ARS ADVANCED ROBOTIC SYSTEMS INTERNATIONAL

Acropolis: A Fast Protoyping Robotic Application

Vincent Zalzal[1], Raphael Gava[2], Sousso Kelouwani[3] and Paul Cohen[3]
[1] Matrox Electronic Systems Ltd.,1055 St. Regis Blvd.,Dorval, Québec, H9P 2T4, Canada
[2] Avisto Telecom, 32 Bd Philippon, 13004 Marseille, France
[3] Perception and Robotic Laboratory, École Polytechnique de Montréal, 2500 Chemin de Polytechnique, Montréal, Québec, H3T1J4, Canada
vzalzal@matrox.com, raphael.gava@polymtl.ca, sousso.kelouwani@polymtl.ca, paul.cohen@ai.polymtl.ca

Abstract: *Acropolis is an open source middleware robotic framework for fast software prototyping and reuse of program codes. It is made up of a core software and a collection of several extension modules called plugins. Each plugin encapsulates a specific functionality needed for robotic applications. To design a robot behavior, a circuit of the involved plugins is built with a graphical user interface. A high degree of decoupling between components and a graph-based representation allow the user to build complex robot behaviors with minimal need for code writing. In addition, the Acropolis core is hardware platform independent. Well-known design patterns and layered software architecture are its key features. Through the description of three applications, we illustrate some of its usability.*

Keywords: *robotic framework, robotic middleware architecture, fast prototyping, process triggering.*

1. Introduction

In order to build a robotic application, hardware and software integration issues need both to be addressed. Hardware integration deals with the physical components required for a robot to interact with its environment. The lack of standardization for such components (sensors and effectors) implies that efforts are required in order to make them work together. Integration mainly focuses on driver design and implementation that enable application software to control the various components.

Since, the complex behavior of a robot in real world requires a variety of heterogeneous pieces of software, integration efforts are also needed to bring all of them to work together. The availability of reusable and flexible tools for building robotic applications is, therefore, a cornerstone of robotic research.

Several approaches have been used to produce robotic development frameworks. Most of them use the middleware design approach. Platform independence, encapsulation, reusability, scalability, simplicity of integration, simplicity of extending existing functionalities and real-time performance are some of the design goals encountered in the robotic community.

In this paper, we present a new middleware called Acropolis for fast prototyping robotic applications. The rest of the paper is organised in four sections. Section 2 is related to previous work done in the robotic community to build development environments. In the third section, we present motivations that led to the development of Acropolis. Architectural aspects are presented in the fourth section. In section 5, we illustrate three practical examples of building applications with Acropolis.

2. Related Work

Much effort has been devoted to fulfill requirements by the robotic community in terms of robot software building tools. Most of them are open source softwares. Roughly, these softwares can be classified in two categories. Custom framework development platforms (usually designed from scratch) are part of the first category (Player, CLARAty, CARMEN, MARIE, etc.). In the second category, softwares are built on existing middleware such as CORBA (TAO CORBA, MIRO, etc.).

2.1. Some Custom Frameworks

One of the well-known middleware software is Player-Stage-Gazebo (Kranz, M. & al. 2006). The Player framework is designed as a robotic hardware abstraction layer. Its architecture relies on the client-server pattern. The Player server plays the role of hardware abstraction. To access a device functionality, a Player client uses a communication protocol built on TCP-IP (Transport Control Protocol - Internet Protocol). This abstraction mechanism hides unnecessary details when building robotic software tools.

CLARAty (Coupled-Layer Architecture for Robotic Autonomy) is designed using object-oriented principles (Urmson, C. & al. 2003). The emphasis is put on the development of reusable components and the definition of a set of simple standard interfaces. It has two separate layers: the first one provides decision tools for robot behavior whereas the second layer is related to functional aspects of the application.

International Journal of Advanced Robotic Systems, Vol. 6, No. 1 (2009)
ISSN 1729-8806, pp. 1-6

SIMOOT-RT is also an object-oriented framework (Becker L. B. & al. 2002). It aims at providing all required support for industrial automation applications: model edition, model validation, real-time code generation and deployment. A top-down design pattern is used within the framework.

CARMEN (Carnegie Mellon Navigation) is a robotic toolbox for navigation (Montemerlo, M. & al. 2003). Components within the framework communicate by using the mechanism of IPC (Inter Process Communication). The design principle used by CARMEN is the Model-View-Controller (Li, L. & al. 2007). Most of navigation sensors and robot hardware are supported.

MARIE (Mobile and Autonomous Robotics Integration Environment) is a middleware developed according to three design principles (Cote, C. & al. 2007). Mediation Design Pattern is the first one and is used to build the Mediator Interoperability Layer (MIL). The MIL allows the management of heterogeneous components within the distributed framework. The second design principle is the layered architecture of the software and the third design key feature of MARIE is the Communication Protocol Abstraction which allows applications to exchange data without knowing the details of the communication protocol.

Microsoft Robotics Developer Studio is an integrated software development framework for robotic applications (Tsai, W.T. & al. 2008). VPL (Visual Programming Language) is a graphical data-flow programming language used with this framework. It is based on service oriented architecture.

2.2. Some CORBA Based Frameworks

TAO CORBA and MIRO are development platforms based on CORBA and therefore are multi-platform distributed middlewares (Utz, H. & al. 2002).

Recently, the Babel framework has been released (Fernandez-Madrigal, J.-A. & al. 2008). It aims at providing support for several programming languages, operating systems and most commonly used communication middlewares (ACE-TAO, CORBA, etc.) In addition, it covers application lifecycle phases including design, implementation and testing.

3. Motivation for Acropolis

In order to support the development and the evolution of increasingly complex robotic applications, the following design objectives must be met (Zalzal, V. & al. 2005):

- Separation between algorithms and data access: it is often necessary to test more than one algorithm on the same data set. In order to allow easy algorithm interchanges without affecting data within the robotic framework, a loose coupling is required;
- For any given algorithm, it may be useful to allow asynchronously or synchronously triggered processes. Asynchronous triggering refers to a process initiated by a component not necessarily embedded within the algorithm. Algorithms that are data sensitive more than time sensitive could be triggered asynchronously, minimizing CPU (Central Processing Unit) cycle waste. For example, a filtering algorithm may react only if its input data set has been updated by another component rather than react at each clock pulse. Synchronous triggering occurs when processing is initiated by a clock signal. The support of both triggering modes allows to easily connect two or more algorithms that were not originally designed to be interconnected. Pre-processing and post-processing of data sets can also be easily and transparently performed;
- For fast prototyping applications, a user-friendly and intuitive building block tool is required;
- In order to allow Acropolis to interact with other robotic development platforms, an easy integration mechanism is required.

In addition to these objectives, flexibility, reusability, high degree of decoupling, robustness, simplicity to use and hierarchical structure are some of the other design objectives.

Most of the frameworks mentioned in the previous section, have some of the identified goals. None of them fulfills the complete list of these design goals. Moreover, most of those architectures allow only the synchronous processing mode.

The rationale behind Acropolis framework is somewhat comparable to VPL and OROCOS-SMARTSOFT goals (Bruyninckx, H. 2001). OROCOS aims at providing a real-time performance of motion execution through an event-driven architecture. SMARTSOFT has been built on the top of this software to provide a dynamic and flexible configuration for both control flow and data flow (Schlegel, C. 2006).

4. Acropolis Software Architecture

The Acropolis framework is an object-oriented software written in C++ and its architecture is mainly based on three components: the Communication Manager, the Core Component and the Extension Modules. Fig. 1 illustrates the architecture of the framework. Robotic applications are represented by a connected graph of extension modules (Acropolis plugins). The Core Component uses this graph to generate and control the robot behavior at runtime. Since the architecture is a distributed framework, the Communication Manager facilitates message exchanges between Acropolis instances. The Hardware Abstractor is used to hide physical component operation details to the other layers. The following sections present in detail each component of the architecture.

4.1. Acropolis Core Component

The Acropolis core is responsible for all tasks related to extension modules creation, management and communication. It uses design approaches such as model-view-controller, singleton, abstract factory and builder.

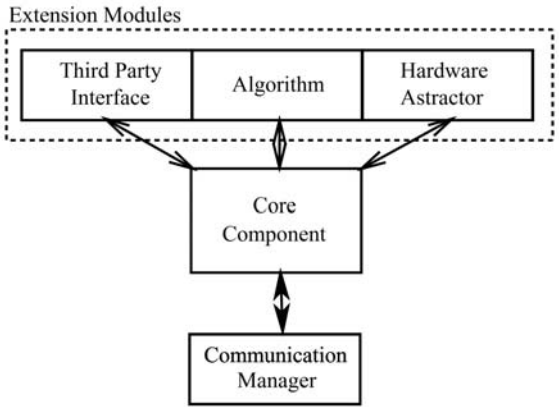

Fig. 1. Acropolis High Level Architecture

To provide unified interfaces between all extension modules, an abstraction factory pattern is used to create these modules. As mentioned before, a robotic application is represented in the Acropolis framework as a graph of extensions modules. This graph is built to reflect the desired behavior of the robot. Cyclic as well as non cyclic graphs are supported. Building a graph is similar to constructing a *Simulink* diagram (Dabney, J. & al. 1997). A user-friendly tool is used for this purpose. Hence, a non expert user could quickly build a robotic application with minimum software code writing. However, if the required extension modules are not available in the Acropolis library, the missing modules need to be designed and implemented before building a graph.

By using a mediator pattern design, the Acropolis core allows easy integration of extension modules from different robotic development frameworks (Player, CARMEN, MARIE, CLARAty, etc.).

4.2. Extension Modules

The framework promotes the use of extension modules (called plugins) in order to reduce the coupling between components. A plugin is a Dynamic Link Library (DLL) designed for a specific task or algorithm. Plugins exchange data through a standard set of interfaces. A plugin has three parts:

- Body: all processing by the plugin takes place in the body;
- Input Interfaces: these interfaces allow other plugin outputs to be connected. A plugin input may be designed to be a sensitive input or a normal input. When the input is sensitive, every change on this input triggers a processing task in the plugin body. This is the asynchronous triggering mechanism. Otherwise, an input may be designed as a normal input and the triggering of processing relies solely on clock events;
- Output Interfaces: they are data structures available to other plugins.

Two types of plugins are defined namely, standard and singleton. Standard plugins are extension modules designed to encapsulate specific algorithms or tasks. Singleton plugins are normally used as virtual object representations, state handlers or robot context handlers on which other extension modules base their actions. Thus, singletons act as information providers to other plugins of the graph via their public functions. This strategy makes it possible to avoid unnecessary duplication of data. Objects such as maps, physical and sensory characteristics of a robot could be represented by singletons. Functional relationships between plugins input and output are represented by a circuit. Acropolis provides an automated plugin design tool and a graph design toolkit (with Graphical User Interface).

Communication Manager

The distributed framework is implemented through a Communication Manager (CM). The adopted architecture is not a communication hub. Instead, a many-to-many approach is used to reduce the coupling due to message exchanges and to avoid the issue of single point of failure. The CM relies on the standard TCP-IP connection for message communication. To discover and join a group of CMs, a broadcast packet is sent over the network. Several CMs may exchange messages in full duplex mode and at any time. Moreover, a CM can join and leave the CM pool at runtime, on-the-fly, without disturbing the already ongoing message exchanges. Fig. 2 is an example of connected CMs in a distributed framework. Three CMs (Communication Managers A, B and C) are initially interconnected. Later, CM D decides to join the group of CMs. Now there are four CMs that can exchange messages.

The adopted architecture allows several Acropolis cores to run on different physical platforms. Each core is attached to one Communication Manager. Any extension module can send a message to other extension modules on any physical platform via this framework without dealing directly with communication concerns.

Fig. 3 is an example of distributed instances of Acropolis. Assume that the Acropolis instance III on physical platform 3 would like to send a message to the instance I on physical platform 2. Since instance III and instance I are respectively attached to CM C and CM B, message sharing is handled by these two CMs within the

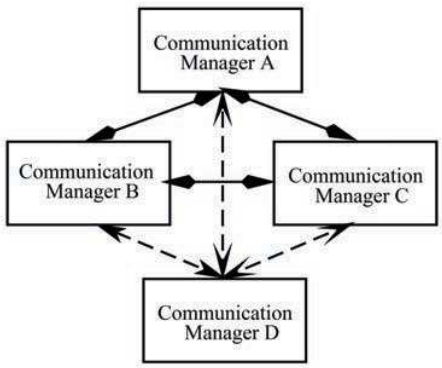

Fig. 2. Pool of Interconnected Communication Managers

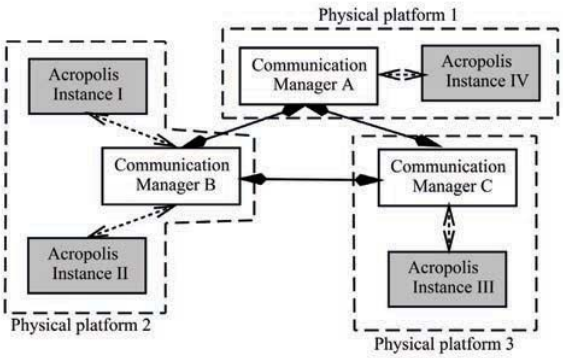

Fig. 3. Distributed Instances of the Acropolis Core

distributed framework. The first step in this process is for the sender to put its message in its CM message list. Each message is labelled with the originator and the destinator instance identifiers. The second step consists of routing the message by the originator CM through the distributed network of CMs.

5. Using Acropolis with Player

Player is a commonly used hardware abstraction layer in the robotic community. We demonstrate how Acropolis can be set to work with Player in order to specify:

- a simple robot move application;
- a navigation and localization task;
- an autonomous navigation task with obstacle avoidance.

However, before presenting the examples, we illustrate one of the key features of Acropolis: the triggering mode.

5.1. Algorithm Processing Triggering Mode

Two process triggering modes have been implemented in Acropolis. In this section a comparison is done between Player and Acropolis in order to put emphasis on one of the innovative feature presented in this paper.

Assume that a non expert user wishes to build an extension module that finds the inverse of a matrix by using the well-known Gauss-Jordan algorithm (Golub, G. & Loan V. 1983). With the Player framework, a driver may be created with the required algorithm.

The user is responsible for designing efficiently the driver so that it does not waste CPU cycles. This task is not necessarily easy to handle for that user because he needs to pay attention to the Player server workflow before starting the design. Even if he carefully designs the driver, CPU cycles are wasted since the process is invoked even in the absence of any change in the input data.

With Acropolis, since the Gauss-Jordan algorithm is data driven, a simple design decision is to select the inverse extension module input to be sensitive to the change in data. Thus, the inverse is performed only when a data change occurs at its input and no unnecessary processing is done within the extension module. Furthermore, there is no need for the user to have knowledge about Acropolis core operation.

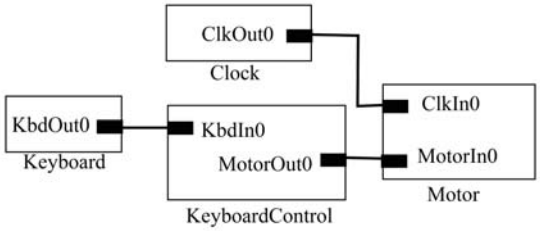

Fig. 4. Acropolis Circuit for a Move-with-Keyboard Application

5.2. Example 1: Simple Robot Motion

In this first example, it is assumed that a keyboard is used to control robot movements. A plugin for capturing keyboard characters is, therefore, needed. This plugin is named *Keyboard* and has an output interface called *GKeyboardData*. The plugin *Motor* is required to drive the motors of the plateform. This plugin has two input interfaces:

- *GClockData*: clock data;
- *GMotorCommand*: motor command.

Since, the *GKeyboardData* interface cannot be directly connected to the *Motor* plugin input, a plugin that performs format translation must be used. This plugin is *KeyboardControl*. The circuit for the first example is shown on Fig. 4.

Two hardware components are involved with this application. The keyboard is considered by Acropolis as one of the basic input devices and is directly handled by the core plugin. On the other hand, robot motors are more complex to handle. Instead of developing its own library to support these devices, Acropolis uses Player as a hardware abstraction layer. In the *Motor* plugin, a *Position2dProxy* from the Player library is created to handle interactions (sending motor command) with the Player server. The relationship between the Acropolis *Motor* plugin and the Player library is depicted on Fig. 5.

The formal description of the above circuit is an XML (eXtended Markup Language) document. Each graph file contains two main sections:

- the list of plugins required to build the circuit;
- the connections between plugins.

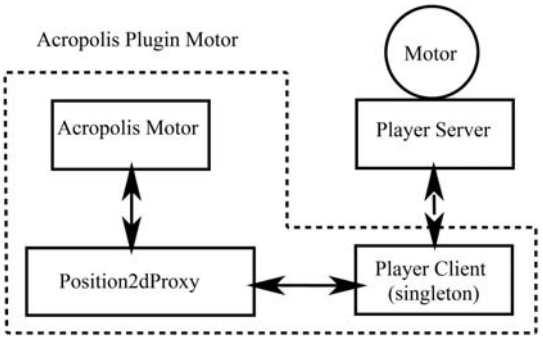

Fig. 5. Interconnection between the Acropolis Motor Plugin and Player

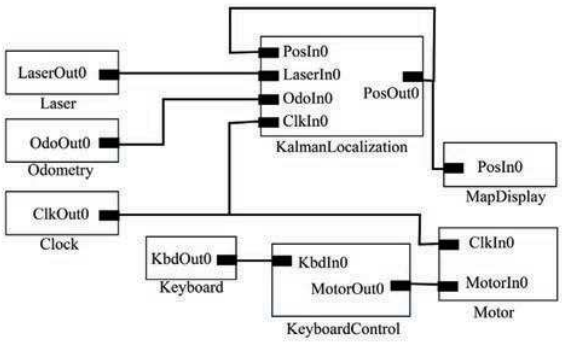

Fig. 6. Acropolis Circuit for Navigation and Localization

The Acropolis core uses the graph description to generate, at runtime, a robot behavior. It is easy to replace the keyboard control with a joystick control on the circuit, since this hardware is also handled by Acropolis. Furthermore, each plugin can be adapted by changing its options in the XML file.

5.3. Example 2: Navigation and Localization Task

The objective of this application is to locate a robot on a map and to display this map on a computer screen. The Acropolis graph that represents this example is shown on Fig. 6.

The robot is controlled with the keyboard, as in the previous example. As the robot is moving, its position and orientation are tracked by a Kalman filtering algorithm encapsulated in the *KalmanLocalization* plugin. To update the robot position, this plugin uses the *Laser*, *Odometry* and *Clock* plugins. By knowing its previous estimation (provided by the output of the *KalmanLocalization* plugin), the predicted robot position and orientation are computed, based upon this previous estimation and odometry data (provided by the *Odometry* plugin). The *Laser* plugin provides proximity range data that allows the correction of the predicted robot position and orientation. The output of the *KalmanLocalization* plugin is used by the *MapDisplay* plugin to display the robot estimated position and orientation.

This example shows another feature of Acropolis that is not usually explicitly available on other robotic frameworks: cyclic graph support. Several robotic applications need the use of loops and Acropolis can handle them as well.

5.4. Example 3: Autonomous navigation with obstacle avoidance

Instead of using the keyboard as control device, the robot autonomously tracks a path previously stored in a file and avoids any obstacle on its path (Gava, R. 2007). The circuit of this application is shown on Fig. 7.

The *TrajFromFile* plugin allows the robot to read the trajectory information from a file. To follow this trajectory, *TrajFollow* is used. It takes the current position combined with the next checked point from the assigned trajectory in order to compute the next position and orientation that the robot should reach. On the way to

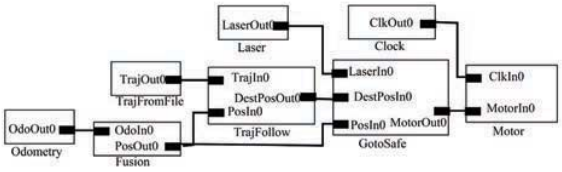

Fig. 7. Autonomous Navigation with Acropolis

reach this point, the robot detects obstacles with a laser range finder through the use of the *Laser* plugin. The *GotoSafe* plugin allows the robot to avoid the obstacle and reach safely the desired point on the trajectory.

Complex applications have also been designed and implemented with the Acropolis framework:

- robot team control and management for search and rescue applications (Gava, R. 2007);
- mutual localization of mobile platforms based on omnidirectional vision and Kalman filtering (Zalzal, V. & al. 2006).

6. Conclusions

We have presented a robotic middleware framework for fast prototyping applications. This framework uses a generic approach for data access and algorithms implementation. It was designed to be easily configurable and portable. Acropolis uses the concept of component graph to represent robot behavior. This approach has two advantages. The first one is the great flexibility since more than one robot behavior can be tested by changing only the components connections. The second benefit is related to the high degree of decoupling of layers and components. Furthermore, each extension module can be designed to be sensitive to change on its inputs. Despite the fact that Acropolis is still in development, several complex robotic applications were designed and successfully demonstrated with this framework.

7. Acknowledgments

This work was supported by the National Science and Engineering Council of Canada, under grant RGPIN3890-01. The authors wish to thank Patrice Boucher and Alexandre Fortin of the Perception and Robotics Laboratory for their help in implementation and testing of plugins and applications.

8. References

Becker, L. B. & al. (2002). SIMOO-RT - An Object-oriented Framework for the Development of Real-time Industrial Automation Systems, *IEEE Transation on Robotic and Automation*, Vol. 18, No. 4, pp. 421-430.

Bruyninckx, H. (2001). Open robot control software: The OROCOS project, *Proceedings - IEEE International Conference on Robotics and Automation*, Vol. 3, pp. 2523 - 2528, Seoul, 2001.

Bruyninckx, H. (2003). The real-time motion control core of the OROCOS project, *Proceedings - IEEE*

International Conference on Robotics and Automation, Vol. 2, pp. 2766 - 2771, Taipei (Taiwan), 2003.

Cote, C. & al. (2007) Robotic software integration using MARIE, *International Journal of Advanced Robotic Systems*, Vol. 3, No. 1, pp. 55-60.

Dabney, J. & al. (1997). *Mastering SIMULINK*, Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.

Fernandez-Madrigal, J.-A. & al. (2008). A software engineering approach for the development of heterogeneous robotic applications, *Robotics and Computer Integrated Manufacturing*, Vol. 24, No. 1, pp. 150 – 66.

Gava, R. (2007). Development of a hybrid decision architecture for an heterogeneous multi-robots team, *Master thesis*, Electrical Department, École Polytechnique Montréal.

Golub, G. & Loan V. (1983). *Matrix Computations*, The Johns Hopkins University Press, Batimore, Maryland, USA, 1983.

Kranz, M. & al. (2006). A Player/Stage System for Context-Aware Intelligent Environments, *Proceedings of UbiSys'06, System Support for Ubiquitous Computing Workshop*, Orange County California. September 17-21.

Li, L. & al. (2007). R-Flow: an extensible XML based multimodal dialog system architecture, *IEEE 9th Workshop on Multimedia Signal Processing*, Crete, Greece, 2007.

Miah, M.S. & al. (2007). Autonomous dead-reckoning mobile robot navigation system with intelligent precision calibration, *IEEE Instrumentation and Measurement Technology Conference Proceedings*, pp. 2179 – 2183.

Montemerlo, M. & al. (2003). Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit, *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2436-2341, Las Vegas, 2003.

Schlegel, C. (2006). Communication Patterns as Key Towards Component-Based Robotics, *International Journal of Advanced Robotic Systems*, Vol. 3, No. 1.

Toby, H.J. Collett & al. (2005). Player 2.0: Toward a Practical Robot Programming Framework, *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*, Sydney, Australia, December 2005.

Tsai, W.T. & al. (2008). A Collaborative Service-Oriented Simulation Framework with Microsoft Robotic Studio, *41st Annual Simulation Symposium*, pp. 263-270, Ottawa, Canada.

Urmson, C. & al. (2003). A Generic Framework for Robotic Navigation, *Proceedings of the IEEE Aerospace Conference*, Montana, March 2003.

Utz, H. & al. (2002). Miro - middleware for mobile robot applications, *IEEE Transactions on Robotics and Automation*, Vol. 18, No 4, pp. 493-497.

Zalzal, V. & al. (2005). Mutual localization of mobile robotic platforms using Kalman filtering, *Master thesis*, Electrical Department, École Polytechnique Montréal.

Zalzal, V. & al. (2006). Mutual localization of mobile robotic platforms using Kalman filtering, *IECON 2006. 32nd Annual Conference on IEEE Industrial Electronics*, pp. 4540 - 4545, Paris, France.