

SANCTA: An Ada 2005 General-Purpose Architecture for Mobile Robotics Research

Alejandro R. Mosteo and Luis Montano

*Departamento de Informática e Ingeniería de Sistemas. Universidad de Zaragoza
Instituto de Investigación en Ingeniería de Aragón, I3A
c/ María de Luna 1, 50018 Zaragoza, Spain
{amosteo,montano}@unizar.es*

Abstract. We present SANCTA, a flexible control architecture for multi-robot teams. It is fully written in Ada 2005, except for the reuse of some C libraries. In this paper we highlight the architectural elements of our implementation and also present our experiences using the cutting-edge 2005 implementation from GNAT, through its GPL 2005 and 2006 iterations. We expect to exemplify the kind of advantages and challenges that developers can find in using the new Ada 2005 features. Since this architecture makes use of a wide range of Ada capabilities, from low level hardware interaction to graphical user interfaces, we believe it is a good example of a successful mid-size project using Ada 2005 in academy.

Key words: multi-robot, control architecture, Ada 2005, GNAT GPL

1 Introduction

Mobile robot teams is a flourishing research topic motivated by problems like planetary exploration, autonomous mapping, search and rescue applications in urban settings, automated transportation and so on. The use of multiple robots has benefits like higher throughput, higher reliability due to redundancy, and the possibility of tackling problems that a single robot could not.

These multi-robot teams present new challenges: coordination needs among robots, long-term mission planning, communication with human operators in manageable ways that are not overwhelming in feedback nor too demanding on inputs, for example.

Such broad necessities, ranging from embedded software to distributed computation and interaction, make Ada a natural language choice due to its ingrained principles in software safety, reusability, portability and maintainability. Hence we want to disseminate our experience with Ada development in this field.

We firstly introduce our research domain in Sect. 2. Next, we show the architectural elements of our implementation in Sect. 3. Ada implementation insights on relevant parts of it are then presented in Sect. 4. Section 5 is devoted to bindings to foreign language libraries. Finally, we comment in Sect. 6 our experience using the new features of Ada 2005.

2 Mobile Robot Teams and Task Allocation

The SANCTA architecture receives its name from *Simulated Annealing Constrained Task Allocation*, since these are the first novel features that it implemented [1] in the field of multi-robot task allocation. Simulated annealing [2, 3] is a probabilistic tool useful for optimization problems with large solution spaces, able to escape local minima and, with enough running time, find good solutions or even the optimal one. It imitates the metallurgical process of annealing, where controlled heating and cooling is used to reach configurations with desired properties. By analogy, the simulated annealing uses a temperature variable to balance greedy and randomized behavior in the optimization process.

Semiautonomous robot teams introduce the critical need of deciding how to assign tasks to robots in order to optimize performance. Ideally, this would be a completely automatic operation without the need of human intervention beyond high-level description of missions. Our research focuses in this kind of problem, aiming for optimality and autonomy of decision in mobile robot teams intended to be deployed in hostile environments.

A typical sequence of events would be: robots are deployed awaiting orders; a human operator relays a high-level description of mission to the robot team; the mission is refined into low-level tasks executable by robots; robots negotiate the ownership of tasks and needs of cooperation; execution of tasks starts; on-line replanning and task allocation is performed when new environment data requires it. All these operations should be preferably done in a distributed manner with good scalability properties. Examples of missions are: build a map of an unknown environment; locate a vehicle in a parking lot; patrol an area in such a way that an intruder could not remain undetected; visit a series of spots of interest to take ground samples.

Several requirements arise in this context: communication among robots themselves and human supervisors; sharing and synchronization of data; ability to cope with faulty networking and lost robots; algorithms for task allocation, path planning, obstacle avoidance; graphical visualization of status; etc.

3 Architecture

SANCTA was born initially as just the testing implementation of our research. Soon it was evident that, better than have stand-alone programs for each element of research, a reusable solution was more desirable. We could have integrated our code in one of the many existing robot control libraries. However, most are oriented to single robot control, so we preferred to create a new Ada layer on top of one of these, namely Player [4]. This way we retain more control over what we could do, while minimizing dependencies on other software. We chose Ada because we believe in its advantageous properties over other popular languages. Also, we had previous exposure to it, since it is actively used for teaching in our university.

There are two principles that are paramount in the design of our system:

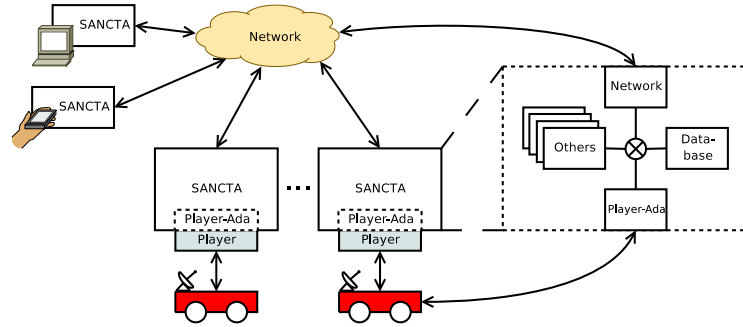


Fig. 1. Key elements in the SANCTA architecture. Components are interconnected as determined by the configuration file. Some important components are explicitly shown. Grayed blocks are C code corresponding to the Player library, accessed through the Player-Ada binding.

- *Simplicity*: Albeit always desirable, a tool that may be potentially used by several researchers must be kept as simple as possible. In SANCTA this is achieved reducing the elements to its simplest expression. The approach taken is modularity, and the building blocks are *components*. Each component operates as a black box, except for its defined inputs and outputs. Several components can thus interact without having to know any internals of each other. Table 1 shows a selection of some relevant components available for our architecture.
- *Flexibility*: We wanted to impose minimum constraints on component programmers, so they must conform to a very simple interface. Additionally, some basic execution scheduling is offered. Namely, programmers can choose between periodic, asynchronous and event-driven execution (or a combination of these). Direct component interaction is achieved connecting matching inputs and outputs, allowing the construction of powerful flows of data. Fig. 2 exemplifies this chaining construction, that corresponds to a *pipes and filters* flow pattern [5]. Indirect interaction is also available through a *shared repository* approach.

Notable components of our architecture are shown in Fig. 1. Nodes can be robots, laptops showing a GUI¹ or in general any connected computing device. A node configuration is defined and constructed with just a collection of component instances and their interconnections. To simplify configuration, some components are predefined and always created automatically, though this could be easily changed if necessary due to hardware constraints.

3.1 Configuration Definitions

A node configuration is maintained in a simple XML file. This configuration is read (using XmlAda [6]) at start-up by the SANCTA executable launched in each

¹ Graphical User Interface

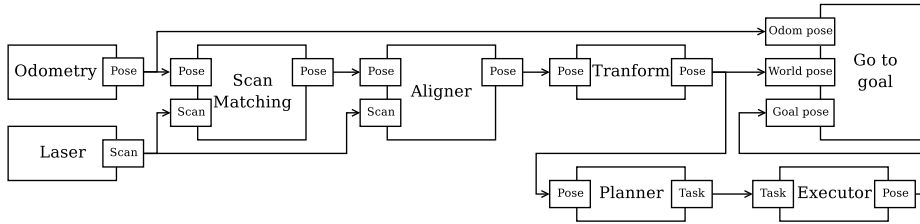


Fig. 2. A node configuration example. *Odometry* and *Laser* are sensor readers. *Scan matching*, *Aligner* and *Transform* are processing algorithms. *Planner* is a task allocation component (i.e. *Annealer* or *Bidder*). *Executor* determines the robot commands necessary to perform a task, while *Go to goal* directly commands the robot hardware.

robot or device. Fig. 2 shows a visualization for a superset of the configuration snippet shown in listing 1.1.

Listing 1.1. Snippet of configuration file

```

<config_example>
  <agent id="Ari" kind="robot">
    <component name="odometry" enabled="true">
      <provides data="pose"
               as="odometric_pose"
               type="types.pose"/>
    </component>
    <component name="scan_matching" enabled="true">
      <requires data="pose"
               as="odometric_pose"
               type="types.pose"/>
      <provides data="pose"
               as="corrected_pose"
               type="types.pose"/>
    </component>
  </agent>
</config_example>

```

Several relevant characteristics are apparent:

- The top-level element names the configuration.
- Each node (robot or not) is defined in an “agent” element. Assuming that every node has synchronized copies of the configuration files, this allows to configure the whole team in one file. Each node can have a different configuration according to its capabilities.
- Each component instance is defined in a “component” element. Attributes are used to configure specific component behavior.

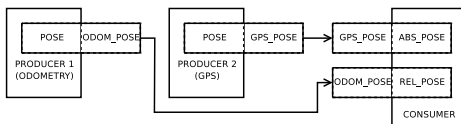


Fig. 3. *data/as* attributes in configuration files correspond to internal/external names of a same value.

- Component inputs/outputs are respectively defined in “requires”/“provides” elements.
- “data” attributes are predefined and detailed in the component documentation. “as” attributes are an arbitrary identifier (Fig. 3). “data” names need not to be unique among components, but external names can only appear once as output in each node (unlimited as inputs). Matching external names create an output \rightarrow input connection. “type” attributes are informative and optional, but data types of connected outputs/inputs should match.
- Unused outputs can be omitted to avoid clutter.

Following a *factory method* pattern [10], what the configuration does is to instance a collection of components, each one providing and requiring a series of data that the user must connect appropriately. Components can be data filters, hardware drivers, network modules or any necessary algorithm. Some predefined components naturally act as *singletons* and are automatically created:

- *Network*: Provides messaging among nodes. A straightforward implementation uses UDP packets. A predefined abstract interface exists for network components, so it can be seamlessly replaced with another network/messaging layer.
- *Local database*: Used for communications among the rest of components.
- *Player-Ada*: Interfaces with our robot hardware (Pioneer3 platforms). Again, an abstract interface exists so a change of platform should have an isolated impact.

3.2 Execution Modes

We have advanced that components have two ways of receiving execution time that are supported by the architecture, plus a third that is self-management. More precisely, the modes are:

- *Periodic*: When created, the component receives an initial call. Each component can specify, at that moment and at each subsequent run slice, the next time to be invoked. There is a single Ada task managing these calls, so this is the mode indicated for soft real-time execution of brief-lasting calls.
- *Event-driven (or synchronous)*: A component can subscribe to database records of interest, using an *observer* pattern [5, 11], and will be notified each time the monitored ones change. Event chains can be constructed this way. It is the responsibility of the user to avoid infinite recursive calls, although some cases can be detected.
- *Asynchronous*: Computationally intensive components should create its own Ada task. This could also be done when hard real-time scheduling is desired (although, in our case, this has not been necessary for our research until date). The predefined components are tasking safe, so they can be used in a typical client-server approach for *rate monotonic analysis* (RMA) or similar techniques.

Table 1. Some available components, ordered in decreasing abstraction level.

Component	Inputs	Outputs	Explanation
Global database	Network link	Database	Globally accesible database for data sharing among nodes with built-in replication
Local database	None	Database	Local data storage and sharing among components
Annealer	Pose, Database	Task allocation	Computes a best effort task allocation for a multi-robot team using simulated annealing techniques [1]
Bidder	Pose, Network link	Task allocation	Bids on auctioned tasks using market-based techniques [7]
Map	Pose, Laser scan	Map	Builds an environment grid map
Network	None	Network link	Provides messaging between nodes
Transformer	Pose	Pose	Transforms a pose in robot coordinates to world coordinates
Scan matching	Pose, Laser scan	Pose	Uses MBICP [8] to improve odometry using laser readings
Aligner	Pose, Laser scan	Pose	Corrects the pose angle when the robot is in an environment with known principal orientations (i.e. orthogonal walls)
GPS	Pose	Pose	Combines an odometry pose with GPS readings to produce global localization
Executor	Task list	Robot commands	Determines the robot actions needed to perform a task
Go to goal	Pose, Goal	None	Issues Player [4] movement calls
GUI relay	Robot state, Network link	None	Relays information to remote GUIs
Logger	Any	None	Logs some input to disk
Watchdog	Any	None	Aborts execution if input remains unchanged for some time
Player_Ada	Robot commands	Robot sensors	Proxy to robot hardware [9]

4 Implementation Details

We will now explore some details of Ada code, highlighting fragments related to the explained aspects of the architecture.

4.1 Component Specification

Listing 1.2 shows that components, as advanced, are really simple. They are just a code container, with an added facility for periodic soft real-time calls, that are managed by a priority queue sorted by time.

Listing 1.2. Root component type.

```
type Object is abstract tagged limited null record;
-- Root type.

type Object_Access is access all Object'Class;
type Object_Creator is access
  function (Config : in Xml.Node)
    return Object_Access;
-- For a factory approach.

procedure Run (This : in out Object;
              Next : out Ada.Calendar.Time)
is abstract;
-- Invoked on component creation.
-- Next should be filled with the next call time.
```

A creation function must be registered (for example on component body elaboration) following a typical factory pattern. This function receives the XML element corresponding to the component configuration. The factory will then be used to create the components as specified in the configuration file.

4.2 Local Database Specification

The database component exports a type that is actually a protected wrapper over a standard `Ada.Containers.Indefinite_Sorted_Maps.Map`. Keys are strings, and stored values are descendants of a root type, so the database can store any variety of non-limited objects. Thus, asynchronous data sharing is simply a matter of using the functions in listing 1.3:

Listing 1.3. Some database accessing functions.

```
function Get (This : in Database;
             Key : in Object_Key)
  return Object_Data'Class;

procedure Put (This : in out Database;
              Key : in Object_Key;
              Value : in Object_Data'Class);
```

A simple view renaming allows for comfortable use of typed data.

Now we present the facilities for event registration, following an *observer* pattern. Components can obtain automatically from the configuration file the equivalence between its internal keys (the “data” attribute) and the particular name bound to it (the “as” attribute), in fact allowing for the connection of inputs/outputs:

The user must define a descendant type of *Key_Listener*, and register an instance in the database with the *Listen* subprogram (listing 1.4). Every time the key receives an update, the *On_Key_Stored* dispatching procedure will be invoked for each listener. Using a tagged type instead of a function pointer has the advantage that the listener can store any necessary contextual data right within it.

Listing 1.4. Database data observing.

```

type Key_Listener is
  abstract tagged limited null record;
type Key_Listener_Access is
  access all Key_Listener 'Class;

procedure On_Key_Stored
  (This : in out Key_Listener;
   Key  : in      Object_Key;
   Value : in      Object_Data 'Class)
is abstract;

procedure Listen
  (This      : in out Database;
   Key       : in      Object_Key;
   Listener  : not null Key_Listener_Access);

```

We provide also a default empty component with a listener member and convenient functions for registration (listing 1.5). This could have been done with the new language interfaces, but we encountered compiler problems.

A typical event-driven component will register itself on creation to the inputs he needs. This is particularly useful for filtering components. In Fig. 2 is depicted a typical chain of components used to correct the robot pose.

Listing 1.5. Component with embedded listener.

```

type Listener (Parent : access Base_Component) is
  limited new Key_Listener with null record;
  -- Base_Component has been declared in public part.

type Base_Component is limited new Component.Object with record
  Listener : aliased Listener (Base_Component 'Access);
end record;

not overriding
procedure Subscribe (This : in out Base_Component;
                    Key   :      String);

```

4.3 Advantageous Ada Features

Our simulated annealing algorithm does not require high precision, but takes advantage of exact repeatability of results when operating on task costs. This repeatability is a key factor to make our algorithm $O(n \log n)$ instead of $O(n^2)$. This can be naturally handled in Ada by means of fixed types instead of floating point ones or integer workarounds.

5 New Bindings

We believe in code reuse, and sometimes a little effort spent in creating a binding to foreign libraries pays over reimplementing the wheel in every favorite language. As part of our work we have implemented some bindings of interest for the Ada community.

5.1 Player-Ada Binding

Our choice for low level robot interaction is the Player library [4]. Its advantages are many: portability across robot platforms, open source nature, dedicated volunteers and developers, acknowledgment within the robotic community, and availability of a quite realistic multi-robot simulator to say a few. Many client libraries existed for it but unfortunately not one written in Ada. Our binding is called Player-Ada [9]. The C library uses object oriented (OO) techniques and so our Ada 95 binding uses the support for OO programming in the form of tagged and controlled types.

Our binding uses some auxiliary C functions to improve the portability of the Ada part in the event of changes in the underlying C library. For example, some constant sized structures that are opaque to the user may likely change in size between versions.

Listing 1.6. C auxiliary file in Player-Ada.

```
#include "libplayerc/playerc.h"

size_t get_device_t_size ()
{
    return sizeof (playerc_device_t);
}
```

Listing 1.7. Player-Ada specification detail.

```
function Get_Device_T_Size return Interfaces.C.size_t;
pragma Import (C, Get_Device_T_Size);

type Pad_Array is array (Positive range <>) of
    System.Storage_Elements.Storage_Element;
pragma Pack (Pad_Array);

type Position2d is record — A new Device_T
    Reserved : Pad_Array (1 .. Natural (Get_Device_T_Size));
    — Note the function call to Get_Device_T_Size.
    — Other fields here.
end record; pragma Convention (C, Position2d);
```

The construct shown in listings 1.6 and 1.7 abstracts the size of the C structure from the Ada binding, thanks to the Ada type elaboration capabilities. A single recompilation will take care of this aspect in the event of changing the Player version. The same technique has been used for other relevant constants.

Table 2. Our perception of some Ada 2005 features.

Feature	Frequency of use	Usefulness
Containers	All the time	Very useful
Limited with	Not used	–
Private with	Frequently	Useful
Anonymous access types	Regular	Useful
Box initializer	Rarely	Somewhat useful
Task termination	All the time	Useful
Overriding marks	All the time	Very useful
Interfaces	Abandoned	Useful
Dot notation	All the time	Very useful

We believe this is valuable over a pure Ada binding that is specially useful when the C library is still evolving. We have been able to successfully use our binding with three different Player versions over the time doing only minor modifications, and indeed this technique has proven useful in at least one version change.

5.2 Concorde Binding

Concorde [12] is an advanced TSP² solver, free for research purposes. Only suited for the single-traveler with symmetric costs problem, with graph transformations it can be used for asymmetric and multi-robot cases. We have implemented a thick binding for it and also the appropriate transformation functions. With our binding, asymmetric, open and MTSP³ problems of moderate size⁴ can be solved.

6 Ada 2005 Experiences

Our attempts at using new Ada 2005 features is a mix of success and failure. Some features are ready for use, whereas other are either partially or totally unimplemented. We must note that the reported failures herein may have been caused in part or in full by a lack of complete understanding in our part, despite our best effort to keep up to date [13, 14]. See Table 2 for a quick overview of our subjective perceptions.

– Standard containers.

This is a very fruitful addition in our experience: Most of our data structures that require collections use the new standard containers. Before its availability we used a mixture of open source libraries but we have completely

² Traveling Salesman Problem

³ Multiple Traveling Salesmen Problem.

⁴ This is due to the use of large numbers in the transformations involved and the use of integers in the C library. We have solved problems involving several hundred cities.

replaced them in favor of the standard containers. We have not found any critical bug in the last two compiler releases.

The safety requirements of the containers, materialized in checks in subprograms with *access-to-subprogram* parameters (`Query_Element` and `Update_Element`, among others), will prevent the improper use of cursors and modification of elements (*tampering with cursors* and *tampering with elements* are the RM expressions). These checks have aided us in preventing some errors, in circumstances that would have been difficult to debug. Once again, the safety emphasis of Ada mandates these checks where other languages would prefer to omit them for a marginal gain in efficiency. In our case, these checks have been proved a valuable feature.

Another related 2005 addition that nicely blends here is the new access scope rules. Declaring trivial subprograms in place for simple processing of container elements has worked flawlessly and contributes to maintain code locality without added burden. See listing 1.8 for an example.

Listing 1.8. Containers and access subprograms.

```
declare
  procedure Do_Something (Key :      Key_Type;
                        Val : in out Element_Type) is
  begin
    — Operate in-place with the Val element
  end Do_Something;
begin
  Some_Map.Update_Element (Some_Map.Find (Some_Key));
end;
```

– **Limited with.**

We were keen to use this new feature but unfortunately the GNAT GPL 2005 version was not behaving as we expected, so we ended not using it. Superficial tests in the most recent 2006 version show that either the problems with this feature have been solved or our understanding of the operations we could do with the incomplete view of a type is more thorough.

We regret not having more up to date information and use cases to report on this major addition to the language.

– **Private with.**

This feature works flawlessly in our experience and helps in adding clarity to specifications.

– **Anonymous access types.**

This is another interesting feature that has contributed to remove some now unnecessary access type declarations. However, functions returning anonymous access types was not completely implemented to the best of our knowledge so we did not use it. The latest patches we have seen submitted to gcc improving support for this feature are [15].

– **Box notation for default initializations.**

This seemingly minor feature is a nice addition to the language, aiding in the simplification of some code: Although intended to allow for some impossible

initializations in Ada 95 related to limited types, we have found it useful for defaults in subprogram specifications. We have identified however a dangerous possibility: it is now easy to indulge in it for the initialization of record components, so neglecting the utility of specifying every record member to get a compilation error if some new components are added.

We reported a bug when used it for controlled types initialization in the 2005 version, which was already corrected for the 2006 version. Due to this bug, some calls to Initialize in controlled members of a type would be omitted. We use it now regularly without problems.

– **Ada.Task.Termination.**

This new package is a nice addition allowing a standard way of tracking unusual task terminations. We have incorporated it into our architecture and it has proved to be useful for extra debugging awareness.

– **Overriding notation.**

Another feature not likely to have a great impact, but which we use and find very useful in aiding to document specifications. It also has helped in preventing a few bugs, identifying some unintended overriding. Being an optional feature, it is in the hand of the programmer to remember to use it. A specific warning devoted to its omission could be a fine addition for the object-oriented practitioners.

– **Interfaces.**

Curiously, we had more success with them in the 2005 version than in the current one. In the former, we were able to use synchronized interfaces to specify network layers, and actually implemented a UDP based component. Other, simpler interfaces (not synchronized), were also used. However, we were unable to get them working in the 2006 version due to compiler crashes, so we had to revert to abstract tagged types (listing 1.9). This was fortunately very little work, since we had made few use of multiple inheritance, and we could simply go back to abstract types in most cases, or move to a vertical hereditary design in the remaining ones.

Listing 1.9. Problems with interfaces

```
type Object is abstract tagged limited null record;  
— Root type as currently defined with GPL 2006.  
  
type Object is limited interface;  
— The version we used with the GPL 2005 compiler.
```

In any case, we think this is one of the most interesting language additions, providing a manageable form of multiple inheritance and also contributing to more informative specifications.

– **Dot notation.**

We close this section with the much anticipated dot notation. The 2005 implementation was certainly buggy, although one could use it if really determined, specially if compilation units were small. Typically, it could be used without noticeable problems until some specific construct would cause

the compiler to crash in any dot notation instance in the compilation unit (which previously compiled fine). We submitted two bugs that were accepted and corrected.

In contrast, the 2006 implementation has been solid in this respect, allowing us to use it without any contempt now.

7 Conclusions

We have presented an Ada architecture currently used for research on heterogeneous multi-robot mobile teams. It makes use of modularity in the form of black box components to allow manageable complexity and enable easy cooperation among developers. During its ongoing development we are testing several Ada 2005 features, following its addition to the GNAT GPL compiler, with mixed but promising results.

Our architecture gives access to a wide range of capabilities through several implemented components and the reuse of available libraries: configuration is parsed with XmlAda; graphical displays are implemented with GtkAda [16]; robot control is achieved via Player-Ada, a binding authored by ourselves. Several ready for use components related with our research are presented.

We value greatly the GNAT effort in bringing a fully usable GPL compiler to the Ada community. Also, the evident improvements between the 2005 and 2006 versions evidence the compromise of AdaCore in having a complete 2005 implementation as soon as possible.

Acknowledgments

This work is partially funded by the Spanish MCYT-FEDER projects DPI2003-07986 and DPI2006-07928.

References

1. A. R. Mosteo and L. Montano, "Simulated annealing for multi-robot hierarchical task allocation with flexible constraints and objective functions," in *IROS'06 workshop on Network Robot Systems: Toward intelligent robotic systems integrated with environments*, 2006.
2. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 4598, no. 220, pp. 671–680, May 1983.
3. V. Cerny, "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm," *Journal of Optimization Theory and Applications*, vol. 45, no. 1, pp. 41–51, 1985.
4. B. P. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *International Conference on Advanced Robotics*, 2003, pp. 317–323.
5. P. Avgeriou and U. Zdun, "Architectural patterns revisited - a pattern language," in *10th European Conference on Pattern Languages of Programs (EuroPlop'05)*, July 2005, pp. 1–39.

6. E. Briot, C. Baillon, and M. Krischik, "XmlAda." [Online]. Available: <https://libre2.adacore.com/xmlada/>
7. M. B. Dias, R. M. Zlot, N. Kalra, and A. T. Stentz, "Market-based multirobot coordination: a survey and analysis," Robotics Institute, Carnegie Mellon University, Tech. Rep. CMU-RI-TR-05-13, April 2005.
8. J. Minguez, F. Lamiraux, and L. Montesano, "Metric-based scan matching algorithms for mobile robot displacement estimation," in *IEEE Int. Conf. on Robotics and Automation*, Barcelona, Spain, 2005.
9. A. R. Mosteo, "Player-Ada." [Online]. Available: <https://ada-player.sf.net>
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. New York, NY: Addison-Wesley Publishing Company, 1995.
11. D. Riehle, "A role-based design pattern catalog of atomic and composite patterns structured by pattern purpose," Union Bank of Switzerland, Switzerland, Tech. Rep. 97-1-1, 1997.
12. D. Applegate, R. Bixby, V. Chvatal, and W. Cook, "Concorde TSP solver." [Online]. Available: <http://www.tsp.gatech.edu/concorde.html>
13. J. Miranda and E. Schonberg, "GNAT and Ada 2005," AdaCore, Tech. Rep., January 2005.
14. J. Barnes, "Rationale for ada 2006," *Ada User Journal*, vol. 26,27, 2006.
15. A. Charlet, "Add support for function returning anon access type," Oct 2006. [Online]. Available: <http://gcc.gnu.org/ml/gcc-patches/2006-10/msg01690.html>
16. E. Briot, J. Brobecker, A. Charlet, and N. Setton, "GtkAda." [Online]. Available: <https://libre2.adacore.com/GtkAda/main.html>