Robot Learning Language (RoLL)
Reference Manual

Alexandra Kirsch

October 5, 2011

Contents

Chapter 1

General

1.1 Introduction

Robots performing complex tasks in changing, everyday environments and required to improve with experience must continually monitor the way they execute their routines and revise them if necessary. To meet this challenge we propose to make learning an integral part of the control program by providing a control language that includes constructs for specifying and executing learning problems.

The Robot Learning Language (RoLL) makes learning tasks executable within the control program. It allows for the specification of complete learning processes including the acquisition of experience, the execution of learning algorithms and the integration of learning results into the program. The experience acquisition in RoLL is designed in a way that experiences can be defined outside the primary control program, using hybrid automata as a tool for declaratively specifying experience and anchoring it to the program. The rich experience concept enables convenient abstraction and an economic use of experiences. RoLL's design allows the inclusion of arbitrary experience-based learning algorithms. Upon the completion of the learning process RoLL automatically integrates the learned function into the control program without interrupting the program execution.

RoLL is based on the plan language CRAM-PL[1], which is written in LISP (using Steel Bank Common Lisp). In this document I assume familiarity with LISP and CRAM-PL.

For background information on RoLL please refer to [3] or to my thesis [2]. The next section contains a complete example of how to use RoLL in a program for collecting experiences and learning. Chapter 2 contains the complete reference of RoLL commands. Chapter 3 complements the reference with detailed examples for the usage of some specific constructs. Chapter 4 provides details on available extensions and how custom extensions can be specified.

1.2 Comprehensive Example

The following example of a complete learning problem including experience specifications assumes that the RoLL core language has been supplied with the learning problem class, the experience

[1] `http://www.ros.org/wiki/cram_pl`

class for storing experiences in a database, and a learning system for regression tree learning. The definitions of these parts are explained in Chapter 4.

The problem to be learned is a time prediction model for the robot's navigation behavior.

1.2.1 Raw Experience Acquisition

For accessing the time needed for a navigation task, the navigation plan must be observed to acquire a raw experience like the one defined in Listing 1.1. The experience automaton is anchored to the control program by defining the activity of the plan `at-location` as an episode (line 3). For describing the task at hand, the start and goal positions must be known (lines 5, 6). In addition, the time stamps of the starting and stopping time points are recorded (lines 4, 7). The goal position is stored in the routine description, which is taken from the designator, which is provided to the `at-location` plan (line 6).

Listing 1.1 Raw experience definition for navigation time model.

```
1 (roll:define-raw-experience navigation-time-exp
2   :specification (nil
3                  :task (:plan at-location)
4                  :begin ( (timestep (get-universal-time))
5                           (start-pose robot-location)
6                           ; assuming that variable robot-location contains
7                           ; the value of the current robot pose
8                           (goal-pose (:desig-value 'pose)) )
9                  :end ( (timestep (get-universal-time)) )))
```

The experience can be identified by its name `navigation-time-exp`. All experiences are stored in a global hash table and can be retrieved with the function `getgv` (get global variable). For observing instances of the specified experience, the generated plan `acquire-experiences-navigation-time-exp` is called.

For learning a model of the navigation routine, both passive and active experience acquisition are conceivable. Here we demonstrate the difference in terms of definition.

Passive Experience Acquisition

Passive experience acquisition means that the robot's actions are not tailored to the learning problem. Rather, the robot performs standard activities. In this case, we use a plan for setting the table. The two processes — observation of the experience and the plan for table setting — are executed in parallel.

```
(pursue
  (acquire-experiences-navigation-time-exp)
  (set-the-table-plan))
```

The `acquire-experiences-...` plan can be used anywhere in the program. However, one motivation for using RoLL is not to modify the program for learning. Therefore, it is best to run the observation process in parallel to the top-level control program. It identifies the interesting parts of the execution automatically.

Active Experience Acquisition

Setting the table involves few navigation commands. Therefore the overall state space needed for a full modeling of the navigation routine can be observed only partially. A different approach is to generate the navigation tasks to be performed with the means provided by RoLL.

Listing 1.2 CRAM-PL plan for active data acquisition.

```
1  (pursue
2    (cram-plan-library:maybe-run-process-modules)
3
4    ; Data Acquisition
5    (acquire-experiences-navigation-time-exp)
6
7    ; Program
8    (seq
9      (roll:with-parameter-samples (:parameters ( (x :random :min -3.0 :max -0.5 :samples 5)
10                                                  (y :random :min -2.5 :max 0.0)
11                                                  (az :random :min (* -1 pi) :max pi) )
12                                  :relation (:indep x y az))
13       (let ( (goal-pose (tf:make-pose-stamped "/map" 0.0
14                                                (tf:make-3d-vector x y 0.0)
15                                                (tf:euler->quaternion :az az))) )
16         (with-designators ( (loc-desig (location `((pose ,goal-pose)))) )
17           (cram-plan-library:at-location (loc-desig))))
18       (sleep 5))))
```

The control program (see Listing 1.2) navigates to different positions in the kitchen (lines 13–17). The parameters defining the goal positions (`x`, `y`, `az`) are generated randomly (lines 9–12).

The problem generator of RoLL is only one way to implement active experience acquisition. Any function returning the necessary parameters — possibly taking into account existing experiences and the robot's overall reward function — can be used for this purpose. For active experience acquisition as it is done in most current systems, the RoLL problem generator is a simple, yet powerful tool for defining and using different parameterizations for control programs.

1.2.2 Experience Abstraction

The position values in the raw experience are absolute values. We make the simplifying assumptions that objects in the kitchen don't affect the navigation time and that the robot has means to decide if a location is accessible. With these premises, learning with absolute positions is not advisable, because navigation tasks that are only shifted or rotated on the 2D plane are treated as different cases. Therefore, we use the abstraction depicted in Figure 3.2 on page 22, which uses the distance of the two points and the angles relative to the connecting line between the two points.

In the definition in Listing 1.3, we use the construct `with-binding` for calculating intermediate values (lines 3–7). Its syntax and functionality correspond to that of the LISP `let*`. The binding of intermediate values is not necessary, but makes the definition better readable. The auxiliary values include the start and goal point from the raw experience, the duration of the navigation task, and the angle of the connecting line of the two points. Based on these values, the distance of the two points and the normalized orientations are calculated (lines 10–16).

When an instance of the raw experience is recorded, it must be processed further immediately. In contrast, we write the abstract experiences to a database, which must be specified in the experience

Listing 1.3 Abstract experience and conversion specification for navigation time model.

```
1  (roll:define-abstract-experience nav-time-abstract-exp
2    :parent-experience navigation-time-exp
3    :specification (roll:with-binding ( (p1 (:var start-pose :begin))
4                                        (p2 (:var goal-pose :begin))
5                                        (timediff (- (:var timestep :end)
6                                                     (:var timestep :begin)))
7                                        (offset-angle (angle-towards-point p1 p2)) )
8                     (roll:with-filter (plusp timediff)
9                       (nil
10                        :begin ( (dist (tf:v-dist (tf:origin p1) (tf:origin p2)))
11                                 (start-phi (tf:angle-between-quaternions
12                                              offset-angle
13                                              (tf:orientation p1)))
14                                 (end-phi (tf:angle-between-quaternions
15                                            offset-angle
16                                            (tf:orientation p2))) )
17                        :end ( (navigation-time timediff) ))))
18   :experience-class roll:database-experience
19   :experience-class-initargs
20     (:database (make-instance 'roll:mysql-database
21                 :host "..." :user "..." :user-pw "..." :name "...")))
```

(lines 18–21). The access to the database is performed by the experience class and is hidden from the programmer of the abstract experience. Likewise, the data is retrieved automatically.

1.2.3 Learning Problem Definition

Finally, we define the learning problem as shown in Listing 1.4. The function to be learned is the time model of the routine `b21-go2pose-pid-player` (line 2). As an experience we use the same abstraction as the one presented in the last section (lines 5–10). But the experience type of the former was a database experience, which doesn't comply with the format required by the WEKA learning system. Therefore, the experience used for learning is defined to be of class `weka-experience` (line 11). The WEKA experience type requires the specification of the WEKA types for the input and output variables (lines 12–16).

As a learning system we use the WEKA M5' algorithm, which supports model and regression tree learning. The adjustment of the learning algorithm here consists only of some path specifications (lines 13–17).

For calling the learned function, the same abstractions as for the experiences must be performed. In the learning problem specification we inform RoLL that this abstraction can be used for calling the learned function, but in the raw experience definition the variable `goal-pose` is not obtained from the designator of a plan, but should be set to the goal pose as contained in the input variable given to the model (lines 18–20). The result value of the regression tree is exactly what the learned function should provide, so no conversion is necessary (line 21).

To initiate the learning process, the learning problem must be executed by calling the function `learn`:

```
(roll:learn (getgv :learning-problem 'jido-go2pose-time-model-tree))
```

After the learning has been completed, the learned function is loaded at once and from then on can be loaded every time with the rest of the robot program.

Listing 1.4 Learning problem definition for navigation time model.

```
(roll:define-learning-problem
  :function (:some-function jido-go2pose-time-model-tree pose)
  :use-experience (:parent-experience nav-time-abstract-exp
                   :specification (nil
                                   :begin ( (dist (:var dist :begin))
                                            (start-phi (:var start-phi :begin))
                                            (end-phi (:var end-phi :begin)) )
                                   :end ( (navigation-time (:var navigation-time :end)) ))
                   :experience-class roll:weka-experience
                   :experience-class-initargs
                     (:attribute-types '((dist numeric) (start-phi numeric)
                                        (end-phi numeric) (navigation-time numeric))))
  :learning-system (roll:weka-m5prime
                     :root-dir (append (pathname-directory (user-homedir-pathname))
                                '("tmp" "roll"))
                     :data-dir (append (pathname-directory (user-homedir-pathname))
                               '("tmp" "roll" "weka")))
  :input-conversion (:generate
                     (:in-experience navigation-time-exp
                      :set-var goal-pose :to pose)))
  :output-conversion (navigation-time))
```

Chapter 2

Reference of Commands

For describing the language constructs we use a BNF-like notation, which is explained in Table 2.1. Each construct is specified formally using this notation. Additionally, the purpose of the construct and a short description are provided.

Table 2.1 Explanation of the notation for RoLL constructs.

`define-...`	Constructs and their syntactical components of RoLL are written in `typewriter` font.
optional	An optional expression is illustrated in gray.
<u>default</u>	Default values are indicated by <u>underlining</u>.
⟨lisp expression⟩	A construct from LISP or CRAM-PL giving the type or a short specification;
⟨name $_{\text{lisp type}}$⟩	A construct from LISP or CRAM-PL giving an explanation with a significant name and providing the desired LISP type.
⟨*expression*⟩	An expression that is defined later or has been defined on the same page.
⟨*expression*^{page}⟩	An expression that has been defined on the page indicated.
a → b	'a' may be of the form 'b'.
a | b	Alternative construct, either 'a' or 'b'.
a^+	→ a | aa^+.
a^*	either nothing or a^+.

2.1 Experience Data

The addressing of data from experiences is solved in a uniform way in RoLL. Before explaining the single constructs, we define the access to experience data as a reference in later definitions.

automaton data

⟨*automaton data*⟩ → ⟨*event*⟩ | `(`⟨*event*⟩ ⟨*automaton*⟩`)`
⟨*event*⟩ →`:begin` |`:end` |`:interval`
⟨*automaton*⟩ →⟨automaton name $_{\text{symbol}}$⟩

Purpose: Access data of experiences.

Description: Access the *begin, end* or *interval* slots of an automaton. If the automaton is anonymous (i.e. the name of the automaton is a keyword or nil), it need not be named explicitly.

automaton var data

⟨*automaton var data*⟩ → `(:var` ⟨variable $_{\text{symbol}}$⟩ ⟨*automaton data*⟩ ⟨*occurrence specification*⟩`)`
⟨*occurrence specification*⟩ → `:first-occurrence` | `:last-occurrence`
| <u>`:only-occurrence`</u> | `:all-occurrences`
| ⟨occurrence-index $_{\text{list of integers}}$⟩

Purpose: Access detailed data of experience structure.

Description: Access single variables from the *begin, end* or *interval* slots of an experience automaton. For all events, an optional occurrence specification may be given. The occurrence index refers to one specific occurrence with the given index.

2.2 Raw Experiences

`(define-raw-experience` ⟨experience name $_{\text{symbol}}$⟩
  `:specification` ⟨*experience automaton*⟩`)`

Purpose: Declaratively define a raw experience.

Description: Define an experience automaton including the data to be acquired during an episode.

Experience Automaton

⟨*experience automaton*⟩ → `(`⟨*automaton identifier*⟩
`:task` ⟨*task*⟩
`:invariant` ⟨fluent⟩
`:begin` ⟨*recording data*⟩
`:end` ⟨*recording data*⟩
`:interval` ⟨*recording data*⟩
`:interval-parameters` ⟨*interval params*⟩
`:children (`⟨*experience automaton*⟩$^+$`)`
⟨*automaton identifier*⟩ → $\langle \text{dummy name}_{\text{keyword symbol}} \rangle$
|$\langle \text{automaton name}_{\text{non-keyword symbol}} \rangle$
| `nil`

Purpose: Describe an experience automaton.

Description: The automaton defines the episodes to be observed either by an invariant (the automaton being active when the given fluent is not `nil`) or a CRAM-PL task specification. The data is attributed to the begin and end events or the complete interval of the automaton execution. With the `children` slot, a hierarchy of automata can be defined.

Episode Specification

⟨*task*⟩ → `(:plan` $\langle \text{plan}_{\text{symbol}} \rangle$`)`
| `(:tagged` $\langle \text{tag name}_{\text{symbol}} \rangle$`)`

Purpose: Describing the program part to be observed.

Description: There are two possibilities: (1) reacting to a plan execution, which is identified by its name (e.g. at-location) (2) addressing a task that is identified by a tag.

Data Specification

⟨*recording data*⟩ → `(`⟨*variable definition*⟩$^+$`)`
⟨*variable definition*⟩ → `(`$\langle \text{variable}_{\text{symbol}} \rangle$ ⟨*value description*⟩`)`
⟨*value description*⟩ → ⟨symbol⟩
| `(:desig-value '`$\langle \text{desig variable}_{\text{symbol}} \rangle$`)`
| `(`$\langle \text{operator}_{\text{symbol}} \rangle$ ⟨*value description*⟩$^+$`)`
⟨*interval params*⟩ → `(:frequency` ⟨number⟩
`:contains-objects` ⟨boolean⟩`)`

Purpose: Specifying the data to be observed.

Description: Each data slot is given a variable name. The values bound to these names are either drawn from a LISP expression (either symbol or LISP function) or from a local variable inside a task. Both sources can be combined by arbitrary LISP expressions. For intervals two parameters may be given: the frequency of the recorder (to be correct the number specified here is the time between two recordings), defaulting to 0.1; and a specification if the data contains objects (which are then automatically copied), defaulting to nil.

2.3 Problem Generation

2.3.1 Defining Problems

```
(generate-parameter-samples
  :parameters (⟨parameter⟩+)
  :relation ⟨relation⟩)
```

Purpose: Generate a list of problems.

Description: Problems are a vector of values attributed to some variables. With the construct `generate-parameter-samples` a list of such problems is generated according to the specification.

Parameter Defintions

```
⟨parameter⟩ → (⟨variables⟩ ⟨parameter source⟩)
⟨variables⟩ →⟨variable symbol⟩ | (⟨variable symbol⟩+)
⟨parameter source⟩ → :random :min ⟨min value⟩ :max ⟨value number⟩
                             :samples ⟨quantity integer⟩
                     |:cover :min ⟨min value⟩ :max ⟨value number⟩
                             :interval ⟨value number⟩
                     |:predefined (⟨value T⟩+)
⟨min value⟩ → 0.0 |⟨value number⟩
```

Purpose: Define how values are generated.

Description: For each variable define a list of values, which can either be random, covering a range of values systematically or by giving an explicit list.

Relations

```
⟨relation⟩ → (⟨relation keyword⟩ ⟨relation argument⟩+)
⟨relation keyword⟩ → :dep |:indep |:indep-min |:indep-max
⟨relation argument⟩ →⟨variable symbol⟩ |⟨relation⟩
```

Purpose: Combine the individual variable value lists into problems.

Description: Every problem consists of one value for each variable. The individual lists can either be combined with a cross product (`:dep`) or by taking one element of each list (`:indep`). `:indep` and `:indep-max` are synonymous.

2.3.2 Using Generated Problems

```
(with-parameter-samples (:parameters (⟨parameter⟩+)
                         :relation ⟨relation⟩)
  ⟨body CRAM-PL code⟩)
```

Purpose: Use a parameter list inside CRAM-PL code.

Description: Runs the body code in a loop, binding the variables defined in the parameters slot to one problem per loop run. Using the variables in the body, the code is executed with different parameterizations.

→ **see example in 3.1 on page 16**

2.4 Abstract Experiences

2.4.1 Experience Definition

`(define-abstract-experience` $\langle$experience name$_{\text{symbol}}\rangle$
  `:specification` $\langle$*abstract automaton*$\rangle$
  `:experience-class` $\langle$*experience class*$\rangle$
  `:experience-class-initargs` $\langle$*keyword pair*$\rangle^{+}$`)`
$\langle$*experience class*$\rangle$ $\rightarrow$ `'transient-abstract-experience` $|$ $\langle$class name$_{\text{symbol}}\rangle$
$\langle$*keyword pair*$\rangle$ $\rightarrow$ $\langle$key$_{\text{keyword symbol}}\rangle$ $\langle$value$_{\text{T}}\rangle$
$\langle$*abstract automaton*$\rangle$ $\rightarrow$ `(`$\langle$automaton name$_{\text{symbol}}\rangle$
    `:begin (`$\langle$variable$_{\text{symbol}}\rangle^{+}$`)`
    `:end (`$\langle$variable$_{\text{symbol}}\rangle^{+}$`)`
    `:interval (`$\langle$variable$_{\text{symbol}}\rangle^{+}$`)`
    `:children (`$\langle$*abstract automaton*$\rangle^{+}$`))`

Purpose: Define an abstract experience.

Description: The definition is analogous to the raw experience. This definition only gives the structure of the automaton including the variable names without the data source to the variables. Because an abstract experience can be generated from several raw experience classes, the conversion is defined separately. A shortcut definition for an abstract experience with the conversion is presented below in Section 2.4.3.

2.4.2 Experience Conversion

`(define-experience-conversion`
  `:from-experience` $\langle$experience name$_{\text{symbol}}\rangle$
  `:to-experience` $\langle$experience name$_{\text{symbol}}\rangle$
  `:operations` $\langle$*conversion*$\rangle$`)`

Purpose: Specify an abstraction step.

Description: Establishes an abstraction between two experiences, the operations being defined in the form of the abstract experience automaton.

Basic Conversion Operations

$\langle$*conversion automaton*$\rangle$ $\rightarrow$ $\langle$*abstract data automaton*$\rangle$
    $|$ `(with-binding` $\langle$*binding definition*$\rangle$
        $\langle$*conversion automaton*$\rangle$`)`
    $|$ `(with-filter` $\langle$*filter definition*$\rangle$
        $\langle$*conversion automaton*$\rangle$`)`
$\langle$*abstract data automaton*$\rangle$ $\rightarrow$ `(`$\langle$*automaton identifier*$\rangle$
    `:occurrence-handling` `:ignore` $|$ `:use-occurrences`
    `:begin` $\langle$*abstraction data*$\rangle$
    `:end` $\langle$*abstraction data*$\rangle$
    `:interval` $\langle$*abstraction data*$\rangle$
    `:children` $\langle$*abstract data automaton*$\rangle$`)`
$\langle$*abstraction data*$\rangle$ $\rightarrow$ `(`$\langle$*variable assignment*$\rangle^{+}$`)`
$\langle$*variable assignment*$\rangle$ $\rightarrow$ `(`$\langle$variable$_{\text{symbol}}\rangle$ $\langle$*value source*$\rangle$`)`
$\langle$*value source*$\rangle$ $\rightarrow$ $\langle$symbol$\rangle$
    $|$ $\langle$*automaton var data*[9]$\rangle$
    $|$ `(`$\langle$operator$_{\text{symbol}}\rangle$ $\langle$*value source*$\rangle^{+}$`)`

Purpose: Operations for experience conversion.

Description: The automaton definition is analogous to raw experiences. The source of the variable values is now the content of the raw experience (the experience given in the `from-experience` declaration).

Sophisticated Conversion Operations

$\langle$*binding definition*$\rangle$ → `(`$\langle$*variable assignment*$\rangle^{+}$`)`
$\langle$*filter definition*$\rangle$ → `(`$\langle$*binding definition*$\rangle$ `:where` $\langle$condition $_{\text{LISP expression}}\rangle$`)`

Purpose: Special operations for abstraction.

Description: These operations make the definition of abstract experiences more comfortable and powerful. In detail, the options are

- local bindings: defining local variables using the values from the raw experience, comparable to a `let` expression;
- filter: a way to omit unwanted experiences, similar to the event handling strategy of the raw experience, but can decide on the basis of abstracted values;

2.4.3 Combined Definition of Abstract Experience and Conversion

```
(define-abstract-experience ⟨experience name symbol⟩
  :parent-experience ⟨automaton name symbol⟩
  :specification ⟨conversion automaton[12]⟩
  :experience-class ⟨experience class⟩
  :experience-class-initargs ⟨keyword pair⟩+)
```

Purpose: Compact definition of abstract experience and conversion.

Description: Abbreviation for one-to-one relationships of experiences. The syntax is almost identical to the pure abstract experience definition, the only addition being the parent experience. The automaton is defined like the one in the conversion specification.

2.4.4 Auxiliary Constructs for Conversion with Occurrences

→ **see explanation and examples in 3.2 on page 18**

```
(aggregate-occurrences ⟨function⟩ ⟨occurrence-list sequence⟩
  :level ⟨integer⟩ | :all
  :keep-occurrence-numbers T | nil)
```

Purpose: Summarize values of several occurrences into one value or a higher-level occurrence list.

Description: Without keyword arguments, `aggregate-occurrences` works like `reduce`, with the only difference that the occurrence specifications in the data are ignored. `:level` specifies the level of aggregation: level 1 triggers aggregation on the hightest level, level 2 on the second highest one etc. (see examples below). With `:keep-occurrence-numbers` one can choose between keeping the occurrence specification as specified by `:level` or to ignore them. If the highest level is aggregated (i.e. the default behavior) the occurrence numbers always disappear, because the highest level doesn't have occurrence numbering.

(**map-occurrences** $\langle$function$\rangle$
  :keep-occurrence-numbers <u>T</u> | nil)
  $\langle$occurrence-lists $_{\text{sequence}}\rangle$

Purpose: Map over occurrences.

Description: `map-occurrences` applies a function to all elements of a sequence (with specified occurrences) or to the first, second, etc. elements of several sequences in turn (like mapcar). As in `aggregate-occurrences` the parameter `:keep-occurrence-numbers` specifies if the result contains the occurrence information.

(**filter-occurrences** $\langle$occurrence-list $_{\text{sequence}}\rangle$ $\langle$occurrence-spec $_{\text{list}}\rangle$)

Purpose: Filtering occurrences to a specific level.

Description: `filter-occurrences` is a more elaborate version of the occurrence specification in $\langle$*automaton var data*$\rangle$: it doesn't only retrieve one specific occurrence, but returns all occurrences whose occurrence sequence ends with the specified `occurrence-spec`.

2.5 Learning Problems

(**define-learning-problem**
  :function $\langle$function specification$\rangle$[1]
  :use-experience $\langle$experience name $_{\text{symbol}}\rangle$ |$\langle$*conversion automaton*[12]$\rangle$
  :learning-system ($\langle$class $_{\text{symbol}}\rangle$ $\langle$*keyword pair*[11]$\rangle^{+}$)
  :input-conversion $\langle$*abstraction data*[12]$\rangle$
                    | (**:generate** $\langle$*conversion*$\rangle^{+}$)
  :output-conversion ($\langle$LISP expression$\rangle^{+}$))

Purpose: Define a learning problem.

Description: Specify all parts of a learning problem:

- the function to be learned, the syntax being defined by the learning problem class;
- the experience, which can either be defined outside the learning problem definition with `define-abstract-experience` or as an anonymous experience inside the learning problem definition;
- the learning system to be used and the bias given as initialization arguments of the learning system class;
- the conversion between the function produced by the learning algorithm and the desired one (see below for more explanation).

2.5.1 Function Conversion

→ see explanation and example in 3.3 on page 20

[1]see specification of learning problem class on page 14

Automatic Input Conversion

$\langle \textit{conversion} \rangle \rightarrow$ `(:in-experience` $\langle\text{experience}_{\text{symbol}}\rangle$ $\langle \textit{substitution}\rangle^{+}$`)`
  `| (:in-conversion` $\langle\text{raw experience}_{\text{symbol}}\rangle$
    $\langle\text{abstract experience}_{\text{symbol}}\rangle$
    $\langle \textit{substitution}\rangle^{+}$`)`
$\langle \textit{substitution} \rangle \rightarrow$ `:set-var` $\langle \textit{var-substitution}\rangle$ `:to` $\langle\text{LISP expression}\rangle$
  `|:replace` $\langle\text{variable}_{\text{symbol}}\rangle$ `:by` $\langle\text{LISP expression}\rangle$
$\langle \textit{var-substitution} \rangle \rightarrow$ $\langle\text{variable}_{\text{symbol}}\rangle$
  `| (`$\langle\text{variable}_{\text{symbol}}\rangle$ $\langle \textit{automaton data}^{9}\rangle$`)`
  `| (`$\langle\text{variable}_{\text{symbol}}\rangle$ $\langle \textit{automaton data}^{9}\rangle$ $\langle\text{automaton name}_{\text{symbol}}\rangle$`)`

Purpose: Describe the input conversion based on the abstraction chain of the learning experience.

Description: Variable substitutions can take place in any experience or conversion definition. (An abstract experience definition that integrates a conversion specification counts as an experience.) They can either replace the initial definition of a variable or substitute all occurrences of a variable in the abstraction definition (see example below). If necessary, the variable to be redefined can be specified unambiguously by giving the event and possibly the automaton name (in hierarchical experience definitions). For replacements no such specification is allowed, it is a pure substition.

2.6 RoLL Extensions

2.6.1 Experience Classes

New experience classes are simply derived from the predefined experience classes (raw experience, transient-abstract-experience, persistent-abstract-experience)

  `(defclass` $\langle\text{class name}_{\text{symbol}}\rangle$ `(`$\langle\text{superclass}_{\text{symbol}}\rangle^{+}$`)`
   `(`$\langle\text{slot definition}\rangle^{*}$`))`

Purpose: Define a new experience class.

Description: The definition a class definition in LISP. In addition, at least the method `deliver-experience` should be provided for the specified class, which takes an incoming experience and processes it in some way (for transient experiences to pass it on to another conversion step or for persistent experiences to store the data). In addition, for persistent experiences that are not used as learning experience the method `retrieve-experience` must be defined.

2.6.2 Learning Problem Classes

`(define-learning-problem-class` $\langle\text{lp-class name}_{\text{symbol}}\rangle$ `(`$\langle\text{superclass}_{\text{symbol}}\rangle^{+}$`)`
 `(`$\langle\text{slot definition}\rangle^{*}$`)`
 `:definition-schema (`$\langle\text{keyword}_{\text{symbol}}\rangle$ $\langle\text{argument}_{\text{symbol}}\rangle^{*}$`)`
 `:name-generation` $\langle\text{LISP expression}\rangle$
 `:initargs (`$\langle \textit{keyword pair}^{11}\rangle$`)`$^{+}$`)`

Purpose: Define a new learning problem class.

Description: The definition is similar to a class definition in LISP. The definition schema is the pattern after which the function in the learning problem must be defined. The keyword gives an identifier, the rest are arguments. The arguments are bound to slots in the slot definitions whose initargs comply with the argument name, if such slots exist. The initargs declaration establishes the connection between arguments and slots explicitly. The name-generation declaration should contain a functional expression returning a string. The generated name is used as an identifier for the learning problem.

2.6.3 Learning Systems

New learning systems aren't defined by a RoLL construct. Rather, a learning system is a class (derived from the RoLL class `learning-system`) defined with the LISP `defclass` construct and an experience class definition for storing the learning data. To integrate the learning system, the following methods must be added:

- `do-learning`
- `integrate-learned-function`

Chapter 3

Additional Explanations and Examples

3.1 Problem Generation

The syntax and working of the problem generator (explained in 2.3 on page 10) can best be explained by examples. The RoLL specification of the problem generator contains two parts: (1) the parameters and a description of what kind of values they should each adopt in subsequent program runs and (2) the relation between the parameters. The default relation between parameters is dependency.

Here is the first example of a problem generator specification:

```
(generate-parameter-samples
 :parameters ((x :random :min 1.0 :max 2.0 :samples 2)
              (y :predefined (4 5 6)))
 :relation (:dep x y))
```

With this declaration six pairs of parameter values are generated in total. For the parameter `x` two random values in the range between 1.0 and 2.0 are generated, for example 1.7654 and 1.286. The three possible values for parameter `y` are given in the specification. The two value lists for both parameters are then combined according to the relation specification. In this case the parameters are defined to be dependent, which corresponds to the cross product of the two lists, so that the resulting list of values would be `((1.8595252 6)(1.8595252 5)(1.8595252 4)(1.6935984 6)(1.6935984 5) (1.6935984 4))`. In each run of the acquisition program, one pair is used, so that in the first run, parameter `x` takes the value 1.7654 and `y` is bound to the value 4. In this example the relation specification is redundant, because the dependency of both values is the default.

In the following example we see how the number of problems is determined automatically. It is almost identical to the previous one.

```
(generate-parameter-samples
 :parameters ((x :random :min 1.0 :max 2.0)
              (y :predefined (4 5 6)))
 :relation (:indep x y))
```

Here the two parameters are declared to be independent and for parameter `x` the number of needed samples is not given. The overall number of problems is determined by parameter `y` and the number of needed samples for `x` is set automatically to three. A possible result of this specification is `((1.2603241 4)(1.228759 5)(1.1549773 6))`.

Now what happens if we declare `x` and `y` to be independent, but additionally specify that we want only two different values for `x` as we have done in the first example? The overall number of problems could be two, because for parameter `x` only two values are generated, but then we have to omit one of the values for `y`. Contrariwise, if we want all the possible values of `y` to be used, one of the values of `x` must be utilized twice. This ambiguity has to be resolved by the programmer. The relation specification `(:indep-min x y)` produces the first variant, for example `((1.6880503 4) (1.7453804 5))`, whereas with the declaration `(:indep-max x y)` a list of three problems is generated: `((1.3849926 4)(1.7848289 5)(1.3849926 6))`.

If two parameters are to take the same values, the declaration can be abbreviated. For example, if a parameter `u` is to take the same possible values as `y`, we could declare `((y u) :predefined '(4 5 6))`. In this case with predefined values we could as well have copied the declaration for `y` and used it for `u`. If the values are determined randomly, however, the two declarations have different results.

```
(generate-parameter-samples
 :parameters (((x y) :random :min 1.0 :max 2.0 :samples 2)))
```

This specification produces two random values, which are both used for parameters `x` and `y`, so that the resulting problem list might look like this: `((1.7241 1.7241)(1.7241 1.8092)(1.8092 1.7241)(1.8092 1.8092))`. In contrast, the following specification generates two separate lists of random values, which are then combined. The result could be `((1.3703 1.3379)(1.3703 1.8953)(1.8114 1.3379)(1.8114 1.8953))`.

```
(generate-parameter-samples
 :parameters ((x :random :min 1.0 :max 2.0 :samples 2)
              (y :random :min 1.0 :max 2.0 :samples 2)))
```

Finally, we present a more complex example with different relations between the variables. We also see how a range of discretized values is covered completely.

```
(generate-parameter-samples
 :parameters ((x :random :min 1.0 :max 2.0 :samples 2)
              (y :predefined (4 5 6))
              (u :cover :min 0.0 :max 2.0 :interval 0.6)
              (w :cover :max 2.0 :interval -0.6))
 :relation (:indep-max (:dep x u) (:indep-min y w)))
```

The value specification of parameters `x` and `y` is the same as in the examples before, the number of possible values for `x` is restricted to two. The additional parameters `u` and `w` cover a range between 0.0 and 2.0 (the specification `:min 0.0` is redundant, as 0.0 is the default minimum value). The discretization step is 0.6 for both values, but the discretization of `u` starts at 0.0, which gives the values 0.0, 0.6, 1.2, and 1.8. In contrast, the values for `w` are 2.0, 1.4, 0.8, and 0.2.

Now let's have a closer look at the dependencies between the variables. Parameters `x` and `u` depend upon each other. As `x` provides two values and `u` four values, we get a total of eight problems for the combination. `y` and `w` are declared to be independent and the lower possible number of values is to be used. This means that three problems are proposed for the combination of `y` and `w`, one of the values of `w` being discarded. The overall combination of the sublists for `x`/`u` and `y`/`w` are to be combined as independent lists using the maximum number of problems that can be generated. The result is a list of eight values (the ordering of the values is `x`, `u`, `y`, `w` as specified

by the dependency relation):
```
((1.4037962 1.8 4 2.0) (1.4037962 1.2 5 1.4)
 (1.4037962 0.6 6 0.8) (1.4037962 0.0 4 2.0)
 (1.4152595 1.8 5 1.4) (1.4152595 1.2 6 0.8)
 (1.4152595 0.6 4 2.0) (1.4152595 0.0 5 1.4))
```
We see that two random variables have been generated for parameter `x`: 1.4037962 and 1.4152595. They are combined in a cross product with the four values of `u`. The values for `y` and `w` are added independently, but the value 0.2, which would be possible for `w` is never used, because it was discarded when combining the lists of `y` and `w` with the minimum independence relation.

Finally, we should mention how the generated values can be used in the program that controls the robot during experience acquisition. The construct `generate-parameter-samples` used in the examples generates a nested list of values as shown. They can be bound to variables by LISP constructs such as `destructuring-bind` and be used in the program. One has to take care of the order in which the values are arranged in the generated list. A more elegant way is provided by the RoLL construct `with-parameter-samples`, whose syntax is similar to the `generate-parameter-samples` construct, but includes a loop and a `let` expression, so that in each run of the loop the parameter variables are bound to new values, which can be used in the body of the construct.

3.2 Auxiliary Constructs for Conversion with Occurrences

3.2.1 Background

If the experience automaton of the raw experience contains several levels, it is possible that lower-level automata are invoked more often than higher-level automata (e.g. when gripping, the robot might need several attempts to close its gripper). This behavior is coded in “occurrences”, which are stored with the data and keeps track of the hierarchial order.

The occurrence of an automaton invokation (and the data observed therein) is given by a list of numbers. A level 0 (or top level) automaton has no occurrence specification, because it can only be detected once per experience. The occurrences of a level 1 automaton are stored in a list of one element: `(0)` is the first occurrence, `(1)` the second and so on. For higher levels, the occurrence specification is a list of more elements, where the numbers further to the right denote higher-level occurrences. For example `(<experience-data> 0 1)` indicates that the outer automaton is in its second iteration and this piece of data is the first occurrence of the subautomaton in this second run of the higher-level automaton.

This numbering is of importance when handling complex experiences. One way of dealing with occurrences is to ignore them. The default access in the experience conversion with (:var ...) (see ⟨*automaton var data*⟩) uses `:only-occurrence` as its default occurrence specification. Even if more occurrences exist, only the first one is used, but a warning is issued to show that some occurrences will be ignored. If all but the first occurrence are ignored deliberately, the warning can be avoided by specifying `:first-occurrence`. Alternatively the keyword `:last-occurrence` is accepted or a list of numbers specifying one specific occurrence (however, this is dangerous, because there is no guarantee that the automaton has been activated a specific number of times).

The syntax of the constructs is explained in 2.4.4 on page 12.

3.2.2 Examples

```
(defparameter occurrence-list-1 '((a 0 0 0) (b 1 0 0) (c 0 1 0) (d 1 1 0) (e 2 1 0)
                                  (f 0 0 1) (g 1 0 1) (h 0 1 1)))
(defparameter occurrence-list-2 '((1 0 0 0) (2 1 0 0) (3 0 1 0) (4 1 1 0) (5 2 1 0)
                                  (6 0 0 1) (7 1 0 1) (8 0 1 1)))
(defparameter occurrence-list-3 '((1 0 0 0) (2 1 0 0) (3 0 1 0) (4 1 1 0) (5 0 0 1)
                                  (6 0 1 1) (7 1 0 1) (8 0 1 1)))
(defparameter occurrence-list-4 '((1 0 0 0) (2 1 0 0) (3 0 1 0) (4 1 1 0)
                                  (5 2 1 0) (6 0 0 1) (7 1 0 1)))
```

Aggregation

```
(roll:aggregate-occurrences #'list occurrence-list-1)
(((((((A B) C) D) E) F) G) H)

(roll:aggregate-occurrences #'+ occurrence-list-2)
36
```

The specification of levels leads to different stages of aggregation: the higher the level, the less values are aggregated:

```
(roll:aggregate-occurrences #'list occurrence-list-1 :level 1)
((((((A B) C) D) E) 0) (((F G) H) 1))

(roll:aggregate-occurrences #'list occurrence-list-1 :level 2)
(((A B) 0 0) (((C D) E) 1 0) ((F G) 0 1) (H 1 1))

(roll:aggregate-occurrences #'list occurrence-list-1 :level 3)
((A 0 0 0) (B 1 0 0) (C 0 1 0) (D 1 1 0) (E 2 1 0) (F 0 0 1) (G 1 0 1) (H 0 1 1))

(roll:aggregate-occurrences #'+ occurrence-list-2 :level 1)
((15 0) (21 1))

(roll:aggregate-occurrences #'+ occurrence-list-2 :level 2)
((3 0 0) (12 1 0) (13 0 1) (8 1 1))

(roll:aggregate-occurrences #'+ occurrence-list-2 :level 3)
((1 0 0 0) (2 1 0 0) (3 0 1 0) (4 1 1 0) (5 2 1 0) (6 0 0 1) (7 1 0 1) (8 0 1 1))
```

The same examples with `:keep-occurrence-numbers` set to nil:

```
(roll:aggregate-occurrences #'list occurrence-list-1 :level 1 :keep-occurrence-numbers nil)
(((((A B) C) D) E) ((F G) H))

(roll:aggregate-occurrences #'list occurrence-list-1 :level 2 :keep-occurrence-numbers nil)
((A B) ((C D) E) (F G) H)

(roll:aggregate-occurrences #'list occurrence-list-1 :level 3 :keep-occurrence-numbers nil)
(A B C D E F G H)

(roll:aggregate-occurrences #'+ occurrence-list-2 :level 1 :keep-occurrence-numbers nil)
(15 21)

(roll:aggregate-occurrences #'+ occurrence-list-2 :level 2 :keep-occurrence-numbers nil)
(3 12 13 8)

(roll:aggregate-occurrences #'+ occurrence-list-2 :level 3 :keep-occurrence-numbers nil)
```

```
(1 2 3 4 5 6 7 8)
```

Mapping

```
(roll:map-occurrences #'1+ occurrence-list-2)
((2 0 0 0) (3 1 0 0) (4 0 1 0) (5 1 1 0) (6 2 1 0) (7 0 0 1) (8 1 0 1)
 (9 0 1 1))

(roll:map-occurrences #'1+ :keep-occurrence-numbers nil occurrence-list-2)
(2 3 4 5 6 7 8 9)

(roll:map-occurrences #'cons occurrence-list-1 occurrence-list-2)
(((A . 1) 0 0 0) ((B . 2) 1 0 0) ((C . 3) 0 1 0) ((D . 4) 1 1 0)
 ((E . 5) 2 1 0) ((F . 6) 0 0 1) ((G . 7) 1 0 1) ((H . 8) 0 1 1))

(roll:map-occurrences #'cons :keep-occurrence-numbers nil occurrence-list-1 occurrence-list-2)
((A . 1) (B . 2) (C . 3) (D . 4) (E . 5) (F . 6) (G . 7) (H . 8))
```

For a reasonable mapping, the occurrence specifications should match, als in `occurrence-list-1` and `occurrence-list-2`. `map-occurrences` tests if the specifications match and issues a warning:

```
(roll:map-occurrences #'cons occurrence-list-3 occurrence-list-2)
WARNING:
   [EXPERIENCE CONVERSION] Incompatible occurrence specification in map-occurrences
WARNING:
   [EXPERIENCE CONVERSION] Incompatible occurrence specification in map-occurrences
(((1 . 1) 0 0 0) ((2 . 2) 1 0 0) ((3 . 3) 0 1 0) ((4 . 4) 1 1 0)
 ((5 . 5) 0 0 1) ((6 . 6) 0 1 1) ((7 . 7) 1 0 1) ((8 . 8) 0 1 1))
```

If the lengths of the lists don't match, the normal mapcar behavior of ignoring additional elements of the longer list is used, in this case no warning is issued:

```
(roll:map-occurrences #'cons occurrence-list-1 occurrence-list-4)
(((A . 1) 0 0 0) ((B . 2) 1 0 0) ((C . 3) 0 1 0) ((D . 4) 1 1 0)
 ((E . 5) 2 1 0) ((F . 6) 0 0 1) ((G . 7) 1 0 1))
```

Filtering

```
(roll:filter-occurrences occurrence-list-1 '(1 0))
((C 0 1 0) (D 1 1 0) (E 2 1 0))
```

Compound Example

Here is a short example of how these constructs can be applied in the context of experience conversion. We start with the definition of the raw experience (with a hierarchical automaton):

```
(roll:define-raw-experience navigation-time-hierarchy-exp
    :specification (top
                    :task (:tagged my-tag)
                    :begin ( (timestep-top (get-universal-time)) )
                    :end ( (timestep-top (get-universal-time)) )
                    :children ( (nav
                                  :task (:plan at-location)
                                  :begin ( (timestep (get-universal-time)) )
```

```
                                      :end ( (timestep (get-universal-time)) )) )))
```

The data of the occurrences can be treated in different ways:

```
(roll:define-abstract-experience navigation-time-hierarchy-exp-format
  :parent-experience navigation-time-hierarchy-exp
  :specification
   (nil
     :begin ( (start-top (:var timestep-top (:begin top)))
              (start-times-nav (:var timestep (:begin nav) :all-occurrences))
              (start-times-nav-sum (roll:aggregate-occurrences
                                      #'+
                                      (:var timestep (:begin nav) :all-occurrences))) )
     :end ( (end-top (:var timestep-top (:end top)))
            (end-times-nav (:var timestep (:end nav) :all-occurrences))
            (durations-nav (roll:map-occurrences #'- :keep-occurrence-numbers nil
                             (:var timestep (:end nav) :all-occurrences)
                             (:var timestep (:begin nav) :all-occurrences)))
            (avg-duration (roll:aggregate-occurrences
                            #'(lambda (&rest args) (/ (apply #'+ args) (length args)))
                            (roll:map-occurrences #'-
                             (:var timestep (:end nav) :all-occurrences)
                             (:var timestep (:begin nav) :all-occurrences)))) ))
  :experience-class roll:format-experience)
```

3.3 Input and Output Conversion of the Learning Problem

3.3.1 Background

Once the learning problem class and an appropriate learning system have been added to RoLL, learning problems can be defined and executed. Every learning problem can be addressed by a unique name. For performing the learning process, the function `learn` must be called with the learning problem as an input parameter. The `learn` function should only be called after the experiences have been acquired. In the current version of RoLL the scheduling of experience acquisition and learning is performed manually, therefore both processes are treated independently.

For specifying a learning problem, it is necessary to state what is to be learned by giving a learning problem class with appropriate parameters. For example, for learning the time model of the routine `navigate`, the specification according to the learning problem class of routine models is `(:model navigate :time)`. The order in which the parameters `navigate` and `:time` are to be given is not part of RoLL, but is defined by the learning problem class `:model`.

Secondly, the experiences that are to be used for learning must be specified. The experience given in the learning problem must be of a type that is understood by the learning system.

The learning system to be used is the third component of the learning problem specification. It also includes the parameterization of the learning algorithm for the problem at hand.

Finally, the abstracted values used for learning must be associated to the input values of the resulting function. We explain this point in more detail, because it is an important step in the context of experience abstraction.

Consider a low-level navigation routine to be learned. The raw experiences might be gathered by controlling the robot in a random way and recording pairs of start and end points together with the low-level navigation commands, which is a vector of the form $\langle x_0, y_0, \varphi_0, x_1, y_1, \varphi_1, rot_0, trans_0\rangle$

with the robot's position at time 0, its position at time 1 and the commands given at time 0. After observing such a vector, we can expect the robot to reach position $\langle x_1, y_1, \varphi_1 \rangle$ from position $\langle x_0, y_0, \varphi_0 \rangle$ if it gives the command $\langle rot_0, trans_0 \rangle$.

This experience representation might not be suitable for learning, because absolute positions are given and therefore identical navigation situations are treated as distinct cases if the positions are translated or rotated in 2D space. Therefore, the experiences are abstracted to a form $\langle distance, \varphi_0, \varphi_1, rot_0, trans_0 \rangle$, the correlation to the original experience being depicted in Figure 3.1. Now the signature of the learned function is $distance \times \varphi'_0 \times \varphi'_1 \rightarrow rot_0 \times trans_0$. However, in a world with many obstacles, the original representation with absolute positions might be more appropriate, since navigation paths must be chosen according to the location of the obstacles. So for learning a navigation routine, different experience abstractions are possible. The call to the learned function should be the same, no matter which abstraction is used for learning, in the example $x_0 \times y_0 \times \varphi_0 \times x_1 \times y_1 \times \varphi_1 \rightarrow rot_0 \times trans_0$. However, the functions produced by the learning system expect to be called according to the abstracted experience.

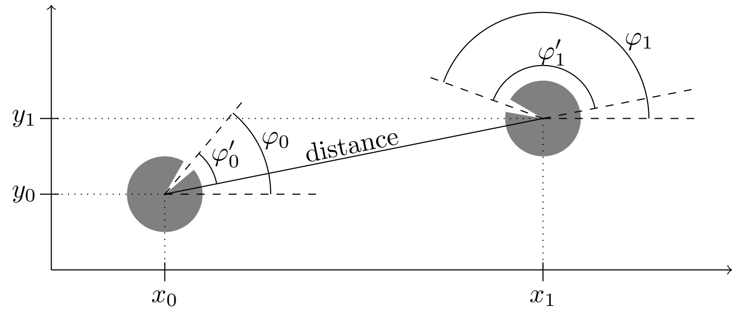
Figure 3.2 illustrates this phenomenon. The function F is the one that is intended to be learned with the signature $x \times y \times z \rightarrow v \times w$, whereas f is the function produced by the learning algorithm with the signature $h \times i \rightarrow j \times k$. The experience data is prepared for learning by a multi-step abstraction process involving abstractions A_0, A_1 and A_2.

When F is called, it gets the originally intended input values x, y and z, which must be converted with the same abstractions as the learning experience and can then be used to call f. The output of f is not exactly what the caller of F expects. Therefore, the output values $\langle j, k \rangle$ must be transformed to $\langle v, w \rangle$ by applying the abstraction chain A_0, A_1, A_2 backwards. This whole procedure has two tricky parts: (1) How can the original abstraction definitions be used, i.e. how do the values $\langle x, y, z \rangle$ correspond to the values of the raw experience? and (2) How can the reverse abstractions be calculated?

To illustrate the first question, consider again the example of the navigation routine to be learned. The position $\langle x_1, y_1, \varphi_1 \rangle$ is obtained by random control of the robot. In contrast, for the resulting function, these values come from the goal position, which is given as the input. Thus, for applying the abstractions, which have already been specified, RoLL must be told that the pose $\langle x_1, y_1, \varphi_1 \rangle$ of the raw experience corresponds to the goal position of the function F. The position $\langle x_0, y_0, \varphi_0 \rangle$ needn't be specified further, because in both cases it denotes the robot's current position. Then the abstraction steps for the input values of f are generated automatically.

RoLL also offers the possibility to specify the complete input abstraction explicitly. This means

Figure 3.1 Illustration of experience abstraction for learning a navigation routine.



to ignore the already defined abstractions for the experiences and to define a function converting the input of F to the input of f. However, one has to take care that when the experience abstraction is changed, the input abstraction to the learned function must be changed as well. Therefore, the automatic generation is to be preferred.

For the back transformation of output values, RoLL only offers the manual method of specifying a function that converts $\langle j, k\rangle$ to $\langle v, w\rangle$. This is because for the input conversion the abstractions are known, but for the output conversion the reverse abstractions would have to be generated. This might be possible in simple cases, for example for algebraic expressions, but can be really hard when complex LISP functions with conditionals and recursion or loops are involved. Therefore, the reverse abstractions for A_0, A_1, A_2 would have to be specified by hand, which is however more costly than just defining the complete back transformation of the output values.

The specification of the starting point for abstracting the input values and the transformation of the output values of f defines the signature of F. A good signature for F is one that corresponds closely to the input given by the learning problem class. For example, routine models are always called with the routine and the goal it is to achieve. The output should fit the chosen kind of routine, e.g. the time needed for performing the task. The input and output abstractions must map these input and desired output values to the abstractions used in the experience abstraction process.

To summarize, a learning problem consists of an instantiation of a learning problem class, a set of experiences, a learning system with parameters suitable for the problem at hand, and a specification of the input and output abstractions, which represents a connection between the learning problem class and the experience abstraction. The learning problem definition for the example presented in this sections is shown in Listing 1.4 on page 7.

3.3.2 Implementation

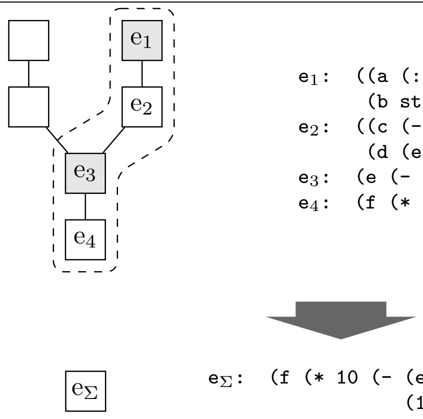
Output Conversion. Because the inverse abstraction cannot be generated in most cases (the conversion from $\langle j, k\rangle$ to $\langle v, w\rangle$ in Figure 3.2), the output conversion is given manually as a list of LISP expressions, the outer brackets replacing an explicit `progn`. This conversion can return any LISP expression including several return values.

Input Conversion. The initial abstraction (from $\langle x, y, z\rangle$ to $\langle h, i\rangle$) can either be given as a list of variable definitions in the style of a `let` expression or it can be generated from the already existing

Figure 3.2 Abstraction in the context of the whole learning process.

$$\begin{tikzcd}
\langle x,y,z\rangle \arrow[r] \arrow[d, "A_0"] & \boxed{F} \arrow[r] & \langle v,w\rangle \\
\langle a,b,c\rangle \arrow[d, "A_1"] & & \langle d\rangle \arrow[u, "A_0^{-1}"] \\
\langle e,f\rangle \arrow[d, "A_2"] & & \langle g\rangle \arrow[u, "A_1^{-1}"] \\
\langle h,i\rangle \arrow[r] & \boxed{f} \arrow[r] & \langle j,k\rangle \arrow[u, "A_2^{-1}"]
\end{tikzcd}$$

Figure 3.3 Collapsing the course of abstractions into one. If there are several paths of abstraction, the one where the substitutions are defined is chosen.



```
e1:  ((a (:internal-value ...))
      (b statevar-pos-x))
e2:  ((c (-1 (:var a e1)))
      (d (expt (:var b e1) 2)))
e3:  (e (- (:var d e2) (:var c e2)))
e4:  (f (* 10 (:var e e3)))
```

```
eΣ:  (f (* 10 (- (expt statevar-pos-x 2)
                 (1- (:internal-value ...)))))
```

experiences and conversions.

3.3.3 Example

Consider the example in Figure 3.3. It shows the abstraction chain defined for learning, the learning experience being `e`$_4$. There are two experiences as possible sources to `e`$_3$. The data of the experiences is shown in a slightly simplified version at the right hand side. When defining a learning problem using `e`$_4$ as experience, a possible input conversion is this:

```
(:generate
  (:in-experience e1 :set-var a :to input-value)
  (:in-experience e3 :replace (:var d e2) :by 5))
```

The first thing this definition tells us is that the right branch of the experience tree is interesting for the abstraction generation, because variables in `e`$_1$ and `e`$_3$ are substituted. The definition must provide a possible solution. It would be incorrect to declare substitutions in both arms of the tree.

With this structural information, a compact experience definition ranging from `e`$_1$ to `e`$_4$ is generated as shown in the figure. It now remains to substitute the variables. The `:set-var` ... `:to` construct replaces the right side of a variable definition. This means instead of binding `a` to `(:internal-value ...)` it should be bound to `input-value` instead. The command `:replace ... :by` does a substitution on all occurrences of one expression by another. This means that in the definition of `e`$_3$, where `d` is used in a calculation, it is replaced by `5`. The resulting abstraction is

```
(f (* 10 (- 5 (1- input-value))))
```

Chapter 4

RoLL Extensions

We have mentioned several ways to customize and extend the basic functionality of RoLL, so that it can be adapted to new developments in learning algorithms and new kinds of learning problems. These extensions are

- experience classes,
- learning problem classes, and
- learning systems.

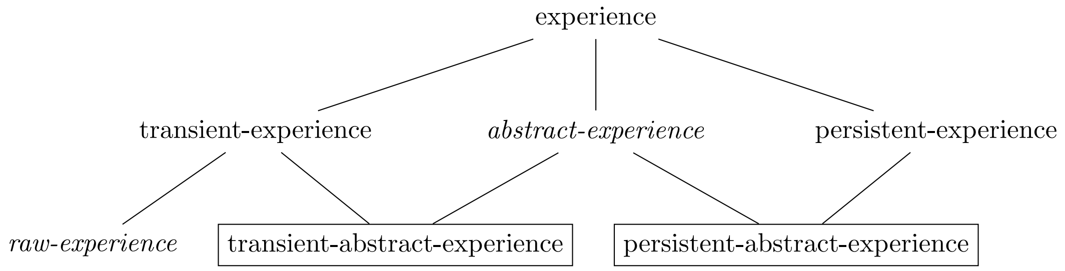
In the following, we explain the language constructs of RoLL and the steps to be taken for defining these modules by first presenting the approach and then giving a detailed example. The following sections are also intended as a reference for the set of extensions implemented so far.

4.1 Experience Classes

Experience classes are defined in a class hierarchy as shown in Figure 4.1. The general classes of experiences can be classified along a logical line into raw and abstract experiences and along a data processing criterion into transient and persistent experiences. As raw experiences come in as a stream of data, they are defined to be transient. This means, they don't store the data only until they are passed on to the next level of abstraction. We don't expect that new classes of raw experiences are defined by the user, because the process of experience acquisition is very complex and an integral part of RoLL.

Abstract experiences are usually of a persistent character. They can either store experiences for longer times or be the interface for the learning data of the chosen learning algorithm. When there are complex relationships between abstractions, the use of transient abstract experiences can be useful. In any case, RoLL allows the definition of new classes of abstract experiences. For permanently storing and managing experiences, we suggest to use the database experience presented below. If no database is available or especially huge sets of experiences are to be stored, log files are an alternative. Examples for abstract experiences containing the data for the learning algorithm are presented in Section 4.3.

Figure 4.1 RoLL standard experience classes. The framed classes are the ones new experience classes can be derived from.



4.1.1 Process of Defining Experience Classes

For defining a new class of abstract experiences, one has to follow three steps:

1. define an experience class (`defclass`),
2. specify how incoming data is to be stored (`deliver-experience`),
3. specify how to retrieve the data (`retrieve-experience`), not necessary for transient experiences or learning experience classes.

The first step simply consists in defining a LISP class derived either from one of the classes `roll:transient-abstract-experience` for transient experiences or `roll:persistent-abstract-experience` for persistent ones. This class definition may contain any slots necessary for storing or retrieving the data, e.g. a directory where a log file is to be stored.

Secondly, each experience class must define the method `roll:deliver-experience`, which receives the reference instance of the class as input and takes care that the data contained in it is not lost. After calling this method, the reference instance will be used to store other data.

The third step is irrelevant for experiences used as input for the learning system. All other experience classes must provide a method that retrieves the ata of an experience.

4.1.2 Database Experience

For storing experiences permanently, relational databases offer a lot of functionality that makes the management and filtering of experiences possible. For one thing, all experiences are available without having to be read from a file and can be modified either by SQL queries or by data mining mechanisms working directly on the database. Besides, additional information concerning the experience instances can be stored, e.g. the time when the experience was made, so that newer experiences can be trusted more than older ones. Moreover, there are lots of tools and frontends available to visualize the contents of SQL databases, which helps to get a better understanding of experiences.

The definition of a database experience presented in the following is not the only way of defining an abstract experience using a relational database as storage method. It uses a specific way of linking LISP to the database and defines a specific mapping from experiences to tables. Besides, it stores the time when an experience was made as additional management information.

Class Definition

```
(defclass database-experience (persistent-abstract-experience)
  ( (table-name-prefix :accessor table-name-prefix)
    (internal-database-spec :accessor database-spec)
    (internal-db-signature-known :accessor db-signature-known :initform nil)
    (internal-max-episode-number :accessor max-episode-number :initform nil) ))
```

The class `database-experience` contains information about the database access, a naming convention for easily finding the tables produced by LISP in the SQL tables, and information about the current status of experience conversion.

For the database experience, we also defined a custom class for the unterlying experience automaton to store the database signature and insertion commands:

```
(defclass db-experience-automaton (experience-automaton)
  ( (db-signature :initarg :db-signature :initform nil :accessor db-signature)
    (db-insert-commands :accessor db-insert-commands) ))
```

Storing Data

For storing experiences in a database, the first design decision is how to map the automaton structure of the LISP experience class with *begin*, *end* and *interval* slots to database tables. Figure 4.2 shows how tables are created from an experience definition. For each data chunk (the data recorded in one automaton at one of the times begin, end, or interval) one table is created, whose name is composed of

- the name of the experience (in the example *cup-experience*),
- the name of the automaton (entity-at-place, grip, pick-up),
- the recording event (begin, end, interval).

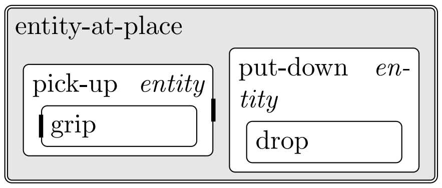
The signature of the table always contains the episode number (a running number counting the number of instances available of this experience type) and the occurrence number. For interval tables, the instance number of the data set is also stored. The rest of the table structure is computed from the variables that are to be recorded according to the names provided in the experience definition. The SQL data types are derived automatically from the LISP types when the first experience is to be stored, although this may be ambiguous (especially the LISP value `nil` can be translated to `false` or `null`, so that the data type can either be Boolean — this is what we chose — or any SQL type).

Beside the tables generated from the experience definition, the database experience adds another table, whose name is composed of the experience name and the suffix *-time-management*, where the episode number and the time when the experience was made are stored.

Retrieving Data

For retrieving data, the data for one episode is read from all tables and reconstructed in an experience-automaton object.

Figure 4.2 Mapping between automaton structure and database tables.



cup-experience:	
entity-at-place$_{\text{interval}}$	$\rightarrow$*robot pose*
grip$_{\text{begin}}$	$\rightarrow$*entity pose, side*
pick-up$_{\text{end}}$	$\rightarrow$*timestep*

CUP-EXPERIENCE-TIME-MANAGEMENT

EPISODE-NR	TIME
...	...

CUP-EXPERIENCE-ENTITY-AT-PLACE-INTERVAL

EPISODE-NR	OCCURRENCE-LEVEL-1	INSTANCE-NR	ROBOT-POSE
...	...	...	...

CUP-EXPERIENCE-GRIP-BEGIN

EPISODE-NR	OCCURRENCE-LEVEL-1	OCCURRENCE-LEVEL-2	ENTITY-POSE	SIDE
...	...	...	...	...

CUP-EXPERIENCE-PICK-UP-END

EPISODE-NR	OCCURRENCE-LEVEL-1	TIMESTEP
...	...	...

4.2 Learning Problem Classes

For RoLL it is important to know what kind of function is to be learned, e.g. a prediction model for a routine. Different kinds of functions need different parameters to be defined unambiguously.

This information is necessary for integrating the learning result appropriately into the program. The integration is done in accordance with the learning system, which means that the generic function `integrate-learned-function` is determined by the learning system and the learning problem class. Here we only show the definition of learning problem classes, an example of integrating learned functions is described in the next section.

Listing 4.1 Definition of learning problem classes.

```
; method-learning-problem
(define-learning-problem-class method-learning-problem (learning-problem)
  ( (generic-fun :initarg :generic-fun :initform nil :accessor generic-fun)
    (specializers :initarg :specializers :initform nil :accessor specializers) )
  :definition-schema (:method generic-fun &rest specializers)
  :name-generation (format nil "~a-METHOD-~{~a~^-~}"
                    generic-fun
                    (mapcar #'second specializers))
  :procedure-generation (declare (ignore generic-fun specializers)))

; general-function-learning-problem
(define-learning-problem-class general-function-learning-problem (learning-problem)
  ( (funname :initarg :funname :initform nil :accessor funname)
    (args :initarg :args :initform nil :accessor args) )
  :definition-schema (:some-function funname &rest args)
  :name-generation (format nil "~a-FUNCTION" funname)
  :procedure-generation (declare (ignore funname args)))
```

The two examples in Listing 4.1 show how a learning problem class is to be defined. The specification looks similar to a LISP class definition. After the name the parent classes are given, usually this is the RoLL class `learning-problem`. Then slots containing the parameters of the learning problem class can be defined. The definition schema specifies the way in which the learning problem class is chosen when specifying a learning problem. The variables in this schema can either correspond to the initargs of the slots defined in the learning problem class or a mapping between the variables in the definition scheme and the initargs of the slots can be established with the parameter `:initargs`. The parameter `:name-generation` describes how to generate a unique name for a learning problem instance of the specified class.

4.3 Learning Systems

The most important dimension of extending RoLL is the possibility to use any experience-based learning algorithm. For RoLL it is of no importance if the learning system is implemented in LISP or in an external program as long as it can be called from LISP. The main differences between learning systems are

- the bias (i.e. the parameters controlling the learning process),
- the format of the input data, and

- the output format.

The procedure of defining a new learning system is oriented along these differences:

1. Define a class derived from `roll:learning-system`, containing all the parameterization needed for running the learning process. This class is the interface for choosing and parameterizing the learning system later.

2. Define an experience class whose storage format fits the requirements of the learning system. For a learning experience class, only steps 1 and 2 of the procedure described on page 25 are necessary since the data needn't be read back into the RoLL system.

3. Implement the invocation of the learning process with the bias given by the user.

4. Specify how to integrate the result of the learning process into LISP. This step consists of two problems: conversion of the output from the learning system to executable LISP code, and integration of this code into the program in a way that fits the chosen learning problem class. As these two steps are strongly interwoven, they have to be specified in one method `integrate-learned-function`. This requires knowledge of both the learning system and the learning problem class.

Although RoLL doesn't assume a specific format for the output of the learning system, it requires that the result of the learning process be written to a LISP file, so that it can be loaded in later runs of the system. Therefore, each learning problem is provided the information of where to put the file containing the learned function (a slot inherited from the class `roll:learning-system`, the information being provided by the programmer) and the LISP package of the function to be learned. Besides, a unique identifier for each learning problem is generated according to the rules of the learning problem.

In the following we present the learning systems implemented so far. Both make use of external programs. The first employs several decision tree algorithms of the WEKA machine learning software, the second provides access to the Stuttgart Neural Network Simulator (SNNS).

4.3.1 Generic Learning System

- Provides the means to learn with an arbitrary function, should be a simple one like max, avg, ...

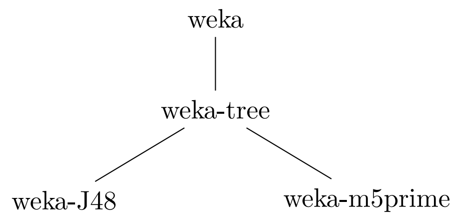
- The generic-learning-experience must always contain an anonymous automaton.

- When specifying a learning problem, the learning function must be provided as a lisp command. For several operations use let or progn.

4.3.2 Learning System WEKA

The WEKA [4] machine learning project is a toolkit providing different learning and data mining algorithms implemented in Java. We only include the algorithms for decision tree learning, these are J48 for classical decision trees and M5' for model and regression trees [1].

Figure 4.3 Class hierarchy for WEKA tree learning algorithms.



Learning System Class

As the WEKA software supports a variety of learning algorithms, we arrange the learning algorithms in a class hierarchy similar to the structure in WEKA. This approach simplifies the integration of new WEKA algorithms and exploits features of the class hierarchy. Figure 4.3 depicts the hierarchy graphically, whereas Listing 4.2 shows the code for defining these classes in RoLL.

The input for all WEKA classifiers is given in the attribute-relation file format (arff) and the result is stored in an output file with the extension `.weka`, whose format differs however when using different learning algorithms. The output produced by WEKA is no executable code, but must be converted to a function in some programming language. Once this conversion is done, the WEKA output file can be deleted or be kept for inspection. This decision can be controlled by a flag in the `weka` class.

The next step in defining the classes for decision tree learning is a class called `weka-tree`. The class definition subsumes parameters that are valid in all tree learning algorithms. Besides, the output format of the different tree learning algorithms is similar, though not identical. By defining the `weka-tree` class, some of the parsing can be handled on the level of the general tree class.

Finally, we define two classes `weka-J48` and `weka-m5prime` for different decision tree learning algorithms. The differences are mainly in the call to WEKA and the parsing of the resulting tree given in the WEKA result file. The J48 algorithm is an enhancement of the well-known C4.5 algorithm and trains classical decision trees. The M5' algorithm handles both model trees and regression trees, the choice being made by the flag `build-regression-tree`.

WEKA Experience

The input format to all WEKA classifiers is independent of the chosen algorithm. All classifiers assume that the data is composed of input and output values. Therefore, the automaton structure of a weka-experience assumes that only one automaton with *begin* and *end* values and no *interval* values is given.

For defining input data files WEKA requires a name for the relation to be learned, the types of the attributes (input and output), and the data itself. The relation name is stored in the `weka-experience` class (see Listing 4.3) and is set automatically to the generated name of the learning problem. The types of the attributes must be specified manually. They could be set automatically like the SQL data types of the database experience presented in Section 4.1.2, but the mapping is sometimes ambiguous and can only be determined after the first experience instance (i.e. the actual data) is known.

The conversion of the learning data to the attribute-relation file format works identically to the conversion from an experience to a more abstract one. The experience data is converted episode-

Listing 4.2 Class definitions for WEKA tree learning algorithms.

```
;; all learning algorithms supported by WEKA
(defclass weka (roll:learning-system)
 ((arff-file        :initarg :arff-file :reader arff-file)
  (weka-output-file :initarg :weka-output-file
                    :reader weka-output-file)
  (delete-weka-output-file :initarg :delete-weka-output-file
                           :initform nil
                           :reader delete-weka-output-file)))

;; tree learning algorithms
(defclass weka-tree (weka)
 ((use-unpruned-tree :initarg :use-unpruned-tree
                     :initform nil
                     :accessor use-unpruned-tree)
  (minimum-number-of-instances
   :initarg :minimum-number-of-instances
   :initform nil
   :accessor minimum-number-of-instances)))

;; classical decision tress
(defclass weka-J48 (weka-tree)
 ((pruning-confidence-threshold
   :initarg pruning-confidence-threshold
   :initform nil
   :accessor pruning-confidence-threshold)
  (reduced-error-pruning :initarg reduced-error-pruning
                         :initform nil
                         :accessor reduced-error-pruning)
  (number-of-folds :initarg number-of-folds
                   :initform nil
                   :accessor number-of-folds)
  (use-binary-splits-only :initarg use-binary-splits-only
                          :initform nil
                          :accessor use-binary-splits-only) ))

;; regression and model trees
(defclass weka-m5prime (weka-tree)
 ((use-unsmoothed-predictions :initarg :use-unsmoothed-predictions
                              :initform nil
                              :accessor use-unsmoothed-predictions)
  (build-regression-tree :initarg :build-regression-tree
                         :initform nil
                         :accessor build-regression-tree)))
```

wise. When the reference instance of the weka-experience is filled with data, its content is written to a file in the required format. The writing of the attribute-relation file is done in two steps: first only the data is written to a file `arff-tmp-file` specified in the WEKA experience class. The final data file is created before the learning process starts (see next section).

Listing 4.3 Definition of experience class for WEKA algorithms. The data is stored in the attribute-relation file format required by WEKA.

```
;; weka-experience class
(roll:define-abstract-experience-class weka-experience
                                       (roll:learning-experience)
 ((arff-tmp-file   :initform nil :accessor arff-tmp-file)
  (relation-name   :initform nil :accessor relation-name
                   :initarg :relation-name)
  (attribute-types :initform nil :accessor attribute-types
                   :initarg :attribute-types)))

;; storing experience data
(defmethod roll:deliver-experience ((experience weka-experience))
 (ensure-directories-exist (arff-tmp-file experience))
 (with-open-file (stream (arff-tmp-file experience)
                  :direction :output :if-exists :append
                  :if-does-not-exist :create)
  (format stream "~{~,5f,~}~{~,5f~^,~}~%"
   (roll:get-automaton-data experience :begin)
   (roll:get-automaton-data experience :end))))
```

Performing the Learning Process

Invoking the WEKA learning software from RoLL is very simple. The two steps to be performed for all algorithms provided by WEKA is to complete the missing information (the attribute names and types) in the data input file and then to call WEKA with the chosen algorithm. This call to WEKA differs for each algorithm. It must contain the correct WEKA class and add the specified parameters to the method invocation. The output of the learning process is written to the output file specified in the learning problem specification. The definition of the general learning procedure for all WEKA algorithms and the specific call to the M5' algorithm are shown in Listing 4.4.

Integrating the Learning Results

The most laborious part in the definition of a new learning system is the integration of the learning result back into RoLL. In the case of WEKA there are no tools translating the output of the learning process to a C or LISP function, which could easily be embedded into the program. Instead, the tree learning algorithms produce a well-readable text file containing the tree structure. We parse this tree in LISP and return executable code, which must then be written to a file either in the form of a LISP function or method or a lambda function in the correct context, according to the learning problem class.

The solution for integrating WEKA results into the RoLL program is shown in Listing 4.5. The method `integrate-learned-function` is specialized to the WEKA learning system and any kind of learning problem. It parses the WEKA output and produces a general frame for the LISP file that is to contain the resulting function.

Listing 4.4 Definition of learning process for WEKA algorithms, in particular M5'.

```
;; invocation of WEKA classification algorithm
(defmethod roll:do-learning ((ls weka) (experience weka-experience))
 ;finish arff file
 (port:run-prog "/bin/bash"
  :args (list
          "-c"
          (format
           nil
           "echo␣-e␣\"@relation␣~a\\n\\n~{@attribute␣~{~a~^␣~}\\n~}
␣␣␣␣␣␣␣␣␣␣␣␣␣\\n@data\";␣cat␣~a"
           (relation-name experience)
           (attribute-types experience)
           (namestring (arff-tmp-file experience))))
  :output (arff-file ls)
  :if-output-exists :supersede)
 (port:run-prog "rm"
   :args (list (namestring (arff-tmp-file experience))))
 ;run specific learning algorithm
 (run-weka ls))

;; call to WEKA for M5' algorithm
(defmethod run-weka ((ls weka-m5prime))
 (port:run-prog "java"
  :args (append
         '("weka.classifiers.trees.M5P")
         (when (use-unpruned-tree ls) '("-N"))
         (when (use-unsmoothed-predictions ls) '("-U"))
         (when (build-regression-tree ls) '("-R"))
         (when (minimum-number-of-instances ls)
           (list (format nil "-M␣~a"
                         (minimum-number-of-instances ls))))
         `("-t" ,(namestring (arff-file ls))))
  :output (weka-output-file ls) :if-output-exists :supersede))
```

Listing 4.5 Integration of WEKA results when learning a prediction model for a routine.

```
(defmethod roll:integrate-learned-function
               ((ls weka) (lp roll:learning-problem))
 (with-open-file (str (learned-function-file ls)
                  :direction :output :if-exists :supersede)
  (format str "(in-package␣~s)~2%"
          (package-name (learned-function-package ls)))
  (format str "~s"
          (make-weka-function-call
           lp
           `(let ((,(first (second (learning-signature ls)))
                   ,(parse-weka-output ls)))
              ,@(output-conversion lp)))))
 (when (delete-weka-output-file ls)
  (port:run-prog "rm"
   :args (list (namestring (weka-output-file ls))))))

; make-weka-function-call for model-learning problem
(defmethod make-weka-function-call ((lp model-learning-problem)
                                     function-body)
 (let ((routine-var (gentemp "ROUTINE")))
  `(add-model ',(routine lp) ,(model-type lp)
    (make-instance 'routine-model
     :execution-entity #'(lambda (,routine-var)
                          (let* ,(make-learning-conversion-code
                                 (input-conversion lp)
                                 `((:input ,routine-var)))
                           ,function-body))))))
```

The function definition according to the learning problem class is done by the method `make-weka-function-call`, which takes the learning problem object and the function body generated by parsing the WEKA output file. In Listing 4.5 we show the integration for the learning problem class for routine models presented in Section 4.2. In this case the function code is surrounded by a lambda function, which is placed in the model slot of the routine specified in the learning problem.

The resulting file is loaded directly after the learning process and then every time the LISP system is started. A prerequisite is of course that the learned file is written to a directory where it is loaded automatically. This can easily be achieved by loading files with the ASDF system.

4.3.3 SVM Learning System

We have implemented a learning system based on libsvm[1]. Using the available LISP bindings of cl-libsvm[2] the integration is straightforward.

```
(defclass libsvm (learning-system)
  ( (model-file :reader model-file :initarg :model-file)
    (parameters :initarg :parameters :initform '(:kernel-type :rbf) :reader parameters) ))
```

The class definition allows the parameterization of where the model file (the result from the learning process) is stored and a very general way to set parameters as defined in libsvm.

[1]LIBSVM — A Library for Support Vector Machines: `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
[2]`http://www.cliki.net/cl-libsvm`

Bibliography

[1] Thorsten Belker. *Plan Projection, Execution, and Learning for Mobile Robot Control.* PhD thesis, Department of Applied Computer Science, University of Bonn, 2004.

[2] Alexandra Kirsch. *Integration of Programming and Learning in a Control Language for Autonomous Robots Performing Everyday Activities.* PhD thesis, Technische Universität München, 2008.

[3] Alexandra Kirsch. Robot learning language — integrating programming and learning for cognitive systems. *Robotics and Autonomous Systems Journal*, 57(9):943–954, 2009.

[4] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, San Francisco, 2$^{\text{nd}}$ edition, 2005.