

# SELECTSCRIPT: A Query Language for Robotic World Models and Simulations\*

André Dietrich, Sebastian Zug, and Jörg Kaiser

**Abstract**—We introduce a new declarative language called SELECTSCRIPT. As its name suggests, it is a scripting language inspired primarily by SQL and its relational algebra. It is intended to be used for complex queries on different kinds of world models. Scripts can be dynamically generated and executed, or embedded into the code of foreign programming languages. A first interpreter was therefore developed for Python. Adapting the ideas of language-oriented programming, which enables developers to create their own domain-specific language, we developed a language stub that can be easily adapted and extended to comply with any (discrete) robotic world model or robotic simulator. We will further show how simple SELECT-statements can be used to extract any kind of valuable information in various return formats, thereby going beyond traditional SQL capabilities.

## I. INTRODUCTION

In 1970 Edgar F. Codd published his remarkable idea of employing a relational model for databases and also introduced the concept of a universal language for data manipulation and querying. The first representative was called Structured English Query Language (SEQUEL) [1] and later renamed into Structured Query Language (SQL). It has become a standard for defining, storing, manipulating, and querying data in databases (cf. [2]). Although databases are in general not directly portable across different database managements systems (DBMS), and different DBMS were developed for different purposes with specific implementation details (which is “proportional” to the number of world models (and simulation environments)), SQL offers a standardized interface to access all data and relations. Even non-relational DBMS, so-called NoSQL-systems (Not only SQL [3]), try to copy (reimplement) the syntax and the semantics of SQL because of its expressiveness and wide acceptance. Popular examples include Cassandra’s CQL [4] or HBase with Phoenix [5].

No one would nowadays try to query a database by directly accessing all data sets then and analyze or filter them manually with an imperative, procedural, or functional programming language. However, all of us access robotic world models in that way! A world model is the linkage between sensory perception and control, which is used to transform, filter, fuse, and abstract sensor data in a way that only relevant information is passed to the control section. An autonomous system may thus operate in the “real world”, but

reasoning, planning, and decision making is always based on the information encoded within the internal (and purely virtual) representation (co-simulation) of the external world (cf. [6] for an early overview).

As in databases, there are a multitude of environmental representations for various purposes. Simple mobile platforms might be satisfied with pattern-based representations (e. g., Roomba’s AWARE [7]) or simple 2D (cf. [8], [9], [10], [11]) or 3D (cf. [12], [13]) representations of the environment, while others require even more complex 3D representations with additional information about objects, semantics, and physical quantities (cf. [14], [15]). The latest achievements of Kim [16] or Ren [17] show that having such complex 3D scenes generated on-line and applied in robotic applications will be a real possibility in the near future. Of course there are also other systems that apply previously mentioned representations in parallel to formal (e. g., [18]) or logic-based representations (e. g., [19]). These additional “non-spatial” environmental representations come into play when other problems, apart from navigation, localization, or manipulation, have to be solved, such as the identification of complex action sequences, the appropriate usage of services (offered by external systems), or for a sophisticated situational awareness. The more complex robotic applications become, the more complex and sophisticated world models or environmental representations are applied.

Thus, different systems, even in overlapping working areas, will apply different types of environmental representations according to their tasks and capabilities. But how will these systems communicate and share their knowledge about the environment? Or in other words, how will a Roomba access the information from a Wakamaru and vice versa? One single standardized protocol surely cannot cover the diversity and complexity of imaginable and not imaginable needs. It will require the expressiveness and complexity of whole language.

Following Codd’s idea of a universal data manipulation language, we have developed a declarative and extendable query language that can be utilized as a general interface to robotic world models (and simulations). It is called SELECTSCRIPT because in contrast to a complete SQL implementation, we only provide the possibility for accessing and querying data (based on SELECT-statements), everything else is left out to the original programming language. We adopted the semantics of SQL and added new language features that enable application specific response formats as well as to extend the language according to different requirements.

\*This work is (partly) funded by the EU FP7-ICT program under the contract number 288195 “Kernel-based ARchitecture for safetY-critical cONtrol” (KARYON).

A. Dietrich, S. Zug and J. Kaiser are with the Department of Distributed Systems, at the Otto-von-Guericke-Universität Magdeburg in Germany (ST). email: {dietrich, zug, kaiser}@ivs.cs.uni-magdeburg.de

## Overview

In the next section we give a brief overview on related topics, discuss our previous research and why it became necessary for us to develop this kind of new query language. Then in Sec. III we give a short introduction to the general concepts, the grammar, and implementation details of SELECTSCRIPT. Following this, we demonstrate our approach (showing how easy it is to solve many common problems with simple SELECT-statements) and show how the language can be extended to meet new requirements. We conclude with a summary and an outlook.

## II. RELATED WORK

In [20] we present an approach that deals with the problem of data, distributed in smart environments, and how it can be shared and transformed into valuable information. By applying cloud based techniques it was possible for us to restore small scale 3D scenes (analogous to the approaches presented in [21], [14]), which can be utilized as central world models (similar to the representation depicted in Fig. 2). But that requires that all participants store (or at least link) their data according to a global hierarchy within the cloud. This hierarchy can be described (for simplicity) as a distributed scene-graph. The reconstructed 3D representation has to be continuously updated with real-time data after its creation. As discussed in [22] even such a simple reconstruction of the vicinity causes problems in extracting information. In some cases an application might be interested in an octomap [12] (to be able to apply a certain trajectory planning algorithm), in another case it might require an occupancy grid map [8] (for a certain region, in order to apply a localization algorithm based on laser scans), or simply the relative positions, geometries, and identifiers of entities that match a certain criteria, or something totally different, such as a representation of the environment in Prolog syntax (to identify appropriate action sequences). We were not able to define an interface in a certain programming language that could satisfy all of these requirements and would be easy to adapt.

Searching the literature for relevant publications, allowing to define on-line queries for world models, or simulations, or at least scene-graphs with the required expressiveness turned up only one publication [23]. The authors of this paper also noted that they could not find something similar to their developed query language for simulated mesh data, which is called MeshSQL. In contrast to our approach, the results of a simulation are indeed stored within a database, according to time and space. MeshSQL is thus a real extension of SQL1999 [2] and specialized on mesh data only. This query language is intended to enable researchers to interactively explore simulation data, to identify new and interesting effects. MeshSQL therefore offers temporal, spatial, statistical, and similarity queries, which require different types of return values (i.e., simple values, surfaces and slices in different formats). For the sake of completeness, we also have to mention LINQ (Language Integrated Query) that extends some of the .NET languages to apply SQL queries onto relational databases as well as on arrays or maps (cf. [24]).

This is similar other approaches in the Java world, JQL (Java Query Language [25]), JoSQL (SQL for Java Objects [26]), etc., but with reduced semantics. However, the major problem that we had with these approaches is that they are linked too much to “real” SQL, in terms of tables and data abstractions. Additionally, it does not meet the requirements of dynamically changing world models and of variable return formats. We will discuss this topic in the next section.

When we started to develop our language we quickly realized that we do not have to develop a whole implementation for one world model (or simulation environment). Instead we only have to provide a small language stub for effective querying. All requested attributes, relations, and output values (abstractions) can be externally handled, which means that the required functionality is still developed within the original programming language, by utilizing model specific interfaces, and it is included in SELECTSCRIPT via function pointers or by simply extending the interpreter class. With this way of implementing, we tried to follow the paradigm of language-oriented programming (LOP), which was firstly described in [27], but better summarized in [28]:

*“... I want to be able to work in terms of the concepts and notions of the problem I am trying to solve, instead of being forced to translate my ideas into the notions that a general-purpose language is able to understand (e.g., classes, methods, loops, ...). To achieve this, I need to use domain-specific languages. How do I get them? I create them ...”*

Prolog [29] can also be interpreted as another popular example of a logical LOP, which was applied to develop GOLOG (although it is called after alGOl in LOGic). It is an implementation of the situation calculus and intended to reason about dynamic environments and robotic behavior (cf. [30]). ConGolog extends this language by including concurrency and the influence of external events (cf. [31]). As mentioned in Sec. I it can be interpreted as a “logical” environmental representation. Before choosing an SQL based syntax for our language, we also discussed the possibility of applying Prolog for our purpose, which was mainly motivated by [32]. These authors discuss the need for an appropriate query semantics based on their concept of a world model for a distributed smart environment, which of course should also incorporate the possibility of defining complex situations over time and space. According to Belkin et al., situations can only be defined on a symbolic level, the simplest situation might be a single symbolic attribute value of an object (e.g., a person is smiling), but most situations can only be inferred by observing the real world over a period of time (e.g., two persons having a conversation). Prolog therefore seemed to be ideal for defining and checking situations, but besides inferring, Prolog and SQL are quite similar. Predicate Logic, used by Prolog, is a subset of the Relational Calculus implemented by SQL. Furthermore, SQL has a wider acceptance. We will show in Sec. IV-C that it is also possible to define situations with database techniques and in Sec. IV-D that it is even possible to reconstruct a Prolog representation from a more general world model.

### III. INTRODUCTION TO SELECTSCRIPT

SELECTSCRIPT was developed with the ANTLR [33] parser generator and Python. We currently provide one basic language stub and two “dialects”, one for the Open Dynamics Engine (ODE) [34] and one for the OpenRAVE robotic simulation environment [35]. All sources are freely available for download<sup>1</sup>, for the OpenRAVE implementation we also provide a PiPy<sup>2</sup> hosted package. Screenshots showing running examples were uploaded to our YouTube-Channel: [www.youtube.com/ivsmagdeburg](http://www.youtube.com/ivsmagdeburg)

#### A. Grammar & Syntax

A simplified version of the grammar is depicted on the right in Lis. 1. It shows four language features, which we will elaborate on in more detail. As already mentioned, SELECTSCRIPT only covers a subset of SQL, namely the possibility of defining SELECT-statements. As listed in line 26 in Lis. 1, the usual column names, which are used to define the return values of a query, are represented by function calls. Functions are identified by an identifier and brackets or by their context within a SELECT-statement (see also line 23 and 26). Functions are defined externally and are applied to the objects which are identified by the WHERE-clause. See also Lis. 4 to get an impression of the definition of SELECT-statements. Furthermore there is an additional keyword AS, as defined in line 32 in Lis. 1. We apply it (in contrast to its common SQL usage) as a key element of our language to define the resulting format of a query, which might be a single value, a list of values, a dictionary (preserving the function names in order to generate an SQL like table), but also something totally different, such as an occupancy grid map, a Prolog representation, or a map showing the sensor coverage of an area (see Sec. IV-D). Additional representations as well as further functions can be defined freely by other developers, which allows to extend the language according to various requirements.

As depicted in line 2 a script can contain multiple statements and the results can also be stored persistently as variables, which allows to define more complex query scripts. The flexible language definition allows it to define also nested SELECT-statements and to apply the results of such statements (or the results of function calls) also as input for further queries, whether it is a value, a list, a dictionary, or a traversable structure like a scene-graph (see Lis. 7). In order to be able to define temporal queries, such as at what point in time has something become true, or for how long has it been so (in some cases it might also be necessary to define complex temporal sequences), we have defined a new type of variable (see line 10). These variables allow to keep previous results of a script over a certain period of time and can also be applied as the base for further queries (see the example in Lis. 6). The time horizon is defined during the first variable assignment within curly brackets, which are afterwards used in combination with the variable to request

a stored values for a specific point in time (`var{22.1}`), all values stored within the last 5.2 seconds (`var{-5.2}`), all values (`var{}`), or the most recently stored value (`var`).

Listing 1: SELECTSCRIPT’s simplified formal grammar.

```

1 # a script is a set of statements
2 <SCRIPT> ::= (<STMT> ";" )+
3 # a statement can be an ...
4 <STMT> ::= <EXPR>           # expression
5 | <ASSIGN>                 # variable assignment
6 | <SELECT>                 # select query
7 | <FUNC>                   # function call
8 # dynamically typed variable assignments
9 # with two possible types of variables:
10 <ASSIGN> ::= <ID>         "=" <STMT>      # standard
11 | <ID>    "{"<Float>"}" "=" <STMT>      # with age
12 # allowed expression
13 <EXPR> ::= <ID>           # identifier
14 | (<ID> "." )? "this"    # pointer
15 | <ID>    "{" <Float> "}" # var. with time horizon
16 | <EXPR> "[" <EXPR> "]"  # slice operator
17 | <EXPR> <bOp> <EXPR>    # binary operators
18 | <uOp> <EXPR>          # unary operators
19 | "(" <EXPR> ")"        # parenthesis
20 | <Float> | <Int>        # literals
21 | <String> | <Bool>     # literals
22 # function call with name and parameters
23 <FUNC> ::= <ID> "(" (<EXPR> "(" <EXPR> "*" ) " " # ID(param, ..)
24 # querying with the possibility for specialized
25 # return values, defined by the keyword AS ...
26 <SELECT> ::= "SELECT" (<ID> | <FUNC> ) "(" (<ID> | <FUNC> ) *
27   "FROM" <STMT> "(" <STMT> ) *
28   ("WHERE" <EXPR> ) ?
29   ("GROUP" "BY" (<ID> | <FUNC> ) "(" (<ID> | <FUNC> ) * ) ?
30   ("ORDER" "BY" (<ID> | <FUNC> ) ("ASC" | "DESC" ) ?
31   "(" (<ID> | <FUNC> ) ("ASC" | "DESC" ) ? ) * ) ?
32   ("AS" ("list" | "dict" | "val" | ... ) ) ?
33 # standard Boolean and arithmetic operators
34 <bOp> ::= "+" | "*" | "=" | "<" | ">" | "<=" | ">=" | "<%" | "OR" | "AND" |
35   "-" | "/" | "!=" | ">" | ">=" | "^" | "IN" | "XOR"
36 <uOp> ::= "-" | "NOT"

```

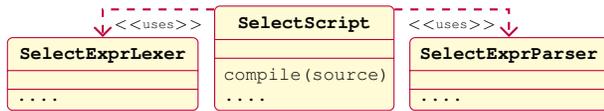
#### B. Implementation

Fig. 1 shows the simplified structure of the implementation, which is segregated into two parts. Fig. 1a depicts the ANTLR generated part, which is responsible for parsing and translating SELECTSCRIPT code into easily interpretable byte-code. Easy means that the basic `SelectScriptInterpreter`, presented in Fig. 1b, can be implemented with only a few lines of code. Our implementation for Python contains less than 250 lines of code. The same could be repeated in other programming languages.

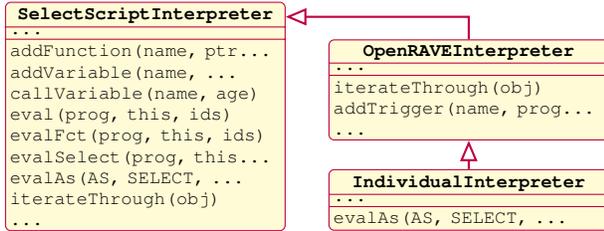
Fig. 1b shows furthermore how to adapt the interpreter to be able to work with different simulation environments and their specific interfaces. It requires the development of a new interpreter class, derived from the basic interpreter as well as redefining or extending some basic methods, `iterateTrough` and `evalAs`. The first method is used to tell the interpreter how to appropriately traverse new objects (structures, scene graphs, robots, sensors, etc.). It is therefore also implemented within `OpenRAVEInterpreter` to iterate through all elements of the OpenRAVE scene graph, while the handling of lists and dictionaries is implemented within the parent class. `evalAs` can be overloaded to define new types of return values and new functions can be attached via method `addFunction`, which requires a function pointer, a name, and some helpful information.

<sup>1</sup>Source-Repo.: [sourceforge.net/projects/selectscript](https://sourceforge.net/projects/selectscript)

<sup>2</sup>PiPy + HowTo: [pythonhosted.org/SelectScript\\_OpenRAVE](https://pythonhosted.org/SelectScript_OpenRAVE)



(a) ANTLR generated lexer, parser, and compile classes.



(b) Manually implemented interpreters, with specializations for different simulation environments.

Fig. 1: SELECTSCRIPT (two-piece) class structure.

#### IV. SELECTSCRIPT IN ACTION

For the demonstrator we choose a (typical) kitchen environment, as depicted in Fig. 2, which is similar to the example presented in [21] or to the indoor scenes that were reconstructed in [16], [17]. We applied OpenRAVE to represent the scene, which was statically defined by an XML description. But as mentioned in Sec. II, similar representations and world models can also be reconstructed from a distributed environment and applied similarly (cf. [20]). To better visualize the sensory coverage area of the cameras, we applied the OpenRAVE “BaseFlashLidar3D” sensor (blue beam in Fig. 2) instead of virtual camera sensors.

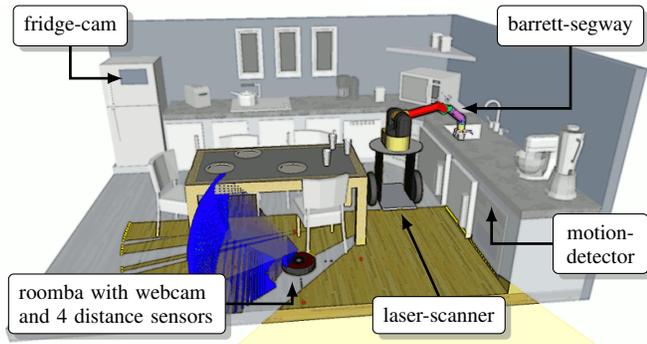


Fig. 2: Kitchen model with two mobile robots and fixed external sensors. Only the sensor beams attached to the robots are visualized.

##### A. Initialization

Loading the SELECTSCRIPT modules is simple (as listed in Lis. 2) and attaching the scene to the SELECTSCRIPT interpreter only requires making the model accessible as a variable from within the script under the name `kitchen` (see line 41). As mentioned earlier, `kitchen` in this case represents the whole OpenRAVE-environment. The interpreter automatically determines the type of variable for appropriate usage and access, whether it is a list, a dictionary, an object, an integer, a float, or a string (as listed in line 42).

Listing 2: Initializing the SELECTSCRIPT parser & interpreter and connecting it to the OpenRAVE model.

```

4 from openravepy import *
5 from SelectScript.SelectScript import *
6 from SelectScript_OpenRAVE.interpreter import *
7 ...
34 env = Environment() # create an OpenRAVE environment
35 env.Load("res/kitchen.env.xml") # loading the model
36 ...
38 ss = SelectScript(None, None) # load the parser
39 ssRave = interpreter() # and interpreter for OpenRAVE
40 # add the environment as variable to the interpreter
41 ssRave.addVariable('kitchen', env)
42 # ssRave.addVariable('today', [27, "Sep", 2014])
43 # ssRave.callVariable('today') => [27, "Sep", 2014]

```

##### B. Simple Queries & Help

As already mentioned, SELECTSCRIPTS are interpreted within other programming languages. A script is a string that can be defined within the source code, read from a file, or dynamically created. A script can also consist only of one function call, shown in Lis. 3. Before a script can be evaluated (line 48) it has to be translated into byte-code (line 47). The translation into byte-code has to be done only once, it can be afterwards evaluated again and again, generating different results based on the state of the simulation.

Listing 3: Calling for help...

```

46 expr = """"help();""""
47 prog = ss.compile(expr) # translation to byte-code
48 print ssRave.eval(prog) # list of function names ...
49 # ["getTime", "within", "isEnabled", "id", "below",
50 # "isRobot", ... "pose", "volume", "position", ... ]
51 expr = """"help("position");""""
52 prog = ss.compile(expr) # translation to byte-code
53 print ssRave.eval(prog) # result is a string
54 # Returns a list of the x,y,z coordinates.
55 # Usage: position(object, begin, end)
56 # position(object) -> [x,y,z]
57 # position(object, 0,2) -> [x,y]

```

Lis. 4 shows an exemplary SELECT query. The previously defined external variable `kitchen` can be directly accessed within this script (FROM). All identifiers after keyword `SELECT` are automatically interpreted as functions with only one parameter. But there might also be functions with multiple input parameters (e.g., `position(this, 0, 2) -> [x, y]`), in this case, the keyword `this` is used to mark the correct position of elements within the function call. As mentioned in the previous section, there might be different requirements for how the result of a query should appear. The comments in Lis. 4 show a snippet of the printed result, which is a dictionary defined by AS dict. The dictionary keys are equal to the function names defined within the SELECT-statement.

Listing 4: Query the model for all identifiers, positions, and types of objects within the kitchen.

```

52 expr = """"SELECT id, position(this,0,2), type
53 FROM kitchen ORDER BY id(this) ASC
54 AS dict;""""
55 prog = ss.compile(expr) # translation to byte-code
56 rslt = ssRave.eval(prog) # result is listed below :
57 #
58 # -----
59 # "barrett_4918" | [ 0.42, 1.30] | Robot
60 # "barrett_4918_laser.." | [ 0.77, 1.30] | Sensor
61 # "chair_083c" | [ 0.52, -0.60] | KinBody
62 # ... | ... | ...

```

The script in Lis. 5 shows a “more” complex example that can be used to query the model for relevant information in a cleaning up task. The requested objects must meet the requirements of being on the table and in reachable distance to the robot. All results are stored persistently within different variables. Note the different types of variables, two are stored “AS value” and one “AS list”. The results can be directly applied (in the FROM part), whereby always the last statement is applied as the return value of the entire script. See Sec. IV-D for more specific return values.

Listing 5: Composition of multiple queries.

```

58 """robot = SELECT obj FROM kitchen
59     WHERE id(this) == "barrett_4918" AS value;
60     table = SELECT obj FROM kitchen
61     WHERE id(this) == "table" AS value;
62     cleanUp = SELECT obj FROM kitchen
63     WHERE above(table, this) AND
64           distance(robot, this) < 1.3
65     ORDER BY distance(robot, this) ASC
66     AS list;"""

```

### C. Complex Queries & Situation Awareness

Lis. 6 briefly introduces the usage of temporal variables, user defined return values and triggers. For the purpose of better localization, by incorporating external sensor measurements, an application (robot) might be interested in identifying all sensors, whether mobile or stationary, that had measured a robot within the last 2 seconds. Such queries might be useful in a dynamically changing environment, where a robot has only vague knowledge about its position and surroundings and thus would need to make use of external sensory systems. Normally this would require various checks for every pair of sensors and robots, if the robot is currently within the measurement range of a sensor and is not hidden behind another obstacle. Furthermore, it requires the storage of results for a certain time.

Variable `measure` (line 125) therefore stores all assigned values over a period of 5 seconds (for demonstration purposes), whereby only the results of the last 2 seconds are returned (line 129). The query itself generates a cross-product of all elements FROM of the model and checks if one of them is a sensor and the other one a robot, as well as if the sensor is currently sensing the robot. The different assignments in line 127 are used in combination with the keyword `this`, to place all parameters correctly within the functions being used. In contrast to the previous example, we do not directly query the model, but instead add a new trigger ‘perceiveRobots’. This method is implemented within the `OpenRAVEInterpreter` (cf. Fig. 1b). The interpreter evaluates the attached byte-code of the script continuously (every 100 milliseconds) and calls a function named `callback`, if and only if the result of the query changes. As the result and structure of a script is always determined by its last statement, the result in this case is a list containing two elements, the current simulation time and a list of all sensors that have measured a robot within the last 2 seconds. The definition in line 95 depicts the body of the callback function with the input parameter `msg`.

Thus, a situation has occurred (or can be interpreted as true), if the result of a `SelectScript` is not empty, and a change of the situation has occurred if the result of a `SelectScript` has changed. In the same way it is possible, for example, to define a “critical” situation such as: “Is there a human within reachable distance to a robot, who is holding a dangerous object (e. g., knife, mug with hot coffee, hand-held blender, etc.)?”. It is furthermore also possible to query the result elements of a timed variable (“FROM measure”), to check whether something is true for a certain period of time, if a situation has occurred within a time frame, or if two or more specific situations have occurred successively within a specific period. This is necessary in order to define aggregated situations, whose individual situations are separated from each other through time.

Listing 6: Triggering a temporal variable in conjunction with a query on the cross-product on the model for identifying the sensors that have perceived a robot within the last 2 sec.

```

95 def callback(msg):
96     simulation_time = msg[0] #float value
97     active_sensors = msg[1] #list of sensor objects
98     ...
125 expr = """measure(5.0) = SELECT obj(env.this)
126     FROM env=kitchen, env2=kitchen
127     WHERE isSensor(env.this) AND isRobot(env2.this)
128     AND isSensing(env.this, env2.this) AS list;
129     [getTime(), measure(-2)];"""
130 prog = ss.compile(expr)
131 ssRave.addTrigger("perceiveRobot", prog, .1, callback)

```

### D. The Final Keyword As

So far we have only worked with the standard result types (i.e., AS value, list, dictionary), but it might also be necessary to generate different abstractions for different purposes. For example, how could the Roomba query the environment model for another abstraction that could be used for trajectory planning? Fig. 3 shows the results of two different requested abstractions to the same query in Lis. 7.

Listing 7: Querying the model for specialized abstractions.

```

140 """ roomba = SELECT obj FROM kitchen
141     WHERE id(this)=='roomba_625x' AS value;
142     SELECT obj FROM kitchen WHERE dist(roomba,this) < 1.5
143     AS environment;
144     # AS occupancygrid(z-pos,grid-size);"""

```

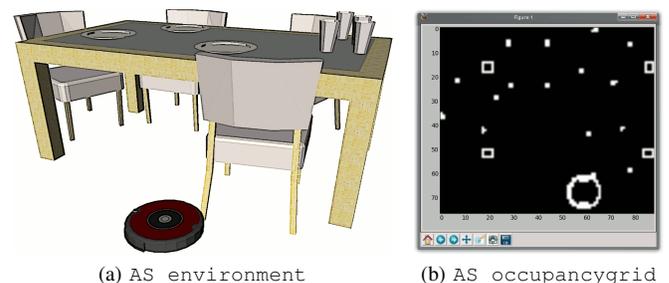


Fig. 3: Different abstractions derived from the same query in Lis. 7, with (a) showing the resulting a sub-model and (b) a binary occupancy grid map (model).

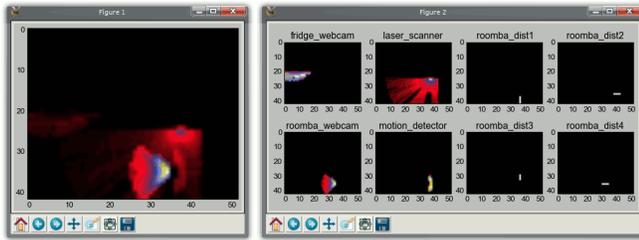
These different abstractions result from the utilization of some of our plugins<sup>3</sup> for OpenRAVE, which can be seamlessly integrated into the language. The next example in Lis. 8 is used to generate a map showing the sensor coverage of the entire model at a certain height. Using GROUP BY in this case makes the difference between one map, containing the areas monitored by all sensors Fig. 4a, or one map per sensor, showing only the area covered by one single sensor Fig. 4b. (We want to apply this kind of mapping for an enhanced trajectory planning in the near future.)

Listing 8: Grouping the resulting sensor coverage grids.

```

180 """ SELECT obj(o.this), obj(s.this)
181 FROM o=kitchen, s=kitchen
182 WHERE isSensor(s.this) AND
183 isRobot(o.this) OR isKinbody(o.this)
184 GROUP BY id(s.this) # one map per sensor
185 AS sensorgrid(zPos=0.3, gridSize=0.025);"""

```



(a) without GROUP BY (b) GROUP BY sensor identifier

Fig. 4: Derived sensor coverage areas for the entire model (a) or as a result set per sensor (b).

In the same way a Prolog representation can also be requested, as depicted in Lis. 9. The return value is a list of clauses that can be directly piped into a Prolog interpreter to be used for further evaluations. All representations were implemented by overloading method evalAs only (cf. Fig. 1). We provide a detailed description on the generation of abstractions on our project-site<sup>2</sup> for OpenRAVE.

Listing 9: Requesting an abstraction AS prolog clauses.

```

211 expr = """SELECT position(a.this), volume(a.this),
212 below(a.this, b.this), isRobot(a.this),
213 above(a.this, b.this), isKinbody(a.this),
214 isEnabled(a.this), ...
215 FROM a = kitchen, b = kitchen
216 WHERE NOT (isSensor(a.this) OR isSensor(b.this))
217 AS prolog;"""
218 # the response is a list of clauses that looks like:
219 # above(table,cup_58a2), position(cup_58a2,[.61, .45,
220 # .922]), volume(cup_58a2,.000157239916373), isKinbo-
221 # dy(cup_58a2), isEnabled(cup_58a2), above(table,p...
222 ...
223
224 from pyswip import Prolog
225 base = Prolog()
226 for clause in ssRave.eval(prog):
227     base.assertz(clause)
228
229 print list( base.query("above(table, X)") )
230 # [{"X": "cup_58a2"}, {"X": "cup_905a"}, {"X": "cup_-
231 # b6af"}, {"X": "plate_835a"}, {"X": "cup_d441"}, ...
232
233 print list( base.query("volume(Obj, V), V < 1.0") )
234 # [{"Obj": "can", "V": 0.00419759182722}, {"Obj":
235 # "shaker", "V": 0.00566537815594}, {"Obj": "pin_a...

```

<sup>3</sup>Plugin-site: [code.google.com/p/eos-openrave-plugins](http://code.google.com/p/eos-openrave-plugins)

## V. CONCLUSIONS & OUTLOOK

Every language has its specific strengths and weaknesses, but the appropriate combination can help to compensate for the weaknesses. If we compare how we solve problems using different programming languages with the command of military forces, then C&C++ would represent the infantry (strong&effective but it requires very detailed control), Python/Ruby/LUA would be comparable to the cavalry (fast programming and dynamically applicable), Matlab/Simulink could be interpreted as the artillery (expensive, domain-specific, and thus very effective in specific fields), and declarative languages could be equated with an intelligence service. Similar to a General, a computer scientist has to select an appropriate programming language and paradigms for his battleground.

The main idea behind the declarative language approach is to express problems in an abstract way so that solutions can be “generated” automatically (by applying specific algorithms in the background) without any need to manually describe each step or the entire control-flow. In other words, I am not interested in how you achieve it, as long as you provide me with that specific information.

For this purpose we introduce SELECTSCRIPT, a new declarative language inspired by the SQL semantic and syntax, which allows to access and treat robotic world models (and simulations) similar to databases. Additionally, we included two language features that allow it to cope with new requirements for dynamically changing models. Temporal queries are enabled by a new kind of variable. The response format of a query is not bound to a particular representation, such as a table, a list, or a value. Instead, different types of representations can be requested for the same information. The language can also be further adapted according the LOP paradigm. Furthermore SELECTSCRIPT enables systems to query foreign world models and to request any kind of information, which is a pretty simple task as long as both systems apply the same type of world model. But SELECTSCRIPT could also be applied as a “lingua franca” for systems with different world models. The complexity of algorithms required for data transformations could be hidden by the language, which would require an interpreter to be on top of every world model (whereby the meaning of functions and representations has to be standardized). But such an interconnection requires further integration of SELECTSCRIPT into different general purpose programming languages (as mentioned in the military analogy).

We plan to extend the language to meet new requirements (parallelization included) of further tasks, such as trajectory planning (or selection), with secondary conditions (e.g., which trajectory is monitored the most by external sensor systems, which route is less frequently crossed by others, etc.) This requires an appropriate selection process and probably also the ability for reasoning. But enabling reasoning capabilities within SELECTSCRIPT that are similar to those in Prolog, requires the implementation of additional mechanisms beyond standard SQL.

## REFERENCES

- [1] D. D. Chamberlin and R. F. Boyce, "SEQUEL: A STRUCTURED ENGLISH QUERY LANGUAGE," in *Proc. of the 1974 ACM SIG-FIDET (now SIGMOD) Workshop on Data Description, Access and Control*, 1974, pp. 249–264.
- [2] A. Eisenberg and J. Melton, "SQL: 1999, formerly known as SQL3," *ACM Sigmod record*, vol. 28, no. 1, pp. 131–138, 1999.
- [3] S. Edlich. The ultimate reference for NOSQL Databases. [Online as at: 10.02.2015]. Available: <http://nosql-database.org>
- [4] DataStax. CQL 3 Language Reference. [Online as at: 10.02.2015]. Available: <http://www.datastax.com/docs/1.1/references/cql/index>
- [5] Salesforce.com. Phoenix - We put the SQL back in NoSQL. [Online as at: 22.09.2011]. Available: <https://github.com/forcedotcom/phoenix>
- [6] E. Angelopoulou, T.-H. Hong, and A. Y. Wu, "World Model Representations for Mobile Robots," in *Proc. of the Intelligent Vehicles Symposium (IV)*. IEEE, 1992, pp. 293–297.
- [7] T. E. Kurt, *Hacking Roomba*. Wiley Publishing, 2007.
- [8] A. Elfes, "Occupancy Grids: A Stochastic Spatial Representation for Active Robot Perception," in *Proc. 6th Conference on Uncertainty in AI*, 1990, pp. 60–70.
- [9] G. K. Kraetzschmar, G. P. Gassull, and K. Uhl, "Probabilistic quadtrees for variable-resolution mapping of large environments," in *Proceedings of the 5th IFAC/EURON symposium on intelligent autonomous vehicles*. Citeseer, 2004.
- [10] B. Lau, C. Sprunk, and W. Burgard, "Improved updating of euclidean distance maps and voronoi diagrams," in *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ, 2010, pp. 281–286.
- [11] B. Kuipers, J. Modayil, P. Beeson, M. MacMahon, and F. Savelli, "Local Metrical and Global Topological Maps in the Hybrid Spatial Semantic Hierarchy," in *Proc. of the International Conference on Robotics and Automation (ICRA)*, vol. 5. IEEE, 2004, pp. 4845–4851.
- [12] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems," in *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, vol. 2. IEEE, 2010.
- [13] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *International Conference on Robotics and Automation (ICRA)*, Shanghai, China, 2011.
- [14] D. Roy, K. Hsiao, and N. Mavridis, "Mental Imagery for a Conversational Robot," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 34, no. 3, pp. 1374–1383, 2004.
- [15] S. Blumenthal, H. Bruyninckx, W. Nowak, and E. Prassler, "A Scene Graph Based Shared 3D World Model for Robotic Applications," in *Proc. of the International Conference on Robotics and Automation (ICRA)*. IEEE, 2013, pp. 453–460.
- [16] Y. M. Kim, N. J. Mitra, D.-M. Yan, and L. Guibas, "Acquiring 3D Indoor Environments with Variability and Repetition," *ACM Transactions on Graphics (TOG)*, vol. 31, no. 6, 2012.
- [17] X. Ren, L. Bo, and D. Fox, "RGD-(D) Scene Labeling: Features and Algorithms," in *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2012, pp. 2759–2766.
- [18] A. Saffiotti, M. Broxvall, B. Seo, and Y. Cho, "The PEIS-Ecology project: A Progress Report," in *Proc. of the ICRA-07 Workshop on Network Robot Systems*. Rome, Italy: IEEE, 2007, pp. 16–22.
- [19] M. Tenorth and M. Beetz, "KnowRob – Knowledge Processing for Autonomous Personal Robots," in *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ, 2009, pp. 4261–4266.
- [20] A. Dietrich, S. Zug, S. Mohammad, and J. Kaiser, "Distributed Management and Representation of Data and Context in Robotic Applications," in *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ, 2014.
- [21] R. Rusu, Z. Marton, N. Blodow, A. Holzbach, and M. Beetz, "Model-based and learned semantic object labeling in 3d point cloud maps of kitchen environments," in *Proc. of the International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ, 2009, pp. 3601–3608.
- [22] A. Dietrich, J. Kaiser, S. Zug, and S. Potluri, "Application Driven Environment Representation," in *The Sixth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM)*, 2013.
- [23] B. S. Lee and R. Musick, "MeshSQL: the query language for simulation mesh data," *Information Sciences*, vol. 159, no. 3, pp. 177–202, 2004.
- [24] P. Pialorsi and M. Russo, *Introducing Microsoft® LINQ*. Microsoft Press, 2007.
- [25] D. Willis, D. J. Pearce, and J. Noble, "Efficient object querying for java," in *ECOOP 2006–Object-Oriented Programming*. Springer, 2006, pp. 28–49.
- [26] G. Bentley. JoSQL (SQL for Java Objects). [Online as at: 10.02.2015]. Available: <http://josql.sourceforge.net>
- [27] M. P. Ward, "Language Oriented Programming," *Software Concepts and Tools*, vol. 15, pp. 147–161, 1995.
- [28] S. Dmitriev, "Language Oriented Programming: The Next Programming Paradigm," *onBoard - Electronic Magazine*, pp. 1–14, 2004. [Online as at: 02.02.2014]. Available: [http://www.jetbrains.com/mps/docs/Language\\_Oriented\\_Programming.pdf](http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf)
- [29] J. A. Campbell, Ed., *Implementation of PROLOG*. Chichester, UK: Ellis Horwood, 1984.
- [30] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl, "GOLOG: A logic programming language for dynamic domains," *The Journal of Logic Programming*, vol. 19, no. 1-3, pp. 59–83, 1994.
- [31] G. De Giacomo, Y. Lesperance, and H. Levesque, "Congolog, a concurrent programming language based on the situation calculus," *Artificial Intelligence*, vol. 121, no. 1, pp. 109–169, 2000.
- [32] A. Belkin, A. Kuwertz, Y. Fischer, and J. Beyerer, "World Modeling for Autonomous Systems," *Innovative Information Systems Modelling Techniques*, vol. 1, pp. 135–158, 2012.
- [33] T. Parr, *The definitive ANTLR reference: Building domain-specific languages*. Pragmatic Bookshelf, 2007.
- [34] R. Smith. (2007) The Open Dynamics Engine. [Online as at: 05.04.2014]. Available: <http://ode.org>
- [35] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, 8 2010.