Introduction to the CARINA Metacognitive Architecture

Conference Paper · July 2018
DOI: 10.1109/ICCI-CC.2018.8482051

CITATIONS
0

READS
168

4 authors, including:

Manuel Fernando Caro Piñeres
Universidad de Córdoba (Colombia)
40 PUBLICATIONS 58 CITATIONS

Adan Gomez
Universidad de Córdoba (Colombia)
16 PUBLICATIONS 3 CITATIONS

Catriona M Kennedy
University of Birmingham
44 PUBLICATIONS 352 CITATIONS

Some of the authors of this publication are also working on these related projects:

Analysis of verbal protocols in the resolution of programming logic problems in sixth grade students of the Mercedes Abrego educational institution View project

CARINA: Cognitive architecture for reasoning failure detection in Intelligent Agents (Focused in Intelligent Tutoring Systems) View project

All content following this page was uploaded by Adan Gomez on 12 October 2018.

The user has requested enhancement of the downloaded file.
Introduction to the CARINA metacognitive architecture

Manuel F. Caro
Dept. Informática Educativa
Universidad de Córdoba
Montería, Colombia
manuelcaro@correo.unicordoba.edu.co

Darsana P. Josyula
Department of Computer Science, Bowie State University,
Bowie, MD – USA
darsana@cs.umd.edu

Adán A. Gómez
Dept. Informática Educativa
Universidad de Córdoba
Montería, Colombia
aagomez@correo.unicordoba.edu.co

Catriona M. Kennedy
School of Computer Science, University of Manchester,
Manchester, UK
catriona.kennedy@manchester.ac.uk

Abstract— Metacognition has been used in artificial intelligence to increase the level of autonomy of intelligent systems. However, the design of systems with metacognitive capabilities is a difficult task due to the number and complexity of processes involved. This paper presents the CARINA architecture, which is based on precise definitions of structural and functional elements of metacognition as defined in the MISM metamodel. CARINA can be used to implement real-world cognitive agents with the capability for introspective monitoring and meta-level control. Introspective monitoring detects reasoning failure (for example, when expectation are violated). Metacognitive control selects strategies to recover from failures. The paper demonstrates a CARINA implementation of reasoning failure detection and recovery in an intelligent tutoring system called FUNPRO. The tutoring system also searches for possible explanations of a failure by searching for known explanations and by analyzing its reasoning trace.

Keywords—metacognition, cognitive architecture, cognitive computing, CARINA

I. INTRODUCTION

Computational metacognition is an area of Artificial Intelligence (AI) that focuses on the ability of intelligent systems to monitor and to control their own learning and reasoning processes [1]–[3]. Computational metacognition distinguishes reasoning about reasoning from reasoning about processes involved [4].

A metacognitive architecture provides a concrete framework for detailed modeling of mechanisms for an AI agent's high-level reasoning about itself, through specifying essential structures, divisions of modules, relations among modules, and a variety of other essential aspects [5], [6].

Metacognitive architectures differ from cognitive architectures in that the agent itself is the referent of the cognitive processing but share a commitment to formalisms for representing knowledge, memories for storing this domain content, and processes that utilize and acquire the knowledge [6], [7].

Nowadays increasingly complex AI agents that make decisions based on multiple variables are developed. The complexity increases the probability that reasoning failures occur in such AI agent [8], [9]. A reasoning failure is defined as an outcome other than what is expected or a lack of some outcome or appropriate expectation [10]. Reasoning failures are generated mainly by unfinished tasks or unexpected results in the performance of a task. Metacognition provides introspective monitoring and meta-level control mechanisms for an AI agent to detect and correct its own reasoning failures.

Some implementations exist and have made progress in the integration of metacognitive functions in cognitive architectures. The ACT-R [11]–[13] is a cognitive architecture based on the theory of rational analysis that can provide a description of the processes from perception through to action for a wide range of cognitive tasks [13]. Although ACT-R theory was not developed with specific metacognitive mechanisms, several metacognition research has been developed based on this architecture. For example, the work of [14] demonstrate through the use of ACT-R that subjects can dynamically control their problem solving process based on the knowledge of the relative demands of tasks and the mental resources needed to complete it. Reitter [15] designed several metacognitive layers in ACT-R that implement several strategies to address the basic control task, as well as the means to classify and select those strategies according to their suitability in a given situation. However, the modeling process has concentrated on multitasking behavior and has not clarified the specific relationship between multitasking and metacognition.

Research on metacognitive architectures includes the CLARION architecture [5] which allows designers to construct models of specific metacognitive processes, which are then used to capture experimental data related to meta-cognition using a a meta-cognitive subsystem. The meta-cognitive subsystem (the MCS) monitors and controls/regulates cognitive processes for the sake of improving cognitive performance in various circumstances [5], [16]. The role of the meta-cognitive subsystem is to monitor, direct, and modify the operations of all the other subsystems [16]. Actions in the
meta-cognitive subsystem can be in form of setting goals, setting parameters and changing an on-going process in other subsystems [5], [16]. However, Clarion does not present processes for the explanation of the cause of a possible poor cognitive performance.

MIDCA is other metacognitive architecture, which is based on earlier work on computational metacognition including the MetaCognitive Loop [17], [18] and Introspective Multistrategy Learning theory [19]. The MIDCA architecture consists of “action-perception” cycles at both the cognitive (i.e. object) level and the metacognitive (i.e., meta-) level (cox). MIDCA is focused on providing agent with greater autonomy in solving problems. Goal insertion is a fundamental process in MIDCA and occurs at both the object level and meta-level. At the meta-level, the cycle achieves goals that change the object level and metacognitive “perception” components introspectively monitor the processes and mental state changes at the cognitive level. MIDCA and CLARION do not present mechanisms of software engineering to simplify the process of developing a intelligent software agent with integration of metacognitive capabilities.

In the context described, the main objective of this paper is to introduce a novel metacognitive architecture for monitoring and control of reasoning failures in artificial intelligent agents. The metacognitive architecture is named CARINA and has support for introspective monitoring and meta-level control. CARINA attempts a thorough integration of action and perception with both cognition and metacognition. These kind of reasoning systems are called “perpetual self-aware cognitive agents” [20], [21].

CARINA allows cognitive systems to detect reasoning failure, to explain the failure and to select strategies for recovery. The implementation of CARINA aims to simplify the process of developing a intelligent software agent and to permit the user to focus on the specifics of the application, hiding the complexities of the generic parts of the model.

The implementation of CARINA aims to simplify the process of developing a intelligent software agent and to permit the user to focus on the specifics of the application, hiding the complexities of the generic parts of the model.

In this paper, first the CARINA architecture is sketched. Next, a domain-specific visual language specifically developed for modeling metacognition in intelligent systems called M++ is described. Finally, a metacognitive model and a computational implementation based on CARINA is presented.

II. THE CARINA ARCHITECTURE

CARINA is a metacognitive architecture for artificial intelligent agents. CARINA is derived from the MISM Metacognitive Metamodel [22]. CARINA integrates self-regulation and metamemory with support for the metacognitive mechanisms of introspective monitoring and meta-level control; in this sense CARINA assumes a functional approach to philosophy of mind, according to [23]–[25]. According to [26], the term “mechanism” includes entities and activities involving entities (i.e., including both static and dynamic aspects). The entities in CARINA are called “cognitive elements”.

In accordance with MISM, CARINA is composed of three types of cognitive elements: structural elements, functional elements and basic elements. Structural elements are containers into which the functional and basic elements are embedded; the main structural element is the “Cognitive level”. The functional elements are tasks that enable reasoning and decision-making. Basic elements consist of the set of elements that participate and interact in reasoning and metareasoning processes. The main functional elements in CARINA are: reasoning task and meta-reasoning task. Reasoning tasks (RT) are actions that enable the processing (transformation, reduction, elaboration, storage and retrieval) of information by applying knowledge and decision making in order to meet the objectives of the system. A meta-reasoning task (MT) may be used to explain errors in some reasoning task or it may be used to select between “cognitive algorithms” to perform the reasoning [2].

A. Cognitive Loop in CARINA

CARINA is composed of two cognitive levels named object-level and meta-level. The object-level contains the model that an artificial intelligent agent has for reasoning about the world (i.e. agent’s environment) to solve problems [22]. The meta-level is a level of representation of the reasoning of an artificial intelligent agent. The meta-level includes the components, knowledge and mechanisms necessary for a system to monitor and control its own learning and reasoning processes.

1) Cognitive Loop at the Object-Level in CARINA

A cognitive loop (action-perception cycle) in CARINA starts when the agent perceives changes to the environment resulting from actions. The situation assessment stage takes as input the perception and processes it. The processing of the information perceived includes the recognition of situations or events as instances of known or familiar patterns and the categorization of objects, situations, and events into known concepts or categories. The output of the situation assessment stage is the combination of perceptual information about many objects and events to compose a comprehensive model of the current environment (i.e. model of the world).
The reasoning process in CARINA takes as input beliefs, assumptions and expectations. It then reaches some mental conclusion (e.g. categorization). This is a central cognitive activity that enables an agent to augment its knowledge state. The problem solving process is closely related to reasoning in the sense that it is given a goal (in the world or within the agent) and the agent then chooses and orders its actions to achieve this goal. Monitoring a plan’s execution can also lead to revised estimates about the plan’s effectiveness, and, ultimately, to a decision to pursue some other course of action. In CARINA, decisions are often associated with the recognition of a situation or pattern, and combines the two mechanisms in a recognize-act cycle that underlies all cognitive behavior. Finally, acting involves the representation and storage of motor skills that enable the activities stored in Procedural Memory.

2) Metacognitive Loop at the Meta-Level in CARINA

The meta-level keeps an updated model of the object-level called the “self-model”. This self-model is based on an internal representation of the reasoning processes that occur at the object-level. The self-model stores the data generated by the reasoning processes, along with the rules that were used to address the reasoning. The self-model allows the interchange of information between the meta-level and the object-level. The self-model enables an intelligent system to reason about its own reasoning, since it describes the internal state of the system. The self-model allows the meta-level to have awareness about reasoning processes that are conducted at the object-level. Introspective monitoring and meta-level control are two meta-reasoning mechanisms implemented at the meta-level in CARINA. Using the common terms of cognitive science, the notion of “mechanism” involves both representations as well as cognitive mechanisms and processes operating on them [27]. Introspective monitoring includes mechanisms for detecting reasoning failures at the object-level. The main purpose of monitoring is to provide enough information to make effective decisions in the meta-level control. The monitoring process is done through information feedback that is gathered at the meta-level from the object level, see Fig. 2. Thus each cognitive task executed in the object level has a performance profile that is continuously updated in the meta-level. The performance profile is used to evaluate the results of each reasoning task. For example, a robot explores a building and uses its current map of each floor by predicting what kind of room it will find at each location. The performance profile summarizes how well the reasoning is performing and may include its prediction accuracy of what it finds when it opens a door.

ProfileGeneration, FailureDetection, FailureExplanation and GoalGeneration are the main monitoring tasks in CARINA: The ProfileGeneration task performs the interface between the meta-level and the object-level.

When a cognitive task is running, it generates computational data. ProfileGeneration reads the computational data and generates a Profile of the CognitiveTask. Each CognitiveTask in the object-level has a performance Profile in the meta-level; thus the meta-level is always informed of the status of the reasoning in the object-level.

The Sensor has the function of monitoring the profiles of cognitive tasks in order to detect disturbances or anomalies that may represent reasoning failures produced by the cognitive task. FailureDetection reads the properties of a Sensor. If the Sensor finds a discrepancy between observations and expectations regarding the performance of the CognitiveTask, then FailureDetection detects a ReasoningFailure in the CognitiveTask being monitored. FailureExplanation generates an
Explanation of the cause of the ReasoningFailure having as inputs the assessment of the failure and reading the ReasoningTrace. GoalGeneration produces new goals based on the Explanation for solving the failure detected. A plan to solve the ReasoningFailure is built the basis of the new Goal. The plan is called FailureSolutionPlan. In the case of the robot, an explanation might be that its map is incorrect and it needs to add more details to it (new goal). For example, this building might be an exception to the norm on which its map is based. Fig. 2 shows the specification of the introspective monitoring process in CARINA.

On the other hand, the meta-level control function decides whether to invoke a scheduler, which scheduler to invoke, and the quantity of resources to invest in the scheduling process [28]. The meta-level control in CARINA includes rules for the recommendation of the best strategy available to correct some reasoning failure at the object-level. The main objective of the meta-level control is to improve the quality of decisions made by the system. Metacognitive tasks associated with meta-level control may explain errors in the reasoning task or it may select between cognitive “algorithms” to perform the reasoning [4].

Therefore, the main control tasks of this mechanism in CARINA are: ControlActivation and StrategySelection. A plan to solve the ReasoningFailure is built based on the new Goal. The plan is called FailureSolutionPlan. A FailureSolutionPlan can activate the metacognitive control. The ControlActivation task starts PlanExecution. StrategySelection is one of the tasks that comprise the plan. The StrategySelection task reads profiles of cognitive tasks and uses MetacognitiveStrategy to recommend computational strategies. The computational strategies are recommended to the CognitiveTask in order to solve the ReasoningFailure.

III. COGNITIVE MODELING LANGUAGE FOR CARINA

Computational cognitive modeling provides detailed descriptions of cognitive mechanisms [26] and embodies descriptions of cognition in algorithms and programs, based on the science and technology of computing [29], [30]. CARINA provides a programming language to create a cognitive model and an interpreter to execute the model. The programming language is called M++ and the development tool with the interpreter called MetaThink.

M++ is a domain-specific visual language specifically developed for modeling metacognition in intelligent systems. In M++ the specifications of the object-level and meta-level are supported in a metamodel configured according to the standard Meta-Object Facility (MOF) of Model-Driven Architecture (MDA) methodology. The architecture also constrains model building because it is based on MISM metamodel. This ensures that the resulting models are products of the theory behind the architecture.

The concrete syntax was defined in order to make the language more usable. The icons were designed bearing in mind their usability when applied by users. Fig. 8 includes a representative sample of the elements of the M++ notation.

In Fig. 4, section (A) shows the icons used to represent object-level tasks and section (B) displays icons representing elements that interact with the tasks at object-level. Section (C) contains the notation related to the tasks of introspective monitoring act from the meta-level on the object level. Section (D) displays icons representing elements that interact with the monitoring tasks. Section (E) displays icons representing the metacognitive control tasks and section (F) displays icons representing elements that interact with the tasks of
metacognitive control. In summary, M++ has approximately 20 notation elements for modeling metacognitive systems.

IV. CARINA COMPUTATIONAL IMPLEMENTATION

The CARINA architecture is an instance of MISM metamodel [22]. MISM metamodel is based on Meta-Object Facility (MOF) standard.

The MOF standard provides a framework based on metamodels to enable the development and interoperability of models to automate some stages of the development process of software applications. As can be observed in Figure 5.1, the architecture of CARINA’s framework is organized into four levels according to the MOF standard.

- **Meta-MetaModel Level (M3).** This level comprises meta-metamodel (MOF 2.0) that is used for implementation of the metamodel MISM (M2).
- **Metamodel Level (M2).** The metamodel MISM is positioned at the M2-level in the MOF CARINA’s framework. Therefore, a Model that is positioned at the M1-level can be modeled by the metamodel. MISM metamodel is specified using MOF standard and implemented in Eclipse Modeling Framework (EMF).
- **Model Level (M1).** This level contains the conceptual models of CARINA that is implemented according to the metamodel specified at M2 level. In this sense, CARINA (M1 level) is a Metacognitive Architecture for monitoring and control of reasoning failures in Artificial Cognitive Agents.
- **User Model Level (M0).** The User Model at the M0-level is the target model that is the aim of the MISM Metamodel. The derived target model represents an Artificial Cognitive Agent in the real-world. In MOF, the domain concept used in a metamodel is presented as a Class. The data for a Class is presented as an Object. As such, the data for the Object are in turn presented as an Instance in User Model. End-Users manipulate real data using Artificial Cognitive Agent generated by a modeling framework from M1, i.e. users can create and use models of entities from real world (M0), using the CARINA architecture (M1).

A. Description of the implementation of CARINA at Model Level (M1)

CARINA is a metacognitive architecture developed under a paradigm oriented around cognitive cycles. Currently CARINA has the following implementations: Prolog, JavaScript and Java. The kernel of CARINA is named CARINA CORE. CARINA CORE is composed of a series of modules that define the basic elements of the computational implementation. The modules are organized into software packages. The CARINA modules are: memory, metacore.

1) Memory Package

The memory module contains the definitions of memory systems that can be used during the cognitive cycle. Memories involve the acquisition, categorization, classification and storage of information. The purpose of memory is to provide the ability to recall information and knowledge as well as events [31]. Memory System in CARINA is structured according to [32]–[34] as follows: Sensory Memory, Working Memory and Long Term Memory, see Fig. 5.

**Fig. 5. Memory process in CARINA is based on [32]–[34]**

Sensory memory is capable of retaining information for a very short period of time [34]–[37], retaining impressions of sensory information after stimuli have ended [36], [37]. Selective attention allows encoding of important information according to goals and expectations established in CARINA. Most of the information in sensory memory is not encoded in this way. The sensory memory is a temporary buffer which
briefly holds information not immediately attended [38], making it possible to attend to some of it slightly later.

Working memory is a system used for temporary storage of information during perception, reasoning, planning or other cognitive tasks [39]–[41]. A selective attention subsystem and a set of “Basic Cognitive Processing Units” (BCPU) compose CARINA’s Working Memory. In CARINA, a BCPU acts as a buffer among the cognitive processes that intervenes in a cognitive loop.

Long-term memory stores information over the time [42]. In Carina, Long-term memory encodes information semantically for storage. Declarative memory (“knowing what”) is a memory of facts and events [43], and refers to those memories that can be consciously recalled (or "declared") [33]. In CARINA, declarative memory is subdivided into episodic memory and semantic memory.

- **Episodic memory**: In Carina episodic memory is implemented as a Case-Based Reasoning System (CBR) and contains cases that represent events. An event consists of detailed sensory-perceptual information on recent experience [44], which includes perception, motor commands, and internal data structures [45], [46]. Episodic memory is designed to support semantic memory [47], [48]. In this sense, episodic memory activity in Carina is analogous to the memory activity concentrated in the hippocampus.

- **Semantic memory**: includes all acquired knowledge about the world [49], not contextualized in time and space. In Carina, semantic memory has the following properties: (i) it simulates the activation of the frontal and temporal cortices; and (ii) it consists of a “mental thesaurus” implemented as an ontology. In this sense, the implementation of semantic memory in CARINA corresponds to Tulving and Binder approaches [49], [50].

Procedural Memory is the memory of actions and behaviors of a system [51], including sequences, categories, rules and routes [52]. It is a non-declarative memory with nonconscious learning, which refers to a “how to” kind of information. In CARINA, this memory consists of a hybrid model based on a rule-based system to support sequences and provide a record of possible motor and behavioral skills.

2) **Metacore package**

The metacore module contains the basic elements and functions of CARINA, which are essential for the development of the meta-cognitive and cognitive processes of the system. Fig. 6 shows the basic package structure in CARINA.

The carina.metacore package facilitates the reuse of elements among different metacognitive mechanisms. carina.metacore facilitates the integration of metacognitive components in the design of intelligent systems because it is based on independent packages that share common design elements in metacore.

The carina.objectlevel package contains the model an AI agent uses for reasoning about the world (i.e. agent’s environment) to solve problems [22].

The carina.metalevel package includes the components, knowledge and mechanisms necessary for self-monitoring and control of the learning and reasoning processes of an artificial intelligent agent. This package is composed of three sub-packages: carina.metalevel.core, carina.metalevel.monitoring and carina.metalevel.control.

The main objective of carina.metalevel.core is to simplify the complexity of carina.metalevel package. This package combines the classes that are common to the subpackages: carina.metalevel.monitoring and carina.metalevel.control. The carina.metalevel.monitoring sub-package includes mechanisms for detecting reasoning failures at the object-level. The carina.metalevel.control sub-package recommends to the object-level the best computational strategy to resolve a reasoning failure.

3) **Validation**

As with any scientific theory or engineered artifact, cognitive architectures require evaluation. However, because architectural research occurs at the systems level, it poses more challenges than does the evaluation of component knowledge structures and methods. In this section, we consider some dimensions along which one can evaluate cognitive architectures. In general, these involve matters of degree, which suggests the use of quantitative measures rather than all-or-none tests. Langley and Messina [53] discuss additional issues that arise in the evaluation of integrated intelligent systems.

4) **Generality**

Recall that cognitive architectures are intended to support general intelligent behavior. Thus, generality is a key dimension along which to evaluate a candidate framework. We can measure an architecture’s generality by using it to construct intelligent systems that are designed for a diverse set of tasks and environments, then testing its behavior in those domains.
The more environments in which the architecture supports intelligent behavior, and the broader the range of those environments, the greater its generality.

5) Versatility
We can define the versatility of a cognitive architecture in terms of the difficulty encountered in constructing intelligent systems across a given set of tasks and environments. The less effort it takes to get an architecture to produce intelligent behavior in those environments, the greater its versatility.

6) Taskability
Generality and versatility are related to a third notion, the taskability of an architecture, which acknowledges that long-term knowledge is not the only determinant of an agent’s behavior in a domain. Briefly, this concerns an architecture’s ability to carry out different tasks in response to goals or other external commands from a human or from some other agent. The more tasks an architecture can perform in response to such commands, and the greater their diversity, the greater its taskability. This in turn can influence generality and versatility, since it can let the framework cover a wider range of tasks with less effort on the developer’s part.

V. COGNITIVE TASK AND MODEL
The real-world task that has been modeled in CARINA is the generation of an instructional plan from the instructional design domain. In this task, an instructional designer wants to design an instructional plan for a specific lesson of a course. The instructional plan will be applied individually and according to the learning style of a new student enrolled in such course. For this, the instructional designer searches among the plans that have been successfully applied to students with learning profiles similar to the new student. If the instructional designer finds an instructional plan then it is applied to the student, otherwise it will design a new instructional plan according to the characteristics of student. In summary, the task of the instructional designer is to determine the learning style of the student and select or build most appropriate instructional plan according to student’s learning style.

A. CARINA-based cognitive agents at User Model Level (M0)
Advancing in the modeling of the instructional planning task, consider an instructional planner agent of a tutor module in an Intelligent Tutoring System. The instructional planner agent generates instructional plans, given the current state of the student, and monitoring the execution of the content plan in order to determine when to re-plan, or generate a new plan. In the context described, we have developed an Intelligent Tutoring System (ITS) called FUNPRO (FUNdamentos de PROgramación) based on the MODESEC methodology [54], in order to show how the computational CARINA architecture can integrate cognitive and metacognitive functions.

The ITS is used as a case study on monitoring a plan’s execution. Monitoring plan execution is a high-level cognitive function. Monitoring and control of reasoning processes are two metacognitive functions supported in CARINA. A metacognitive model of the instructional planner is constructed using the principles described in Section 2. The model of the object-level of the instructional planner is based on an extended Belief, Desire and Intention (BDI) framework [55]-[57].

In FUNPRO, the instructional planning task includes a subtask for recommendation of learning resources identified as playResource. The implementation of FUNPRO in CARINA includes defining the metacognitive mechanisms for introspective monitoring and meta-level control.

FUNPRO makes replanning to the instructional plan in the following cases: (i) if a resource for some reason cannot be deployed in the lesson, e.g. resource has the URL broken; (ii) if a recommended resource has received a poor evaluation; (iii) if the student obtains a low performance in the lesson. The instructional planning constantly refines the pedagogical strategy for each student using three types of recommendation strategies: (i) matching simple query, the search query in a simple SQL type; (ii) exclusive search is similar to (i), but excludes some results and; (iii) vote-based search, this strategy is based on the nearest neighbor algorithm. Fig. 7 shows the partial view of the metacognitive model for the instructional planning function in the FUNPRO, the view is focused in playResource subtask.

![Fig. 7. Metacognitive model generated in MetaThink using M++ notation](image)

The partial mapping from MOF metamodel (MISM) from CARINA to FUNPRO metacognitive model in M++ is listed as follows (Table I).

<table>
<thead>
<tr>
<th>CARINA - Concept</th>
<th>Artifact of FUNPRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReasoningTask</td>
<td>playResource</td>
</tr>
<tr>
<td>Strategy</td>
<td>SQL_simple_query;</td>
</tr>
<tr>
<td>Goal</td>
<td>SQL_exclusion_query;</td>
</tr>
<tr>
<td>ComputationalData</td>
<td>play_resource_data</td>
</tr>
<tr>
<td>ReasoningTrace</td>
<td>play_resource_trace</td>
</tr>
</tbody>
</table>

TABLE I. ITS – mapping table
The meta-level intervenes in the process of instructional planning by deciding whether to continue reasoning for a better plan or execute the current plan. When a plan is generated, the meta-level analyzes the possibility of refining the plan (reasoning about the planning process) using as evidence the effectiveness of similar plans in the past. If the meta-level finds that expected performance of the current plan is sufficient to achieve the planned goals, then it proceeds to execute the plan. But if there are possibilities to improve the plan in a reasonable time, then the meta-level decides to continue planning further.

1) Implementation Issues

The instructional planer in FUNPRO was implemented using SWI-Prolog according to metamodel presented in Fig. 7. FUNPRO is an ITS for teaching Introduction to Programming in Engineering. Fig. 8 shows a screenshot of the graphical user interface of FUNPRO.

<table>
<thead>
<tr>
<th>Process</th>
<th>Reasoning task – Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student profile identification</td>
<td>hasLearningStyle(S, Y)</td>
</tr>
<tr>
<td>Verification of prerequisites</td>
<td>hasLesson(C, L)</td>
</tr>
<tr>
<td>and selection of lesson</td>
<td>hasPrerequisite(L, P)</td>
</tr>
<tr>
<td>Recommendation of pedagogical tactics</td>
<td>recommendPedagogicTactic(C, T)</td>
</tr>
<tr>
<td>Recommendation and deployment of</td>
<td>hasLearningResource(L, R)</td>
</tr>
<tr>
<td>learning resources</td>
<td>isResourceOfPedagogicTactic(R, T)</td>
</tr>
<tr>
<td></td>
<td>isResourceOfLessonComponent(R, I)</td>
</tr>
<tr>
<td></td>
<td>playResource(L, I, T, U)</td>
</tr>
</tbody>
</table>

In Table I: S: Student - T: Pedagogic tactic - L: Lesson - R: Resource - U: URL - I: Lesson component - Y: Learning style - C: Course - P: Performance

Introspective monitoring starts when playResource generates new computational data (a resource path or an error message). The generation of a new computational data activates the sensor associated with the reasoning task. In the following code snippet we can see the implementation of the activation of a sensor in FUNPRO.

```
sensorActivation(ID_reasoning_task, Sensor):-
  reasoning_task(ID_reasoning_task),
  newReasoningTaskOutput(ID_reasoning_task, U),
  sensor(ID_sensor, _),
  sensorMonitors(ID_sensor, ID_reasoning_task),
  updateSensorState(D_sensor, active).
```

Reasoning failure detection starts when a sensor is activated. The meta-level gets updated observations from the sensor associated with the MRP, using the instructions...
showActiveSensor(Action, Sensor) and updateSensorObservation (Sensor, Observation).
Once the current reading of the active sensor is obtained, then meta-level checks whether the observation is consistent with the expectation of the sensor. In the following piece of code we can see the identification of a reasoning failure.

isReasoningFailure(Action, Observation):-
  showActiveSensor(Action, Sensor),
  updateSensorObservation(Sensor, Observation),
  anomalyInExpectation(Sensor),
  generateReasoningFailure(ID_reasoning_task, ReasoningFailure),
  updateFailureCounter(V).

After reasoning failure is detected, the metalevel generates an explanation. There are three possible explanations for a reasoning failure occurred in playResource:

• Resource available refers to an unavailable resource when deploying on FUNPRO.
• Inappropriate resource refers to a resource that was not adequate to the student profile.
• URL broke refers to an available or valid resource that has a broken URL.

Explanations can be generated in two ways: search for known explanations and reasoning trace analysis.

Strategy search for known explanations queries for explanations given to reasoning failures in the past and then evaluates and prioritizes explanations, see the following piece of code.

explanationGeneration(ReasoningFailure, Explanation):-
  hasExplanation(ReasoningFailure, Explanation ),
  explanationPriorization(Explanation).

The reasoning trace analysis strategy performs a more complex reasoning because it makes queries on the trace of the structures of reasoning performed by the failed task looking for anomalies, see the following piece of code.

explanationGeneration(ReasoningFailure, Explanation):-
  hasReasoningFailure(ReasoningTask, Reasoning Failure),
  anomalyInReasoningTrace (ReasoningTask, Anomaly),
  anomalyExplanation(Anomaly, Explanation),
  explanationPriorization(Explanation).

Finally, the meta-level generates a goal based on the explanation in order to solve the reasoning failure.

goalGeneration(Explanation, Goal):-
  hasReasoningFailure(ReasoningTask, Explanation),
  reasoningTaskHasProfile(ReasoningTask, Profile),
  reasoningTaskUsesStrategy(ReasoningTask, Strategy),
  reasoning_task_profile(Profile_r, Goal_r, Performance, _,-,-,-),
  strategy_profile(Profile_s, Goal_s, Performance, _,-,-,-),
  goalCandidate(Profile_r, Profile_s, Goal).

Meta-level control starts after goal generation. The main function of metalevel control is to address the failure of reasoning. The selection of strategies receives as a parameter the Goal to be achieved and searches through the available strategies those which satisfy the Goal. Meta-level control then evaluates the performance of each strategy by selecting the best, in the following piece of code can be see the general implementation.

recommendStrategy(Goal, Strategy):-
  goalAchievedWith(Goal, Strategy, Performance),
  strategyPriorization(Strategy, Performance).

VI. CONCLUSION

CARINA is a metacognitive architecture for artificial intelligent agents. CARINA is derived from the MISM Metacognitive Metamodel. CARINA integrates self-regulation and metamemory with support for the metacognitive mechanisms of introspective monitoring and meta-level control. The main purpose of monitoring in CARINA is to provide enough information to make effective decisions in the meta-level control. The meta-level control in CARINA includes rules for the recommendation of the best strategy available to correct some reasoning failure at the object-level. CARINA provides a programming language to create metacognitive models and an interpreter to execute the models.

Currently CARINA has implementations in Prolog, JavaScript and Java. The kernel of CARINA is named CARINA CORE. CARINA CORE is composed of a series of modules that define the basic elements of the computational implementation.

The CARINA computational implementation allows the creation of intelligent software agents that conform to the CARINA metacognitive architecture and the MISM metamodel. This paper has demonstrated how a CARINA implementation can be added to an Intelligent Tutoring System (FUNPRO). A real-world task related to generation of an instructional plan from the instructional design domain was modeled in CARINA and implemented in the tutor module of FUNPRO. The added metacognitive capabilities allow the tutoring system to detect reasoning failure, to explain the failure and to select strategies for recovery.
The implementation of CARINA aims to simplify the process of developing a intelligent software agent and to permit the user to focus on the specifics of the application, hiding the complexities of the generic parts of the model.

ACKNOWLEDGMENT

This work was funded in part by the Internal Call for Sustainability of Research Groups of the University of Córdoba and by the project COL / 86815 – CONV. 647/15 of the Ministry of Information Technologies and Communications of Colombia (MinTIC).

REFERENCES


