Advances in the Framework for Automatic Evaluation of Obstacle Avoidance Methods

J.L. Jimenez I. Rañó
Dpto. de Informática e Ing. de Sistemas
Universidad de Zaragoza, Spain
irano@unizar.es

J. Minguez
I3A, Dpto. de Informática e Ing. de Sistemas
Universidad de Zaragoza, Spain
jminguez@unizar.es

***Abstract*— This paper will describes the advance in our project for benchmarking obstacle avoidance techniques for mobile robots. The core of the project is to create a methodology/software to evaluate the performance of the methods given a wide range of work conditions. These work conditions usually include scenarios with very different nature (dense, complex, cluttered, etc). The performance is measured in terms of robotic parameters (robustness, optimality, safety, etc). In the paper we will give an overview of the project and we will focus on the project analysis from a software engineering point of view. At this state the software design decisions are critical and could imped a proper later development, therefore we have developed a great effort in the analysis and design of the project.**

I. Introduction

There is currently a great effort in the robotic community to find standards as a way to measure the quality of a wide range of technologies. Good examples are the RosTa EU project, the special group of interest raised by EURON and some other launched by the NIST. Many of these efforts are focussed on creating benchmarks for a given area. Our project is concerned with the standarization but from the automatic evaluation point of view. Instead of working on benchmarks, we address an automatic evaluation system is being constructed for obstacle avoidance algorithms. On the one hand, benchmarking just will try to match the results of an algorithm with some expected output, a desired algorithm result. On the other hand, our automatic evaluation framework will provide the needed results to perform the benchmarking, but also will generate a comparison performance of the obstacle avoidance algorithms in very different situations.

In this direction, we have been working in the development of a software tool evaluate the obstacle avoidance algorithms in the context of service robotics. The objective of this paper is to present the first steps: the analysis and design of the problem from a software engineering perspective. The project is expected to grow up, hence this software engineering perspective is very important at this stage. Since this is currently an open research field, the application is expected to change as new ways of evaluation appear. In order to design a useful application in the future, including those desirable characteristics, it is important to follow good software engineering practices. An early implementation has been done and some tests have been performed as the final steps of the analysis of requirements and design of modules and system. The rest of the paper is organised as follows: Section II introduces the evaluation system global analysis and its requirements. In Sections III, IV and V the analysis and design of the identified independent applications is presented. Some details of our early implementation and preliminary results are depicted in Section VI, while Section VII depicts some conclusions and next steps in our project on automatic evaluation system for obstacle avoidance.

II. Overview of the Evaluation System Framework

In this section we will present an overview of the whole evaluation system framework to analyse and identify the possible modules and its requirements. Some aspects of the evaluation system has been introduced in [2]. The idea behind the application is to evaluate obstacle avoidance methods (measuring quantitative parameters of the solutions) given a wide range of working conditions (different scenarios). As can be seen from Figure 1 three independent modules can be clearly identified. The colored blocks denoted *Module 1*, *Module 2* and *Module 3* correspond to the scenario generator and characteriser part, the robot simulator and trajectory descriptor and the final analysis of the results module. They jointly from the complete automatic evaluation system and can be implemented as separate applications even some interaction exists between them.

Figure 2 draws the interaction between the three modules of the automatic evaluation system as a data flow diagram representation. Instead of keeping direct data flows between the modules, the applications can use two data repositories, namely the scenario files (obtained from the scenario generator) and the execution result files (provided by the robot simulator module). The user will introduce to the system some parameters like the amount of scenarios to generate, the number of test to run, and so on. In view of Figures 1 and 2 the automatic evaluation has been decided to be implemented as three independent applications.

Following [5] the requirement analysis of a software system can be viewed as a set of functional requirements, what the system should do, and a set of non-functional requirements, like efficiency, portability, extensibility and so on. As a ongoing research project some points are not currently defined or are prone to change, therefore software modularity, flexibility and extensibility are mandatory as

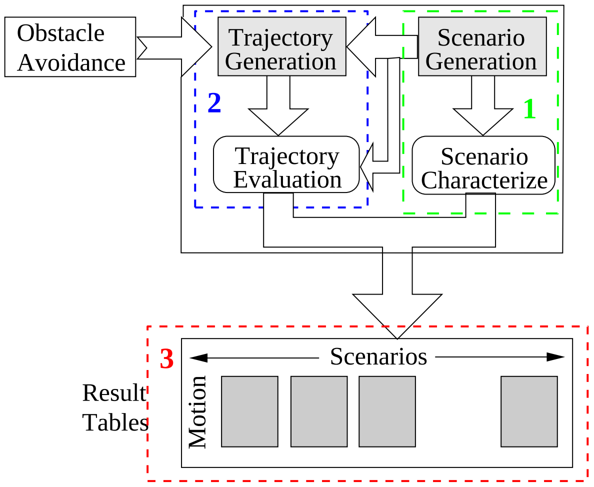


Fig. 1. Framework Modules

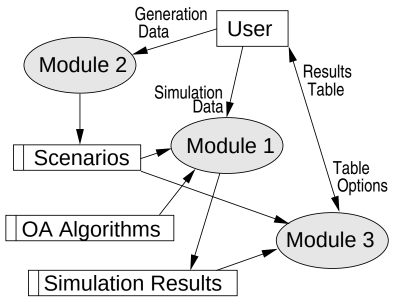


Fig. 2. Data Flow Diagram

non-functional desirable features. On the other hand the functional requirements of the system have to be defined before the analysis stage. The rest of this section is devoted to present the modules and its functional requirements.

a) Module 1: represented in Figure 1 is the scenario generator and characteriser. Its general purpose is to generate simulated environments (randomly or following some criteria) and compute descriptors measuring interesting properties from an obstacle avoidance point of view. The descriptors will be used to classify the different sets of environments according to its nature (dense, complex, cluttered, etc). This is a key module on the system since the obstacle avoidance algorithms must be tested on all the possible situation or environments a robot will face up. A high level analysis shows that its functionality must include:

- It must be able to characterise the scenarios using the defined descriptors and compute their values.
- It must generate scenarios, randomly (for a given number of scenarios) or such that they descriptors cover a given range with a minimum number of them. While in the first case the user must only introduce the number of scenarios to be generated, the former must ask for a number of bins and minimum scenario number in each bin.
- It must handle scenario files and files with scenario lists. The module will create, open, save, move, copy and select (following several criteria) those files or parts of them.
- It should allow to create and delete indexes on a list of scenarios, according to a descriptor and user defined descriptor ranges.
- It must include a graphical user interface to show single scenarios or scenario lists, and make them accessible for user manipulation.

b) Module 2: will compute the trajectories of an obstacle avoidance algorithm for a set of scenarios. It also should provide some performance measurements of the trajectories refereed to an, in some sense, optimal path. The functional requirements found for this module are:

- It should be able to dynamically load any obstacle avoidance algorithm that matches a given function prototype.
- It must simulate a generic robot over a set of scenarios and characterise the robot motion according to some trajectory descriptors.

c) Module 3: builds the evaluation results based on the outputs provided by the two other modules. Given an obstacle avoidance algorithm, for each scenario descriptor and trajectory descriptor this module will generate a result table with the algorithm performance behavior. This module can bee seen as the most general part of the system, since it can be used for other purposes where scalar results from two different features need to be drawn. The minimal functional requirements of this application are:

- It must build tables by crossing scenario descriptors and trajectory descriptors.
- It must be able to open and to create result files for an obstacle avoidance algorithm benchmark in a predefined format.

III. Scenario Generation and Characterisation

In this section we present the application for scenario generation and characterisation (*Module 2*) analysis and design, while implementation and testing will be treated in a separated section. We choose to follow the OMT (Object Modelling Technique) methodology [3] to build the analysis diagrams of each application in the system and some UML (Universal Modelling Language) tools has been also used to build those diagrams. However, since UML [4] has been derived from OMT and they have many common elements.

The structure of the sections describing the individual applications is the same for all three parts of the whole evaluation framework (see also Sections IV and V). First the analysis section focuses only on the static relations of data, not treating the dynamic and functional relations, because its smaller significance in our case. Then the design section includes the application architecture grouping the analysed data models into modules.

A. Analysis of the Scenario Generator and Characteriser Application

The key concept in this application is the scenario for a full technical description of scenarios see [2]. They will be characterised according to its qualitative features and will be used to carry out the robot motion simulations. A concept with such an importance must be included as a class to collect the scenario functionality on the application, the *Scenario* class. The scenarios will have associated sets of characteristics, the descriptors, and encoding models, the way scenarios are implemented in the application, that need to be included in the static class view. Therefore a *Descriptor* virtual class needs to be created to reflect the scenario characterisation. The general class represents the functionality of any descriptor prone to be implemented in the system, and collects the scenario characterisation functional requirement. Moreover, as the *Descriptor* class is virtual the concrete defined descriptors will be derived from it. This also fulfils the extensibility non functional requirement above mentioned. Since the calculation of some descriptors can be computationally expensive in terms of time and memory we decided to use the *Proxy* software pattern [1]. In this way the computation of the descriptor is delayed until it is really necessary, and once the numerical value is get it will be stored for later use. Figure 3 shows part of the UML class diagram generated on the analysis stage of the application. As can be seen from figure, currently only three descriptors have been implemented as *Density*, *Clearness* and *Confinement* derived classes.

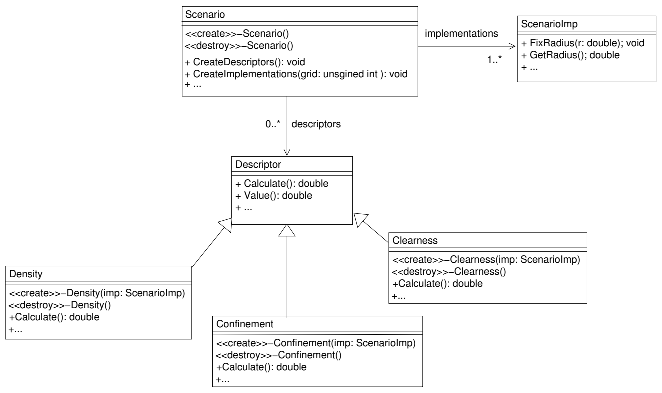


Fig. 3. Part of Scenario Class Diagram

The right class box on Figure 3 is the scenario implementation *ScenarioImp* virtual class. Since the computation of some descriptors can be quite difficult for some scenario representations the *ScenarioImp* class has been designed to allow different scenario representation on the application. Even not drawn in the figure two classes are derived from this virtual class, the discrete *DiscreteImp* and continuous *ContinuousImp* scenario implementations. While the later stores the environment as a set of obstacles defined by its geometrical primitives, the former consists on a binary occupancy grid with configurable size. This allows to compute the descriptors with different resolutions and algorithms for a given scenario. The capability of having different implementations introduces the need for an implementation converter in order to pass from one implementation to another. A *Converter* virtual class has also been designed using the *Strategy* software pattern. This pattern defines and encapsulates a family of algorithms, in our case for implementation conversion. From the *Converter* class any implementation converter can be derived. Currently only a continuous to discrete implementation algorithm is available.

Another functional requirement of this application stated in Section II is the ability to generate scenarios in different ways. A fixed number of random scenarios, for instance, could be desired to perform test on some algorithms. In other cases a set of scenarios need to be generated to cover all the possible value ranges of a given descriptor. A virtual class *Generator*, not reflected on the Figure 3, has been defined to fulfil the corresponding functional requirement of the application. Since the generation of scenarios can be performed in different ways, the *Strategy* software pattern has also been selected to design the generator class. Any instance of a generation algorithm must be derived from the *Generator* class. Moreover, as there are several implementations of the scenario, the virtual class has a method called `FixImplementation()` to select the implementation kind to be generated.

Up to this point the analysis has only included individual scenarios, however the evaluation system has to perform test on all possible scenario conditions. Therefore a scenario list *ScenarioList* abstract class needs also to be used to jointly store sets of scenarios. Any other scenario list can be derived from it. On the other side, the lists usually need to be acceded in a sequential way and an iterator software pattern is necessary. Besides this access method for the scenarios on the lists, another functional requirement is the capability of scenario indexing according to the value of some descriptor. A class for a scenario list index has been modelled to serve as an interface for scenario lists allowing to access groups of scenarios with some given ranges of descriptor values.

An important aspect of the scenario generator is data persistence, because all the computed descriptors and its scenarios must persist when the application finishes. To generate and characterise scenarios can be a resource consuming task. To allow this data persistence it was necessary to create a scenario storage file system. Since the internal form in which data is stored is not a part of the analysis stage, but just to take into account the persistence needs, two abstract classes have a reader and a writer been implemented, with some appropriate derived classes to store the scenario lists in a preliminary format. This structure allows for a change in the internal storage way, fulfilling also the flexibility non functional requirement.

The final requirement is to allow the user generate and handle the scenarios and sets of them through a Graphical User Interface. It is common practice to change an application GUI, and therefore it is important to have a small number of classes involved, to have a uncoupled application and GUI implementation. Any visual element on the application is modelled as a *GUIElement* class, and are grouped hierarchically. The composite software pattern has been used since it allows to treat visual elements in the same way either being compound or not. It is a good practice to separate the user interaction in two parts: the visual part and the functional part, such that if any needs to be changed the other can be kept. The functional part are implemented through the application commands that represent the simplest actions a user can perform. The Command software pattern proses the creation of command objects to encapsulate and parametrise actions.

B. Design of the Scenario Generator and Characteriser Application

The previous section has depicted the five main concepts related to the scenario generation and characterisation application with its corresponding classes. A set of classes (Scenario, Scenario Implementation, Generator…) and its derived subclasses are grouped around the scenario concept. There are other classes related to the scenario list; the Scenario List itself, the Iterator, Reader, Writer and Indexes. Some others providing Graphical User Interface functionality and finally the command related classes. All of the above can be joined into a new class representing the application itself. Figure 4 groups and relates the main modules to build application, an arrow starting in one module means that it depends on the target arrow module. As can be seen all the modules, except the application, use services of the Scenario module while this one does not use any service of the rest of the layers. This makes the scenario module quite critical in case any change should be done, because all the layers could need an adaptation to the new module interface. The Scenario I/O layer is used by the Commands and GUI modules making it the second most critical module to changes.

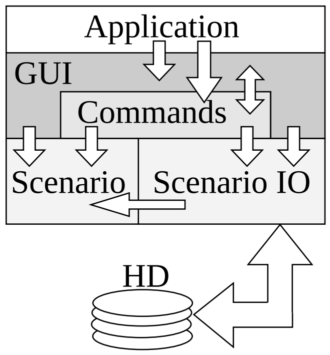


Fig. 4. Scenario Generator and Characteriser Application Architecture

The Commands and GUI layers interact with each other, since the GUI uses Commands functionalities and introduces information parameters from the end user to the commands. Finally, the higher layer, the Application is available to the user for the lower level Commands and GUI layers access.

IV. Robot Simulator and Trajectory Evaluation

This section presents the analysis and design performed on the simulation application, the so called module 1 in Section II. This module must interact with the scenario generator presented in Section III by reading the different scenarios where obstacle avoidance algorithms have to be tested. As stated in Section II the functional requirements of this application are the simulation of a generic robot motion for any dynamically loaded obstacle avoidance technique and trajectory descriptor calculation.

A. Analysis of the Robot Simulator and Trajectory Evaluation

The simulated robot model has a sensorial and motor part. The sensor of the robot is assumed to be a proximity range sensor providing distance measures in a $180°$ range around the robot front, one for each degree. On the other hand, the robot always move forward with limited speeds and accelerations. As presented in Figure 5 a robot class models all those robot characteristics. We reflect in this class diagram takes the fact that the motion control is provided by an external library. Through the MovementCalculator class we provide an interface for the implementation of the dynamic library loading process and function call. As can be seen in the figure two classes are derived from this basic one that must take into account the way each operating system loads the dynamic libraries. Using this class heritage to load obstacle avoidance algorithms the non functional requirement of a multi-platform application is accomplished.

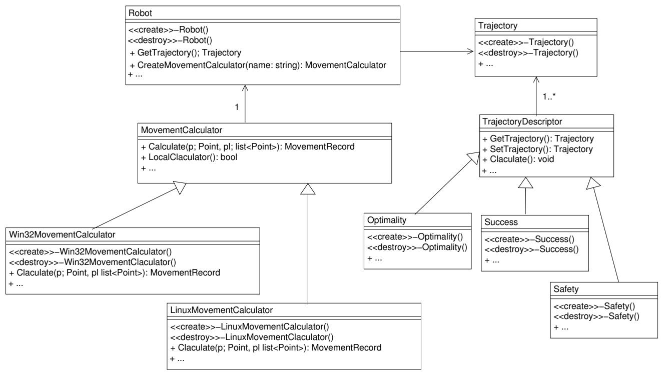

Robot
<<create>>–Robot()
<<destroy>>–Robot()
+ GetTrajectory(); Trajectory
+ CreateMovementCalculator(name: string): MovementCalculator
+ ...
Trajectory
<<create>>–Trajectory()
<<destroy>>–Trajectory()
+ ...
1
1..*
MovementCalculator
+ Calculate(p; Point, pl; list<Point>): MovementRecord
+ LocalClaculator(): bool
+ ...
TrajectoryDescriptor
+ GetTrajectory(): Trajectory
+ SetTrajectory(): Trajectory
+ Claculate(): void
+ ...
Win32MovementCalculator
<<create>>–Win32MovementCalculator()
<<destroy>>–Win32MovementClaculator()
+ Claculate(p; Point, pl list<Point>): MovementRecord
+ ...
Optimality
<<create>>–Optimality()
<<destroy>>–Optimality()
+ ...
Success
<<create>>–Success()
<<destroy>>–Success()
+ ...
Safety
<<create>>–Safety()
<<destroy>>–Safety()
+ ...
LinuxMovementCalculator
<<create>>–LinuxMovementCalculator()
<<destroy>>–LinuxMovementClaculator()
+ Claculate(p; Point, pl list<Point>): MovementRecord
+ ...

Fig. 5. Class Diagram for the Robot Simulator and Trajectory Evaluation

The trajectory evaluation is performed by the computation of some trajectory descriptors also represented in Figure 5. The motion simulator obtains as output the robot trajectory that must be compared to some defined optimal paths to evaluate the trajectory generated by the loaded obstacle avoidance algorithm. The robot class includes a trajectory class which can be evaluated through the trajectory descriptors, modelled as a virtual TrajectoryDescriptor class. Currently three such derived descriptors have been defined and implemented, success, optimality and safety. The success descriptor has a boolean value indicating if the target position has been reached following the motion commands. Optimality is a comparison with the optimal path obtained using the visibility graph from the start to the end position, while safety is computed through comparison with the safest path, obtained from the Voronoi diagram. Finally, as for the scenario characterisation application the results need to be stored, therefore the robot class also includes a result writer abstract class that allows to store descriptors in a format that can be changed in the future without affecting the rest of the classes.

B. Design of the Robot Simulator and Trajectory Evaluation

The robot simulator and trajectory evaluation application design is presented in Figure 6. As can be seen some of the application layers are Scenario and Scenario I/O, defined for the scenario generator but also used here. The design reflects that the only way to access the hard disk storage is through the I/O modules. An operating system layer has been added since the obstacle avoidance library, the Motion Calculator module, is OS dependant. The Robot layer is related is related with the Scenario, from which simulation data is extracted, with the Trajectories and Characterisation layer, the one that computes trajectories descriptors

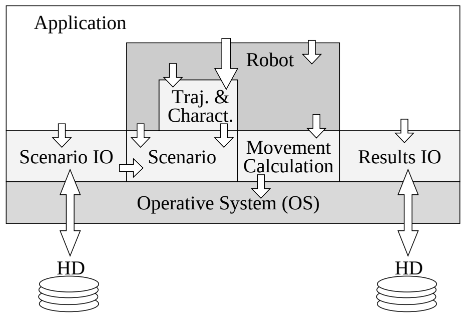


Fig. 6. Simulation Application Architecture

Since two of the designed modules on this trajectory evaluation application are directly dependent on the Scenario layer, as for the scenario generator and characteriser the classes related with the scenario class may change if the scenario is changed. Therefore the scenario layer must have a stable interface with the rest of classes.

V. Result Analysis Application

This is the simplest application on the current implementation of the automatic evaluation system. Actually this application could be used to perform data analysis of any system with similar features, since it only builds result tables by crossing descriptor ranges. Besides the capability of building result tables the only functional requirement is to store them into a given format.

A. Analysis of the Application for Data Analysis

The key concept in for this application is the result interpreter, an element that for a given result sequence and a scenario file is able to statistically summarise results in an automatic way. Since there are multiple table formats and programs to handle them we decided to use in out preliminary implementation a plain text format. However, it can be interesting to build binding with some office applications for data analysis. The main classes include the ResultReader abstract class that forms a base for the result file reader, the reading functionality for results not included on Figure 5. When the reader finished the process of data loading it generates an element of the ResultInterpreter hierarchy which is responsible of statistically cross the data. Once again the Result interpreter class is an abstract one allowing different data interpretation classes to be defined in later researching steps, that is providing extensibility to the application.

B. Design of the Application for Data Analysis

This is the smallest application to design, only an interpreter for the results needs to be added as represented in Figure 7, while the Scenario and Scenario I/O layers are again used. The main application layer uses the functionality of the results interpretation and the Results I/O, while the statistical interpretation module writes its own result files, making it independent of the Results I/O layer.

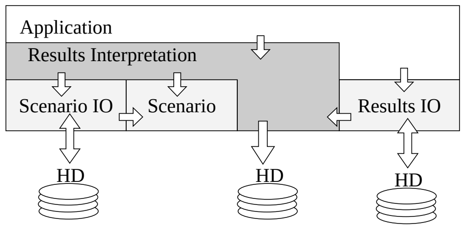


Fig. 7. Analysis Application Architecture

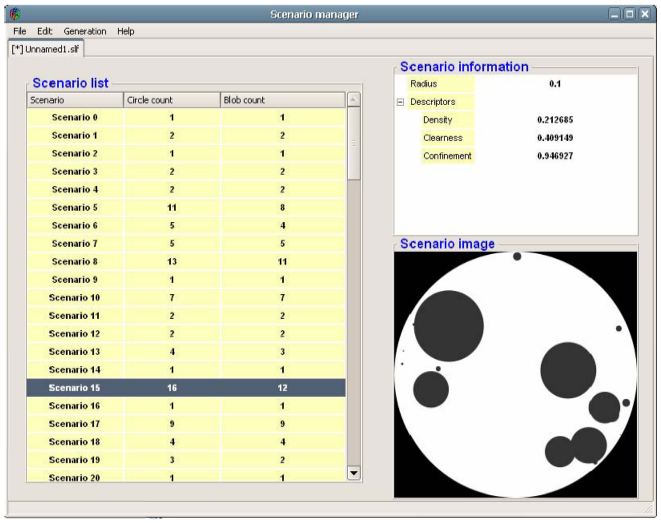
VI. Implementation Issues and Application Testing

Besides all the functional and non-functional requirement for our applications we should choose the proper development tools. Since the obstacle avoidance algorithms will usually be provided as compiled dynamic libraries and the evaluation problem is run on a intensive test basis, an efficient programming language must be selected. Another interesting requirement is to make a multi-platform project, that, as stated before, must be highly modular. On the other hand, both application analysis and design have been done Object Oriented and therefore the implementation language should support objects. We choose C++ as our implementation language since it fulfils all out requirements, but mainly it provides modularity and produces efficient programs. We also used a set of highly standard tools like the Standard Template library (STL) and GTK+ for the application parts that need a Graphical User Interface.

An early implementation of the whole framework has been performed contain more than 11,000 lines of C++ code. The performed tests over the system has been done in two steps; a component test and integration test. The component test where performed over all the classes and modules, and the joint application was also tested once the parts were integrated. Most of the test were black-box testing, where different inputs were provided to the elements and its outputs were checked with the expected output. Once a class or layer was fully tested the integration was performed and again tested in a bottom-up way.

A. Scenario Generation and Characterisation

Figure 8 shows a final view of the application Graphical User Interface. The left part of the main window shows the working scenario list, while the right part is split in the current scenario display at the bottom and the descriptors sub-window, where descriptor values are displayed. The scenarios can be generated in different ways, the non-parametric way just generates a random number of scenarios on a scenario list. The parametric generator allows the user to define bins over one descriptor and select a minimum number of scenarios in each bin. Of course all the generated scenarios or lists can be saved to the hard disk with an appropriate format using the application.



Scenario	Circle count	Blob count
Scenario 0	1	1
Scenario 1	2	2
Scenario 2	1	1
Scenario 3	2	2
Scenario 4	2	2
Scenario 5	11	8
Scenario 6	5	4
Scenario 7	5	5
Scenario 8	13	11
Scenario 9	1	1
Scenario 10	7	7
Scenario 11	2	2
Scenario 12	2	2
Scenario 13	4	3
Scenario 14	1	1
Scenario 15	16	12
Scenario 16	1	1
Scenario 17	9	9
Scenario 18	4	4
Scenario 19	3	2
Scenario 20	1	1

Radius	0.1
Descriptors	
Density	0.212685
Clearness	0.409149
Confinement	0.946927

Fig. 8. Generation and Characterisation Application

B. Robot Simulator and Trajectory Evaluation

For the simulation application the optimal path between a starting and target positions needs to be obtained. The current implementation state includes a A^* algorithm to compute the optimal path as a sequence of discrete cells, therefore using the discrete implementation of the scenarios. The Voronoi diagram on the discrete scenario implementation has been used to get the safest path. Both, paths are used to compute optimality and safety trajectory descriptors, which are also evaluated using the discrete implementation by performing an integration over the grid cells. Since the amount of simulations to perform is big and done as a batch process, no graphic interface has been designed for this application.

C. Evaluation Results

For the final evaluation results, we implemented a software able to connect the different descriptors of the scenarios with the performance paramenters of the methods. This is displayed in the form of tables of performance for visualization and comparison in between methods. Figure **??** shows the performance of one technique selected **??**. In this case, one can see the performance and evolution of the performance descriptors as a function of the different scenarios measured by density, clearness and confinement for example. In fact this type of tables are a great help of researchers, engineers and developers in order to asses their results and to search for possible techniques to work in a given range conditions.

VII. Conclusions and Further Work

This paper present the current state of the automatic evaluation software for obstacle avoidance algorithms. As an ongoing research project some aspects of the software framework development could change. The initial idea is to create an open source software system. A great effort has been performed from a software engineering perspective to start building an extensible and flexible framework in three mostly independent modules. Extensibility and flexibility must be key features on such a system. Some major aspects of the analysis and design have been presented. Our framework allows to include new descriptors to both, scenarios and trajectories, that on the other hand need to be studied and implemented to better evaluate obstacle avoidance mechanisms.

The future steps include the definition and implementation of new scenario descriptors, and a extensive scenario lists generation as a test-bed for the algorithms. Even the system has been developed to be multi-platform this feature has not been tested yet, and maybe some small changes and further development will be needed to actually have such an application. A Developer's Guide document and online API documentation will be also created in order to help both developers and users.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1998.
[2] J. Mínguez and I. Rañó. Steps toward the automatic evaluation of robot obstacle avoidance algorithms. In *Workshop of Benchmarking in Robotics, in the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006.
[3] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
[4] J. Rumbaugh, I. Jacobson, and G. Booch. *The UML Reference Manual*. Addison Wesley, 2004.
[5] I. Sommerville. *Software Engineering*. Addison Wesley, 2005.