motion estimates of both the vehicle in world coordinates and sensors/effectors in vehicle coordinates.

In the ALS project, DRDC augmented two ATV class vehicles with a mixture of range and position sensors, geometric models, and hybrid arbitrated control to generate limited vehicle autonomy. This report reviews the design, implementation, and use of this system's *geometric* modeling and localization services.

In general, mechanical modelling trys to predict a system's physical response to motion through three steps:

- geometric modeling, the estimation of body position and shape.
- kinematic modeling, the estimation of body motion and acceleration.
- dynamic modeling, the estimation of internal and external body forces.

This report focusses on the design of an elementary, prototype *geometric* model *server* and briefly describes the localization portion of the server.

2 Why a Geometric Model?

To develop a consistent world representation, observations from exteroceptive sensors must be consistent with one another. This is particularly true of large vehicles where sensor displacement from the vehicle's center of gravity (CG) must be considered. For example, under the right conditions differential global positioning systems (DGPS) can place the vehicle with a 2cm spherical error. Thus, any vehicle using DGPS should record the displacement of the DGPS antenna from the vehicle's CG, often the preferred reference point of the vehicle in space. Motion detected through DGPS and inertial systems must then be compensated with equations of motion to produce a consistent estimate of vehicle motion at the CG, even though these devices may be neither colocated nor at the CG. Similarly, high precision range sensors must be accurately positioned on the vehicle with respect to the CG to correctly portray terrain elevation around the vehicle.

Since the vehicles dynamics will be expressed in terms of the vehicles CG, all detected motion – from odometry to inertial measurements – must be transformed through kinematics equations to the CG. Further, dynamic and kinematic models demand accurate velocities and accelerations. Without accurate first and second derivatives of position, models cannot predict vehicle performance or accurately predict relative motion in the outside world. For example, vehicle and ground orientation must be known precisely to ensure vehicle stability during high velocity cornering. Similarly, accurate target tracking requires accurate estimates of position and its derivatives.

As a first step in this process, a geometric model must store the position and orientation of significant features or devices on the vehicles physical structure *and* maintain a running estimate of the vehicle's position in the world coordinate system.

3 Geometric Model Design

Given that CAD, solid, and dynamic modelers have developed many geometry storage methods, DRDC should attempt to follow a development process that incrementally moves towards industrial modeling standards, but in a control and autonomy context. CAD engines store geometry as linked graphical elements. Solid modelers capture geometry through sculpting operators. Dynamic modelers use this same representation, adding assembly operators and numerical methods. Autonomous vehicles need similar methods to predict vehicle performance and plan through known obstacle fields. This section reviews the approach DRDC adopted for this first, elementary prototype system.

3.1 Pose

Geometry engines represent the position and orientation, or *pose* of coordinate frames on a rigid body. How is this expressed? Homogeneous transformations are the most universal

pose representation (see Annex A for further detail), for example: (e.g.[1]).

$$\mathbf{X} = \begin{bmatrix} \mathbf{n}_x & \mathbf{o}_x & \mathbf{a}_x & \mathbf{p}_x \\ \mathbf{n}_y & \mathbf{o}_y & \mathbf{a}_y & \mathbf{p}_y \\ \mathbf{n}_z & \mathbf{o}_z & \mathbf{a}_z & \mathbf{p}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

However convenient it may be, the Homogeneous Transform consumes computer memory and, worse, cannot be differentiated into the well-defined velocity vector:

$$\dot{\mathbf{x}} = \begin{bmatrix} \mathbf{v}_x & \mathbf{v}_y & \mathbf{v}_z & \omega_x & \omega_y & \omega_z \end{bmatrix}^T \tag{2}$$

Where $\mathbf{v}$ is the linear velocity vector and ω is the angular velocity vector. To work around this problem, some representations combine position and roll-pitch-yaw angles into pose vectors of the form:

$$\mathbf{x} = \begin{bmatrix} \mathbf{p}_x & \mathbf{p}_y & \mathbf{p}_z & \theta_x & \theta_y & \theta_z \end{bmatrix}^T \tag{3}$$

though this form has even less mathematical meaning than homogeneous transforms, it is, at least, compact. Unfortunately, RPY representations suffer singularity in some regions, a condition known as *gimbal lock* and equivalent to the mechanical jamming of aviation attitude indicators during aerobatics. To date, a mixture of position and quaternions[2] possess the best compromise between compact size, differentiability, and numerical stability, unlike roll-pitch-yaw equivalents.

$$\mathbf{x} = \begin{bmatrix} \mathbf{p}_x & \mathbf{p}_y & \mathbf{p}_z & q_s & q_x & q_y & q_z \end{bmatrix}^T \tag{4}$$

The 6DOF displacement between two frames or *transformation* can also be represented as a pose. Therefore, pose, regardless of representation, represents both coordinate frames and transformations between frames.

Without reviewing homogeneous transform algebra in detail, it is sufficient to understand the pose of an object, q, in coordinate system B, $\mathbf{X}_q^B$, expressed in coordinate system A, $\mathbf{X}_q^A$, is the *matrix product* of the pose of B in A and the pose of the object, or analytically:

$${}^A\mathbf{X}_q = {}^A\mathbf{X}_B {}^B\mathbf{X}_q \tag{5}$$

The succession of frames on the RHS of this equation form a *kinematic chain*. Thus the position of the Nth frame in base (or 0th frame) coordinates is simply:

$${}^0\mathbf{X}_N = \prod_{i=1}^{N} {}^{i-1}\mathbf{X}_i \tag{6}$$

PSfrag replacements

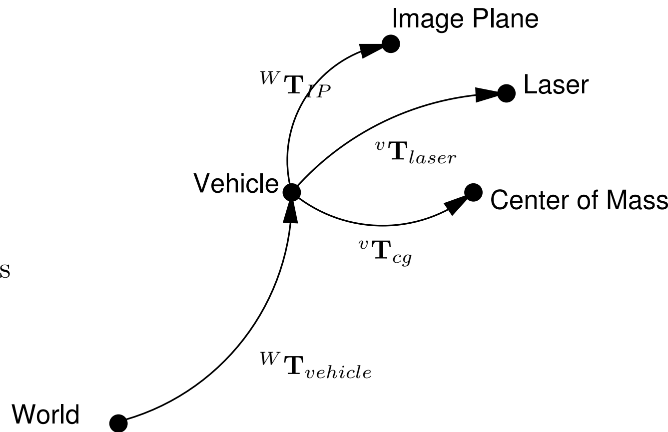


Figure 1: *A simple flat hierarchy*

3.2 The Single Rigid Body Geometric Model

The simplest vehicle geometric model stores a simple list of significant vehicle coordinate frames expressed with respect to a single vehicle coordinate system. To ensure consistency between measurements, a survey measures coordinate locations in 3-space over a short period. Usually, these points share a single coordinate system and are stored as a list of homogeneous transforms from the vehicle origin. This treats the vehicle and all its components as a single rigid body, an example of which is depicted in Figure 1.

Such 'whole vehicle' surveys keep the geometry database simple and reduce measurement error, but complicate its use and maintenance. Users retrieve labelled positions in absolute vehicle coordinates simply by recovering a similarly labelled entry in the database. Since all positions are measured with respect to a single vehicle origin, additions or changes to the database may entail direct measurment on the assembled vehicle using the same methods as the original survey. The frequent, often iterative, field changes common to a research setting makes this approach awkward and error prone. Clearly, to achieve greater flexibility demands a modular, maintainable – and more complex – solution.

3.3 The Hierarchical Rigid Body Geometric Model

Segmenting systems into frequently changed subcomponents eases maintenance of the geometry database but complicates the database itself. Rather than placing all vehicle components into a single coordinate system, parts can be grouped into smaller rigid body systems *attached* to one another forming a composite assembly or *body*. This forms a hierarchy of *bodies* containing multiple coordinate frames expressed with respect to a single body coordinate system. This transforms the database into a set of bodies, each containing a set of coordinate frames describing the position and orientation of significant body features.

Constraints bind these bodies together into an assembly. Constraints are geometric relationships between body frames that mathematically limit or prevent relative motion between

Figure 2: *A hierarchical rigid body model.*

two frames. The constraint concept, central to dynamic modeling, limits the degrees of freedom between frames. Normally, any two 'free' frames possess 6 mutual degrees of freedom (DOF).Constraints fix the relationship between points (or frames) and limit these DOF. Dynamic models vary the number and type of constraints applied to frames to achieve common mechanical linkages. For example, fixing the relative distance between two 3-space points in X and Y only, permits points to slide in pure linear motion along Z – to form a linear slider. Through similar techniques, revolute joints, linear sliders, spherical joints, cams and motors may be simulated. Instead of a simple list of coordinate frames, this form of geometry database now relies on software to compute the global coordinate of any requested body frame through methods similar to equation 6.For example, DRDC converted an ATV class vehicle to autonomous operation, adding a number of sensors and radios to the surface of the cab and computer enclosure. The hierarchical model for this vehicle is depicted in figure 2. While complex, this hierarchical rigid body model accepts additions and changes more easily than an equivalent single rigid body model simply by isolating dependent measurements to frequently moved or duplicated components – such as the Nodding Lasers.

CAD and robotics research commonly use such hierarchical models(e.g. [3]) as do dynamic modeling systems [4]. However, dynamic modelers seek to predict the dynamic state in

addition to position, from the resolution of forces, inertia, friction, and geometry. The computation in these systems is far more complex than involved in equation 6.

Dynamic simulators compute the accelerations of the bodies given the applied forces, displacements, and velocities, integrating the results to develop the next time step's position and velocity. A typical newtonion form for a mechanical system:

$$\mathbf{D}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{J}^T(\mathbf{f}_e) = \tau \tag{7}$$

where $\mathbf{q}$ are the system's generalized coordinates [5], $\mathbf{D}(\mathbf{q})$ is the *mass matrix*, $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$ contains all coriolis and damping terms, $\mathbf{G}(\mathbf{q})$ expresses gravitational terms, and $\mathbf{f}_e$ are the external forces on the system. The system's equation can be rewritten:

$$\mathbf{D}(\mathbf{q})\ddot{\mathbf{q}} = \tau - \mathbf{b} \tag{8}$$

$$\mathbf{b} = \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{J}^T(\mathbf{f}_e) \tag{9}$$

where $\mathbf{b}$ is a *bias vector*, the output of the recursive Newton Euler dynamics algorithm when $\ddot{\mathbf{q}}$ have been set to zero. Given $\mathbf{D}(\mathbf{q})$, the accelerations may then be computed simply through:

$$\ddot{\mathbf{q}} = \mathbf{D}(\mathbf{q})^{-1} [\tau - \mathbf{b}] \tag{10}$$

Numerical integration moves the system state forward through time [6].

4 DRDC's Geometry Server: ModelServer

With a long term interest in exploiting more elaborate mechanical models, DRDC adopted dynamic modeling conventions compatible with the dynamic modelers such as the open source Open Dynamics Engine[4]. These systems simulate rigid body dynamic systems and often model contact dynamics, friction, and closed kinematic chains. The first step in this process is to capture basic geometry through an hierarchical rigid body model.

The DRDC hierarchical model manages three data types: a *Body*, $\mathcal{B}$, a *BodyFrame*, $\mathcal{F}$, and a *Constraint*, $\mathcal{C}$, accessible through a *BodyList*, $\mathbf{L_B} = [\mathcal{B}_1, \mathcal{B}_i, ..., \mathcal{B}_m]$ a *ConstraintList*, $\mathbf{L_C} = [\mathcal{C}_1, \mathcal{C}_i, ..., \mathcal{C}_q]$, and a directed graph *Model*, $\mathcal{M}$. The relationship between these types appears in Figure 3.

The following subsections will detail the theoretical design and practical implementation of each of these types.

4.1 Pose Representation

Surprisingly, there are few open software sources for pose representation. The most exhaustive toolkit comes from the US National Institute of Standards and Technology (NIST) in the form of the somewhat dated Real Time Control Systems Posemath library[7]. This library contains operators transforming pose between various representations including RPY, rotation matrices, and quaternions. DRDC used this library within a simple *Pose* object.

PSfrag replacements

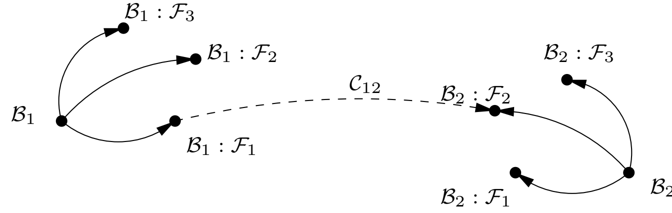


Figure 3: *The symbolic relationship between bodies $\mathcal{B}_1$ and $\mathcal{B}_2$, their frames, $\mathcal{F}_j$, and constraint, $\mathcal{C}_{12}$.*

The object shown in Figure 4 merges Posemath structures with efficient numerical structures from the BOOST library[8], an object oriented layer of the uBLAS libraries, and CORBA IDL data types (discussed in greater detail in section 4.9).

The class maintains three representations, a 7-vector position-quaternion state, an homogeneous transform, and a quaternion. All orientations `set` through this class are converted into quaternions and then into rotation matrices. All `get` orientation operations extract orientation from the quaternion. The isolation of orientation to quaternions avoids gimbal lock common to RPY representations.

Pose inherits from the simple `namedObject` class, providing inheriting objects with a common labelling structure.

4.2 The Body

A *Body* contains a unique string name identifier, s_b, and a list of *BodyFrames*, $\mathbf{L}_{i\mathcal{F}} = [\mathcal{F}_1, \mathcal{F}_j, ..., \mathcal{F}_n]$ or for the ith body: $\mathcal{B}_i = \langle s_b, \mathbf{L}_{i\mathcal{F}} \rangle$.

Clearly, the *Body* construct groups BodyFrames together through an intermediate coordinate system. In this simple prototype, loose rules define what is or is not a body. For fixed assemblies such as the stereo camera and its mounting bracket, differentiating components within the assembly provides no real benefit. In general, bodies aggregate infrequently moved components into a single assembly. Hence a component and a coupling bracket often constitute single bodies. Using this simple guideline, any movable component collection falls into a single body definition.

This strategy permits the quick duplication of bodies if multiple devices (e.g. cameras) are installed on a single vehicle.

4.2.1 The Body Object

The Body Object, a C++ construct, currently inherits from Pose, a simplified listing appears in figure 5:

```
class Pose : public namedObject
{
public:
     Pose (std::string * aName);
     Pose (std::string * aName, MatrixParameters * matrixParams);
     Pose ();
     ~Pose ();
     void                print();
     int                 getType();
     void                setType(int value);
     vector<double>    * getStateVector();
     void                setStateVector(vector<double> * value);
     void                setStateVector(double px, double py, double pz,
                                 double qs, double qx, double qy,
                                 double qz );
     matrix<double>    * getHTransform();
     PoseTransformIDL  * getPoseTransformIDL();
     matrix<double>    * getHTransformInv();
     PoseTransformIDL  * getInversePoseTransformIDL();
     void                setHTransform(matrix<double> *aTransform);
     void                setHTransform(PoseTransformIDL *aTransform);
     matrix<double>    * getRotationMatrix();
     matrix<double>    * Pose::getRotationMatrixInv ();
     void                setRotationMatrix(matrix<double> *aRotMatrix);
     void                setRotationMatrix(PoseTransformIDL *PoseTransformIDL);
     vector<double>    * getRollPitchYaw();
     void                setRollPitchYaw(double roll, double pitch, double yaw);
     vector<double>    * getPositionVector();
     void                setPositionVector(vector<double> *aVector);
     PM_HOMOGENEOUS    * getPM_HOMOGENEOUS();
     PM_QUATERNION     * getPM_QUATERNION();
     Pose        operator    = ( Pose B);
friend ostream&  operator   <<(ostream& os, Pose& p );

private:
     int Type;
     vector<double>     * State;
     PM_HOMOGENEOUS     * HTransform;
     PM_QUATERNION     * Quaternion;
};
```

Figure 4: *A summarized view of the The Pose Object*

Though many constructors are provided, the most important uses parameters to automatically construct bodies based on an XML parameter description (described in Annex C).

The object uses the access methods `addBodyFrame()`, `getBodyFrame()` to add and retrieve bodyframe objects from the `bodyFrames` list, itself an STL object. The methods `setInertiaMatrix`, and `getInertiaMatrix()` are placeholder access methods for future dynamic inertial parameters.

4.3 The BodyFrame

A *BodyFrame* contains a unique string identifier, s_f, a homogeneous transform from the ith body's origin to the jth frame, $\mathbf{A}_{ij}$, and pointers to the parent Body, $\mathcal{B}_i$, and to a constraint, $\mathcal{C}_{ij}$. For the jth bodyframe: $\mathcal{B}_i : \mathcal{F}_j = \langle s_f, \mathcal{B}_i, \mathcal{C}_{ij}, \mathbf{A}_{ij} \rangle$.

```
class Body : public Pose
{
public:

  Body ( );
  Body (std::string * aName);
  Body (BodyParameters * bP);
 ~Body ();

  AFA_BFRAME_ERRCODE verifyBodyFrame(BodyFrame * aFrame);
  void   addBodyFrame (BodyFrame * aFrame) ;
  AFA_ERRCODE  delBodyFrame (string * aName);

  BodyFrame * getBodyFrame (string * aName);
  BodyFrame * getBodyFrame (char * aCharName);
  list<BodyFrame *> * getBodyFrameList ();

  BodyFrame *  findRelBodyFrame (string theBodyName,
                                 string theFrame,
                                 Pose * theTransform,
                                 list<Body *> & path);

  matrix<double> * getInertiaMatrix ( );
  void setInertiaMatrix (matrix<double> * value );
  void print();
  Body operator =(Body B);

private:
   matrix<double> * InertiaMatrix;
   list<BodyFrame *> bodyFrames;
}
```

Figure 5: *A summarized view of the The Body Object*

BodyFrames represent a body *frame of interest*. However, to be measurable and consistent, these frames were generally applied to surface locations at the center or edge of features (e.g. holes and corners). Though rarely a frame of interest and, therefore, strictly not necessary, the origin bodyframe held significance in any body as the point of reference for all other measures. Convenience, consistency, and accessibility drove the selection of a body's origin.

No single rule guides the orientation of frames relative to the origin. While some simply aligned the frame with the origin, others aligned one principal plane (i.e. either X-Y, Y-Z, or X-Z) with a component surface. In general, frame alignment fell to convenience, either for the measurer or for later constraint construction.

4.3.1 The BodyFrame Object

As summarized in Figure 4.3.1, the BodyFrame object inherits from *Pose*, adding a *Constraint* pointer and a pointer back to the parent *Body* object.

Though the bodyframe object can be constructed through conventional default-parameter methods, the preferred technique uses parameter files to drive the construction of all body objects, particularly bodyframes.

Figure 4.3.1's code segment plainly reveals the core data members of the object are pointers

```
class BodyFrame : public Pose
{
public:
     BodyFrame (BodyFrameParameters * bFP);
     BodyFrame ( );
     BodyFrame (std::string * aName);
     ~BodyFrame();

     Body * getOnBody ( );
     AFA_BFRAME_ERRCODE setOnBody (Body * value );

     Constraint * getConstraint ( );
     void setConstraint (Constraint * value );
     AFA_BFRAME_ERRCODE verifyConstraint();
     AFA_BFRAME_ERRCODE unConstrain ();

     BodyFrame * nextBodyFrame(matrix<double> * theTransform);

     void  print();
     BodyFrame & operator = (BodyFrame & B);
 private:
     Constraint * constraint;
     Body * onBody;
};
```

Figure 6: *A summarized view of the The BodyFrame Object*

to the constraint object and the parent body. Access methods set, get, and verify (confirm the existence of) these pointers.

When constrained, the `nextBodyFrame` method may be used to interrogate the partner constrained frame. This method 'crosses' the constraint to discover the other bodyframe sharing the constraint.

4.4 The Constraint

Constraints bind distinct body-bodyframe pairs through a (currently) time invariant homogeneous transform, $\mathbf{T}_k$. For the kth constraint: $\mathcal{C}_k = \langle s_c, \mathcal{F}_{\mathbf{fr}}, \mathcal{F}_{\mathbf{to}}, \mathbf{T}_k \rangle : \mathcal{F}_{\mathbf{fr}} \neq \mathcal{F}_{\mathbf{to}}$. To capture the transformation 'direction', a *Constraint* contains pointers to *From* and *To* bodyframes, $\mathcal{F}_{\mathbf{fr}}$ and $\mathcal{F}_{\mathbf{to}}$ respectively.

The constraint object ties bodies and bodyframes together into a single model. In effect, a properly composed constraint list *defines* the model uniquely. Careful design of both bodyframes and constraints can make the constraint design either very simple or very difficult. Poor selection of bodyframes can lead to counterintuitive constraint relationships and invite human error. Well designed frames lead to simple, intuitive constraints that can be easily verified..

Constraints mathematically (and, of course, geometrically) relate two bodyframes through a transform *from* one bodyframe *to* another. Currently, ModelServer uses a *static* or *time-invariant* transform. Dynamic modeling systems use both time-invariant and time-varying transforms to implement *joints*, such as hinges, sliders, and axles. Indeed, time-varying

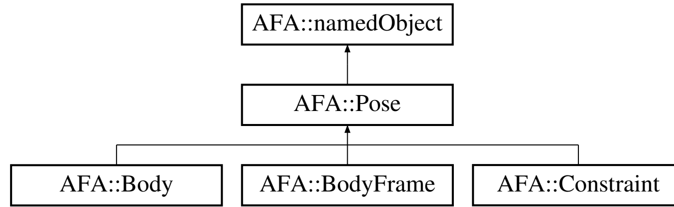


Figure 7: *Inheritance Diagram for Body, BodyFrame, and Constraint objects*

constraints can be complex functions of both geometry and time. In any case, DRDC uses simple fixed constraints to 'bolt' two bodies together.

Each transform has a *sense* of direction *from* one frame and *to* another. Traversing the constraint in the to-from direction reverses this sense, meaning the transform must be *inverted* mathematically to capture the correct sense of transformation. Thus any kinematic chain assembly must keep track of sense and perform any necessary inversions.

4.4.1 The Constraint Object

Just as for both *Body* and *BodyFrame* objects, the Constraint object of Figure 8 is normally invoked during ModelServer initialization using the ConstraintParameters constructor method.

The constraint object provides access to both *To* and *From* bodyframe pointers through the `AFA_BFRAME_ERRCODE` types. This method works in conjunction with the `BodyFrame::verifyConstraint` method to discover the sense of a given constraint when assembling transformations.

The `setTransform` method sets the constraint transform matrix.

The inheritance relationship between *Body*, *BodyFrame*, and *Constraint* objects appears in Figure 7.

4.5 The Model

The *Model* $\mathcal{M}$, $\mathbf{L}_{\mathcal{B}}$ constrained by $\mathbf{L}_{\mathcal{C}}$, resembles a cyclic directed graph of $\mathcal{B}_i$:$\mathcal{F}_j$ vertices with $\mathbf{A}_{ij}$ and $\mathbf{T}_k$ 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. $\forall \mathcal{C}_k \in \mathbf{L}_{\mathcal{C}}$, $\mathcal{F}_{\mathbf{fr}}$ and $\mathcal{F}_{\mathbf{to}}$ must exist. Constraints can exist only between real body frames.

```
class Constraint : public Pose
{
public:
    Constraint();
    Constraint(std::string * name,
                    BodyFrame * From,
                    BodyFrame *To,
                    matrix<double> * anHTransform);
    Constraint::Constraint(ConstraintParameters *cP,
                           BodyFrame * From,
                           BodyFrame * To );
    ~Constraint();
    static void Constraint::Constrain(ConstraintParameters *cP,
                                      BodyFrame * From,
                                      BodyFrame * To );
    static void Constraint::Constrain(std::string * name,
                                      BodyFrame * From,
                                      BodyFrame * To,
                                      matrix<double> * anHTransform);
    BodyFrame *  getBodyFrame (AFA_BFRAME_ERRCODE type );
    BodyFrame * copyBodyFrame (AFA_ERRCODE type ) ;
    AFA_ERRCODE setBodyFrame (AFA_ERRCODE type, BodyFrame  * aFrame );
    AFA_ERRCODE unSetBodyFrame (AFA_BFRAME_ERRCODE type);
    AFA_ERRCODE setTransform (matrix<double> * anHTransform );
private:
    BodyFrame *  fromBodyFrame;
    BodyFrame *  toBodyFrame;
    static int  numConstraints;
};
```

Figure 8: *A summarized view of the The Constraint Object*

4. $\forall \mathcal{B}_i$:$\mathcal{F}_j$, $\mathcal{F}_j$ if constrained, may have only one constraint which must exist. In this prototype, a bodyframe cannot be constrained to more than one other bodyframe.

When ModelServer launches, the process consults a common XML vehicle process configuration file that, in turn, identifies the location of the vehicles geometry XML file. ModelServer parses this latter file and instantiates the necessary *Body*, *BodyFrame*, and *Constraint* objects based on the file entries.

ModelServer assigns the 'root' body to the first body in the geometry file and keeps a running tally of whether new bodies are constrained to the root. When the geometry file parse begins, ModelServer loads all the bodies and sets a counter to the total body count. As constraints are loaded, ModelServer verifies the constraints and attempts to trace a path from the first or root body to each body in the list. If successful, the counter is decremented. A properly formed model constrains all the bodies to a single model tree. Currently, ModelServer declares trees not traceable to the root body as orphans and deletes them. Similarly, badly formed constraints are removed once parsing is completed. Future versions may preserve orphan trees as subassemblies.

4.6 The Search Engine: PlatformBase

ModelServer must 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 a pose representation based on a *frame of reference* and a *frame of interest*.

The frame of reference can be interpreted as a *base frame* in which the pose of other frames, the frame of interest, are to be expressed. Referring to figure 2, one might express the location of the Digiclops Imageplane in Raptor Frontbumpercenter coordinates. This would make the Digiclops Imageplane the frame of interest and the Raptor Frontbumpercenter the frame of reference.

Given the database structure described above, the process must start at a 'root' frame (simply a known entry point to the model – any bodyframe will do). The system must then:

- find the frame of reference, FOR.
- build the transform from the root frame to the frame of reference,${}^{root}\mathbf{T}_{FOR}$.
- find the frame of interest, FOI.
- build the transform from the root frame to the frame of reference${}^{root}\mathbf{T}_{FOI}$.
- determine the spanning transform from the frame of reference to the frame of interest,${}^{FOR}\mathbf{T}_{FOI}$ using:

$$ {}^{FOR}\mathbf{T}_{FOI} = {}^{FOR}\mathbf{T}^{-1}_{root}\,{}^{root}\mathbf{T}_{FOI} \qquad (11) $$

To find the frame of reference from a starting 'root' frame, ModelServer must perform a search of the model tree. Many search methods are possible, most variations on *Depth First* or *Breadth First* methods, using top down or bottom up directions (i.e. from the root to the frame of interest or vice versa).

4.6.1 Searching for frames

A typical tree is composed of 'parent' nodes, each having one or more 'children'. The 'root' node is simply the first parent of the tree from which all others spring. If two parents are equally displaced from the root, they are in the same 'generation'. While simple trees have no cycles i.e. children are not their own ancestors, the geometric model can have cycles and must be treated carefully.

Easy to implement, depth first searches do not guarantee the shortest path from root to target. In a depth first search, the algorithm interrogates a parent node and then its children before interrogating another parent of the same 'generation'. Since each child may be a parent, the algorithm will 'drill down' to the furthest generation before 'backing up' to interrogate a less distant generation.

Depth first methods need protection from cyclic paths or they will become locked in an endless loop, never finding the target node. The simplest protection is to maintain a list of the current 'ancestral line', adding children as the search burrows down through generations and removing them when the search 'backs up' a generation. By not interrogating children

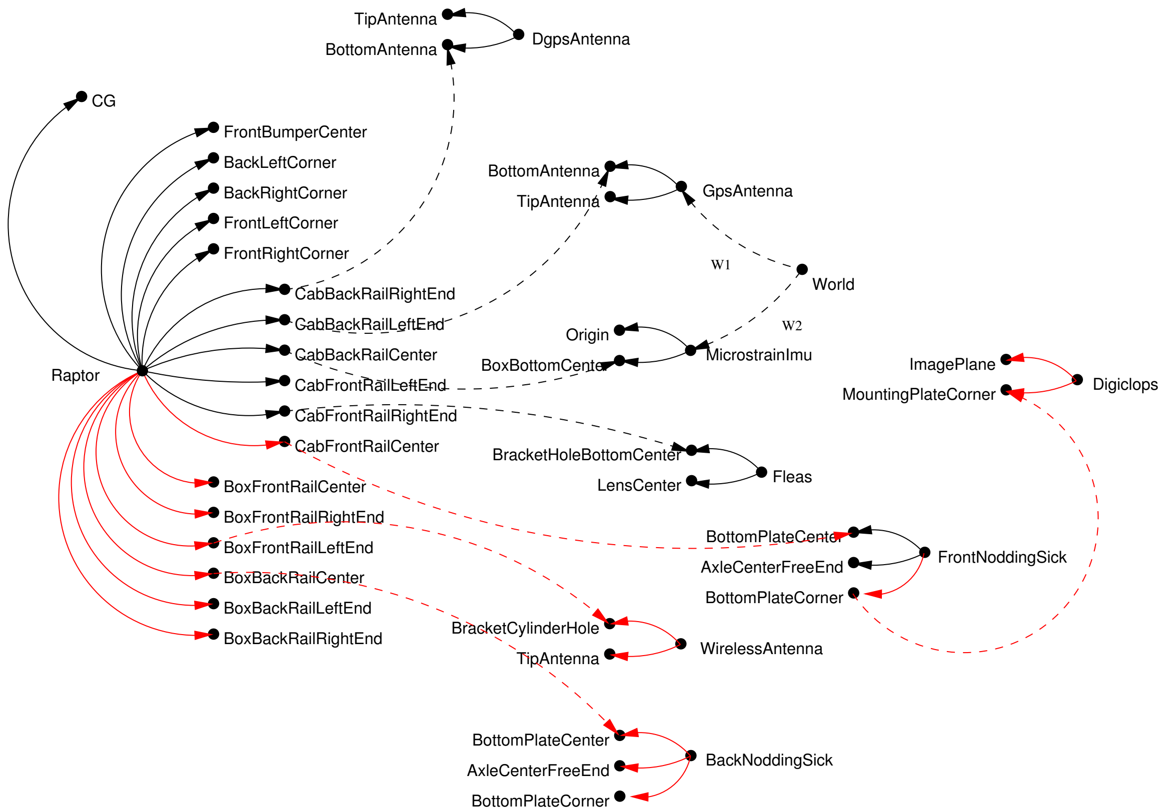


Figure 9: *A Depth first search example superimposed in red on DRDC's model tree. With the digiclops imageplane as the frame of interest and the raptor origin as the frame of reference, this figure assumes the search order is bottom to top.*

that appear in this list, cyclic loops can be avoided. Not surprisingly, the path length from root to target depends on the route taken during a depth first search and is a function of the child order. Since the tree is assembled arbitrarily, the path lengths cannot be guaranteed the shortest.

Breadth first searches find the shortest path and are somewhat more immune to cyclic paths, but are harder to implement. These methods will find solutions despite cyclic paths, but benefit from cyclic protection in any case. In a breadth first search, all parents of a given generation are interrogated before their children. This makes bookkeeping in the algorithm complex, but ensures that the path to the target spans the minimum number of generations. Cyclic paths slow, but do not interrupt, the search. Unfortunately, the search grows combinatorially as the number of nodes in each generation expands. Breadth first search underlies many important algorithms in AI, notably A*. For more information on breadth first methods consult Mackay [9].

4.6.2 ModelServer search

A simple depth-first search recursively constructs a transformation between reference and target vertexes. Starting at the $\mathcal{B}_R$:$\mathcal{F}_R$ vertex, the search compares $\mathcal{B}_i$:$\mathcal{F}_j$ in $\mathbf{L}_{i\mathcal{F}}$ against the $\mathcal{B}_I$:$\mathcal{F}_I$. 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 $\mathcal{B}_I$:$\mathcal{F}_I$ is found. The search then unwinds the recursion, building the transform product, ${}^{\mathcal{F}_R}T_{\mathcal{F}_I}$, according to the directed graph.

4.6.3 Building Transforms

Once the path from the root to target has been found, the transformation between these frames must be built. Both bodyframes and constraints possess transformations describing a displacement. For bodyframes this displacement is with respect to the body origin and for constraints it is with respect to the *From* bodyframe. As the search develops, these transformations must be collected with the sense preserved and, finally, a transformation product developed, of the form:

$$\mathbf{T}_{target}^{root} = \prod_{j=root}^{target} \mathbf{T}_j \tag{12}$$

where $\mathbf{T}_{target}^{root}$ is the total transformation and $\mathbf{T}_j$ is the jth transformation from root to target with sense preserved.

Clearly, from figure 9, an actual path in the tree is composed of both forward and inverted transforms. The following product describes the figure's final transformation between root, the raptor origin ($R:O$), and target, the digiclops imageplane ($D:I$):

$${}^{R:O}\mathbf{T}_{D:I} = {}^{R:O}\mathbf{T}_{R:CFRC}\,{}^{R:CFRC}\mathbf{C}_{F:BPCtr}\,{}^{F:BPCtr}\mathbf{T}_{F:O}^{-1}\,{}^{F:O}\mathbf{T}_{F:BPC}\,{}^{F:BPC}\mathbf{C}_{D:MPC}\,{}^{D:MPC}\mathbf{T}_{D:O}^{-1}\,{}^{D:O}\mathbf{T}_{D:I} \tag{13}$$

This transformation traces the path through constraints, $\mathbf{C}$, and the front nodding sick, F.

Given possible field measurement error of $\pm 1mm$, the cumulative error in any particular path could approach centimeter scales. Though concievably large, this was judged tolerable for the gross motions of the vehicle.

4.7 Localization

Localization determines the physical location of the vehicle within a World coordinate system through specialized sensing and filtering. For ground vehicles, typical sensing includes GPS, accelerometers, magnetometers, and odometry. Filtered together, these values can provide a consistent, optimal estimate of world location. Since many processes may draw from these sensor *services*, DRDC used a service based philosophy and advanced communications software to *publish* sensor data to *subscribing* processes.

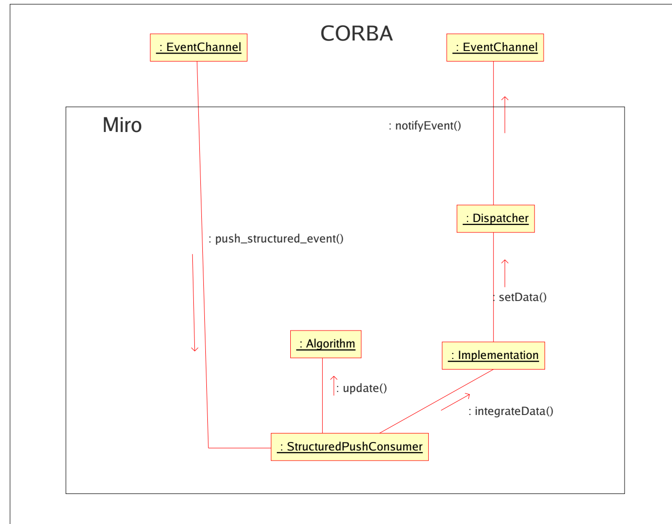


Figure 10: *Publish-Subscribe Server Design Pattern*

Encapsulated within ModelServer, a Kalman Filter based process subscribed to GPS, IMU, and odometry data to provide an estimate of the Raptor body. A future technical report will review this system in detail.

4.8 ModelServer Communication

ModelServer and its client processes are constructed using one of two basic design *patterns*[10, 11]. All software implemented using these design patterns have defined CORBA object interfaces, or in software parlance these design patterns yield *components*.

The following sections provide details on the implementation and use of these design patterns.

4.8.1 Subscribe-Publish Server

ModelServer exemplifies a *Subscribe-Publish server* design pattern, illustrated in the collaboration diagram shown of Figure 10. This pattern allows ModelServer to receive data events, process the data, and publish new events. ModelServer then becomes an independent component with interfaces defined by the CORBA objects for both event subscription and publication as well as the CORBA objects enabling client polling.

In ModelServer, an object derived from Miro's Structured PushConsumer class responds to every published GPS, IMU, and odometry event (e.g. published by SokkiaService, a GPS server) via the `push_structured_event()` method. In turn, this method calls the localization filter, effectively updating the location estimate with each new sensor event.

ModelServer publishes this new location estimate to any subscribing process. The `integrate_data()`

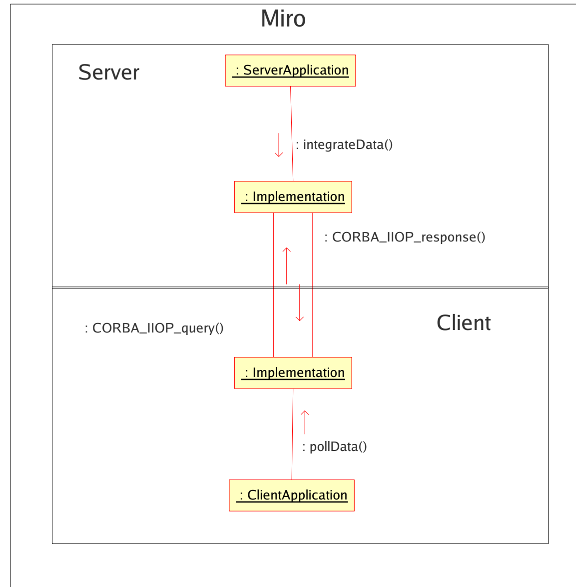


Figure 11: *Client Design Pattern*

method finally calls a *Implementation* object, that assembles the outbound event for final transmission on the event channel through another *Dispatcher* object. The implementation object also handles any client polling requests, each of which is described in the IDL interface for the implementation. ModelServer supports the Platform IDL interface and publishes *Pose* events corresponding to the *PoseTransformIDL* data structure.

4.8.2 Typical ModelServer Client

To poll ModelServer, a client design pattern follows a structure similar to Figure 11. This pattern allows a client process to poll data from a ModelServer. The client resolves a CORBA object reference using an object name lookup in the Naming Service. Polling itself uses the CORBA Internet Inter-ORB Protocol to request query and receive data from Miro server application.

Clients subscribing to ModelServer *Pose* events use a similar design pattern to ModelServer itself.

4.9 ModelServer IDLs

Generally speaking, CORBA software development for the ALS project fell into two camps: those that founded their algorithms on CORBA objects and those that translated native types into suitable object forms for transport. Since CORBA IDL structured types do not support pointers and the search algorithm is heavily pointer dependent, ModelServer fell into the latter group. Therefore, ModelServer translated between native and CORBA objects immediately after event reception and prior to transmission.

```
struct PoseTransformIDL
{
    TimeIDL time;
    double HTransform[4][4];
    double initialUTM[3];
    char poseValidFlag;
};
struct PoseVectorIDL
{
    TimeIDL time;
    double vec[AFA_POSE_STATE_DIM];
};
interface Pose
{
    PoseVectorIDL      getStateVector ();
    void               setStateVector (in PoseVectorIDL value);
    PoseTransformIDL   getHTransform ();
    PoseTransformIDL   getHTransformInv ();
    void               setHTransform (in PoseTransformIDL aTransform);
    sequence<PoseIDL>  getHistory(TimeIDL Start,
                                  TimeIDL End,
                                  float RateInHz);*/
};
```

Figure 12: *A summarized view of the The Pose CORBA Object*

ModelServer relies on the Platform IDL to describe output events. This IDL inherits type definitions from the Pose IDL.

4.9.1 Pose

The Pose CORBA object in Figure 12 provides type definitions for the Platform CORBA object in Figure 13, the most significant being `PoseTransformIDL`. This time stamped structure contains a pose validity flag to warn consumers of unreliable data, an initial UTM coordinate, and a homogeneous transform.

The validity flag provides consumers with an indication of the event's reliability. When the vehicle is stationary, GPS heading becomes erratic and contaminates the filtered solution. DRDC used this flag to warn clients that the pose might be questionable.

Since world coordinates are in UTM, vehicle position were expressed in very large meter displacements from a UTM zone origin. DRDC quickly found these large numbers problematic, often causing downstream numerical errors in the terrain map service. To prevent these errors, yet retain the true world coordinate, the *Pose* event stores a startup UTM position in `initialUTM`. `HTransform`'s position vector is measured from this point and updated as the vehicle moves.

4.9.2 Platform

The Platform CORBA object in Figure 13 provides the primary poll interface to ModelServer. Through this interface, clients can interrogate the model for:

- all available bodies,through `getBodyList()`.

```
typedef sequence<string> StringSequenceIDL;
interface Platform
{
   PoseTransformIDL  getTransformation (in string frameOfInterest,
                                        in string frameOfReference);
   StringSequenceIDL getBodyList ();
   StringSequenceIDL getBodyFrameList (in string BodyName);
};
```

Figure 13: *A summarized view of the The Platform CORBA Object*

- framelists for any body `getBodyFrameList(in string BodyName)`.
- transformations between any two body/bodyframe nodes through `getTransformation()`.

5 Operation

DRDC selected the Koyker Raptor as its Unmanned Ground Vehicle (UGV) demonstration platform, based on payload and drivetrain requirements. In each, a 25Hp gasoline engine powered a 4x4 hydrostatic drivetrain while generating an additional 1.5 kW of on-board power. The vehicles on-board intelligence enclosure housed power inverters, quad and dual Pentium servers, ethernet and USB hubs. The SpeedLan mesh networking router implements an 802.11b class wireless communications network. XJ Design of Ottawa, Canada converted the vehicles to drive-by-wire using an MPC555 microcontroller.

5.1 Proprioceptive Sensing

The sensing system collected raw position and orientation data from a GPS, an IMU, and odometry. The Sokkia GSR2600 GPS, combined with a Pacific Crest PDL RVR radio, supplied differentially corrected GPS positions with an accuracy of 2-5cm at an update rate of 4 Hz. Equipped with two Honeywell 1GT101DC hall effect sensors, on-board software decoded quadrature pulse trains into the displacement and direction of each front wheel, while a frequency-to-voltage divider transformed this signal into wheel speed. Using magnetometers, gyros, and accelerometers, the Microstrain 3DM-GX1 produced orientation and angular rates with respect to gravity and magnetic north.

5.2 Exteroceptive Sensing

DRDC and Scientific Instrumentation Ltd. developed a nodding mechanism for a 2-D SICK, creating a system that returns 3-D data. Communicating through an ethernet interface, the embedded RTEMS controller nods the laser from 2-90 degrees/sec with 0.072 degree resolution and 4cm accuracy over 1 - 30 metres.

DRDC also adopted Point Grey's Digiclops system to provide high speed range image streams. The Digiclops develops a disparity map between three camera image streams,

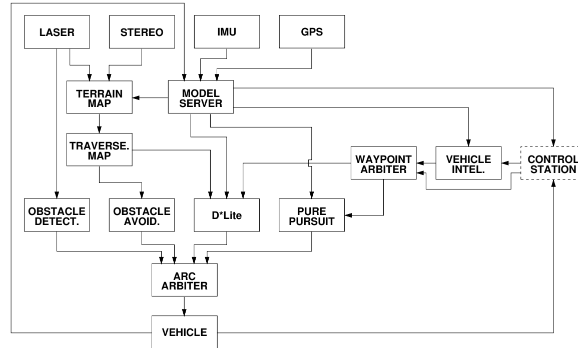


Figure 14: *The ALS Project Process diagram*

publishing the resulting 3D range image stream over an IEEE-1394 Firewire digital connection.

5.3 Miro Services

The vehicle control system depicted in figure 14, a network of intercommunicating CORBA-based processes, ran on two high level Linux servers. DRDC developed CORBA-based services that allowed both software services and hardware interfaces to publish and subscribe to CORBA events, as well as to respond to polling requests. The Raptor vehicle published *wheelevent* and *vehicle status data* and received vehicle control commands such as the forward velocity and the steering angle. The proprioceptive sensing published *GPS* and *Imu* events, while the exteroceptive sensing published *Stereo* and *Laser* data as generic 3D data events. A Terrain mapping service consumed *Pose* events and transmitted *Terrain* events to the Traversability Mapping system. In turn, an arbitrator consumed *Arcvote* events generated by a Pure Pursuit waypoint tracker, a traversability-based obstacle avoidance behaviour, and D* lite, a path planner. The arbitrator issued steering commands to the low level Raptor vehicle controller.

5.4 ModelServer Operation

5.4.1 Launch

Launched with other control processes through a launch script, ModelServer parses the geometry file, assembles the vehicle model, and establishes CORBA connections with GPS, IMU, and RaptorBase services. A late addition to this final step included the capture of IMU bias at startup. When using GPS heading, ModelServer flagged *Pose* events as *invalid* until the vehicle began moving. Once underway, the flag was set to *valid*, indicating consumers were safe to use *Pose* events. At 2Hz, GPS heading could not provide smooth orientation changes during vehicle turns. Under timed events using the IMU, ModelServer produced *Pose* events at upward of 10Hz, making for smoother turning results.

5.4.2 Steady State

In the steady state, ModelServer processes *GPS*, *Imu*, and *wheelevent* odometry events and poll requests for geometry data. The events trigger the localization processing to produce an optimal estimate of the vehicle's current position. This estimate, an homogeneous transform, was packaged with a validity flag and initial UTM coordinates, to emerge as a *Pose* event issued to all subscribing processes. Clients seeking the relative transformation between two bodyframes could poll ModelServer at any time after launch.

Again, when using GPS heading, ModelServer recaptured the IMU drift bias when ever the vehicle remained stationary for a user-defined time period. This recapture ensured that the drift rate remained small (typically 1-2 degrees/min).

6 An Example: the DRDC Raptor

This section describes the bodyframe configuration assembled for the September 2005 demonstration supporting photographs and explanations. This section illustrates the measurement strategy adopted by one DS in attempting to consistently measure frames on a variety of devices and the Raptor vehicle itself.

6.1 Raptor Body

The Raptor vehicle's body origin was assigned to be forward looking **x**-axis and **z**-axis up. The Raptor description includes all fixed, though technically movable, components such as the computer enclosure.

The vehicle coordinate system is attached to the Raptor at a point easily accessed for measurements, the rear bottom right corner of the green box shown on figure 15.

The CG frame represents the centre of gravity and is set approximately at centre of the vehicle at 1/3 of its height. The four corners and the centre of the front bumper of the vehicle are also set as frames. Figure 16 presents the points considered as corners and the one located on the front bumper.

Four rails are installed on the top of the vehicle to fix the sensors. Two rails are installed on the cabin roof and two on the box top. Each rail has three slots. BodyFrames were constructed on the top of the rail in the centre of the centre slot. Three points are defined: the right side, the centre and the left side of the rail. Figure 17 shows the three points on a rail. As can be seen on figure 15, the cabin front rail is tilted forward 12 degrees.

6.2 Nodding SICK Lasers

For the front and the back nodding SICK laser sensors, the measurements are identical. Two frames are involved in this body: the mounting plate and the nodding axle. As the bottom

of the plate is mounted centered on the rail's centre slot, the coordinate is (0.00,0.00,0.00). The axle is measured relative to the bottom centre of the plate. The end of the axle, opposite the motor, is used as measurement point. See figure 18 for visual explanation of the measurement points.

6.3 Digiclops Stereo Camera

The digiclops counts four measurements points: the three lens and the camera pivot. Figure 19 gives a visual representation of those points. The lens are measured from the pivot and the pivot is positioned compared to the front nodding SICK axle. The camera tilt angle will be fixed later.

In a later version, the Digiclops was given a fixed bent mount and attached to the stationary housing of the Nodding Sick Laser.

6.4 FLEA camera

The FLEA camera is installed on the right of the front SICK laser. Its bracket is fixed on the front slot of the cabin front rail. The important point to measure is the lens with respect to the bracket fixing. Figure 19 shows these points. The bracket angle is fixed, however, the lateral orientation of the camera can be changed.

6.5 GPS and DGPS antenna body

The GPS and the DGPS antennas have two important points to measure: the mounting holes on the bottom and the tip of the antenna. See figure 21 for illustration of the points. The bottom mesasurements is in the centre of the screw hole.

6.6 Microstrain 3DM-GX1

The Microstrain IMU sensor position is measured at the bottom of the box in the centre. This point is fixed in the centre of the rail slote. Figure 22 shows the back of the IMU with its measurement point. The location of the sensor can be seen in figure 21.

6.7 SpeedLan Wireless antenna body

This body has two bodyframes: the centre of the antenna tip and the bottom centre of the bracket hole. This bracket holds the cylindrical support of the antenna. Figure 23 presents the wireless antenna with the two bodyframes.

Figure 15: *World coordinate system on the Raptor body*

7 Discussion and Conclusion

DRDC held vehicle trials over the summer and fall of 2005. These trials exercised the geometry file format, geometry engine, and localization filter. The experience produced a number of important conclusions.

7.1 Geometry Server (Poll Mode)

In general, the XML files proved difficult to read and more difficult to safely assemble. Though man-readable, XML can prove difficult for typical users to understand. In particular, the size of these text files made cross checking name identifiers difficult. During file assembly, errors in syntax were common and revealed ModelServer's relatively weak error handling properties.

Inappropriate constraint construction posed a common problem for new users. Constraints limit the relative motion between two frames. New users incorrectly assumed multiple constraints could be attached to a single frame, causing faults in ModelServer or incomplete models. Nevertheless, once familiar with the syntax and file structure these files were easily modified in the field. Additional surveys were common, as predicted. Proving that estimating absolute positions and orientations would have been extremely inconvenient.

Given a correctly formed XML geometry file, ModelServer repeatedly and successfully responded to polled arbitrary frame-to-frame tranformation queries through the CORBA interface. For users, the poll operation proved simple and convenient to use since methods appeared local to the client.

Figure 16: *Corners of the vehicle*

7.2 Localization Server (Event Driven Operation)

The localization system's event driven communication mechanics functioned without error, though the localization filter itself proved difficult to tune. To improve performance the localization system, DRDC altered the structure from pure event-driven to a user defined event-driven or timer-driven operation. Despite these changes, the team reached the conclusion that pure kalman filtering was inadequate.

The filter optimally combines GPS, IMU, and odometry data. Each device has unique statistical measurement characteristics that must be tuned into the filter for an optimal estimate to emerge. However, each device also has characteristics that prevents continuous optimal results, described in detail below.

The GPS provides position, heading, and velocity estimates at approximately 2Hz. Unfortunately, the GPS only reflects accurate heading and velocity when on the move. This means for stationary vehicles any filtered solution drawing on GPS heading and velocity will diverge from reality.

Odometry produces consistent data during constant velocity, straight course runs. Turns and/or accelerations add error through wheel slip. Unfortunately, the Raptor possessed relatively poor quadrature encoders that compounded this error.

The Microstrain IMU adopted by DRDC for the Raptor trials proved to be a significant

Figure 17: *Rail measurement points*

source of error. Differences between Raptor vehicles and IMUs led the team to conclude that the IMUs could only be trusted in 'vertical gyro' mode. In this mode, the IMU ignores magnetometer data and the gyro alone drives the IMU's orientation estimate. However, this increased the vehicles reliance on GPS heading. Under these circumstances, the procedure to gather heading became a mixture of GPS heading under motion and estimated IMU heading while stationary.

Ultimately, however, further investigation revealed the IMUs could, indeed, be run in magnetometer mode after an appropriate 'hard iron calibration' – a procedure available in the final weeks prior to the September demonstration.

In any case, ModelServer became the focus of additional logic around the filter including startup IMU bias capture, and various strategies to isolate and correct sensor induced filter errors, including the construction of homogeneous transform based on raw GPS and IMU results. This 'raw' mode, ignoring odometry and including switching logic between IMU and GPS headings, proved adequate for basic obstacle avoidance. Since both Terrain and Traversability maps were 'egocentric' or centred on the vehicle and ignored global position estimates, cumulative orientation error rarely posed any problem for local vehicle navigation. However, this strategy was inadequate for D* path planning that requires coherent, smooth estimates of world vehicle position.

7.3 Conclusions and Future Work

Though the XML database and Localization subsystems proved awkward and difficult, ModelServer succeeded in providing field maintainable geometry to client processes. Labour saved in repeated resurveys was partially lost to the XML file's composition, often an error prone and confusing process for the uninitiated. In the short term, XML composition should

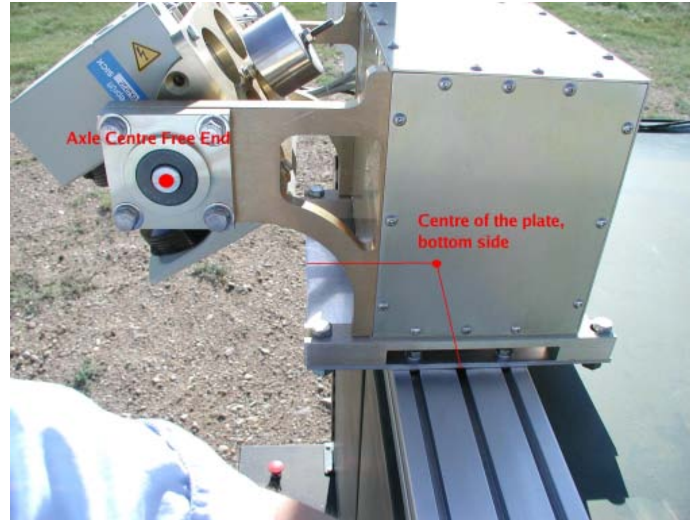


Figure 18: *Measurement points on the nodding SICK body*

be partially automated through GUI data entry and constraint checking. In the long term, an alternate file format should be adopted, such as a standardized solid modeling format. The search engine, though simple and effective, should be replaced with a more expressive dynamic modeler such as Gazebo or ODE[12, 4]. The current system captures position well, but fails to provide velocity or acceleration estimates that might improve future approaches to control, capture suspension effects, and, thus, an improved estimate of sensor positions.

The Kalman filter alone could not adequately compensate for nonlinear sensor behaviour without additional logic. This widely documented problem [13] became significant during the Raptor trials and emphasizes the necessity for an integrated solution binding logic to localization within a separate Miro service. The current elementary filtering strategy assumes GPS, IMU and odometry are sufficient estimates for position. In a multirobot environment, additional resources will be available including radar, visual ranging, third party observations, etc. The current service should be expanded to a generic localization service with the necessary discrete and filter logic to cope with these additional data streams.

ModelServer successfully fulfilled the role of an elementary geometry database engine. However, in the long term, changes to this concept must include database editting/visualization tools, a dynamic modeling engine, and a distinct localization service. Visual editors and constraint resolvers would remove human errors in XML file composition and verification.

Though the depth first search engine performed well for this first application, the long term solution incorporates dynamic modelers to maintain geometry, kinematics and dynamics. This step will add complexity to the server, having to draw on all internal motion sensors to accurately reflect vehicle configuration. However, this will lead to more accurate vehicle geometries, and provide a means for predictive vehicle control.

Figure 19: *Digiclops measurement points*

Figure 20: *FLEAS measurement points*

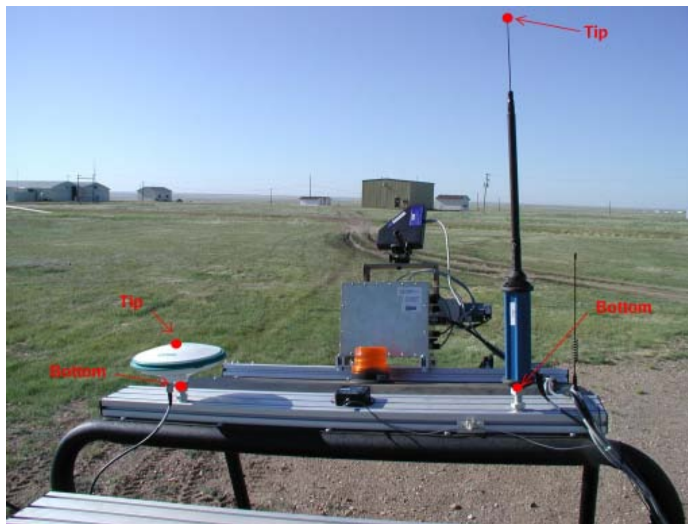


Figure 21: *GPS and DGPS antenna measurement points*

Figure 22: *Microstrain IMU measurement point*

Figure 23: *Wireless antenna measurement points*

References

[1] Paul, R.P. (1981), Robot Manipulators-Mathematics, Programming and Control, MIT Press.

[2] Yuan, J.S.C. (1988), Closed Loop Manipulator Control Using Quaternion Feedback, *IEEE Transactions on Robotics and Automation*, 4(4), 434–440.

[3] Monckton S., Toogood R. (1989), A Compact PROLOG Assembly Graph World Model., In *1989 ASME Winter Annual Meeting Dynamic Systems and Control, San Francisco, Dec. 1989.* Published.

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

[5] Goldstein, H. (1981), Classical Mechanics, Second Edition, Addison Wesley.

[6] Walker, M.W. and Orin, D.E. (1982), Efficient Dynamic Computer Simulation of Robotic Mechanisms, *Journal of Dynamic Systems, Measurement and Control*, 104, 205–211.

[7] (1999), RCS Posemath Library, Manufacturing Engineering Laboratory, National Institute of Standards and Technology.

[8] Walter J., Koch M. (2002), Boost: Basic Linear Algebra, www.boost.org.

[9] Mackay, David (2005), Path Planning with D*-Lite, (Technical Report TM2005-242), DRDC Suffield.

[10] Broten, G., Monckton, S., Giesbrecht, J., and Collier, J. (2006), Software Sysetms for Robotics, An Applied Research Perspective, *International Journal of Advanced Robotic Systems*.

[11] Broten, G., Moncton, S., Giesbrecht, J., and Collier, J. (2006), Software Engineering for Experimental Robotics, Ch. UxV Software Systems, An Applied Research Perspective, Springer Tracts on Advanced Robotics.

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

[13] Borenstein, J. and Feng, L. (1994), UMBmark - A Method for Measuring,Comparing, and Correcting Dead-Reckoning Errors in Mobile Robots, *University of Michigan*.

This page intentionally left blank.

Annex A: Position and Orientation

A.1 Orientation

Traditionally orientation is uniquely expressed as a 3×3 *orientation matrix*.

$$\mathbf{R} = \begin{bmatrix} \mathbf{n}_x & \mathbf{o}_x & \mathbf{a}_x \\ \mathbf{n}_y & \mathbf{o}_y & \mathbf{a}_y \\ \mathbf{n}_z & \mathbf{o}_z & \mathbf{a}_z \end{bmatrix} \tag{A.1}$$

Each column of the matrix is the projection of a rotated coordinate axis $< \mathbf{x}, \mathbf{y}, \mathbf{z} >$on a global coordinate system $< \mathbf{a}, \mathbf{b}, \mathbf{c} >$. The matrix is *orthonormal*, meaning all the columns have a vector magnitude of 1, or for any column vector $\mathbf{x}$:

$$|\mathbf{x}| = \sqrt{\mathbf{x} \cdot \mathbf{x}} = 1 \tag{A.2}$$

the inner product of any two columns is 0 (no axis projects on any other),

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{y} \cdot \mathbf{z} = \mathbf{z} \cdot \mathbf{x} = 0 \tag{A.3}$$

and the cross product of any two columns will produce a vector parallel to the third column (all axes are orthogonal)

$$\mathbf{x} \times \mathbf{y} = \mathbf{z}, \quad \mathbf{y} \times \mathbf{z} = \mathbf{x}, \quad \mathbf{x} \times \mathbf{z} = -\mathbf{y} \tag{A.4}$$

Since orientation is a relative measure, orientation matrices are also *transformation* matrices, meaning they can also be used as a mathematical operator to rotate other coordinate systems.

Unfortunately, differentiating the orientation matrix does not produce an angular velocity vector, causing consternation amongst roboticists. A 3×1 representation of orientation integrable from angular velocity is only possible (to date, anyway) through Quaternions, a 3×1 vector and a scalar.

A.2 Homogeneous Transform

The homogeneous transform [**?**] hybridizes a position vector,$\mathbf{p}$ with an orientation matrix.

$$\mathbf{T} = \begin{bmatrix} \mathbf{n}_x & \mathbf{o}_x & \mathbf{a}_x & \mathbf{p}_x \\ \mathbf{n}_y & \mathbf{o}_y & \mathbf{a}_y & \mathbf{p}_y \\ \mathbf{n}_z & \mathbf{o}_z & \mathbf{a}_z & \mathbf{p}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.5}$$

Originally used for *two* dimensional graphics and later expanded to three, the Homogeneous transform uses the bottom row for graphical scaling and, therefore, has no real purpose in robotics other than squaring the matrix. Note that while the above representation dominates literature, occasionally investigators assume a transposed form producing a seemingly reversed matrix algebra.

This page intentionally left blank.

Annex B: Geometry Parameter Files

Miro uses the `makeParam` utility build XML parsers. The 'source code' used by `makeParam` to construct the parser is itself written in XML.

To represent geometry on a vehicle, it is necessary to describe the position of many devices, some of which will be repositioned between trials. DRDC used an XML configuration file to record the location of these devices for run-time lookup.

The essential steps to building this Parameter parser are:

1. construct a *compile-time* `Parameter.xml` file that prescribes how the *run-time* parameter file should be read.
2. 'compile' this file through `makeParams` to generate`Parameters.h` and `Parameters.cpp` files.
3. build the application.
4. construct a run-time parameter 'somename'.xml file. This file holds the necessary parameters for the process.

At run time, parameters are read into a process through Miro's parameter facilities.

B.1 Compile-Time Parameter Files

The structure of the compile-time parameter file is XML. Consider the example XML snippet:

```
1 <!DOCTYPE xml SYSTEM "PolicyEditor_behaviour.dtd">
2 < config>
3     <config_global name="namespace" value="Test" />
4     <config_global name="include" value="miro/SvcParameters.h" />
5     <config_global name="Include" value="vector" />
6     <config_group name="test">
7       <config_item name="My" parent="Miro::Config" instance="true" >
8          <config_parameter name="TestVector" type="std::vector&lt;int&gt;" />
9       </config_item>
10    </config_group>
11 </config>
```

Parameters are grouped through opening and closing symbols. The config file opens with document type header, typically something like line 1 above.This is followed by the configuration data block, always within the `<config> - </config>` pair. Note that a 'slash' denotes the closure of an XML block.

Of course this configuration data configures the structure of C++ source files, so the next block is a series of statements for `namespace` declarations and `#include` files. To work with Miro's parameter system, `SvcParameters.h` must be included [1]. To use Sets or Vectors,

[1] SvcParameters contains a series of 'service' paramater types and ultimately includes `Miro::Config`

their headers must also be included. Since makeParams can be instructed to read *and* output arbitrary filenames, there is no need to combine parameter definitions into a single file. Modular Parameter definitions can be generated and included much like C++ headers (e.g. SvcParameters.h is machine generated).

The next block describes the data to be used at run-time and is contained within a <config_group> pair. The config group name on line 6 becomes the 'section' name in the run-time XML file.

Line 7 describes the config_item name, parent and instance.

name is the name of the Parameter class made and will be appended by "Parameters". So in this example, the compiled XML file will produce Parameters.h that contains a class called MyParameters, that will ultimately hold the parameters read at run-time. Dropping the name simply produces "Parameters" as the class name. This gives the user control over Parameter Class naming and permits the generation of multiple different parameter parsers within the same XML compile-time file.

parent identifies the parent class of the parameter class to be defined. To work in the Miro system, it must *ultimately* inherit from Miro::Config, hence the inclusion of SvcParameters.h.

instance is a boolean value (i.e. "True" or "False") that determines whether Miro should permit multiple instances of this parameter set ("False") or simply permit multiple use of a single instance("True").

Note that the config_item identifier of Line 11 is closed at the end of the parameter definitions in typical XML fashion in Line 13. Finally, the parameter definitions follow config_parameter with name, type (and others, see the Miro manual):

name is the name of the Parameter. If config_item name is the name of the class described by this file, config_parameter name is the name of the data member within the class. It seems necessary to *capitalize the leading letter* of this name in the compile and run-time references to this name, despite the fact that the leading letter of the C++ method variable is *always in small case*. More on this later.

type identifies the type of parameter, usually simple types (e.g. double or int) see the Miro manual for the full list. However it is possible to declare more elaborate structures as we shall see.

The parameter item, group and block sections are then closed as mentioned earlier.

B.2 Parsing Run-Time Parameter Files

How is a run-time file parsed? The following code block is one way:

```
1  Test::MyParameters * params = Test::MyParameters::instance();
2
3  try
4  {
5     Miro::ConfigDocument * config = new Miro::ConfigDocument(argc, argv,
"Config.xml");
6     config->setSection("test");
7     config->getParameters("Test", *params);
8     delete config;
9  }
10 catch (const Miro::CException& e)
11 {
12     cerr << "Miro exception: " << e << endl;
13     rc = 1;
14 }
```

Line 1 declares an instance of `MyParameters` using the `instance()` operator, available to us since we declared `instance= "True"` in the XML file. In the try block, we suck up the parameter file "Config.xml" into a `config` variable. Note that if we leave out the filename, the default file is the truly awkward filename `<hostname>.xml`. In line 7`params` is assigned to the parameters. With this assignment, the `config` object is no longer needed and is deleted. From this point on the variable `params` contains the parameters we want, for example:

```
1  for (i=0;i<3;i++) cout << params->testVector[i] << endl;
```

Note: all members of params have small case first letters...even though you must have leading Caps in the compile and run time files. As an aside, the `cout` expression above is always handled natively by Miro's parameter system, you can simply write `cout << *params` to achieve similar output.

B.3 Run-Time Parameter Files

After compiling, the XML run-time file can be built.

```
1<config>
2  <section name="test">
3    <parameter name="Test">
4      <parameter name="TestVector">
5        <parameter value="2" />
6        <parameter value="4" />
7        <parameter value="2" />
8        <parameter value="5" />
9      </parameter>
10    </parameter>
11  </section>
12</config>
```

The structure is very similar to the compile-time version, but carefully note the grouping of parameters with matching open/close statements. Note further, that on line 4 the end bracket has no slash, this means *the parameter expression is not terminated.* For both Vectors and Sets, the `<parameter name...>` entry is followed by as many `<parameter value=" "/>` expressions as desired. Note that, without exception, all parameters values are enclosed in quotes in the XML file.

B.4 Nested Data Structures

Now `int`, `double`, `std::string`, `vector`, and `set` are useful for most situations, but to structure parameters further, or have multidimensional parameters, nested structures can

be assembled through careful use of compile-time XML files. Consider the example below (the headers have been removed for space):

```
<config_item name="Matrix"  parent="Miro::Config" final="false" >
  <config_parameter name="Row1" type="std::string" />
  <config_parameter name="Row2" type="std::string" />
  <config_parameter name="Row3" type="std::string" />
</config_item>
<config_item name="BodyFrame"  parent="Miro::Config" final="false" >
  <config_parameter name="FrameName" type="std::string" />
  <config_parameter name="HTransform" type="MatrixParameters"/>
</config_item>
<config_item name="Body" final="false" parent="Miro::Config"  >
  <config_parameter name="BodyName" type="std::string" />
  <config_parameter name="HTransform" type="MatrixParameters"/>
  <config_parametername="Frames" type="std::vector&lt;BodyFrameParameters&gt;"/>
</config_item>
```

Three structures are described, a `Matrix`, a `BodyFrame`, and a `Body`. Some important points to note:

- all structures inherit from `Miro::Config`.
- when a type with a `config_item` of *name* becomes *name*Parameters when nested in other structures.
- nesting can be of arbitrary depth.

The greatest difficulty with nested parameters lies in the run-time file construction, which can easily become confusing as illustrated in the following example chunk:

```
<parameter> <!-- Body -->
  <parameter name="BodyName" value="Raptor"/> <!-- BodyName -->
  <parameter name="HTransform" > <!-- HTransform -->
       <parameter name="Row1" value="1.00 0.00 0.00 0.00"/>
       <parameter name="Row2" value="0.00 1.00 0.00 0.00"/>
       <parameter name="Row3" value="0.00 0.00 1.00 5" />
     </parameter> <!-- end of HTransform -->
   <parameter name="Frames" > <!-- FrameList -->
     <parameter> <!-- BodyFrame -->
     <parameter name="FrameName" value="CG"/> <!-- Frame -->
     <parameter name="HTransform" > <!-- HTransform -->
       <parameter name="Row1" value="1.00 0.00 0.00 0.00"/>
       <parameter name="Row2" value="0.00 1.00 0.00 0.00"/>
       <parameter name="Row3" value="0.00 0.00 1.00 0.00"/>
       </parameter> <!-- end of HTransform -->
     </parameter> <!-- end of BodyFrame  -->
     <parameter> <!-- BodyFrame -->
     <parameter name="FrameName" value="CabRailCentre"/> <!-- Frame -->
     <parameter name="HTransform" > <!-- HTransform -->
       <parameter name="Row1" value="1.00 0.00 0.00 0.00"/>
       <parameter name="Row2" value="0.00 1.00 0.00 0.00"/>
       <parameter name="Row3" value="0.00 0.00 1.00 5.00"/>
       </parameter> <!-- end of HTransform -->
     </parameter> <!-- end of BodyFrame  -->
   </parameter> <!-- end of FrameList  -->
</parameter> <!-- end of Body  -->
```

Comments, using the angle bracket notation `<!-- comment here -->` greatly clarify the situation. Still, care must be taken to ensure list types are properly opened (i.e. "no slash

on the parameter”) and closed. Note that the `Matrix` type uses `std::string` instead of vectors of doubles. A major problem with the Miro Parameter system is the dependence on quoted parameter values. This forces each vector or set element to be a single parameter entry and makes XML parameter files very large and difficult to read. In this case, the solution was to collapse rowvectors into a single string, parsed later in the C++ source.

This page intentionally left blank.

Annex C: A Raptor XML Body Model

The following file demonstrates the use of XML to describe vehicle geometry – in this case DRDC Raptor Vehicle circa August 2005.

C.1 Structure

The XML file has two parts:

- *BodyList*, simply a list of *Body* objects. Each *Body* object in the *BodyList* has
  - a *BodyName*, a *unique* string, and a
  - *FrameList* composed of a
    - *FrameName*, a string *unique* to this *Body*.
    - *Description*, a text description of the physical location of the frame on the body.
    - *HTransform*, the position and orientation of the frame expressed as a homogeneous tranform in *Body* coordinates.
- *ConstraintList* a list of *Constraint* objects. Each Constraint object is composed of:
  - *ConstraintName*, a *unique* string, typically relating the two constrained frames.
  - *From*, a *BodyFrame* expressed in `BodyName:FrameName` notation.
  - *To*, a *BodyFrame* expressed in `BodyName:FrameName` notation.
  - *HTransform*, the transform between the *From* frame and the *To* frame in *From* coordinates expressed as a homogeneous tranform in local or component coordinates.

The overall XML structure always has the form:

where the vertical ellipsis is filled with either body or constraint entries described below.

C.1.1 The Body XML entries

The XML file is built around a list of bodies. The example below typifies each body entry. The front nodding sick laser is represented by a body with four frames:

```
<parameter> <!-- Body  -->
    <parameter name="BodyName" value="FrontNoddingSICK"/>
    <parameter name="Frames" >
        <parameter>     <!-- BodyFrame -->
            <parameter name="FrameName" value="Origin"/>     <!-- Frame -->
            <parameter name="Description" value=""/>
            <parameter name="HTransform" >     <!-- HTransform -->
                <parameter name="Row1" value="1.00 0.00 0.00 0" />
                <parameter name="Row2" value="0.00 1.00 0.00 0" />
```

```
                <parameter name="Row3" value="0.00 0.00 1.00 0" />
            </parameter>     <!-- end of HTransform -->
        </parameter>     <!-- end of BodyFrame  -->
        <parameter>     <!-- BodyFrame -->
            <parameter name="FrameName" value="BottomPlateCenter"/>
            <parameter name="Description" value="  Measured on the bottom of the plate,
                          in the center."/>
            <parameter name="HTransform" >     <!-- HTransform -->
                <parameter name="Row1" value="1.00 0.00 0.00 0.00" />
                <parameter name="Row2" value="0.00 1.00 0.00 0.00" />
                <parameter name="Row3" value="0.00 0.00 1.00 0.00" />
            </parameter>     <!-- end of HTransform -->
        </parameter>     <!-- end of BodyFrame  -->
        <parameter>     <!-- BodyFrame -->
            <parameter name="FrameName" value="AxleCenterFreeEnd"/>
            <parameter name="Description" value="  Measured in the Center of the free
                          end of the axle, the one without motor."/>
            <parameter name="HTransform" >     <!-- HTransform - Undo the 12 angle of the cab
                           and add a +1 degree roll-->
                <parameter name="Row1" value=" 0.978 0.00   0.208  0.24" />
                <parameter name="Row2" value=" 0.00  1.00   0.000  0.15" />
                <parameter name="Row3" value="-0.208 0.00   0.978  0.18" />
            </parameter>     <!-- end of HTransform -->
        </parameter>     <!-- end of BodyFrame -->
        <parameter>     <!-- BodyFrame -->
            <parameter name="FrameName" value="BottomPlateCorner"/>
            <parameter name="Description" value="  Measured at the bottom corner
                          (right-back) of the SICK mounting plate.">
            <parameter name="HTransform" >     <!-- HTransform -->
                <parameter name="Row1" value="1.00 0.00 0.00 -0.133" />
                <parameter name="Row2" value="0.00 1.00 0.00 -0.135" />
                <parameter name="Row3" value="0.00 0.00 1.00 0.00" />
            </parameter>     <!-- end of HTransform -->
        </parameter>     <!-- end of BodyFrame -->
    </parameter>     <!-- end of FrameList -->
</parameter> <!-- end of Body -->
```

Similarly the Digiclops stereo camera is represented by a body of three frames:

```
<parameter> <!-- Body  -->
    <parameter name="BodyName" value="Digiclops"/>
    <parameter name="Frames" >
        <parameter>    <!-- BodyFrame -->
            <parameter name="FrameName" value="Origin"/>
            <parameter name="HTransform" >    <!-- HTransform -->
            <!-- Measured at the center of the pivot screw. Tilt angle 32.13deg referenced to a
           vertical line. Substract the cab front rail tilt angle. -->
            <parameter name="Row1" value="1 0.00 0.00 0.00" />
            <parameter name="Row2" value="0.00 1.00 0.00 0.00" />
            <parameter name="Row3" value="0.00 0.00 1 0.00" />
            </parameter>    <!-- end of HTransform -->
        </parameter>    <!-- end of BodyFrame  -->
        <parameter>    <!-- BodyFrame -->
        <parameter name="FrameName" value="ImagePlane"/>
        <parameter name="HTransform" >    <!-- HTransform Angle of 19.25 degrees -->
            <parameter name="Row1" value=" 0.9441  0.0000 -0.3297 0.093" />
            <parameter name="Row2" value=" 0.0000  1.0000 0.0000 0.072" />
            <parameter name="Row3" value=" 0.3297  0.0000 0.9441 0.093" />
            </parameter>    <!-- end of HTransform -->
        </parameter>    <!-- end of BodyFrame -->
        <parameter>    <!-- BodyFrame -->
```

```
            <parameter name="FrameName" value="MountingPlateCorner"/>
            <!-- Measured at the bottom corner (right-back) of the mounting plate. -->
            <parameter name="HTransform" >    <!-- HTransform -->
                <parameter name="Row1" value="1.00 0.00 0.00 0.00" />
                <parameter name="Row2" value="0.00 1.00 0.00 0.00" />
                <parameter name="Row3" value="0.00 0.00 1.00 0.00" />
            </parameter>    <!-- end of HTransform -->
        </parameter>    <!-- end of BodyFrame -->
    </parameter> <!-- end of FrameList -->
</parameter> <!-- end of Body -->
```

C.1.2 The Constraint XML entries

The constraint list can have many entries each similar to the following example. Here the Digiclops is fixed atop the Nodding Sick:

```
<parameter>     <!-- Constraint -->
    <parameter name="ConstraintName" value="DigiclopsFixed"/>     <!-- Constraint Name -->
    <parameter name="From" value="FrontNoddingSICK:BottomPlateCorner"/>     <!-- From frame -->
        <parameter name="To" value="Digiclops:MountingPlateCorner"/>     <!-- To Frame -->
    <parameter name="HTransform" >     <!-- Constraint Transform -->
             <parameter name="Row1" value="1.00 0.00 0.00 0.339" />
             <parameter name="Row2" value="0.00 1.00 0.00 0.007" />
             <parameter name="Row3" value="0.00 0.00 1.00 0.34" />
        </parameter>     <!-- end of Constraint Transform -->
</parameter>     <!-- end of Constraint  -->
```

Here the constraint fixes the relationship between the digiclops mounting plate corner to the front nodding sick's bottom plate corner. Note the constraint relates the two points through a translation and that the two frames are coaligned.

This page intentionally left blank.

Annex D: AFA::namedObject Class Reference

`#include <NamedObject.h>`

Inheritance diagram for AFA::namedObject::

Public Member Functions

- **namedObject** (std::string *aName)
- AFA_ERRCODE **setName** (std::string *aName)
- AFA_ERRCODE **print** ()
- **namedObject operator=** (**namedObject** B)
- std::string * **getName** ()
- char * **getCharName** ()
- AFA_ERRCODE **initialize** ()

Protected Attributes

- std::string **name**

Detailed Description

This object is an abstract superclass to many of the objects in this architecture. It provides common services such as (so far) unique names. Additional services could include logging and state reporting mechanisms.

Some things that should be added:

- Miro::Log should be a parent of this class.
- An exception class/structure should be declared in this header or in **afa_errcode.h**(p. ??).

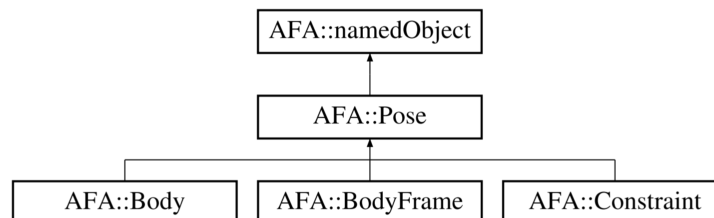


Figure D.1: *The Hierarchy for key geometry objects*

Constructor & Destructor Documentation

AFA::namedObject::namedObject (std::string * *aName*)

Constructors

Member Function Documentation

char * AFA::namedObject::getCharName ()

METHOD: **getCharName()**(p. 46) NOTES: returns the agent's unique string name, NIL otherwise.

string * AFA::namedObject::getName ()

METHOD: **getName()**(p. 46) NOTES: returns the agent's unique string name, NIL otherwise.

AFA_ERRCODE AFA::namedObject::setName (std::string * *aName*)

METHOD: setName(string aName) NOTES: sets the name attribute to aName.

Parameters

aName

Member Data Documentation

std::string AFA::namedObject::name `[protected]`

The a unique string identifier for the object.

The documentation for this class was generated from the following files:

- NamedObject.h
- NamedObject.cpp

Annex E: AFA::Body Class Reference

`#include <Body.h>`

Public Member Functions

- **Body** ()
- **Body** (std::string *aName)
- **Body** (BodyParameters *bP)
- ~**Body** ()
- void **updateState** (matrix< double > *position)
- AFA_BFRAME_ERRCODE **verifyBodyFrame** (BodyFrame *aFrame)
- void **addBodyFrame** (BodyFrame *aFrame)
- AFA_ERRCODE **delBodyFrame** (string *aName)
- BodyFrame * **getBodyFrame** (string *aName)
- BodyFrame * **getBodyFrame** (char *aCharName)
- list< BodyFrame * > * **getBodyFrameList** ()
- BodyFrame * **findRelBodyFrame** (string theBodyName, string theFrame, **Pose** *theTransform, list< **Body** * > &path)
- matrix< double > * **getInertiaMatrix** ()
- void **setInertiaMatrix** (matrix< double > *value)
- void **print** ()
- **Body operator=** (**Body** B)

Detailed Description

A body is any rigid element that can be monitored or controlled.

Constructor & Destructor Documentation

AFA::Body::Body ()

Constructors

AFA::Body::Body (std::string $*$ *aName*)

Allocate new memory for a 6x6 inertia matrix.

AFA::Body::Body (BodyParameters $*$ *bP*)

Allocate new memory for a 6x6 inertia matrix.

AFA::Body::~Body ()

Destroys the **Body**(p. 47) object and deletes all the BodyFrames. If BodyFrames are important to keep after the destruction of the **Body**(p. 47), *then maintain a copy of these frames elsewhere*!

Member Function Documentation

void AFA::Body::addBodyFrame (BodyFrame $*$ *aFrame*)

This method appends a BodyFrame to the BodyFrame list maintained by the **Body**(p. 47) object. A BodyFrame is a coordinate frame expressed in the local coordinates of the **Body**(p. 47) Object. In general, the only way to connect one body with another is through a BodyFrame and Constraint.

AFA_ERRCODE AFA::Body::delBodyFrame (string $*$ *aName*)

This method removes a BodyFrame from the BodyFrame list maintained by the **Body**(p. 47) object. A BodyFrame is a coordinate frame expressed in the local coordinates of the **Body**(p. 47) Object.

BodyFrame $*$ AFA::Body::findRelBodyFrame (string *theBodyName*, string *theFrame*, Pose $*$ *theTransform*, list< Body $*$ > & *path*)

This method returns the pointer to a named BodyFrame from somewhere in the **Body**(p. 47) Tree and the Homogeneous Transform from this body to the named. Deleting this pointer will lead to bad things!!. The method returns AFA_FAIL if the name cannot be found.

Returns	
AFA_FAIL	no matching body frame
bodyFrame $*$	a named bodyFrame

BodyFrame $*$ AFA::Body::getBodyFrame (char $*$ *aCharName*)

This method returns the pointer to a named BodyFrame. Deleting this pointer will lead to bad things!!. The method returns AFA_FAIL if the name cannot be found.

Returns	
bodyFrame ∗	a named bodyFrame
Throws	
AFA_NULL	no such body
AFA_FAIL	bad name

BodyFrame ∗ AFA::Body::getBodyFrame (string ∗ *aName*)

This method returns the the pointer to a named BodyFrame. Deleting this pointer will lead to bad things!!. The method returns AFA_FAIL if the name cannot be found.

Returns	
AFA_FAIL	no matching body frame
bodyFrame ∗	a named bodyFrame

list< BodyFrame ∗ > ∗ AFA::Body::getBodyFrameList ()

This method returns a pointer to the BodyFrame List. Deleting this pointer will lead to very very bad things!!. The method returns AFA_FAIL if the name cannot be found.

Returns	
AFA_NULL	no matching body frame
list<BodyFrame ∗ > ∗	a bodyFrame List pointer

matrix< double > ∗ AFA::Body::getInertiaMatrix ()

This method returns a pointer to freshly allocated memory containing a **copy** of the inertia matrix. It does NOT return a pointer to the actual Inertia matrix.

Allocate new memory for a 6x6 inertia matrix.

Copy InertiaMatrix into allocated memory.

Return a pointer to the allocated memory.

Body AFA::Body::operator= (Body *B*)

This method copies the contents of one body into another. Bodyframes are recursively copied NOT the pointers.

Returns	
AFA_NULL	no body frame
bodyFrame ∗	a named bodyFrame

void AFA::Body::print ()

This (somewhat crude) method prints the contents of the **Pose**(p. 53) object to stdout in the following format:

```
Name: <a pointer in hex> <a character string>
 Type: either static or dynamic
Quaternion: <a 4x1 vector follows>
HTransform: < a 4x4 matrix follows>
```

Reimplemented from **AFA::Pose** (p. 57).

void AFA::Body::setInertiaMatrix (matrix< double > * *value*)

This method sets a 6x6 Inertia Matrix to the **Body**(p. 47) object. An InertiaMatrix is characterized by a 6x6 boost matrix of doubles expressed in the local coordinates of the **Body**(p. 47) Object.

Returns	
AFA_DIMENSION	bad matrix dimension
AFA_OK *	successful assignment

void AFA::Body::updateState (matrix< double > * *position*)

Now there are a bunch of ways of doing this. We could

- query each body recursively and force it to dynamically compute its own state.
- OR build a static list of transformations through some function and evaluate it at run-time.
- OR we could perform a breadth-first evaluation of the entire tree starting at a 'world' frame.

At the moment, I am biased towards the latter. This would make the **updateState()**(p. 50) function simply a version of the setHTransform....at least, I think so anyway.

AFA_BFRAME_ERRCODE AFA::Body::verifyBodyFrame (BodyFrame * *aFrame*)

This method determines whether the BodyFrame is either is properly connected to a **Body**(p. 47) Object. If not, the BodyFrame is unrelated to the **Body**(p. 47) (and probably an illegitmate orphan).

Returns	
AFA_OK	this frame is attached to this body.
AFA_FAIL	this frame does not appear in the frame list (the BodyFrame is an orphan)
AFA_NULL	the frame pointer is NULL

The documentation for this class was generated from the following files:

- Body.h
- Body.cpp

This page intentionally left blank.

Annex F: AFA::Pose Class Reference

#include <Pose.h>

Public Member Functions

- **Pose** (std::string *aName)
- **Pose** (std::string *aName, MatrixParameters *matrixParams)
- **Pose** ()
- ~**Pose** ()
- int **evaluate** ()
- AFA_ERRCODE **localize** ()
- void **print** ()
- int **getType** ()
- void **setType** (int value)
- vector< double > * **getStateVector** ()
- void **setStateVector** (vector< double > *value)
- void **setStateVector** (double px, double py, double pz, double qs, double qx, double qy, double qz)
- matrix< double > * **getHTransform** ()
- PoseTransformIDL * **getPoseTransformIDL** ()
- matrix< double > * **getHTransformInv** ()
- PoseTransformIDL * **getInversePoseTransformIDL** ()
- void **setHTransform** (matrix< double > *aTransform)
- void **setHTransform** (PoseTransformIDL *aTransform)
- matrix< double > * **getRotationMatrix** ()
- matrix< double > * **Pose::getRotationMatrixInv** ()
- void **setRotationMatrix** (matrix< double > *aRotMatrix)
- void **setRotationMatrix** (PoseTransformIDL *aPoseTransformIDL)
- vector< double > * **getRollPitchYaw** ()

- void **setRollPitchYaw** (double roll, double pitch, double yaw)
- vector< double > * **getPositionVector** ()
- void **setPositionVector** (vector< double > *aVector)
- PM_HOMOGENEOUS * **getPM_HOMOGENEOUS** ()
- PM_QUATERNION * **getPM_QUATERNION** ()
- **Pose operator=** (**Pose** B)

Friends

- ostream & **operator**<< (ostream &os, **Pose** &p)

Detailed Description

The **Pose**(p. 53) object encapsulates the position and orientation of a rigid body. Normally this would be the product of a set of transformations from a base frame. However, these transformations must, sometimes, be estimated from observed sensor data. The meaning of the pose homogeneous transformation is application dependent. Within the **Body**(p. 47) derived class, the pose is the state of the object in world coordinates, in the BodyFrame derived classes, the pose is the location of the frame in **Body**(p. 47) Coordinates. In the Constraint object, the pose is a *transformation* and not a pose per se. In this case the pose is used to transform coordinates between BodyFrames.

Constructor & Destructor Documentation

AFA::Pose::Pose (std::string * *aName*)

Creates a **Pose**(p. 53) object of Name, *aName* . The homogeneous transformation is initialized to an identity rotation matrix and is positioned at the origin.\

Parameters: ***aName*** a pointer to a std::string. This should be a unique identifier of the object, but there are no checks on uniqueness (yet).

AFA::Pose::Pose (std::string * *aName*, MatrixParameters * *matrixParams*)

Creates a **Pose**(p. 53) object of Name, *aName* . The homogeneous transformation is initialized to values in a MatrixParameters structure.\

Parameters: ***aName*** a pointer to a std::string. This should be a unique identifier of the object, but there are no checks on uniqueness (yet).

AFA::Pose::Pose ()

Creates a **Pose**(p. 53) object of Name, *anon* . The homogeneous transformation is initialized to an identity rotation matrix and is positioned at the origin.

AFA::Pose::∼Pose ()

Destroys a **Pose**(p. 53) object including the contents of the pointers to the Homogeneous Transform and Quaternion.

Member Function Documentation

matrix< double > ∗ AFA::Pose::getHTransform ()

This method returns the 4x4 homogeneous transform with respect to an external coordinate system. The matrix includes an upper left 3x3 rotation matrix and a 3x1 position vector on the upper left. The lower row is typically unused, but is meant as a scaling row vector. Below is the basic structure:

$$\mathbf{H} = \begin{bmatrix} \mathbf{R} & \mathbf{p} \\ \mathbf{0} & 1 \end{bmatrix}$$

Returns	
matrix<double> ∗	0-indexed 4x4 matrix
AFA_NULL	

matrix< double > ∗ AFA::Pose::getHTransformInv ()

returns the inverse of the 4x4 homogeneous transform with respect to an external coordinate system.The matrix includes an upper left 3x3 rotation matrix and a 3x1 position vector on the upper left. The lower row is typically unused, but is meant as a scaling row vector. The example below reveals the basic structure.

$$\mathbf{H}^{-1} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T\mathbf{p} \\ \mathbf{0} & 1 \end{bmatrix}$$

PoseTransformIDL ∗ AFA::Pose::getInversePoseTransformIDL ()

returns the 4x4 homogeneous transform with respect to an external coordinate system

PM_HOMOGENEOUS ∗ AFA::Pose::getPM_HOMOGENEOUS ()

For those that wish to use the PoseMath utilities, this method returns a copy of the `Posemath` homogeneous transform representation. *Caution should be used in adopting this representation as its long term use is uncertain in the pose object.*

Returns	
PM_HOMOGENEOUS ∗	the Posemath transform

PM_QUATERNION * AFA::Pose::getPM_QUATERNION ()

For those that wish to use the PoseMath utilities, this method returns a copy of the `Posemath` quaternion representation. *Caution should be used in adopting this representation as its long term use is uncertain in the pose object.*

Returns	
PM_QUATERNION *	the Posemath quaternion

PoseTransformIDL * AFA::Pose::getPoseTransformIDL ()

returns the 4x4 homogeneous transform with respect to an external coordinate system

vector< double > * AFA::Pose::getPositionVector ()

Returns the X, Y, and Z components of the position vector,**p**, with respect to an external coordinate system.

Returns	
vector<double> *	0-indexed 3 vector
AFA_NULL	

vector< double > * AFA::Pose::getRollPitchYaw ()

This method returns the Roll, Pitch, and Yaw orientation angles with respect to an external coordinate system.

Returns	
vector<double> *	0-indexed 3 vector

matrix< double > * AFA::Pose::getRotationMatrix ()

The method returns a 3x3 rotation matrix, **R**, with respect to an external coordinate system.**R** is an orthonormal matrix meaning that the magnitude of the columns (or rows) is always 1.

vector< double > * AFA::Pose::getStateVector ()

This method returns a 7x1 'state vector' composed of the 'position vector' and the quaternion vector and magnitude. The vector appears as:

$$\mathbf{x} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ q_s \\ q_x \\ q_y \\ q_z \end{bmatrix}$$

int AFA::Pose::getType ()

This method gets the 'type', either static or dynamic, of the pose. The type is a means of discriminating static transformations from computed or sensor driven transformations. Currently this field is unused.

Pose AFA::Pose::operator= (Pose *B*)

This operator Copies **Pose**(p. 53) B into **Pose**(p. 53) A, by copying *the contents* of the HTransform and Quaternion structures into **Pose**(p. 53) A. This means that when the program leaves the scope of the operator it is free to delete B's HTransform and Quaternion without damaging **Pose**(p. 53) A.

Returns	
Pose(p. 53)	the RHS pose.

void AFA::Pose::print ()

This (somewhat crude) method prints the contents of the **Pose**(p. 53) object to stdout in the following format:

```
Name: <a pointer in hex> <a character string>
 Type: either static or dynamic
Quaternion: <a 4x1 vector follows>
HTransform: < a 4x4 matrix follows>
```

Reimplemented from **AFA::namedObject** (p. 45).

Reimplemented in **AFA::Body** (p. 50).

void AFA::Pose::setHTransform (PoseTransformIDL * *aPoseTransformIDL*)

sets the 4x4 homogeneous transform with respect to an external coordinate system

void AFA::Pose::setHTransform (matrix< double > * *aTransform*)

Sets the 4x4 homogeneous transform with respect to an external coordinate system.

Parameters: ***aTransform*** a 4 by 4 BOOST matrix of doubles.

void AFA::Pose::setPositionVector (vector< double > * *aVector*)

Returns the X, Y, and Z components of the position vector with respect to an external coordinate system.

Returns	
AFA_OK	success
AFA_NULL	

void AFA::Pose::setRollPitchYaw (double *roll*, double *pitch*, double *yaw*)

This method sets the Roll, Pitch, and Yaw orientation angles with respect to an external coordinate system.

Parameters: ***roll*** the rotation of the coordinate system about the x-axis

pitch the rotation of the coordinate system about the y-axis

yaw the rotation of the coordinate system about the z-axis.

Returns	
AFA_OK	success
AFA_NULL	

void AFA::Pose::setRotationMatrix (PoseTransformIDL * *aPoseTransformIDL*)

sets the 3x3 rotation matrix with respect to an external coordinate system

void AFA::Pose::setRotationMatrix (matrix< double > * *aRotMatrix*)

This method returns a 3x3 `rotation` , **R**, with respect to an external coordinate system.Of course the inverse is the transpose of the rotation matrix.

$$\mathbf{R}^{-1} = \mathbf{R}^T$$

Parameters: ***aTransform*** a 3 by 3 BOOST matrix of doubles.

Returns	
matrix<double> *	0-indexed 3x3 matrix
AFA_NULL	

void AFA::Pose::setStateVector (vector< double > * *value*)

This method assigns a set of doubles composed of the position vector' and the quaternion vector and magnitude in the format:

$$\mathbf{x} = p_x, p_y, p_z, q_s, q_x, q_y, q_z$$

Parameters: ***value*** is a AFA_POSE_STATE_DIM sized BOOST vector of doubles.

Returns	
AFA_OK	success
AFA_NULL	

void AFA::Pose::setType (int *value*)

This method sets the 'type', either static or dynamic, of the pose. The type is a means of discriminating static transformations from computed or sensor driven transformations. Currently this field is unused.

Parameters: ***value*** is either AFA_POSE_STATIC or AFA_POSE_DYNAMIC

Returns	
AFA_OK	success
AFA_NULL	

Friends And Related Function Documentation

ostream& operator< < (ostream & *os*, Pose & *p*) [friend]

writes the pose object to the output stream.

The documentation for this class was generated from the following files:

- Pose.h
- Pose.cpp

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
   PO Box 4000, Medicine Hat, AB, Canada T1A 8K6

2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable).

   UNCLASSIFIED

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

   A Prototype Vehicle Geometry Server

4. AUTHORS (last name, first name, middle initial)

   Monckton, S. ; Vincent, I. ; Broten, G.

5. DATE OF PUBLICATION (month and year of publication of document)

   January 2006

6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc).

   74

6b. NO. OF REFS (total cited in document)

   13

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 Report

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

   Defence R&D Canada – Suffield
   PO Box 4000, Medicine Hat, AB, Canada T1A 8K6

9a. PROJECT NO. (the applicable research and development project number under which the document was written. Specify whether project).

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.)

   DRDC Suffield TR 2005-240

10b. OTHER DOCUMENT NOs. (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).

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).

Defence R&D Canada is developing autonomy technologies for unmanned ground vehicles (UGV) and sensors (UGS); air (UAV) and subsea and surface (UUV and USV) vehicles – all operating together with minimal human oversight. To act autonomously, such devices must understand and change themselves and their surroundings through internal and external sensing and control. Autonomous systems must often model their physical structure as well as *localizing* or estimating their position in the world. Together, internal geometric models, localization, and range sensing produce an interpretation of the system's surroundings. For motion over and through complex environments, DRDC must further consider the use of dynamic models inside vehicle control loop to predict the necessary control forces for maneuver. These issues drove the adoption of modeling conventions compatible with systems that simulate mechanical dynamics including friction and closed kinematic chains. With sufficiently capable processing and high fidelity sensing, these systems can provide predictive control for even the most dynamic maneuvers. The first step, however, is to develop a regimented method of describing and storing system geometry. This report reviews the design of an elementary, prototype *geometric* model server and briefly describes the localization portion of the server.

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 not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

vehicle geometry, world representation, world model