P-IndiGolog: An Integrated Agent Architecture

Programmer and User Manual
Beta Version 0.3

Sebastian Sardiña

May 2, 2005

Contents

1 OVERVIEW 3

2 DIRECTORY LOGICAL STRUCTURE 4
- 2.1 Libraries Provided 4

3 THE CORE 5
- 3.1 The Top-Level Main Cycle 5
- 3.2 EM: The Environment Manager 8
  - 3.2.1 The EM Interface 9
  - 3.2.2 EM Passive-mode Cycle 10
  - 3.2.3 Socket Communication and Messages 11

4 TRANSITION SYSTEM AND TEMPORAL PROJECTOR 12
- 4.1 The Transition System 12
- 4.2 The Temporal Projector 13

5 DEVICE/ENVIRONMENT MANAGERS 14
- 5.1 Device Manager Operation 15
  - 5.1.1 How to Develop a New Device Manager 17

6 DEVELOPING DOMAIN APPLICATION (USER MANUAL) 17
- 6.1 The Domain Axiomatization and High-Level Programs 18
- 6.2 The main file "main_xxx.pl" 18
  - 6.2.1 Execution information: how to execute programs in the world? 19

7 TODO's 21
- 7.1 A Full Example: The Wumpus World 22

8 RELATED AND USEFUL LINKS 23

9 CONCLUSIONS 24

1 OVERVIEW

P-IndiGolog [8] is an agent architecture completely programmed in Prolog which intends to realize the IndiGolog logic-based interleaved agent account of sensing, planning, and execution [2, 11, 6, 4, 1, 12]. P-IndiGolog is based on LeGolog
citeLevesque00-Legolog, a logic-programming architecture for running Golog [7] agent programs on the LEGO MINDSTORMS Robotic Invention System (RIS).[1] P-IndiGolog, however, can be used with any real robotic or virtual platform, provided the correct device managers are written. At this point, we have used P-IndiGolog to control the LEGO MINDSTORM robot already mentioned as well as the ER1 EVOLUTION robot[2] and several Internet and System agents. We plan to use it to control the RWI R21 robot too.[3]

This manuscript is intended to give a quick overview of the P-IndiGolog architecture *internal implementation*. Thus, it is mainly aimed for the programmer/developer of the architecture. Nevertheless, the regular user may just jump directly to Section 6 in order to find instructions and guidelines on how to develop domain applications for the P-IndiGolog agent system.

A quick graphical overview of the complete architecture is shown in Figure 1. The outline of this manual is as follows:

Section 2: Describes the logical directory structure of P-IndiGolog. The architecture is modular divided into many files and each file is placed in a particular directory depending on its role within the framework.

Section 3: Discusses the core of the architecture, namely, the main top-level control cycle and the module in charge of communicating with the external devices and environments.[4]

Section 4: Discusses the role of the evaluation procedure or temporal projector.

Section 5: Outlines the form and interface of the device managers, which are the modules in charge of managing each external device or environment (e.g., a robot platform). After reading this section, the reader should be able to design new device managers to operate new robots or environments.

Section 6: Provides detailed guidelines on how to use P-IndiGolog for a real domain application. This section amounts to the "User Manual" and should be sufficient for anybody wanting to use P-IndiGolog.

Section 7: Discusses future improvements.

Section 8: Lists related pointers and references.

Section 9 Draws conclusions.

---

[1] See `http://mindstorms.lego.com` and `http://www.cs.toronto.edu/cogrobo/Legolog/index.html`

[2] See `http://www.evolution.com/`

[3] See `http://www.irobot.com/rwi/p06.asp`

[4] From now on, the external, real-world, systems that will interact with P-IndiGolog will be refer to devices (e.g., a real robotic platform) or environments (e.g., the Internet, a file system).

2 DIRECTORY LOGICAL STRUCTURE

The files conforming the whole implementation are scattered among many different directories. Knowing the logical structure facilitates the task of finding files and locating specific code depending on its nature. In order to use P-IndiGolog, the global environment variable `PATH_INDIGOLOG` must point to the architecture root directory (e.g., `PATH_INDIGOLOG=/home/ssardina/Code/indigolog`). Environment variable `PATH_INDIGOLOG` is often used to localize libraries and specific files. From the architecture initial directory, the logical directory structure is as follows:

`Doc/` Documentation, manuals, guides, logos, etc. For instance, this manual is inside this directory.

`Env/` All code related to the handling of the external environments. All device managers (see Section 5) and the environment manager itself (see Section 3.2) are located in this directory.

`Eval/` Temporal projectors or evaluation procedures (see Section 4).

`Interpreters/` Top-level architecture module (see Section 3.1) and the any transition system available.

`Lib/` Compatibility and tool libraries, global definitions.

`Examples/` Domain applications (e.g., elevator controller, delivery coffee robot, etc.). For legibility and modularity, each application should be stored in its own subdirectory (Section 6).

`Old/` Old code that is out of use but that we may want to keep.

`Temp/` Temporal directory.

2.1 Libraries Provided

Among others, these are some of the libraries provided in directory `Lib/`:

- `eclipse_swi.pl`: SWI library for compatibility with ECLIPSE Prolog.
- `common.pl`: Prolog independent common library.
- `tools_xxx.pl`: tool library for Prolog `xxx`. Currently, we have special libraries for ECLIPSE Prolog (`tools_ecl.pl`) and SWI Prolog (`tools_swi.pl`). These libraries usually include library `common.pl`
- `systemvar.pl`: library used by domain applications' main files to perform Prolog initializations.
- `er1actions.pl`: library for translating ER1 actions to their low-level representations.

3 THE CORE

The core of the P-IndiGolog is made of two modules, namely, the top-level *main cycle* and the *environment manager*. The former implements the main loop of the system which is based on the well-known *sense-think-act* loop in the agent community [5]. The latter module provides the interfaces with the external world, which is viewed as a set of different devices and environments. So, for example, whenever the main cycle produces an action to be executed, it passes the action to the environment manager which, in turn, will communicate with the corresponding device in order to execute the action in question. Similarly, whenever a device or environment produces an exogenous event, this is passed to the environment manager which, in turn, will pass it to the main cycle as it corresponds.

3.1 The Top-Level Main Cycle

The *top-level main cycle* is intended to execute a *sense-think-act* interleaved loop in the spirit of many state of the art agent systems [5]. In a nutshell the cycle repeats the following four steps continuously: (i) roll forward the database if necessary; (ii) check for exogenous events that have occurred; (iii) calculate the next program step; and (iv) if the step involves an action, execute the action.

The execution of an agent program `E` is started in P-IndiGolog by calling predicate `indigolog(E)`. First, `init/0` is called to perform some system-wide initializations. Then, the top-level *main cycle* is started by calling `indigo/2` with the main program and the empty history. When the cycle terminates, `fin/0` is called to perform system-wide finalization routines.

So, `indigo/2` implements the top-level cycle. It be found in file `Interpreters/indigolog.pl` and is depicted in Figure 2. Roughly speaking, the cycle can be described as follows:

1. Perform **mandatory rolling forward** of the database by calling `handle_rolling/2` (e.g., whenever the history has grown excessively long).

2. Handle all pending **exogenous events** by calling `handle_exog/2`.

3. Compute the next **legal transition step** by calling `mayEvolve/5`, which, in turn, uses the underlying transition system. Depending on the outcome of this step, the cycle proceeds as follows:

   3.1. If the step was **interrupted** by an exogenous event before termination, then abort step and jump to step (1) without changing the current configuration (i.e., the program and history).

   3.2. If the current configuration was found to be **final**, then *terminate successfully* the top-level cycle. At this point, `indigo/2` just succeeds.

   3.3. If a **legal transition step** was indeed found, then do the following in priority order:

   3.3.1. If the step found *does not involve a new action* (i.e., the history remains the same), then restart the cycle in (1) with the new remaining program but the same history.

   3.3.2. If the step found involves a *simulated* action (i.e., action of the form `sim(A)` for some real action `A`), then ignore the action and restart the cycle in (1) with the new remaining program but the same history.

3.3.3. If the step found is the *waiting* meta-action `wait`, then perform optional rolling forward and, then, wait for an exogenous action to occur by using predicate `doWaitForExog/2`. After an exogenous event happen, restart the cycle in (1) without including the action `wait`.

3.3.4. If the step found is a `show_debug` action, then start debugging by calling `debug/3` projector tool. When finished, restart cycle in (1) without including the `show_debug` action.

3.3.5. If the step involves a `halt_exec` action, then restart the cycle in (1) with the empty program.

3.3.6. If the step involves an `abort_exec` action, then restart the cycle in (1) with the always failing program `?(false)`.

3.3.7. If the step involves an `pause_exec` action, then a PROLOG break point is inserted to the main cycle. It is not possible to use the PROLOG prompt to perform queries (for debugging mainly) and restart the program execution by just typing `Ctrl+D`.

3.3.8. If the step found is a `stop_interrupts` action, then just restart the cycle in (1).

3.3.9. Finally, if the step found involves a particular *domain action*, then execute the action in question in the real world by calling `indixeq/3` and, after that, restart the cycle in (1) with the remaining program and the new updated history.

The main cycle makes use of the following auxiliary tools:

- `now(-H)`: `H` is the current history in the system. All the actions in `H` have already been executed.
- `indi_exog(-A)`: `A` is an exogenous action that has occurred but has not yet been handled, i.e., has not been incorporated into the current history.
- `doingStep/0`: states whether a next-step-transition is being computed.

Predicate `indigo/2` uses several important tools:

- `handle_rolling(+H, -H2)`: implements the *mandatory* rolling forward of history `H` to history `H2`. The history must be rolled forward in certain circumstances (e.g., the history has grown up to long). The predicate uses `must_roll/1` and `roll_db/2` from the temporal projector (Section 4).
- `pause_or_roll(+H, -H2)`: implements the *optional* rolling forward of current history `H` to history `H2`. In this case, the history should be roll forward if there is sufficient time to do so. The predicate uses `can_roll/1` and `roll_db/2` from the temporal projector (Section 4).
- `handle_exog(+H, -H2)`: handles the pending exogenous actions which are stored under clause `indi_exog/1`. Basically, this amounts to adding all pending exogenous actions into the current history defined by `now/1`.
- `indixeq(+A, +H, -S)`: execute domain action `A` at history `H` in the real world with sensing outcome value `S`. The predicate interacts with the *environment manager* via predicate `execute_action/4`, which will order the execution of the action in the corresponding device

manager. If the sensing outcome is bound to term `failed`, then `action_failed/2` is called. Otherwise, `indixeq/3` updates the `now/1` predicate by appending action `A` to the current system history.

- `mayEvolve(+E1,+H1,-E2,-H2,-S)`: this may be the most important predicate as it defines the transition between one state of the system and the next one. In a nutshell, it says to system configuration `(E1,H1)` can evolve to configuration `(E2,H2)` under step type `S`. If a legal step could be found (i.e., `S=trans`), `E2` would stand for the remaining program and `H2` for the new history.

  `S=trans` configuration `(E1,H1)` can legally advance one step to configuration `(E2,H2)` w.r.t. the underlying transition system.

  `S=final` configuration `(E1,H1)` can legally terminate successfully w.r.t. the underlying transition system. Variables `E2` and `H2` have no meaning here.

  `S=exog` an exogenous event occurred during the computation of `mayEvolve/5`. So far, variables `E2` and `H2` have no meaning in this case. However, in the future, these variables may be used as partial information regarding the computation performed by `mayEvolve/5` up to the point of interruption.

  `S=failed` the current configuration `(E1,H1)` cannot terminate and cannot make any legal transition w.r.t. the underlying transition system. In this case, the configuration has reached a so-called dead-end and the program is "stuck". Variables `E2` and `H2` have no meaning in this case.

  As expected, predicate `mayEvolve/5` depends on an underlying transition system. Such transition system should be defined by two predicates: `trans/4` and `final/2`, which are implemented in file `Interpreters/transfinal.pl` and have the following interpretation:

  – `trans(+E1,+H1,-E2,-H2)` succeeds if there is a legal transition from configuration `(E1,H1)` to configuration `(E2,H2)`.

  – `final(+E,H)` succeeds if `(E,H)` is a terminating configuration.

- `indigo2/2`: implements the *second* phase of the main cycle whenever a transition was found. If the transition found involves no new action, then `indigo/2` is called directly with the new program. However, if the transition involves a new action, then there are two general cases. If the action in question is a *system* action (i.e., an action that is not intended to be performed by our agent, but one that carries some special meaning in the top-level loop), then some particular task is performed and the main cycle is restarted by calling `indigo/2`. At the moment, the following are considered system actions:

  – Simulated actions: this is an action of the form `sim(_)` and it means that the action was just used for simulation and it should not be actually performed. Hence, it the step is just ignored and the `indigo/2` is called.

  – Wait for an exogenous action: this is an action called `wait` and it sets the system to just wait until some exogenous action happens. Then, the main cycle is restarted.

  – Call for debugging: this is an action called `show_debug`. The debugging predicate `debug/3` is called and the main cycle restarted.

– Halt/Abort system: this are meta-actions called `halt_exec` and `abort_exec`, respectively, that causes the system to terminate abruptly either successfully or by failing completely.

– Pause system: this is a meta-actions called `pause_exec` that causes the top-level cycle to pause by introducing a Prolog `break` point. It is possible then to perform queries using the Prolog top-level prompt and return to the program execution by doing `Ctrl+D`.

– Stop interrupts: this is an action called `stop_interrupts` and it is used to stop the handling of high-level interrupts.

If the action corresponding to the transition is *not* one of the above system actions, then it ought to be a domain specific action, i.e., one that the agent/robot should perform in the real world. As a result, the action is sent for real execution using predicate `indixeq/3` and the main cycle is restarted after that.

- `abortStep/0`: this predicate is called when the main cycle is in the process of computing an system evolution step (i.e., `mayEvolve/5` is executing) and an exogenous event arrives asynchronously. The predicate is generally coupled with the implementation for `mayEvolve/5` by appealing to event handling mechanisms in order to interrupt the execution of a Prolog goal. For example, `abortStep/0` may throw an exception that `mayEvolve/5` can recognize and act upon.

3.2 EM: The Environment Manager

The environment manager (EM) is tightly coupled with the main cycle and provides a complete interface with all the external devices, platforms, and real-world environments. In a nutshell, the EM is responsible of executing actions in the real world and gathering information from it in the form of sensing outcome and exogenous events. The full EM implementation can be found at `Env/env_man.pl`

Given a domain high-level action, the EM is in charge of: (i) deciding which device should execute the action; (ii) order its execution to the appropriate device manager; and (iii) collect the corresponding sensing outcome. Furthermore, the EM is continuously listening to the devices for the occurrence of exogenous events.

When the system starts, the EM starts up all device managers required by the application and sets up communications channels to them using TCP/IP stream sockets. Recall that each real world device or environment has to have a corresponding device manager that understands it. After this initialization process, the EM enters into a *passive mode* in which it asynchronously listens for messages arriving from the various devices managers. This passive mode should allow the top-level main cycle to execute without interruption until a message arrives from some device manager. In general, a message can be an exogenous event, a sensing outcome of some recently executed action, or a system message (e.g., a device being closed unexpectedly). The incoming message should be read and handled in an appropriate way, and, in some cases, the top-level main cycle should be notified of the occurred event.

Three different implementations are provided via `env_man_cycle/1` for realizing this passive mode. The first implementation is based on *software signals* or interrupts; the second one is based on systematic *after-events*; and the the third one is based on *multi-threading.* Whereas

the third one works on BSD systems only (e.g., Unix, Linux, etc), the second one works with PROLOG implementations supporting after-events (e.g., ECLIPSE PROLOG), and the first one works only on multi-threading PROLOG implementations (e.g., SWI-PROLOG). More details on these implementations are described below.

3.2.1 The EM Interface

We now list and explain the predicates which conform the EM interface to the main top-level cycle of the architecture.

- `initializeEM/0`: this predicate is called just once to start the EM. Generally, this is done at the very beginning of the architecture initialization. Basically, it performs the following two steps:

  1. Initializes all the device managers required for running the application using `start_env/2`. The domain application specifies which devices are required to load by using predicate `load_environment/3` (see Section 6). This process involves, among other things, establishing a socket connection with each device manager in order to be able to communicate with them during the complete execution.
  2. Starts the EM passive-mode cycle using `start_env_cycle/1` depending on they kind of implementation chosen: *threads*, *after-events*, or *signals/interrupts*. The passive mode cycle is responsible of watching, passively, the connections to the various device managers for asynchronous messages (e.g., exogenous events, sensing outcomes, and system messages). This task should be programmed in such a way that an asynchronous event coming from any device manager could eventually interrupt the top-level execution if required. More information on how this cycle is implemented is given in the following section.

- `finalizeEM/0`: this predicate is the counterpart of `initializeEM/0` and is called when the application has completely finished. The predicate performs the following sequence of actions:

  1. Closes all open device managers by calling `close_dev/1`, which, in turn, sends each device a specific "closing" message.
  2. Terminates the EM passive-mode cycle using `finish_env_cycle/1`.

- `execute_action(+A, +H, +T, -S)`: this predicate is called by the top-level main cycle whenever an action needs to be executed in the real world, i.e., in one of the opened devices. The following actions are performed in sequence:

  1. `execute_action/4` finds out which device manager is in charge of the actual execution of the action in question. In principle, each action must only be executed by one, and only one, device specified by the application via predicate `how_to_execute/3` (see Section 6). Such predicate should also specify the code associated with the corresponding domain action (e.g., the high-level action `rotateLeft` may correspond to action number id 3).
  2. Once the device in question is found, a special message of the form `[execute, N, T, Code]` is sent to its corresponding device manager to order the action execution. `Code` is

the action id code, `T` is the action type (sensing, nonsensing, etc.), and `N` is the current action number sequence.[5]

3. `execute_action/4` predicate *waits* until the action sensing outcome is received from the corresponding device manager. This will happen, asynchronously, when `got_sensing(N, Outcome)` is asserted into the database.[6] Finally, the sensing outcome is bound to variable `S` and the predicate succeeds.

3.2.2 EM Passive-mode Cycle

There are currently three different implementations for realizing the EM passive-mode cycle. However, the three are based on a single predicate `em_one_cycle/1` which performs one single interation waiting for data comming from the open devices. Predicate `em_one_cycle/1` takes as input the numbers of seconds it should wait for incomming data and basically it waits that much for incomming data from the device managers and if any `handle_levents/1` is called to handle these data (see below). So, there are three ways to use predicate `em_one_cycle/1`:

(a) **Threads:** an independent thread is in charge of executing `em_one_cycle/1` with ID (alias) `em_thread`. In that way, the main thread can continue the execution of the agent program independently. The thread `em_thread` calls `em_one_cycle/1` with parameter `block` which means to wait indefinitely so that the thread in question would block itself waiting for data to arrive at any of the device managers' sockets. When a message arrives at any of these sockets, the special thread reads them all, calls `handle_levents/1` and, finally, restart its cycle again.

(b) **After-events:** an *event-after* mechanism provided by some PROLOG implementations like ECLIPSE is used. An *after-event* is a syncrhonous PROLOG event that is triggered systematically after some fixed period of time (e.g., 1 second). A predicate is associated with an event to handle it. So, this EM implementation defines a special event-after `em_cycle_eventAfter` to be fired every 2 seconds. The event is handled by clause `em_cycle_eventAfter_handler/0` which is basically a call to `em_one_cycle/1` with waiting of 0 seconds so that the process does not block at all. So, if no message from the device managers is waiting, the handler succeeds immediately. If, however, there are messages waiting at any of the corresponding sockets, they are collected and passed to `handle_levents/1` to be processed. Notice that the event handler must always succeed.

(a) **Signals:** a special *input-output interrupt* hanlder is defined to asynchronously handle the messages coming from the device mangers. The handler predicate is called `handle_io/0`, which is asynchronously called whenever a message arrives to any socket associated with a device manager. Predicate `handle_io/0` then calls `em_one_cycle/1` without blocking; in this case it is know already that some message is waiting at some socket so the messages will be callected and processed with `handle_levents/1`. Notice that this implementation is not recommended as it relies on interrupts and only works in BSD systems.

[5]The EM carries a counter of executed actions.

[6]Notice that this approach assumes that sensing outcome is received sufficiently quickly after ordering the execution of an action.

The kind of EM to be used can be specified by the domain application by using `set_option/2` with option `type_EM`. The different types can be `thread`, `eventafter`, or `signal`. By default, the *thread* implementation is used. The type of the EM can always be accessed by querying `type_manager/1`.

Now, whenever there is some data waiting to be read from one or more device managers, predicate `em_one_cycle/1` will collect all of them into a single list and call `handle_levents/1` to handle them as it corresponds. Predicate `handle_levents/1` uses `handle_events/1` to handle each individual message at a time in the following prioritized way:

1. First, all messages corresponding to a *sensing outcomes* are handled. The sensing outcome is translated if needed using `translateSensing/3` and asserted into the database with a `got_sensing/2` clause.

2. Second, all messages corresponding to a *exogenous events* are handled. The message is translated with `translateExogAction/2` if nedded and asserted, temporarily, into the database with clause `got_exogenous/1`.

3. Third, all other messages are processed (e.g., device managers that have closed or unknown events).

4. Finally, if there was indeed or or more exogenous actions they are callected and passed to the top-level cycle predicate `exog_action_occurred/1` which would decide what to do with them.

As can be seen, the EM uses some sophisticated system tools which are only present in advance Prolog implementations. First, it appeals to TCP/IP sockets to communicate with the various device managers. Second, it either uses *interrupt handling*, *multi-threading*, or *synchrounous events* capabilities to implement the passive mode cycle. We think that this is not a major problem as most recent Prolog implementations (e.g., ECLIPSE, SWI, SICSTUS, etc.), support at least some of these features.

3.2.3 Socket Communication and Messages

As already explained, the communication between the EM and all device managers is achieved using TCP/IP sockets. The EM itself registers its own socket with id `em_socket`. A special socket will also be opened for *each* device manager. Each socket would carry the device manager own name as alias, which will be stored as the third argument in clause `env_data/3`. Two predicates, provided as part of a library, are provided to perform all communication along the sockets: `send_data_socket/2` and `receive_list_data_socket/2`.

The messages exchanged between the EM and the device managers are terms of the following form:

```
[SenderId, Message]
```

where `SenderID` identifies the sender of the message (either with its socket id or with a symbolic high-level name), and `Message` is a *list* containing the actual message. So far, there are four messages types recognized by the EM; two of them for reporting exogenous events and sensing outcomes and two others for reporting system messages. Given that the messages used are of a uniform form, tools for sending and reading messages from sockets are provided as part of library

tools_xxx. In concrete, send_data_socket/2 is used to send a message via a socket; whereas receive_list_data_socket/2 is used to read *all* messages waiting at a socket.

In the presence of one or more messages from the device managers, the EM collects all messages into a list of current events and calls handle_levents/1 to handle all them. A message can be one of the followings:

[SenderId, [end_of_file]] Device SenderId has been closed unexpectedly. The device is, therefore, deleted from the set of opened devices.

[SenderId, [sensing, N, CodeO]] Device SenderId has reported the sensing outcome represented by code CodeO corresponding to the recently executed action number N. After translating the code into the domain representation using translateSensing/3, the corresponding got_sensing/2 clause is asserted into the database.

[SenderId, [exog_action, CodeA]] Device SenderId has reported the exogenous event represented by code CodeA. After translating the code into its domain representation using translateExogAction/2, the corresponding got_exogenous/1 is asserted into the database.

After handling all messages as just described, the EM performs one final task before setting itself into its (default) passive mode. Namely, if at least one of the messages received was an exogenous event, it collects all of them and calls clause exog_action_occurred/1 from the top-level main cycle. The top-level main cycle is, hence, responsible of handling all these exogenous events as it corresponds. For example, if the top-level is in process of computing a next step (i.e., doingStep/0 succeeds), it may decide to abort it by calling abortStep/0.

4 TRANSITION SYSTEM AND TEMPORAL PROJECTOR

In this section, we quickly explain two other modules that, though not technically part of the architecture's core, are very important and mandatory for the evolution of the main cycle execution. These are the *transition system* and the *temporal projector*. The former is used to compute the evolution of the high-level program, whereas the latter is in charge of the projection task.

4.1 The Transition System

A configuration in P-IndiGolog is a formed by the current high-level program and the current history. A transition system states the rules under which a configuration may evolve to another configuration. The evolution may or may not involve a new domain action, which is consequently executed in the right device or environment. As noted in Section 3.1, computing this evolution step is required in step 3 of the top-level main cycle.

Every transition system must provide an implementation for the following two predicates:

- trans(P,H,P2,H2): configuration (P,H) can perform a single step to configuration (P2,H2).
- final(P,H): configuration (P,H) is terminating.

Optionally, the transition system could also provide the following transitive closures of the above two predicates:

- ttrans/4: reflexive transitive closure of trans/4.
- ttrans/5: finite reflexive transitive closure of trans/4 with an upper bound on the number of transitions stated by the last argument.
- tfinal(P,H): transitive closure of trans/4 and tfinal combined. Notice all transitions should involve no action whatsoever.

Every transition system implementation should be stored in directory Interpreters. The default transition system provided in file transfinal.pl corresponds to the agent language IndiGolog, an extension of ConGolog to support incremental execution of programs.

4.2 The Temporal Projector

The temporal projector or evaluation procedure is in charge of evaluating formulas w.r.t. some system histories. To that end, every projector should provide a clause eval/3 as its main predicate. A goal eval(+F,+H,?B) is meant to say that formula F has truth value B (usually true or false) at history H.

In the context of the situation calculus, there are many evaluation procedures available depending on the type of action theory chosen (basic action theory [9], guarded theories [3], fluent calculus theories [13], etc.). For the sake of uniformity, each projectors are meant to be inside directory Eval and with name eval_ttt.pl where ttt stands for the type of projector. For example, eval_bat.pl is the basic action theory evaluation procedure and eval_gat.pl is a guarded action theory projector.

The temporal projector is used in two places. First, it is heavily used by the transition system to compute the system evolution. Second, the evaluation procedure provides a number of tools which are called from the top-level main cycle of the architecture to perform some bookkeeping tasks.

Besides eval/3, the following list shows the predicates that should be provided by any evaluation procedure. We first list the predicates that will be used by the top-level main cycle:

- initializeDB/0: initializes the projector.
- finalizeDB/0: finalize the projector.
- must_roll(+H1): succeeds if it is completely necessary to roll forward.
- can_roll(+H1): succeeds if it is worth rolling forward in case of sufficient idle time.
- roll_db(+H1, -H2): rolls forward the database from history H1 into the new history H2.
- handle_sensing(+A, +H, +S, -H2): H2 is H plus new action A with sensing result S.
- debug(+A, +H, -S): perform debug task with current action A, sensing outcome S, and history H.
- system_action(+A): action A is system action for the projector (e.g., the action e(A,S) is used to store sensing outcomes inside the history).

Finally, the following predicates shall be used, in general, by the underlying transition system to compute evolutions of the high-level program being executed:

- `eval(+F, +H, -B)`: formula `F` has truth value `B` at history `H`
- `sensing(+A, -L)`: action `A` is a sensing action with a list `L` of possible outcomes
- `sensed(+A, -S, +H)`: action `A`, when executed at history `H`, got sensing result `S`
- `inconsistent(+H)`: last action turned history `H` inconsistent, i.e., impossible
- `domain(-V, +D)`: object `V` is an element of domain `D`
- `getdomain(+D, -L)` : `L` is the list representing domain `D`
- `calc_arg(+A1, -A2, +H)` : action `A2` is action `A1` with its arguments replaced at history `H`
- `before(+H1, +H2)`: history `H1` is a prefix (i.e., previous) of history `H2`
- `assume(+F, +V, +H1, -H2)`: `H2` is the history resulting from assuming fluent `F` to have value `V` at history `H1`

We finally note that the evaluation procedure to be used has substantial impact on the way that a domain application is specified. Therefore, the user should refer to each evaluation procedure in order to learn how to axiomatize a specific domain. In fact, different domains may require different temporal projectors.

5 DEVICE/ENVIRONMENT MANAGERS

A P-IndiGolog application will generally operate in complex scenarios by interacting with different devices (e.g, a robot) and environments (e.g., the Internet). For instance, opening a door may be performed by some real robot, whereas opening a web page may be performed by some software agent in charge of downloading multimedia files.

Each external device or environment (e.g., a robot platform, the Internet network, or just a simulator device) can be available to a P-IndiGolog domain application by simply implementing a so-called *device manager*. A device manager is analogous to software *drivers* in operating systems. As it is well-known, a *driver* is a program that determines how a computer will communicate with a peripheral device. Similarly, a *device manager* is a program that determines how P-IndiGolog will communicate with an external device or environment. Therefore, P-IndiGolog "talks" to the real artifact/environment via its specific device manager.

Basically, a device manager should be able to perform the following three tasks:

1. Execute domain actions in the device.
2. Gather actions' sensing outcomes from the device.
3. Gather exogenous events generated from the device and potentially generate them itself.

All device managers are programmed in Prolog and are meant to execute *independently* (i.e., as a separate independent process) of the P-IndiGolog core. In order to facilitate the creation of new device managers, the general structure of the managers is fixed in advance. Hence, all device managers have a common structure specified in file `env_gen.pl` and a private, local, implementation with a fixed and clear interface. In the common structure, a device manager interface is composed of just two predicates, namely, `start/0` and `finalize/0`. The former is in charge of starting up the device manager, whereas the latter is in charge of terminating it.

There are three important issues to understand about how device manager operate:

- As explained before, device managers communicate with the EM using TCP/IP sockets. Hence, a device manager needs to be told, when started, the specific address where the EM will be listening for messages. To that end, device managers should be called with two arguments of the form `host=<host-id>` and `port=<n>`, where `<host-id>` and `<n>` stand for the host address (e.g., an IP or a host domain name) and the port number, respectively, where the EM's communication socket is.
- Device managers usually print debugging information as they execute and may potentially interact with the user (e.g., the simulator device asks the user for the actions' sensing outcomes). Therefore, device managers need a terminal window to print information and potentially interact with the user.
- A device manager must be called so that the goal `start/0` is automatically started.

To better explain these three issues, let us consider a typical device manager initialization as performed by the EM:

```
xterm -e ''pl host=192.168.9.1 port=8023 -b Env/env_rcx.pl -e start''
```

This command opens a Unix xterm window in which an SWI-Prolog engine is executed with file `env_rcx.pl`, the device manager of the LEGO MINDSTORM RCX robot, being consulted. The Prolog engine receives two command line parameters, namely, "`host=192.168.9.1`" and " `port=8023`" defining the EM's socket address at 192.168.9.1:8023. Finally, goal `start` is set as the top-level goal for the Prolog engine. If everything goes as expected, an xterm window should be opened, a Prolog engine should be started there, Prolog code `env_rcx.pl` ought to be consulted, and goal `start` must be started.

5.1 Device Manager Operation

In this section, we briefly give an overview of how device managers function. This corresponds, actually, to the core of every device manager which is fixed in advance and is, mainly, device independent.

When goal `start/0` is started, it first collects the socket address of the EM that was passed as command line argument and stores it for future reference by asserting a `env_manager/2` clause. Then, the optional `debug` command line argument is read, if available, and the corresponding debug level for the device manager is set. Finally, `start/0` sets up a permanent socket connection to EM by creating a socket connection with id `env_manager` to the above address. Predicate `start/0` continues by collecting any other command line argument of the form `namearg=value` into a list `L` and calls `initializeInterfaces/1` which is in charge of the initialization of any device manager specific interface or process required by the such as a TCL/TK window or the RCX listener process.

In general, a device manager will open several stream-connections (sockets, pipes, etc) to communicate with various other processes. At least, there will be one such stream to communicate with the EM, but there may be others to communicate with TCL/TK windows, remote processes (e.g., the ER1 server), or local processes (e.g., a process downloading a file). In order to communicate with all these processes in an asynchronously fashion, each stream is is *registered* to the device manager by asserting a `listen_to/3` fact into the database.[7] In concrete, a clause `listen_to(T, Name, S)` states that the channel-stream `S` of type `T=stream/socket`, and identification `Name` must be listened to in an asynchronously. As expected, one of the channels to watch for is the socket `env_man` corresponding to the connection with the EM. In general, the channels to watch for are set at the outset of the device manager and released at the end of it. Nonetheless, there may be cases where a device appeals to *dynamic* streams that are active only for a short period of time (e.g., a temporal process implementing the action of downloading a file from the Internet). Hence, in the general case, clauses `listen_to/3` are asserted and retracted from the database dynamically.

After all interfaces and processes required to run the device manager are initialized, `start/0` enters into its *passive-mode cycle* by calling predicate `main_cycle/0`. In a nutshell, `main_cycle/0` waits for messages to arrive at one of the channels being listened to (e.g., a device interface/process socket or the socket associated to the EM). At that point, predicate `handle_streams/1` is called with each "ready" channel-stream.

The only provided handler common to all device managers is the one corresponding to the stream associated to the EM. In other words, `handle_stream(env_manager)` is fixed in advance and already provided. The most important message comming from the EM is the one of the form `[execute, N, Type,CodeAction]`. Such a message is ordering the device manager to *execute* the action `CodeAction` of type `Type` and number `N`. Upon receiving this special message, `handle_stream(env_manager)` would first call the device-specific predicate `execute/4` to perform the actual execution of the action in the device, and, after that, it will send a `[sensing, N, S]` message to the EM to report the corresponding sensing outcome `S` by appealing to tool `report_sensing/4`.

The device-manager local predicate `execute(+Action, +Type, +N, -S)` is responsible of the actual execution of `Action` as well as of returning the action sensing result in variable `S`. For example, if the device in question corresponds to the simulation environment, executing the action would simply amount to printing the action term to standard output, and reading its outcome would reduce to reading the sensing outcome from the user. On the other hand, if the device corresponds to the LEGO RCX brick, the predicate would send the action to the brick via infrared communication and receive its sensing result from it too.

Another important message that a device manager can receive from the EM is `[terminate]`. Such a message is ordering the device manager to terminate its execution. In that case, the corresponding handler `handle_stream(env_manager)` calls predicate `finalize/0`, which is in charge of cleanly terminating the device manager. This involves closing and deregistering all streams (including the one associated with the EM), finalizing all local interfaces and processes, and finally halting the execution.

[7]Notice that this database is totally independent of the P-IndiGolog core module as each device manager runs in its own Prolog engine.

5.1.1 How to Develop a New Device Manager

As already said, the core of any device manager is fixed by the architecture and common to all device managers. Here we explain what code should be appended to the static code already provided in order to develop a new device manager for some new device or environment. To that end, the programmer should provide the following extra predicates:[8]

- `initialize_interfaces/0`: starts up all the necessary interfaces and processes that are required for the device manager. We recall that for every channel-stream used, a corresponding `listen_to/3` fact has to be registered in the database. The fact in question would usually store the type of channel (stream or socket), the channel identifier, and a symbolic name.
- `finalize_interfaces/0`: ends up all local open interfaces and active processes (e.g., close a TCL/TK window and sends special terminating codes to the RCX brick.). The corresponding `listen_to/3` clauses should also be removed from the database.
- `execute(+A, +T, +N, -S)`: executes action `A` of type `T` and number `N` in the device. Variable `S` is then bound to the action's sensing outcome. If the action fails to execute, then `S` should be bound to atom `failed`.
- `handle_stream(+C)`: specifies how to handle messages from the registered channel `C`. There must be one `handle_stream/1` clause for each registered channel.
- `name_dev(+NameDev)`: this is a fact defining the name of the device manager to be `NameDev` (e.g., `er1` for the ER1's device manager and `sim` for the simulator environment).

In order to report exogenous events and sensing outcomes to the EM, the programmer must make use of the following two special tools already defined in the static part of every device manager (i.e., in file `env_gen.pl`):

- `report_exog_event(+A, ?M)`: reports the occurrence of exogenous action `A` to the EM. Optionally, if bound to some ground atom, message `M` is printed in the device manager's standard output.
- `report_sensing(+A, +N, +S, ?M)`: reports sensing outcome `S` for action `A` with number `N` to the EM. Optionally, if bound to some ground atom, message `M` is printed in the device manager's standard output. If the device manager wants to report a failure on the action execution, it can do so by reporting the term `failed` as sensing outcome.

6 DEVELOPING DOMAIN APPLICATION (USER MANUAL)

Probably the most relevant thing to learn for the P-IndiGolog user is how to specify and develop a real-world domain application. We shall address this issue here. Any domain application must specify three well-defined sections:

1. An *axiomatization of the dynamics of the world.* This axiomatization would strongly depend on the temporal projector to be used (see Section 4).

[8] These predicates are provided in a separated file, usually named `env_xxx.pl`, which would include file `env_gen.pl`

2. One or more high-level *agent programs* that will dictate the different agent behaviors available. These programs depend on the transition system to be used (e.g., IndiGolog).

3. *Execution information* stating how to run the domain application in the real-world devices. This accounts to providing what external devices the application relies on (e.g., the device manager for the ER1 robot), and defining how high-level actions are actually executed in these devices (e.g., what device is in charge of performing each high-level action). In addition, information on how to translate high-level symbolic actions and sensing results into the device manager codes, and vice-versa, could be provided.

An application would generally consist of two files inside a special subdirectory in directory `Examples` (e.g., `Examples/Elevator/`). One file would be called the *main* file and would be Prolog-dependent. A main file is usually named `main_xxx.pl` where `xxx` stands for the Prolog to be used (e.g., `swi` for SWI-Prolog). The main file would contain all information described in point (3) above. The other file would be the *application* file and it would contain above points (1) and (2) (e.g., `Examples/Elevator/elevator.pl`).

6.1 The Domain Axiomatization and High-Level Programs

The user should write a domain axiomatization for the application in accordance with the the evaluation procedure to be used. Both the axiomatization and the high-level programs should be stored in a single file with a name referring to the application (e.g., `elevator.pl`).

By convention, we assume that all high-level controllers will be defined as procedures with names of the form `mainControl(ID)`, where `ID` stands for the *unique* identification of a controller (e.g., an application may have two controllers defined: `mainControl(0)` and `mainControl(1)`).

6.2 The main file "`main_xxx.pl`"

As expected, the "main file" is the head file of a domain application. It is the file to be loaded by the user and the one responsible of loading all required files to run the application. In general, this file will be sensitive to the Prolog implementation and should be named `main_xxx.pl` where `xxx` stands for the Prolog platform.

The main file starts by including a common library named `systemvar.pl` which provides global variable definitions and Prolog-dependent initializations (e.g., loading specific Prolog-dependent libraries) that are required by the whole architecture. After that, the main file loads the following modules/files:

1. Extra libraries required to run the specific application (e.g., constraint libraries, etc.);

2. the top-level main cycle (i.e., `Interpreter/indigolog.pl`);[9]

3. the environment manager (i.e., `Env/env_man.pl`);

4. the evaluation procedure to be used (i.e., `Eval/eval_bat.pl`);

5. the application specification, that is, the domain axiomatization and the high-level programs to be used.

[9]Currently, the main cycle would also load the transition system, but it could eventually be loaded independently.

In addition, the main file should provide the following three extra predicates:

- type_prolog(-T) specifies the Prolog platform to be used (so far, SWI (swi), ECLIPSE (ecl), and "vanilla" Prolog (van) are recognized). Currently, this predicate is implemented in the systemvar library.
- server_port(-N) specifies N to be the port number for the EM socket. Some Prolog socket implementations, like ECLIPSE and SWI's one, are able to automatically find a free port and N can be left unbound. Other Prolog platforms, however, may require a concrete specific port number.
- main/0 collects all procedures with name of the form mainControl(ID) and asks the user to determine which controller to start. Then, the corresponding high-level program is started with the empty history.

Lastly, global Prolog-dependent settings may be stated in this file. Examples of these settings are compilation directives, garbage collector options, variable representation scheme, etc.

6.2.1 Execution information: how to execute programs in the world?

The main file also contains all information required to execute the high-level programs into the real-world devices and environments. This amounts, basically, to the specification of which device managers should be loaded at the outset of the architecture initialization, which device manager executes each high-level domain action, and the translation between high-level domain action names and sensing outcomes to their low-level device manager representations.

The main file must specify the following predicates:

- load_device(+Name, -Command, [+Host, +Port]): device manager Name must be loaded using the shell system command Command. Host and Port provide the EM socket address.

  The command line should be designed such that: (i) a terminal is provided to the device manager (e.g., an xterm); (ii) the socket address of the EM provided in variables Host and Port are passed as two arguments of the form host=<Host> and port=<Port>; and (iii) the start/0 predicate should be set as the entrance goal. Optionally, an argument of the form debug=n could be used to set the debug level of the device manager to level n.

  A typical ECLIPSE and SWI commands under Unix/Linux would look as follows:

```
xterm -e ‘‘eclipse host=<Host> port=<Port> -b Env/env_sim.pl -e start’’

xterm -e ‘‘pl -t start -f Env/env_sim.pl host=<Host> port=<Port> debug=1’’
```

  Some device managers, like the one for the ER1 robot, may require extra, device manager dependent, parameters to be passed along (e.g., the robot's own socket address):

```
xterm -e ‘‘eclipse host=Host port=Port \
                  er1host=’er1.cs.toronto.edu’ er1port=9000 \
      -b Env/env_er1.pl -e start’’
```

The corresponding clauses for the device managers in charge of the RCX and the ER1 robots would look are given bellow. Notice that both device managers are meant to run in independent ECLIPSE PROLOG processes.

```
load_environment(rcx, Command, [Host, Port]):-
      concat_atom(['xterm -e ', 'eclipse -g 10M',
                   ' host=', Host, ' port=', Port,
                   ' -b Env/env_rcx.pl', ' -e ', ' start'], Command).

load_environment(er1, Command, [Host, Port]):-
      concat_atom(['xterm -e ', 'eclipse -g 10M',
                   ' host=', Host, ' port=', Port,
                   ' er1host=', 'er1.cs.toronto.edu', ' er1port=','9000',
                   ' -b Env/env_er1.pl', ' -e ', ' start'], Command).
```

To be able to reuse code, a file `Env/dev_managers.pl` is provided in order to predefine a set of device managers that can be used in multiple domain applications.[10] In concrete, the file contains definitions for predicate `device_manager(+Env, +P, -C, [+Host, +Port])` with the following interpretation: device manager `Env` on PROLOG platform `P` (`swi` or `ecl`) should be loaded using command `C` with EM socket address `Host/Port`. Hence, one can promptly reused these definitions and define `load_device/3` as follows:

```
% Load simulator, RCX and internet device managers
load_device(Env, Command, Address) :-
 member((Env,Type), [(simulator,ecl), (rcx,ecl),(internet,swi)]),
 (var(Address) ->
      Host=null, Port=null
 ;
      Address = [Host, Port]),
 device_manager(Env, Type, Command, [Host, Port]).
```

This definition would dictate to load three device managers: the simulator and LEGO RCX devices under ECLIPSE PROLOG, and the Internet environment under SWI PROLOG.

- `how_to_execute(+A, -Env, -Code)`: ground high-level domain action `A` is to be executed by device manager `Env` under the low-level code `Code`. The name `Env` of the device manager must correspond exactly to one of the devices specified with `load_device/3`. Otherwise, the EM will not be able to order the execution of the action in question to any device manager.

  For instance, if the action `lift_arm` is intended to be executed on the RCX robot under code 23, whereas action `moveFwd(Dist)` is intended to be executed on the ER1 platform under the low-level representation '`move <Dist> cm`', the following facts should be asserted:

[10] Currently, there are device managers for a simulator environment; the RCX LEGO robot; an internet-web, file-system and speech environment; the EVOLUTION ER1 robot; and a Wumpus World simulator.

```
how_to_execute(lift_arm, rcx, 23).
how_to_execute(moveFwd(Distance), er1, X) :-
            concat_atom(['move ',Distance,' cm'], X).
```

- `translateExogAction(+CodeAction, -Action)`: `CodeAction` is the low-level device manager representation for high-level domain action `Action`. If no translation is found, the high-level and low-level representations are assumed to coincide.

  For example, the ER1 exogenous event "`move done`" generated by its device manager should be translated into the high-level exogenous action `arrive`:

  ```
  translateExogAction('move done', arrive).
  ```

- `translateSensing(+Action, +SensorCode, ?Value)`: low-level sensing outcome representation `SensorCode` for a high-level action `Action` stands for the high-level sensing outcome representation `Value`. Again, if no translation is found, `SensorCode` and `Value` are assumed to coincide.

  For instance, the following two clauses translate a thermometer reading into a high-level boolean value representing whether it is hot or not:

  ```
  translateSensing(senseHot, N, true)  :- N>30.
  translateSensing(senseHot, N, false) :- N<=30.
  ```

7 TODO's

The following issues should need to be addressed:

1. The RCX's device manager provided in `Env/env_rcx.pl` may end up never consulting the RCX for exogenous events. This happens if the device is always required to execute an action. This is because execution of actions have, implicitly, more priority than the periodic query for exogenous events.

   This problem arises, for instance, with the elevator application where an action `ring` is, at some point, ordered to be executed continuously after some `reset` exogenous event is read. In that case, the device manager keeps sending execution commands to the RCX without ever querying for exogenous events.

2. Several improvements should be performed on predicate `abortStep/0`. Such predicate is the one called whenever an exogenous event occurs while IndiGolog is computing the next transition. So far, the predicate merely aborts the transition computation, so that the top-cycle can be re-initiated. However, if one wants to implement more "intelligent" behaviors (such us evaluating and, maybe transforming, the computation computed so far), extra parameters should be passed along (e.g., current program, current history, time, etc.)

3. Describe the action state in the device managers.

4. Talk about string compatibility between the environment manager and the device controllers.

5. Develop a more robust environment manager that deals with incomplete actions, no response from device managers, etc. The environment manager may query the devices for the state of certain incomplete actions.

6. When the main cycle hits a "wait" action it calls "doWaitForExog/0" which waits for an exogenous action to occur. The problem is that such predicate does a busy cycle looking for a new exogenous action entrance with indi_exog/1. It would be better to avoid the polling and instead use a socket at which the top level can just block.

7.1 A Full Example: The Wumpus World

Inside directory `Examples/Wumpus`, one can find the code for implementing the Wumpus World domain [10, Chapter 7]. The Wumpus World is a well-known example for reasoning and acting with incomplete knowledge. According to the scenario, the agent enters a dungeon in which each location may contain the Wumpus (a deadly monster), a bottomless pit, or a piece of gold. The agent moves around looking for gold and avoiding death caused by moving into the location of a pit or the Wumpus. The agent has an arrow which she can throw as an attempt to kill the Wumpus. Also, the agent can sense the world to get clues about the extent of the dungeon, as well as the location of pits, gold pieces, and the Wumpus.

To implement the domain, the following specific files are used:

`Examples/Wumpus/main_swi.pl`: This is the main file for the example.

`Examples/Wumpus/wumpus.pl`: This file contains the axiomatization for the domain and the different agent controllers in the IndiGolog programming language.

`Env/env_wumpus.pl`: This is the device manager for a virtual wumpus world. It provides an interface to execute the domain actions (e.g., `turn`, `go`, `smell`, `shoot`, etc.) as well as to produce sensing outcomes and exogenous events. In addition, the simulator can display the virtual domain using a Java Applet.

`Examples/Wumpus/WumpusApplet/WumpusApplet.java`: This file provides a Java-based graphical interface for displaying the behaviour of the agent within the Wumpus World.

`lib/alpha_star`: This file is a library providing path-planning algorithms which is used to program the agent's controller.

The example main file `main_swi.pl` is in charge of consulting the following files: (i) the main-cycle and transition system `indigolog.pl`; (ii) the temporal projector `eval_know.pl`; (iii) the environment manager `eval_man.pl`; (iv) the wumpus world axiomatization `wumpus.pl`. Furthermore, the main file states that the device `virtual_wumpus` should be loaded as follows:

```
load_device(Env, Command, Address) :-
        member((Env,Type), [(virtual_wumpus, swi)]),
        (var(Address) -> Host=null, Port=null ; Address = [Host,Port]),
        device_manager(Env, Type, Command, [Host, Port]).
```

The corresponding clause for `device_manager/4` is defined in file `Env/dev_managers.pl` as follows:

```
device_manager(virtual_wumpus_silent, swi, Command, [Host, Port]):-
        main_dir(Dir),
        wumpus_location(IPW, PORTW),
        wumpus_config(TIDRun,Size,PPits,NoGolds,TIDScenario),
        term_to_atom(TIDRun, IDRun),
        term_to_atom(TIDScenario, IDScenario),
        concat_atom([Dir,'Env/env_wumpus.pl'], File),
        concat_atom(['pl ', ' -t ', ' start', ' -f ', File,
                     ' host=', Host, ' port=', Port,' debug=1',
                     ' ipwumpus=', IPW, ' portwumpus=', PORTW,
                     ' ppits=', PPits, ' nogolds=', NoGolds, ' size=', Size,
                     ' idrun=\'', IDRun, '\' idscenario=\'', IDScenario,'\'',
                     ' 1>/dev/null 2>/dev/null'], Command).
```

Notice that the definition of the virtual Wumpus simulator relies on several options defined via predicates `wumpus_location/2` and `wumpus_config/5` which are both defined also in central file `main_swi.pl`.

Finally, the main file also contains the following two directives to set-up the debug level and the waiting step option to zero:

```
:- set_option(debug_level,0).
:- set_option(wait_step,0).
```

8 RELATED AND USEFUL LINKS

Robot platforms:

EVOLUTION Robotics: http://www.evolution.com/

B21 RWI robot: http://www.irobot.com/rwi/p06.asp

AIBO (Sony Dogs): http://www.us.aibo.com/

LEGO MINDSTORMS:

http://mindstorms.lego.com/eng/default.asp

http://www.vorlesungen.uni-osnabrueck.de/informatik/robot00/ftp/lego.html

http://www.crynwr.com/lego-robotics/

Prolog Systems:

SWI Prolog http://www.swi-prolog.org/

ECLIPSE Prolog http://www.icparc.ic.ac.uk/eclipse

SICSTUS Prolog http://www.sics.se/sicstus/

Agent Systems:

Golog/ConGolog/IndiGolog: http://www.cs.toronto.edu/cogrobo/systems.html

3APL: http://www.cs.uu.nl/3apl/

Fluent Calculus and FLUX: http://www.fluxagent.org/

CologNet: http://mas.colognet.org/implementation.html

9 CONCLUSIONS

References

[1] Giuseppe De Giacomo, Yves Lespérance, Hector Levesque, and Sebastian Sardiña. On the semantics of deliberation in IndiGolog – from theory to implementation. In .F̃ensel, F. Giunchiglia, D. McGuinness, and M. A. Williams, editors, *Proceedings of Eighth International Conference in Principles of Knowledge Representation and Reasoning (KR-2002)*, pages 603–614, Toulouse, France, April 2002. Morgan Kaufmann. 3

[2] Giuseppe De Giacomo and Hector Levesque. An incremental interpreter for high-level programs with sensing. In Hector J. Levesque and Fiora Pirri, editors, *Logical foundation for cognitive agents: contributions in honor of Ray Reiter*, pages 86–102. Springer, Berlin, 1999. 3

[3] Giuseppe De Giacomo and Hector Levesque. Projection using regression and sensors. In *Proceedingsof the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 160–165, Stockholm, Sweden, 1999. 13

[4] Giuseppe De Giacomo, Hector J. Levesque, and Sebastian Sardiña. Incremental execution of guarded theories. *ACM Transactions on Computational Logic (TOCL)*, 2(4):495–525, October 2001. 3

[5] R. A. Kowalski. Using meta-logic to reconcile reactive with rational agents. In K. R. Apt and F. Turini, editors, *Meta-Logics and Logic Programming*, pages 227–242. MIT Press, 1995. 5

[6] Yves Lespérance and Ho-Kong Ng. Integrating planning into reactive high-level robot programs. In *In Proceedings of the Second International Cognitive Robotics Workshop*, pages 49–54, Berlin, Germany, August 2000. 3

[7] 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. 3

[8] Hector Levesque and Maurice Pagnucco. LeGolog: Inexpensive experiments in cognitive robotics. In *Proceedings of the Second International Cognitive Robotics Workshop*, Berlin, Germany, August 2000. 3

[9] Fiora Pirri and Ray Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):261–325, 1999. 13

[10] Stuart Russell and Peter Norving. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003. 22

[11] Sebastian Sardina. IndiGolog: Execution of Guarded Action Theories. Master's thesis, Dept. Computer Science, University of Toronto, 2000. 3

[12] Sebastian Sardina. *Deliberation in Agent Programming Languages*. PhD thesis, Dept. of Computer Science, University of Toronto, 2005. 3

[13] Michael Thielscher. The fluent calculus. Technical Report CL-2000-01, Computational Logic Group, Artificial Intelligence Institute, Department of Computer Science, Dresden University of Technology, April 2000. 13

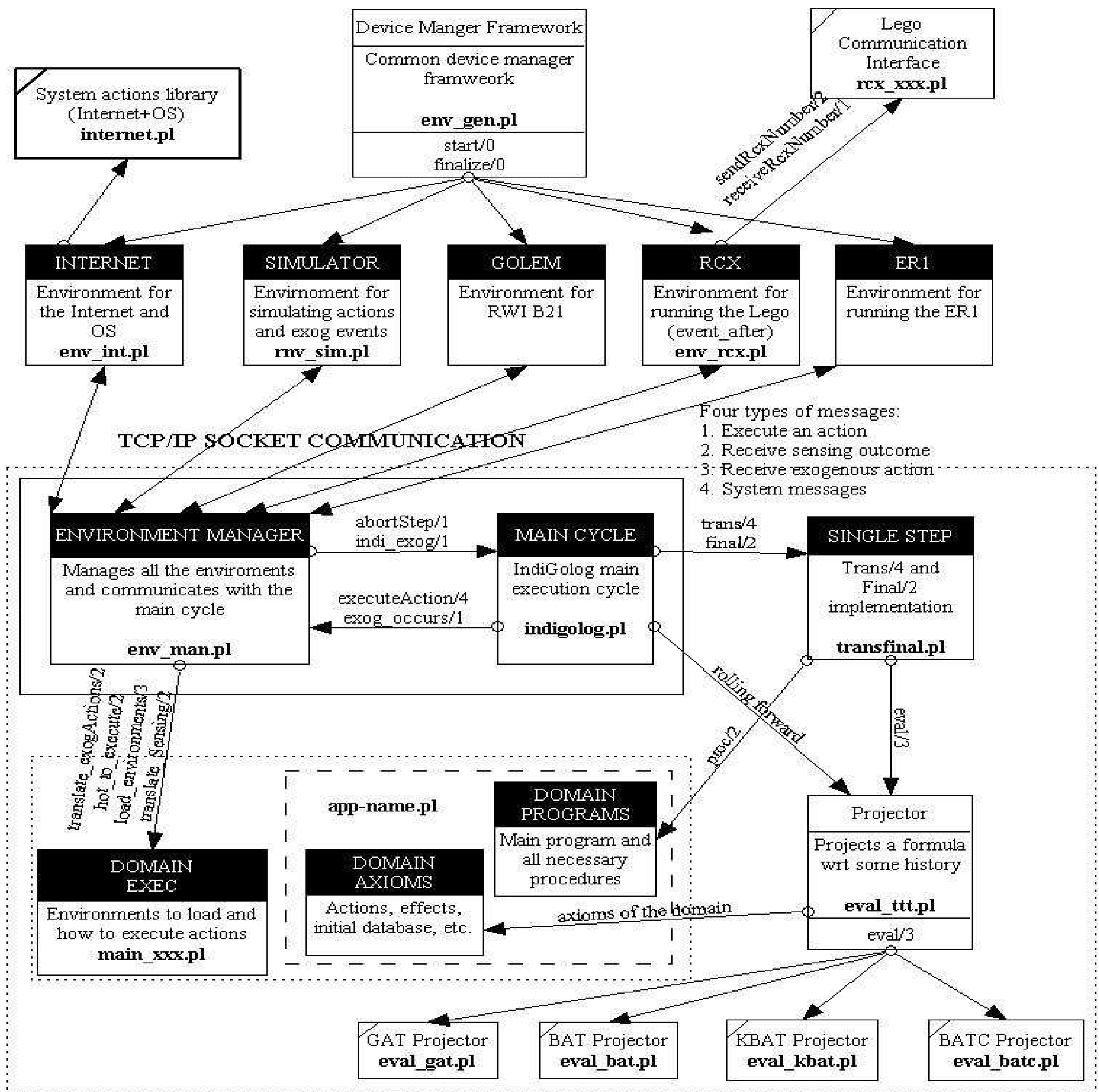

Figure 1: The P-IndiGolog architecture

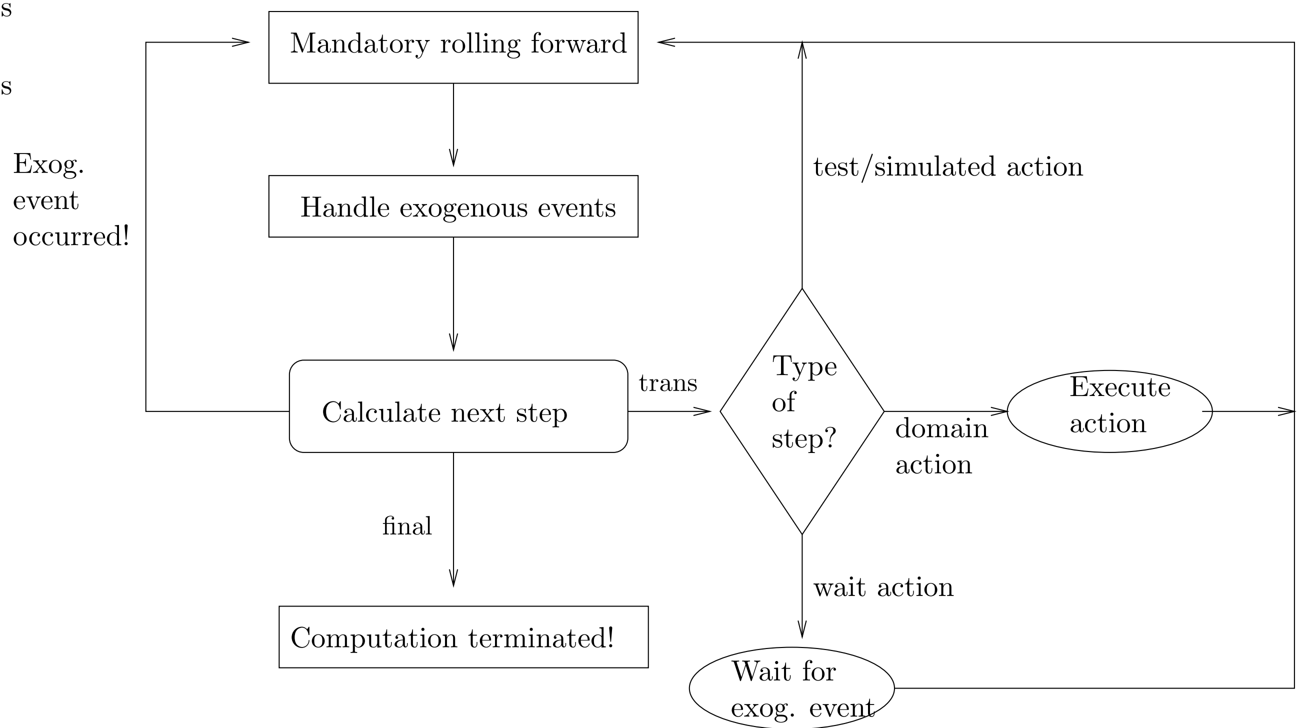

Figure 2: The P-IndiGolog top-level cycle