International Journal of Advanced Robotic Systems

SitLog: A Programming Language for Service Robot Tasks

Regular Paper

Luis A. Pineda[1,*], Lisset Salinas[1], Ivan V. Meza[1], Caleb Rascon[1] and Gibran Fuentes[1]

1 Universidad Nacional Autónoma de México
* Corresponding author E-mail: lpineda@unam.mx

Received 05 Dec 2012; Accepted 06 Aug 2013

DOI: 10.5772/56906

Abstract In this paper we present SitLog: a declarative situation-oriented logical language for programming situated service robot tasks. The formalism is task and domain independent, and can be used in a wide variety of settings. SitLog can also be seen as a behaviour engineering specification and interpretation formalism to support action selection by autonomous agents during the execution of complex tasks. The language combines the recursive transition network formalism, extended with functions to express dynamic and contextualized task structures, with a functional language to express control and content information. The SitLog interpreter is written in Prolog and SitLog's programs follow closely the Prolog notation, permitting the declarative specification and direct interpretation of complex applications in a modular and compact form. We discuss the structure and representation of service robot tasks in practical settings and how these can be expressed in SitLog. The present framework has been tested in the service robot Golem-II+ using the specification and programming of the typical tasks which require completion in the RoboCup@Home Competition.

Keywords Robot Programming Languages, Service Robot Task Structure, Service Robot Programming, Service Robot Architecture, The Golem-II+ Robot

1. Introduction

In this paper we present a programming language and environment for the specification, representation and interpretation of service robot tasks. Programming service robots is a complex exercise involving several kinds of programs defined in at least three, mostly orthogonal, dimensions: the first consists of the algorithms for programming the robot's basic perception and action behaviours (e.g., vision, speech recognition and interpretation, navigation); the second involves system programming which deals with processes and agents, process communication and coordination, and also with the drivers for the diverse input and output devices; finally, the third dimension addresses the representation and programming of the service robot task structure. This is known in the literature as behaviour engineering. Thanks to recent advances in computing, algorithms and sensor and mechatronic technologies, current robotic systems are often equipped with a wide range of functionalities such as navigation, grasping and object recognition. As a result, robots are now capable of performing complex tasks that involve many of these functionalities working concurrently. In order for a robot to perform tasks autonomously, a way to specify its behaviour is needed, i.e., an action selection mechanism which decides what to do based on what the robot perceives and the model of the task. This is especially true for service robots which operate in highly dynamic and unstructured environments. Several strategies have been proposed for behaviour engineering, ranging from using general programming frameworks and conventional languages to developing domain-specific languages. However, the challenge of how to specify behaviour in a flexible and concise way still remains. In this paper we are concerned with specifying and programming service robots in this latter dimension.

In Section 2 we overview several approaches for representing and programming task structure, with focus on service robot tasks, and place the present approach in such a context.

In Section 3 we present our approach for representing the structure of service robot tasks. Tasks are conceptualized in terms of generic protocols, which we call dialogue models (DMs), which the robot needs to perform in order to achieve its goals. The underlying computational model of the present approach is the functional recursive transition network (F-RTN) [1], an extension of the recursive transition network (RTN) [2]. RTNs have an expressive power equivalent to a push-down automata and the model goes beyond the finite state automata (FSA) or finite state machine (FSM) model (i.e., in the formal sense in which a FSA corresponds to a regular language) commonly used in service robot task programming, but preserves the graph-oriented structure, providing a very good compromise between expressive power and efficient computation. The programming environment is embedded within the robot's architecture and operating system through a simple interface, permitting fast prototyping and development. In addition, the system provides a simple and flexible interface to knowledge bases and deliberative resources, like planners, theorem provers and problem solvers which can be used on demand during task execution.

The specification and interpretation of SitLog's programs are presented and illustrated in Section 4. These follow closely the Prolog notation and the core of the system consists of two interpreter programs working in tandem: the first implements the F-RTN model (i.e., for traversing the recursive graph), and the second interprets expressions of a functional language called *L* through which the content and control information of DMs and situations is expressed. DMs have a diagrammatic representation which is also presented in this Section. The framework is illustrated with the specification and program of a typical behaviour required in many service robot scenarios, such as the scenarios in the *RoboCup@Home Competition*.

DMs are represented independently of perception and action modalities, and SitLog's programs need to be related to the actual interpretations and actions performed by the robot during the execution of a task. For this, the SitLog interpreter is embedded in the robot's architecture. In Section 5 we describe the interaction-oriented cognitive architecture (IOCA) [1] that we have developed in conjunction with SitLog for this purpose.

In Section 6 the implementation of the present formalism and programming environment in the robot Golem-II+ is presented. For validation purposes we have developed the full set of tasks of the RoboCup@Home competition (Rulebook 2013) with very promising results. SitLog is independent of task and domain, and can be embedded in different architectures and operating systems, and the paper is concluded in Section 7 with a brief reflection on the generality of the present formalism.

2. Robotic Programming Languages

In this section we review a variety of strategies for specifying and programming the structure of robotic tasks. Among the first domain-specific languages designed for behaviour engineering was the Behaviour Language [3], built on an extension of the subsumption architecture [4]. In the Behaviour Language, behaviours are defined as collections of rules written in a subset of Lisp. From these rules, behaviours are compiled into augmented finite states machines (AFSMs) and these in turn into assembler code. Other prominent early instances of robotic programming languages are Colbert [5] and the Task Description Language (TDL) [6]. Colbert is a sequencer language created for the Sapphira architecture and used for developing intermediate modules connecting motion control and planning. The semantics of a Colbert program is based on FSMs, which are written in a subset of ANSI C. TDL, on its part, defines a syntax as an extension of C++ for task-level control such as task decomposition, synchronization, execution monitoring and exception handling. TDL code is compiled into pure C++ code with calls to a platform-independent task-control management (TCM) library.

Recent instances of domain-specific languages include the Extensible Agent Behaviour Specification Language (XABSL) [7], XRobots [8] and b-script [9]. The semantics of both XABSL and XRobots is based on hierarchical state machines. In XABSL, a given state of the whole system is determined by a subset of state machines. XABSL has been implemented on many platforms, mainly on robots that participate in the RoboCup soccer competitions. XRobots treats states as behaviours which are first class objects in the language and can be passed as parameters to other behaviours. It also integrates template behaviours, allowing generalized behaviours to be customized and instantiated. In contrast, b-script describes hierarchical complex behaviours via specialized generators. b-script's syntax is built on a combination of Python and C++. Programs written in b-script can be executed by an interpreter or compiled into C++ code.

Another common strategy for behaviour engineering is the use of general programming frameworks. GOLOG [10], a logic programming language based on the situation calculus, has been shown to be applicable to robot programming [11]. Some extensions of this language have added additional features such as concurrency [12], continuous change and event-driven behaviours [13], and execution and monitoring systems [14] to make it more suitable for some applications. Frob [15], a functional reactive programming framework embedded in Haskell, was introduced by Peterson et al. for programming robots at various levels of abstractions. For time-critical applications such as space rovers, the reactive model-based programming language (RPML) [16, 17] has been widely adopted. Ziparo et al. [18] presented a new formalism based on Petri Nets, called Petri Net Plans (PNP), for describing robot and multi-robot behaviours. UML statecharts have been applied for the same purpose, especially in soccer robots [19, 20]. An important advantage of the statechart and Petri Net Plans approaches is that agent behaviours can be naturally designed through intuitive graphical tools.

In the case of service robots, applications and tasks are commonly implemented with variants of FSMs. For instance, the behaviour of HERB 2.0 [21] is modelled by the behaviour engine (BE) [22] through three different layers using hybrid state machines. This approach has allowed the service robot HERB2.0 to carry out complex manipulation tasks in human environments. In [23], the TREX framework was used to control the behaviour of a PR2 robot. TREX integrates task planning, discrete and continuous states, and durative and concurrent actions. Under this framework, the robot was able to navigate through an office environment, open doors and plug itself into electrical outlets. Bohren et al. developed a Python library called SMACH [24, 25] based on hierarchical concurrent state machines. SMACH served as the task-level executive of a PR2 robot in a drink fetching and delivery application [24]. For the GRACE robot [26], FSMs were created under TDL to structure the tasks of the AAAI Robot Challenge. In the context of RoboCup@Home, the robots Cosero and Dynamaid [27] build hierarchical state machines to perform the tests of the competition, whereas the robot Caesar [28, 29] used an enhanced version of GOLOG named ReadyLog [30].

In summary, several different strategies and domain-specific languages have been proposed for robot behaviour engineering. Most of these strategies are implemented as extensions of conventional programming languages, mainly in the imperative, specification and functional paradigms. The formalisms used in these strategies are commonly built upon the FSM or extensions of it, thus often translating in a limited expressive power and unwieldy task programming. In contrast, SitLog is a logic programming language especially designed for service robot tasks and constructed on the more expressive formalism of DMs, which exploit the context and history of the task to decide the actions to be executed. In the logical programming paradigm, on its part, ReadyLog is focused on reasoning about actions, while SitLog has the notions of situation and task structure as its main representational objects. Here, we argue that SitLog is particularly suitable for specifying task-level behaviours of service robots in a flexible and concise way, as illustrated in the example in Section 4.4.

3. Task Structure, Situations and Dialogue Models

Service robot tasks can be construed in terms of states, in which the robot is capable of performing some specific set of actions, depending on the input information at the state and possibly on the previous states visited by the robot during the execution of the task. States can be seen from a dual perspective as states in the world and as informational objects in the robot's memory. Here, we say that the robot is *situated* if the memory state corresponds to the state in the world (i.e., the robot is in the state in which "it thinks" it is); in this case, interpretations and actions are performed in context and contribute to the successful completion of the task. On the other hand, if the external information does not match the memory state, the robot is out of context; whenever this is the case, the robot's interpretations and actions need to be directed to place the robot in context again, before proceeding with the task.

An important question from this perspective is how much information needs to be contained within information states in service robot tasks. Here there is a trade-off between expressive power and computational cost: if states have a small amount of information and the next state is determined to a great extent by the external input, computations are efficient, but complex tasks are cumbersome and difficult to model; on the other extreme, if states contain large amounts of information and the determination of the next state requires inference, expressing the task structure may be simple but at a high computational cost. These two extremes correspond to two opposing traditions in interaction and dialogue modelling: one is the use of FSMs, where states have a minimal amount of information expressed by the constant labels on their arcs; this approach is common in dialogue systems and interactive applications, and also in many service robots, as mentioned above. The other is the artificial intelligence tradition in dialogue modelling involving complex inference and the notion of conversational moves that need to be identified dynamically through searching during the execution of the task (e.g., [31] and derived work). In this later case, a state may contain a full "mental state" including a temporal and spatial situation, the history of the task, domain knowledge, the beliefs, desires and intentions of the agent, and even common sense knowledge.

In the present framework we adopt the view that the state's information content consists of the knowledge of the potential actions performed by another agent (the interlocutor) at such a state, with or without communicative intent, in addition to the knowledge of the potential events which can occur in the world at the particular state too; we refer to this knowledge as the *expectations*; this information state contains also the knowledge of the *intentional actions* that the robot needs to perform in case a particular expectation is met. On this basis, we define a *situation* to be an information state consisting of the representation of the set of expectations and potential actions in the context of the task, in addition to the control information required within the task structure. Expectations, actions and next situations can be concrete, but these can also be dynamic and depend on the context and, in the present formulation, situations can be highly abstract knowledge objects supporting large spatial and temporal abstractions. For instance, the robot may be in a *finding situation* in which it has the expectation of seeing an object or failing to see an object at each point in the search path. Another expectation in this situation may be reaching (or failing to reach) a search point; in this situation the robot may navigate and change its spatial position and orientation continuously for a large amount of time, and nevertheless remain in the same *situation*.

From another perspective, expectations are representations of the potential interpretations (i.e., the outputs of perception) that are plausible in a given context within a task structure, and differ from the raw data produced by low-level sensors, that need to be handled by low-level processes in a context independent fashion (i.e., independently of task and domain). A situation is then an abstraction over a collection, possibly large, of perceptions and actions, and tasks can be modelled often through a small set of situations. In addition, while an FSM changes the state when a symbol labelling one of its arcs is consumed (i.e., an event occurs in the world), an agent changes its situation when its set of expectations is changed. Although a change of expectations is often due to an external event, there are many external events that do not change the expectations, and the expectations can also be changed as a result of an inference, which is internal to the agent. Hence, situations are intentional knowledge objects in opposition to FSM states, which are extensional and deal directly with concrete input.

Other kinds of knowledge stored in the robot's databases and/or knowledge bases, like domain specific or general concepts, or even the robot's beliefs, desires and intentions, are not included in the situation in the present framework. These knowledge objects can be retrieved and updated from memory during the interpretation of a situation, but such knowledge is to a large extent context independent and not directly relevant to communication and interaction.

More generally, situations are contextual informational objects and the notion of expectation presupposes that the robot is placed in context in relation to the task. For this, the present notion of situation is restricted to tasks where this condition is met. We refer to tasks that can be construed in this form as practical tasks with their associated practical task and domain-independent hypotheses, extending Allen's corresponding practical dialogues notion and hypotheses [32] as follows: the structure of practical tasks, although complex, is significantly simpler than open human tasks (i.e., practical task hypothesis) and within the genre of practical tasks the structure of the tasks and task management are independent of the task domain and the particular task being performed (i.e., domain-independent hypothesis). We also advance the hypothesis that practical tasks lie somewhere between FSMs, which are too limited in their expressive power, and open search engines which demand an unbounded computational cost. SitLog has been developed to support this notion of situation and task structure. Next, we introduce the language and illustrate the framework with a simple application.

4. Specification of SitLog

4.1. Situations and Dialogue Models

A task `T` is represented in SitLog as a set of DM types `T` = [dm_1, dm_2, ..., dm_n][1]. In turn each DM consists of a set of situation types dm_i = [s_1, s_2, ..., s_n]. During the interpretation process, instances of DM and situations are created and executed dynamically, unfolding a graph that corresponds to the structure of the task. Identifiers and variables within DMs and situations have a local scope and these structures can be thought of as abstract data-types in the standard sense. The arguments of DMs and situations are called by reference and these abstract objects support recursive calls, providing a simple and expressive way to represent complex cyclic behaviours as illustrated below.

Dialogue model and situation names or IDs can be constant or predicate symbols with one or more arguments (e.g., dm1, dm2(X, Y, Z), s1, s3(Z, W)). Following Prolog's notation, identifiers starting with lower case letters stand for constants and predicates, and those starting with capital letters stand for variables. In addition to Prolog's variables, SitLog supports the definition of global and local variables; these are atoms (i.e., constants) with an associated value that can be assigned, modified or recovered within the situation's body and is preserved along the execution of the task in the case of global variables, or within the execution of a dialogue model in the case of local variables, as explained below.

A DM is expressed as a clause with three arguments: an identifier, a set of situations, and a set of local variables, as follows:

```
diag_mod(id, Situations, Local_Vars).
```

In this definition *Situations* stands for the list of situations of the DM; each situation is specified as a list of attribute-value pairs including the following:

```
[
  id ==> ID,
  type ==> Type,
  prog ==> Local_Prog,
  in_arg ==> In_Arg,
  out_arg ==> Out_Arg,
  arcs ==> [
             Expect1:Action1 => Next_Sit,
             Expect2:Action2 => Next_Sit2,
                       ...
             Expectn:Actionn => Next_Sitn
           ],
  diag_mod ==> Diag_Mod_ID
]
```

The symbols at the left of ==> are the attribute names, and the symbols at the right stand for variables or expressions through which the actual expectations, actions, next situations and control information of the situation are expressed. These expressions are evaluated during the execution of the situation, and their values correspond to the concrete interpretations and actions performed by the robot in the situation, including the selection of the next situation, which can be a dynamic choice depending of the context.

Situations have three mandatory attributes: `id`, `type` and `arcs`. The value of the first is an identifier (possibly with a list of arguments, as mentioned above) for each situation within the DM; whenever an instance of the DM

[1] Although we use a Prolog list (a sequence of elements enclosed in brackets: "[e_1, e_2, ..., e_n]") to represent DMs and situations, the order is not considered and such a list is interpreted as a set.

is created and executed, an instance of the first situation in the situation's list is created and executed too, and this corresponds to the initial node of the situation's graph that unfolds during the execution of the DM.

The value of the `type` attribute is an identifier of the information's input modality or combination of modalities (e.g., speech, vision) and has an associated perceptual interpretation algorithm.

The value of the attribute `arcs` is a set of objects of form $\texttt{Expect}_i\texttt{:Action}_i$ `=>` $\texttt{Next\_Sit}_i$ codifying the expectation, action and next situation of the $\texttt{arc}_i$ of the DM. During the interpretation of the situation all expressions representing expectations are evaluated, rendering a concrete value, which should match one of the interpretations made by the robot at the situation. Once an expectation is met, the expressions representing the corresponding action and next situation are evaluated too, and the robot performs the concrete action and moves to the concrete situation that results from this latter evaluation process. In the case that no situation is met, the system executes a recovery protocol that is also specified as a DM; then, a new instance of the situation where the failure occurred is created and executed.

In addition to the interpretation modality types, there are two generic types: `recursive` and `final`. Every DM must contain one or more `final` situations, each standing for a possible conclusion of the task and, consequently, do not have the attribute `arcs`. The type `recursive`, in turn, stands for a situation that contains a full DM within it, so situations of this type abstract over embedded tasks. Situations of type `recursive` must include two attribute-value pairs as follows:

```
[       ...
        type ==> recursive,
        embedded_dm ==> Dialogue_Model_ID,
        ...
]
```

Whenever a recursive situation is executed, the current task is pushed onto the stack, and the initial situation of the embedded model is interpreted. In addition, whenever a final situation is reached, the current DM is popped from the stack, with the identifier of the particular final situation in which the task was concluded. Recursive situations permit structuring tasks at different levels of abstraction, and behaviours are grounded in generic situations which perform actual interpretations and actions in concrete input and output modalities.

There are also four optional attributes: `in_arg`, `out_arg`, `prog` and `diag_mod`. The first two correspond to the input and output arguments of the situation, providing a simple mechanism to pass control and content among situations and DMs, in addition to the arguments of DM and situation IDs. The `in_arg` and `out_arg` arguments are defined in a pipe-like fashion and the value `in_arg` propagates along DMs and situations unless `out_arg` is explicitly assigned a value, which propagates as the value of the `in_arg` of the next actual situation, either within the current DM or through the next DM executed in the task. This pipe mechanism is explicitly handled by SitLog's interpreter and the arguments' values are carried along by the interpretation process even if they are not explicitly stated in the specification of one or more situations.

The value of the `prog` attribute, in turn, is a list of expressions, which we call the `local program`, that is interpreted unconditionally when the situation is created and executed, providing the means for representing control and content information that is local to the situation. The variables within the local program, and also within the arc's attribute, are encapsulated and have a local scope, hence their evaluation does not affect the value of other variables within the situation, even if they have the same name.

Finally, the attribute `diag_mod` permits assigning output values to the DM ID's arguments from within a situation's body. For this the value of this attribute is unified with the DM's identifier when the interpretation of the situation is concluded. Hence, despite the fact that variables in the local program and within the arc are encapsulated within their attributes' scopes and within the situation's body, their values can nevertheless be accessible to the DM and to the task as a whole, as will be illustrated below.

During interpretation, the system keeps track of all concrete expectations and actions performed by the robot, with the corresponding situation, and these objects are assembled in a structured list, which corresponds to the structure of the task. This structure is called the *task history*, which is accessible in all DMs through functions included in the situation's body. The same functional mechanism can be used to access other external knowledge resources during the interpretation of a situation; for instance, to query the robot's knowledge bases, or to use deliberative resources on demand, like planners, theorem-provers or problem-solvers.

In summary, a DM stands for a schematic task and each DM instance unfolds according to the expectations met by the robot along the way, generating a concrete graph whose nodes are the actual situation instances and its arcs correspond to the concrete interpretation and actions performed by the robot during the execution of the task. In this way, a DM specifies an implicit RTN that is explicitly rendered during the execution of the task, and the expressive power of the formalism corresponds at least to a push-down automata, which is in turn equivalent to a context free grammar; in this latter view, recursive situations correspond roughly to variables, modality specific situations to constant symbols and productions to the *rewriting* of a recursive situation by its content, although each recursive situation may stand for several productions, corresponding to the possible situation's paths. In addition, the arcs labelled with functions that have the history as their arguments make the interpretations, actions and next situations sensitive to the context, and hence the extension to F-RTNs.

SitLog's programs are executed by two interpreters that work in tandem. The first is the F-RTN interpreter which interprets DMs and situations and unfolds the recursive graph. For this, the F-RTN interpreter inspects the value of the situation's attributes, selects the expectation that is matched with the current perceptual interpretation, executes the corresponding actions and selects the next situation.

The situation's content is specified through expressions of a functional term-language that we call *L*. The second interpreter of SitLog is the interpreter of this latter language, and it is used systematically by the F-RTN interpreter for evaluating the variables and expressions in the value slot of the attribute-value pairs, during the interpretation of situations. Next we present and illustrate the language *L*.

4.2. The Functional Language L

Expressions of the language *L* are built out of variable symbols, atoms, numerals and predicate symbols (unary, binary, etc.). In addition, there are function symbols with an arbitrary cardinality. The basic operators of *L* include `assign`, `=`, `is`, `->`, which stand for variable assignment, the standard unification operation, Prolog's arithmetic operations (e.g., `X is Y + Z`), the conditional and the standard arithmetic predicates; the language includes a set of standard operators on lists like `append`, `member`, etc., as well as a number of operators for abstract data-type operations like `push` and `pop`; additional operators can be included as needed by simply declaring the operator in the operators list and stating the corresponding function. Finally, the language includes the binary predicate `apply` whose arguments are a function and a list of arguments. Expressions are built compositionally in the standard way.

For instance, the following are all well-formed expressions of *L*: `X`; `a`; `3`; `p(a)`; `p(a, 2, X)`; `q(X, Y)`; `assign(X, p(a, X))`; `X = q(a, b)`; `X is 1 + Y`; `X == Y`; `[a, p(a), p(a, b), p(X), p(a, X), p(X, Y)]`; `(p(a), p(X))`; `[a, p(a), p(X, Y), [a, b], c])`; `apply(f(X), [3])`; `apply(g(X, Y), [1, apply(f(X), [3])])`, where symbols starting with lower case letters are constant or predicate names, and those starting with capital letters are standard Prolog variables. A value is assigned to a local or global variable through expressions using the operators (`get`, `set` and `inc`); these expressions return also a value which is the variable's value itself once the assign operation has been performed; for instance, if `count` is defined as a local or global variable, `set(count, 0)`, `get(count, X)` and `inc(cont, Y)` are well-formed expressions of *L* which produce the value of `count` that results form evaluating the expression.

The interpretation of expressions of *L* is defined compositionally, and the value of a composite term is a function of the values of its parts and its syntactic structure in the standard way. So expressions are evaluated by the interpreter of *L* in relation to the programming environment, which is constituted by input and output arguments, the set of local and global variables and the task history at the current instances of DMs and situations. The interpreter reduces an expression to its basic form which can be a variable, a constant, a grounded predicate, a predicative function with one or more variables, and also an `apply` term with its corresponding function and arguments. The source code of this interpreter is available in the reference included in Section 7.

In the context of the situation, grounded terms stand for concrete expectations, actions or next situations; for instance, a constant or a grounded proposition may stand for a specific proposition expressed by the interlocutor, or for a concrete (fully determined) action, or for a specific next situation. Predicative functions resulting from the evaluation process may stand for a predication whose variables need to be instantiated by the perceptual interpreter out of the recognition of intentions expressed by the interlocutor, or out of the interpretation of natural but expected events in the world. The actions resulting from the evaluation can also be concrete and can be performed directly by the robot, or predicative functions with open variables that need to be further specified before these are sent to the actual rendering mechanisms. The next situations can be expressed by a constant or a grounded predicate (i.e., a situation with arguments), but can also be stated through functions, whose evaluation results in the actual next situation. In addition, the history can be accessed through a special predicate defined within a function's body, and it determines in part the function's value. Additionally, other deliberative resources and memory can be accessed through a similar mechanism.

4.3. Diagrammatic Notation

The graph of situation types have a diagrammatic representation, where nodes stand for situations and arcs are labelled with pairs of form $\alpha{:}\beta$ standing for expectations and actions respectively, and the boundaries of a DM are depicted by dotted lines, as in Figure 1. In addition to standard finite state machine graphs, the next situation may be determined dynamically through a function from the task history (i.e., the discourse context) into the set of situations; in this case, the corresponding output arc forks at the thick dot labelled with the selection function's name (e.g., *h*), out of which there are directed links to the possible next situations. In this notation, information flow is depicted by continuous thin lines, while control flow by thicker lines. In particular, recursive situations embedding subordinated DMs have an output link to the first situation of the embedded DM, and include a number of thick dots depicting the return entry points when the execution of the embedded DM is concluded; there is also a control link from each final situation of the embedded DM to the corresponding return entry point, and from this to the situation that is executed next to the recursive one in the embedding DM.

For instance, the diagram in Figure 1 illustrates a task with two DMs (i.e, *md_main* and *md_sub*), where *md_main* has the situations *is*, *rs* and *fs* of types *speech*, *recursive* and *final* respectively. The situation *is* has two output arcs, one that cycles on it if the expectation $e1$ is met, and the second which is labelled with the expectation f and the action g that goes either to itself or to the recursive situation depending on the value of the function h labelling the junction dot. In this illustration, f, g and h stand for functions that are evaluated dynamically in relation to the task history and become concrete expectations, actions

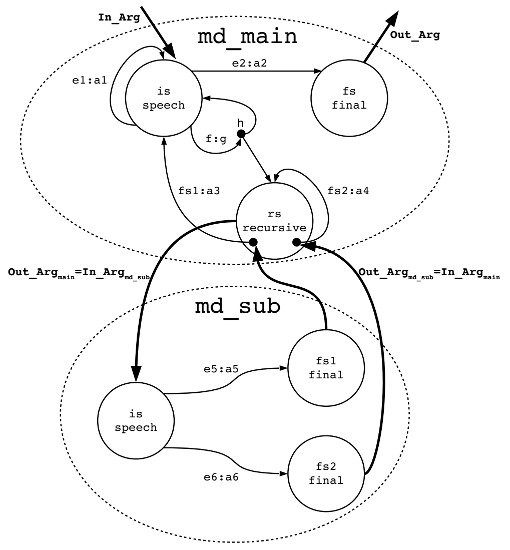

Figure 1. Graphical representation of the dummy application.

and next situations each time the corresponding arc is transversed. The embedded DM *md_sub*, in turn, also has three situations, two of which are of type *final*; there is also a control link from each of these to an entry point of the *md_main* DM, and from this to the its corresponding next situation; so, if the embedded task finishes in the situation *fs1*, the recursive situation is resumed at its left entry point and the next situation in the *md_main* DM is *is*. In this way, the *expectations* of a recursive situation are the final situations of its embedded DM. Figure 1 also illustrates the flow of information between DMs and situations through their arguments, where the `out_arg` of one DM or situation is the `in_arg` of the next DM or situation executed in the actual task.

We also consider that a situation may have more than one instance; this is the case for situations with the same identifier but with different parameters or parameters values. This expressivity is useful for specifying modular or recursive behaviour (i.e., when one situation codifies the basic case and another the inductive pattern), where an arc labelled with an arbitrary α:β pair may be followed by a number of disjunctive situations. This pattern also has a diagrammatic representation analogous to the thick dot depicting a disjunction of next situation but in this latter case, the disjunction is depicted with a small circle including the symbol $\vee$ and the links out of it are labelled with a symbol or a string standing for the condition that is met when such link is transversed. This case is illustrated in Figure 2, where a DM is constituted by a situation *a* that is followed by two instances of situation *b* via a disjunction symbol; in this case, the link from the disjunction to the left instance of *b* is labelled by the string *cond1* and the right instance by *cond2*; in the illustration, whenever *cond2* is met the system is engaged in a cyclic behaviour, but when *cond1* is met the computation comes to an end.

4.4. An Example Program in SitLog

In this section we illustrate SitLog with a program implementing the find task in which a robot searches for a person, an object or a list of objects through a discrete path until such entity or list of entities is found or the path

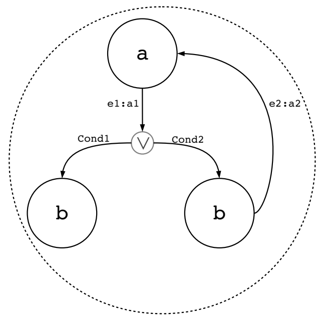

Figure 2. Diagrammatic representation of recursive DM structures.

is exhausted; for this definition we assume that the robot is already in the initial position of the search path. This is a common behaviour required in many service robot tasks, like the *Cocktail Party*, *Clean it Up*, *Emergency Situation* and *Restaurant* tests of the *RoboCup@Home Competition* (Rule book 2013). This behaviour is quite complex and can be structured in a hierarchy of DMs as shown in Figure 3. The *find* DM uses the *scan* DM to make a number of horizontal observations at each path position and move to the next position until the object is found or the path is exhausted. The *scan* DM, in turn, uses the *tilt* DM to make a number of vertical observations at each of the robot's neck scanning positions using the *see* DM at each tilt orientation. Finally *see* selects the kind of object sought and *see_object*, *see_face*, *detect* and *recognize* make the basic observations. Each DM in the hierarchy includes the main logic of the behaviour at the corresponding level, the specification of the perception and action capabilities that are relevant at that level, and the specification of one or more recursive situations embedding the DM codifying the next level behaviour down the hierarchy until the bottom DMs which implement the basic perceptions and actions.

We illustrate SitLog with the actual definition of the *find* DM. We first introduce the program through its diagrammatic representation in Figure 4. The DM has six situations altogether: the initial situation *scan* is a recursive situation that uses the embedded *scan* DM. The return entry points of this situation are *fs_found* and *fs_not_found* corresponding to whether or not the sought entity was found in any of the observations made at the current position of the robot. In the former case, SitLog's interpreter selects the final situation *fs_found* and terminates the task with the success status. In the latter, the interpreter selects one instance of the *search* situation. In case the situation *search* stating that the task has been exhausted is selected (the left instance), the system moves the *fs_error* final situation and the execution of the DM is ended. On the other hand, if there are more positions to explore, the system moves to one instance of the *scan* situation, depending of whether or not is able to reach the next observation position. In the first case, the *scan* DM is executed again one position down the path, and in the latter the task is ended with the corresponding status error.

The actual SitLog code for this DM is shown in Listing 1. The specification of the six situations is declarative and corresponds quite directly to the diagram. The arguments of the *find* DM are as follows: (1) the type of entity to be

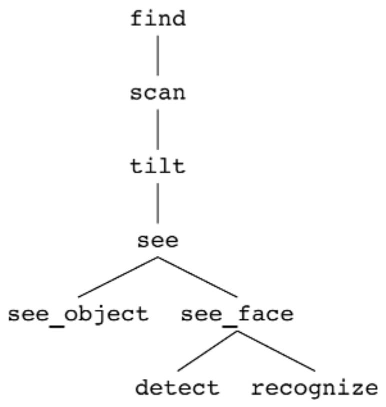

Figure 3. Abstract decomposition of find task.

found: a person or an object; (2) the identifier of the sought person or list of objects; in the case where the sought entity is not specified, the MD returns the first object or list of objects found in the exploration process; (3) a list of search positions (i.e, the search path); (4) the list of scan horizontal orientations that the robot needs to inspect at every search position; (5) the list of vertical tilt orientations that the robot needs to inspect at each scanning orientation; (6) a mode of observation in the case where the entity sought is a person: memorize or recognize; (7) the list of objects found that were specified as sought objects, with their parametric information (e.g., position and pose in relation to the current search position); (8) the list of search positions that remained unexplored when that find task was accomplished; so, if the robot is engaged in the search of multiple objects, it can resume that task at its current position after the first successful observation; and (9) the status of the task (e.g., ok, not_found, not_detected, empty_scene, move error, etc.), reporting the successful completion of the task or the status of the last observation made in the search process. The code also shows the arguments' pipe equating the `out_arg` of a situation with the `in_arg` of the next actual situation, and these arguments must have exactly the same form. Situations with a single output arc have the corresponding expectation labelled `empty`; also, if there is no action in an arc its corresponding position is also labelled `empty`. The specifications of the rest of the DMs in Figure 3 is stated along similar lines.

The format of a call to the *find* DM is as follows:

```
find(object, ['lemon tea',gatorade,pepsi],
         [pt1,pt2,pt3,pt4], [left,right],
         [-30,-15], _, Found_Objects,
         Rest_Positions, Status).
```

This DM call specifies that the robot should search for three objects through a path constituted by four positions, and in each position must look at the current orientation of the neck (as a default convention) and also at the left and right. In addition, at each neck orientation it must look at the current tilt orientation, and also at -30 and -15 vertical degrees. In the execution of the path the robot searches in these three dimensions until a scene containing at least one of the sought objects is seen, and returns all the objects in the scene that are specified as been sought in the variable `Found_Objects`, the positions yet to be explored

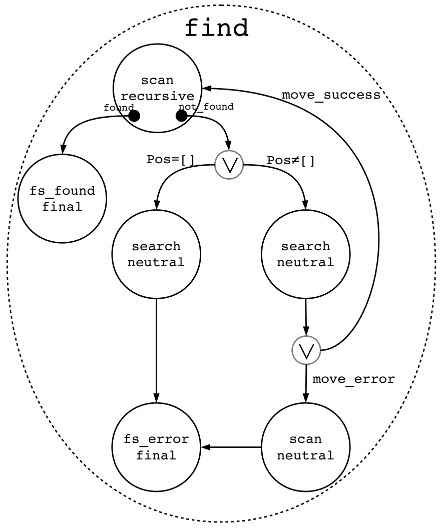

Figure 4. Diagrammatic representation of find DM.

in `Rest_Positions` and the status of the task which is `success` or the status error of the last observation made in the search process. The sixth argument is unspecified as it is only used to state the mode of observation (i.e., memorize or recognize) in the case where the sought object is a person.

In addition, a SitLog application, like the tests of the RoboCup Competition, requires that the global specification of all expectation and actions names (e.g., the `navigate` predicate) are included in all DMs and situations, and also the specification of the global variables and functions. The expressions within the arc attributes belong to the language L and are evaluated by the corresponding interpreter each time an arc is interpreted. Although the present example does not use composite expressions or functions, local programs or local variables, the construction of a full application may require the use of this expressive power. The full code of all DMs in Figure 3, which does make use of these expressive devices, is available at http://golem.iimas.unam.mx/sitlog/sitlog.tar.gz.

5. Dialogue Models and the Cognitive Architecture

Within the present framework we have developed the interaction-oriented cognitive architecture (IOCA), which is centred on SitLog's interpreter as illustrated in Figure 5. IOCA has three main layers corresponding to recognition/rendering at the bottom level, interpretation/action-specification at the middle and expectation/action-selection at the top processing level. These three layers are involved in the main communication cycle. The bottom layer of the architecture consists of the speech and vision recognition modules, for instance, that translate the external information into the corresponding internal codes on the input side, and of the actual realization devices for navigation and manipulation behaviours on the output. The middle layer on the input side corresponds to the interpreter that matches the expectations of the current situation, which are passed top-down from SitLog's interpreter, with the output of the recognition systems, which

```
diag_mod(find(Kind, Entity, Positions, Orientations, Tilts, Mode, Found_Objects, Remaining_Positions, Status),

%First argument: List of situations
[
% Scan situation: already in the scan position
 [id ==> scan(Entity, Positions, Orientations, Tilts, Found_Objects, success),
  type ==> recursive,
  out_arg ==> [Entity, Scan_Parameters, Positions, Scan_Status],
  embedded_dm ==> scan(Kind, Entity, Orientations, Tilts, Mode, Scan_Parameters, Scan_Status),
  arcs ==> [fs_found:empty => fs_found,
            fs_not_found:empty => search(Entity, Positions, Orientations, Tilts, Found_Objects)]
 ],

% Error reaching next position
 [id ==> scan(_, Positions, _, _, _, move_error),
  type ==> neutral,
  out_arg ==> [_, _, Positions, move_error],
  arcs ==> [empty:empty => fs_error]],

% Search situation 1: no more search points
 [id ==> search(_, [], _, _, _),
  type ==> neutral,
  arcs ==> [empty:empty => fs_error]],

% Search situation 2: move to the next search point and scan
 [id ==> search(Entity, [Next_Position|Rest_Positions], Orientations, Tilts, Found_Objects),
  type ==> neutral,
  arcs ==> [empty:navigate(Next_Position, true, Status_Move)
                          => scan(Entity, Rest_Positions, Orientations, Tilts, Found_Objects, Status_Move)]
 ],

% Final Situation
 [id ==> fs_found,
  type ==> final,
  in_arg ==> [Entity, Found_Objects, Positions, Final_Status],
  diag_mod ==> find(_, Entity, _, _, _, _, Found_Objects, Positions, Final_Status)
 ],

% Final Situation
 [id ==> fs_error,
  type ==> final,
  in_arg ==> [Entity, Found_Objects, Positions, Final_Status],
  diag_mod ==> find(_, _, _, _, _, _, _, Positions, Final_Status)
 ]
], % End situation list

% List of Local Variables
[]
). % End Find Task DM
```

Listing 1. SitLog's specification of find behaviour.

proceed bottom-up, and produces the representation of the expectation that is met in the situation. Structural processes are defined at this level; for instance, in the case of linguistic interpretation, the output of speech recognition is a text string, and the interpreter includes the parser which produces a syntactic and semantic representation. However, perceptual interpretation depends also on the context set by the expectations, and also on the interaction history, and the output of the interpreter is then a contextualized representation of the intention expressed by the interlocutor or a representation of an expected natural event.On the output side, the action scheme selected by SitLog is fully specified and realized by the corresponding rendering devices.

In actual implementations there is a particular interpreter for each situation type defined at the level of SitLog. Interpreters involve one or more modalities, and are also relative to a particular perspective or aspect of the world; for instance, there is visual interpreter for face recognition and another for object recognition, regardless of the fact that these involve the visual modality. However, the output of perceptual interpretation is a propositional representation already independent of the particular vision algorithms and associated data structures. On the output side SitLog selects the action schemes, which are propositional and modality independent representations, that need to be specified with the relevant modality specific information and realized by the corresponding rendering devices.

IOCA has also a reactive behavioural cycle that establishes a link between recognition and rendering devices through the autonomous reactive systems, which are involved in the robot's reactive behaviour, as shown by the corresponding path in Figure 5. This cycle is embedded within the main communication cycle as it does not involve interpretations and the construction of fully fledged representations; this cycle involves mostly signal processing mechanisms that take place much faster than the communication cycle. IOCA also involves a coordinator between communication and reactive behaviour which permits that these two cycles proceed concurrently.

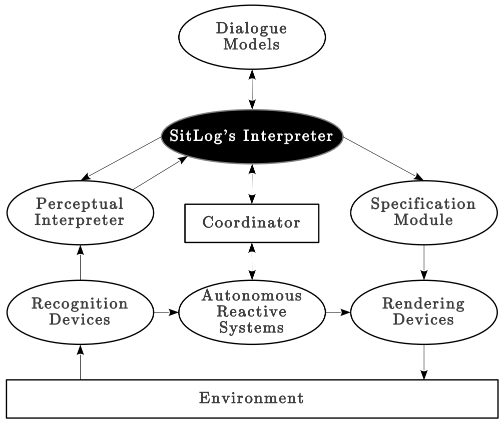

Figure 5. The interaction-oriented cognitive architecture.

6. Practical Tasks and the Golem-II+ Robot

Service robots have the purpose of supporting people to perform common daily life tasks. In practical settings, such tasks have the purpose of achieving one or more goals through the execution of a number of actions, possibly in a previously unspecified order. Simple examples of these tasks are the tests of the *RoboCup@Home Competition* (Rule book 2013), like the *Cocktail Party*, *Clean it Up*, *Emergency Situation*, *Restaurant* and *General Purpose Service Robot*. Although these kinds of tests are defined as bench-marks for demonstration purposes, they illustrate the potential settings in which service robots will be useful in the future: simple scenarios involving a few designated people and objects, actions and events, limited in time and space, where the robot needs to achieve a few specific goals, while collaborating with people. In our terminology, these are instances of practical tasks.

For the construction of this kind of application our methodology distinguishes two main kinds of DMs: those targeted to model the structure of the practical task as a whole, like serving as a waitress in a restaurant, cleaning a room or assisting clients in a supermarket, and those directed to model general capabilities that can be used recurrently within different tasks, like learning a person's face, learning his or her name, finding people in a room, finding, grasping and delivering drinks, etc., which need to be coordinated in order to accomplish the goals of the task successfully. These latter kinds of actions are generic behaviours that need to be defined independently of task and domains, and constitute the library of behaviours that can be used systematically by full applications, like the find behaviour and their associated DMs which constitute a particular library. SitLog and IOCA have been developed with the purpose of supporting the high-level declarative definition of practical tasks in terms of a library of behaviours.

In order to test the present framework, SitLog and IOCA have been implemented in the Golem-II+ robot. Next, we list the main functionalities used by SitLog behaviours through IOCA and their associated devices and algorithms:

1. Face detection and recognition are carried out by standard OpenCV functions, employing the Viola-Jones method [33] for detection and Eigenfaces for recognition [34] .
2. Visual object recognition is performed using MOPED [35]; this framework uses different images of an object to create a 3D model based on SIFT [36].
3. Person tracking is performed via a module based on the OpenNI driver.
4. Speech recognition is carried out via a robust live continuous speech recognizer based on the PocketSphinx software [37], coupled with the Walt Street Journal (WSJ) acoustic models for English speaking users, and the DIMEx100 acoustic models [38] for Spanish speaking users. Hand-crafted language models for each task are able to be switched on, depending on the context of the dialogue. Noise filtering is carried out by estimating the noise spectral components via a quantile-based noise estimator [39] and subtracting it from the speech signal in a non-linear form.
5. Speech synthesis is produced with the Festival TTS package.
6. User localization is carried out via audio using a triangular microphone array audio-localization system [40].
7. Route planning is carried via the Dijkstra algorithm [41] over a hand-crafted topological map of the environment.
8. Obstacle evasion is carried out via the nearness diagram [42] for long-range obstacles, and smooth nearness diagram [43] for close-range obstacles.
9. For object manipulation, two in-house 4-degrees-of-freedom robotic arms are used. Arm control is carried out by estimating a constrained polar-coordinate plane and via coordinated inter-motor movement. The grips are equipped with infrared sensors, which reactively make adjustments when taking objects, overriding vision errors.

7. Conclusions

In this paper we have presented SitLog: a programming language and environment for the specification and interpretation of behaviour engineering for service robot tasks in a simple and flexible way. The formalism has an associated diagrammatic notation that facilitates greatly the design and specification of complex tasks. The core computational mechanism consists of two interpreters working in tandem, one for interpreting the structure of the task and the other for interpreting content and control information. These two interpreters are implemented in Prolog and programs in SitLog follow closely the Prolog notation, supporting the definition of applications in a declarative and compact form.

We have also introduced the notion of situation which is an information state containing the expectations and potential actions of a robotic agent in the context of the task structure. Situations are related in a recursive directed graph giving rise to the notion of DMs, which can also be seen as abstract behavioural models. Alternatively, DMs can be seen as schematic and parametric plans to achieve complex tasks. In addition, we have presented the notion of practical tasks, and the practical task and domain independent hypotheses, and suggest that the tests of the RoboCup@Home Competition and similar demonstration scenarios are instances of practical tasks.

SitLog permits defining a library of general robot behaviours, like finding people or objects in a room, interpreting speech input, navigating to a designated place, coordinating visual object recognition and manipulation, etc., that can be used by different applications on demand, in addition to the main DMs representing the structure of a service task. This modularity is particularly useful for the implementation of general service robots, which need to assemble task structures dynamically, by the composition and interpretation of complex DMs out of the basic DMs stated in advance. This functionality can also be seen as the dynamic construction and execution of parametric plan schemes out of the basic schematic plans.

These notions permitted the definition of the interaction-oriented cognitive architecture (IOCA). SitLog is at the heart of this architecture and relates the flow of perception and intentional actions, articulating the robot's main communication cycle, which subsumes reactive behaviour, on the one hand, and manages symbolic representations and deliberative resources, on the other. SitLog is also task and domain independent and can be easily ported to different robotic architectures and operating systems.

We have also presented the specification and interpretation of complex robotic behaviour, illustrating the expressive power of the formalism.In addition, we have implemented the full set of tests of the *RoboCup@Home Competition* (Rule book 2013) in the Golem-II+ robot, and we have found no limitations on the expressive power of SitLog for this purpose.

The source code of SitLog's interpreter together with the DM library of generic behaviours presented in this paper are available at http://golem.iimas.unam.mx/sitlog/sitlog.tar.gz. A video of Golem-II+ executing a RoboCup@Home task fully written in SitLog can also be seen at http://youtu.be/99XhhEkyIz4.

8. Acknowledgements

We thank the support of the members of the Golem Group who participated in the development of the Golem-II+ robot as follows: Mario Peña, Ivan Sánchez, Arturo Rodríguez, Joel Durán, Mauricio Reyes, Hernando Ortega, Esther Venegas and Varinia Estrada. We also acknowledge the support of grants CONACYT's 81965 and 146636, PAPIIT-UNAM's IN-115710 and IN-107513, Secretaría de Economía FP2009-385 and ICyT-DF-PICCO12-024. The last author is supported by ICyT-DF.

9. References

[1] Luis Pineda, Ivan Meza, and Lisset Salinas. Dialogue model specification and interpretation for intelligent multimodal hci. In Angel Kuri-Morales and Guillermo Simari, editors, *Advances in Artificial Intelligence – IBERAMIA 2010*, volume 6433 of *Lecture Notes in Computer Science*, pages 20–29. Springer Berlin / Heidelberg, 2010.

[2] William A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606, 1970.

[3] Rodney A. Brooks. The behavior language; user's guide. Technical report, Massachusetts Institute of Technology, 1999.

[4] Rodney A. Brooks. A robust layered control system for a mobile robot. Technical report, Cambridge, MA, USA, 1985.

[5] Kurt Konolige and Motion Control. Colbert: A language for reactive control in sapphira. Technical report, SRI International, 1997.

[6] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Proc. of the Conference on Intelligent Robots and Systems*, 1998.

[7] Martin Lotzsch, Max Risler, and Matthias Jüngel. XABSL - A pragmatic approach to behavior engineering. In *Proc. of IEEE/RSJ International Conference of Intelligent Robots and Systems*, pages 5124–5129, 2006.

[8] Steve Tousignant, Eric Van Wyk, and Maria Gini. An overview of XRobots: A hierarchical state machine-based language. In *Proc. of the Workshop on Software Development and Integration in Robotics*, 2011.

[9] Thijs Jeffry de Haas, Tim Laue, and Thomas Röfer. A scripting-based approach to robot behavior engineering using hierarchical generators. In *Proc. of the International Conference on Robotics and Automation*, pages 4736–4741, 2012.

[10] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[11] Frank Dylla, Alexander Ferrein, Er Ferrein, and Gerhard Lakemeyer. Acting and deliberating using golog in robotic soccer - a hybrid architecture. In *Proc. of the International Cognitive Robotics Workshop*, 2002.

[12] Giuseppe de Giacomo, Yves Lespérance, and Hector J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[13] Henrik Grosskreutz and Gerhard Lakemeyer. cc-golog: Towards more reaslitic logic-based robot controllers. In *Proc. of the AAAI Conference on Artificial Intelligence*, 2000.

[14] Dirk Hähnel, Wolfram Burgard, and Gerhard Lakemeyer. Golex–bridging the gap between logic (golog) and a real robot. In Otthein Herzog and Andreas GÃijnter, editors, *Advances in Artificial Intelligence*, volume 1504, pages 165–176. Springer Berlin / Heidelberg, 1998.

[15] John Peterson, Greg Hager, and Paul Hudak. A language for declarative robotic programming. In *Proc. of the International Conference on Robotics and Automation*, 1999.

[16] Brian C. Williams, Michel D. Ingham, Seung H. Chung, and Paul H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 91(1):212–237, 2003.

[17] Brian C. Williams, Michel D. Ingham, Seung Chung, Paul Elliott, Michael Hofbaur, and Gregory T. Sullivan. Model-based programming of fault-aware systems. *AI Magazine*, 24(4), 2004.

[18] Vittorio A. Ziparo, Luca Iocchi, Daniele Nardi, Pier F. Palamara, and Hugo Costelha. Petri net plans: a formal model for representation and execution of multi-robot plans. In *Proc. of the International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 1, pages 79–86, 2008.

[19] Oliver Obst. Specifying rational agents with statecharts and utility functions. In *RoboCup 2001: Robot Soccer World Cup V*, pages 173–182, 2001.

[20] Jan Murray. Specifying agent behaviors with uml statecharts and statedit. In *RoboCup 2003: Robot Soccer World Cup VII*, pages 145–156, 2003.

[21] Siddhartha Srinivasa, Dmitry Berenson, Maya Cakmak, Alvaro Collet Romea, Mehmet Dogar, Anca Dragan, Ross Alan Knepper, Tim D Niemueller, Kyle Strabala, J Michael Vandeweghe, and Julius Ziegler. Herb 2.0: Lessons learned from developing a mobile manipulator for the home. *Proceedings of the IEEE*, 100(8):1–19, 2012.

[22] Tim Niemuller, Alexander Ferrein, and Gerhard Lakemeyer. A lua-based behavior engine for controlling the humanoid robot nao. In *Proc. of the RoboCup Symposium 2009*, pages 240–251, 2009.

[23] Sachin Chitta, E. Gil Jones, Matei Ciocarlie, and Kaijen Hsiao. Mobile manipulation in unstructured environments: Perception, planning, and execution. *IEEE Robotics and Automation Magazine*, 19(2):58–71, 2012.

[24] Jonathan Bohren, Radu Bogdan Rusu, E. Gil Jones, Eitan Marder-Eppstein, Caroline Pantofaru, Melonee Wise, Lorenz Mösenlechner, Wim Meeussen, and Stefan Holzer. Towards autonomous robotic butlers: Lessons learned with the pr2. In *Proc. of the International Conference on Robotics and Automation*, pages 5568–5575, 2011.

[25] Jonathan Bohren. ROS SMACH package.

[26] Reid Simmons, Dani Goldberg, Adam Goode, Michael Montemerlo, Nicholas Roy, Brennan Sellner, Chris Urmson, Magda Bugajska, Michael Coblenz, Matt Macmahon, Dennis Perzanowski, Ian Horswill, Robert Zubek, David Kortenkamp, Bryn Wolfe, Tod Milam, Metrica Inc, and Bruce Maxwell. Grace: An autonomous robot for the AAAI robot challenge. *AI Magazine*, 24:51–72, 2003.

[27] J. Stuckler, D. Holz, and S. Behnke. Robocup@home: Demonstrating everyday manipulation skills in robocup@home. *IEEE Robotics Automation Magazine*, 19(2):34–42, 2012.

[28] Stefan Schiffer, Alexander Ferrein, and Gerhard Lakemeyer. Caesar: an intelligent domestic service robot. *Intelligent Service Robotics*, 5(4):259–273, 2012.

[29] Stefan Schiffer, Alexander Ferrein, and Gerhard Lakemeyer. Reasoning with qualitative positional information for domestic domains in the situation calculus. *Journal of Intelligent and Robotic Systems*, 66(1-2):273–300, 2012b.

[30] C. Fritz. Integrating decision-theoretic planning and programming for robot control in highly dynamic domains. Master's thesis, RWTH Aachen University, Knowledge-based Systems Group, Aachen, Germany, 2003.

[31] James F. Allen and C. Raymond Perrault. Analyzing intention in utterances. *Artificial Intelligence*, 15(3):143–178, 1980.

[32] James F. Allen, Donna K. Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. Toward conversational human-computer interaction. *AI MAGAZINE*, 22(4):27–38, 2001.

[33] Paul Viola and Michael J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.

[34] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[35] Alvaro Collet, Manuel Martinez, and Siddhartha S. Srinivasa. The MOPED framework: Object recognition and pose estimation for manipulation. *The International Journal of Robotics Research*, 30:1284–1306, 2011.

[36] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[37] David Huggins-Daines, Mohit Kumar, Arthur Chan, Alan W. Black, Mosur Ravishankar, and Alex I. Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 1, 2006.

[38] Luis A. Pineda, Hayde Castellanos, Javier Cuétara, Lucian Galescu, Janet Juárez, Joaquim Listerri, Patricia Pérez, and Luis Villaseñor. The corpus dimex100: transcription and evaluation. *Language Resources and Evaluation*, 44(4):347–370, 2010.

[39] Volker Stahl, Alexander Fischer, and Rolf Bippus. Quantile based noise estimation for spectral subtraction and wiener filtering. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1875–1878, 2000.

[40] Caleb Rascón, Héctor Avilés, and Luis A. Pineda. Robotic orientation towards speaker for human-robot interaction. In *Proc. of the Ibero-American Conference on Advances in Artificial Intelligence*, pages 10–19, 2010.

[41] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[42] Javier Minguez and Luis Montano. Nearness diagram (nd) navigation: Collision avoidance in troublesome scenarios. *IEEE Transactions on Robotics and Automation*, 20:2004, 2004.

[43] Joseph W. Durham and Francesco Bullo. Smooth nearness-diagram navigation. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 690–695, 2008.