Robotica (1997) volume 15, pp 105–110. © 1997 Cambridge University Press

LL(1) descriptions for robots

Albert Hoogewijs

Department for Pure Mathematics and Computer Algebra, University of Gent, Galglaan 2, B-9000 GENT (BELGIUM)
Albert.Hoogewijs@rug.ac.be.

SUMMARY

Syntax-directed development is a software development method in which, the syntax of the input of the application plays a central role. The syntax description forms a frame on which semantic actions, attributes, local and global information can be attached. Much research has been done to develop theories and algorithms needed for robots to process information and interact with the environment. This paper describes how LL(1) descriptions can be used to produce automatically motion control software, object recognizers and human-computer interfaces.

KEYWORDS: LL(1) description; Software method; Motion control; Human-computer interface; Robots.

1. INTRODUCTION

Until recently, robots were primarily employed for carrying out programmed repetitious tasks. With the new methodologies and algorithms for autonomous functioning and the rapid advances in semiconductor and computing technology, it has become feasible to build robots that can function at reasonable speed. From literature[1] we learn that within the next 5 years we may expect autonomous unmanned ground vehicles that can perform military missions with a minimum of human oversight. Just what are intelligent robotic systems, anyway? One working definition is that an intelligent robot is a machine that can extract information from its environment and use knowledge about its world to move safely in a meaningful and purposeful manner. Hence, human computer interface, image processing, motion control are the key issues in robotics.

2. HUMAN-COMPUTER INTERFACE

The key to the real effectiveness of a robot is its usability by people other than computer professionals. The possibilities of a machine are limited not by its power to compute, but rather by its power to communicate with its human users. Hartson and Hix[2] propose an environment in which good interfaces can be constructed. They present useful, consistent definitions for key terms, to overcome misunderstanding that arise from conflicting usage of terms in literature. The terms ''human-computer dialogue'' and ''human-computer interface'' (also called the ''user interface'') are defined separately to denote, respectively, the communication between a human user and a computer system and the medium for that communication. Thus, a *dialogue is* the observable two-way exchange of symbols and actions between human and computer, whereas an *interface* is the supporting software and the hardware through which this exchange occurs. The two terms, however, are tied closely together in the development process, and we shall use them synonymously here just as they are in most of the literature.

As research has progressed in the field of interface management, some concepts have emerged upon which a framework for human-computer interface development can be based. Hartson and Hix distinguish the following concepts:

- *Dialogue independence.* The characteristic of an interactive software system that separates the design of dialogue from the design of computational software so that changes in either tend not to cause changes in the other.
- *Structural modelling of the human-computer interface.* The description of the general process of human–computer interaction that can be used to direct the design of dialogue and dialogue development tools.
- *Representation of the human-computer interface.* The techniques used to represent definitions of specific instances of human-computer interaction.
- *Interactive tools for human-computer interface development.* Software and hardware to help automate system development, especially for the human-computer interface.
- *Rapid prototyping.* The process of building executable versions of partially constructed interactive systems to allow early observation of system behaviour, especially its interface.
- *Methodologies for interactive system development.* System development processes and life cycles that treat interface development as an equal and integral part of the overall software development process.
- *Control structures.* The organization of dialogue and computational components and the mechanisms that govern logical sequencing and internal communication among dialogue and computational events.

The idea of language is involved in both structural modelling and interface representation, but in different ways. In the first case, structural models typically relate to the language of the end-user, that is, the interaction language in which the human communicates with the computer. In the other case, an interface representation language used by the dialogue developer is a metalanguage for defining the end-user's interaction language.

https://doi.org/10.1017/S026357479700012X Published online by Cambridge University Press

It is clear that human-computer dialogue, especially sequential dialogue, can formally be modelled as the vocabulary and the grammar of a human-computer "interaction language". To understand the idea of dialogue as an interaction language, consider the use of an ordinary command language. Each typed command line is accepted, lexically analysed, parsed according to a grammar, recognized as either a valid command or an error, and acknowledged as either a valid command (sometimes implicitly through the presentation of the next prompt) or an error. The action requested by the command, if valid, is performed. From the properties of LL(1) syntax descriptions[3,4] there is no difficulty in seeing this kind of interaction sequence as one involving language that can be formally described by a ELL(1) grammatical specification, that is combining the advantages of the LL(1)-techniques and the expressive power of extended regular expressions.[4] The MIRA application generator environment[5] implements these techniques. The syntax of the language is represented in Backus-Naur Form (BNF). This is a highly structured, hierarchical metalanguage that results in a so-called "fan-out" problem. That is, nonterminals in an expression can be replaced by more nonterminals through several successive iterations before a terminal symbol is finally reached. This multilevel tree structure is difficult for human beings to follow, since by the time the terminals are reached, the highest level expression and the language structure may long be forgotten. In the Visual MIRA environment, these descriptions can be visualized in corresponding transition diagrams, which helps in overcoming this problem.

The BNF-like definition of the interaction language as well as the programming code that is invoked to perform the related semantic actions are both used as input of the generator. Output is a recursive descent parser for the interaction language, as well as a scanner and screen manager.

As an example we consider a dialogue language, taken form a MIRA-specification that guides a virtual robot through a castle. It accepts short phrases as commands, such as READ THE SIGN, GO WEST. In BNF we get the following representation for that language:

GET ⟨Object⟩; TAKE ⟨Object⟩; DROP ⟨Object⟩: READ ⟨Object⟩; PUSH ⟨Object⟩; GO ⟨Direction⟩; HELP; INVENTORY;

The last two commands are somewhat special. HELP gives an overview of the most important phrases. INVENTORY tells what the robot is carrying. There are six possible directions the robot can GO: EAST, WEST, NORTH, SOUTH, DOWN and UP. It accepts these as a short for the GO command.

The first step in handling this language is performed by the *lexical analyzer* or *scanner*. The principle linguistic object is a "token", an abstraction representing the smallest unit of end-user input that can have formally defining meaning in terms of the application or task. An example is a simple command or data value that can be entered, say as a series of typed keystrokes, a menu choice, the press of a programmed function key, a mouse selection of an icon, a word of a voice recognizer of an image of object recognizer.

The lexical analyzer maps the "raw" (unintepreted and device dependent) input to a "normalized" token value. Token values are validated within interactions according to their lexical definitions and constraints. In addition, tokens themselves can be grouped together in semantically related sequences called "sentences". Sentences are processed by constructional objects called dialogue "transactions". The set of all valid sentences comprises the "transaction language". Relationships among tokens and constraints relating their values make up the grammar of the transaction language. In this case three types of tokens are build: a word, an end of line and an end of file. To recognize full words as well as their abbreviations, a two-level context sensitive search method can be used:

– The first context level is syntactic. It consists of the set of expected words, passed as a parameter to the read routine. The contents of this set is dynamically calculated by the generated parser. This set is passed unchanged as a parameter to this routine. So, the first refinement step is to look for an abbreviation of a word, represented in this set.

– The second context level is already semantical and is only applicable for abbreviations of objects. Only the objects in the current place or the objects the robot is carrying are valid. So, the second refinement step is to look for an abbreviation of a word, corresponding with such an object. The semantical refinement is not used for the operators.

Listing 1 shows a part of the MIRA-specification RobotScanner.mir that generates the lexical scanner:

```
#RULE
⟨Identifier | Keyword⟩
 = InitName⟨Letter⟩
   (AddName (⟨Letter⟩
   |⟨Digit⟩
   )
   ) * OutIdKey
#DECLARE
   long int NbOfChars = 0L;
#InitName
   @[Identifier] [NbOfChars + +] = Inp.c;
   @[Identifier] [NbOfChars] = "\0";
#AddName
NbOfChars + +;
if (NbOfChars⟨IDSTRLENGTH) {
   @[Identifier] [NbOfChars-1] = Inp.c;
   @[Identifier] [NbOfChars] = "\0";
}
#OutIdKey
if (NbOfChars⟩ = IDSTRLENGTH)
   LexError(M_IDENTRUNCATED);
if (! strcmp(@[Identifier], "get"))
   OutRoutine(%[get]);
```

https://doi.org/10.1017/S026357479700012X Published online by Cambridge University Press

```
else
if (! strcmp(@[Identifier], "take"))
   OutRoutine(%[take]);
else
if (! strcmp(@[Identifier], "put"))
   OutRoutine(%[put]);
else
if (! strcmp(@[Identifier], "read"))
   OutRoutine(%[red]);
else
if (! strcmp(@[Identifier], "go"))
   OutRoutine(%[go]);
else
if (! strcmp(@[Identifier], "help")
   OutRoutine(%[help]);
else
if (! strcmp(@[Identifier], "inventory"))
   OutRoutine(%[inventory]);
else
   OutRoutine(%[Identifier]);
                        #END
```

Listing 1: RobotScanner.mir

The *parser* gets the tokens from the scanner and performs the corresponding semantical actions. In Listing 2 we present a part of specification of the RobotParser.

```
#RULE
   ⟨Move⟩ = ("go" | &)("north" GoNorth
                | "south" GoSouth
                | "east" GoEast
                | "west" GoWest
                )
   #GoNorth
   Go(North)
   #GoSouth
   Go(South)
   #GoEast
   Go(East)
   #GoWest
   Go(West)
#END
#Rule
   ⟨GetIt⟩ = ("get" | "take")⟨Object⟩
             ("in" ⟨Object⟩ GetFluids | GetIt)
   #GetFluids
   GetObject(@⟨Object⟩1, @⟨Object⟩2)
   #GetIt
   GetObject(@⟨Object⟩1, NoObject)
#END
#Rule
   ⟨TakeIt⟩ = ("get" | "take")⟨Object⟩
              ("in" ⟨Object⟩ GetFluids | GetIt)
   #GetFluids
   GetObject(@⟨Object⟩1, @⟨Object⟩2)
   #GetIt
   GetObject(@⟨Object⟩1, NoObject)
#END
```

Listing 2: RobotParser.mir.

Within the MIRA application generator environment[5] the generation of the robot interface can be represented as in Figure 1.

3. OBJECT RECOGNITION AND IMAGE PROCESSING

Object recognition can be a challenging task, even for humans. With automated object recognition, a general model of the object must be defined and algorithms developed to search for the model within the scene. Such models are analogous to a mental image used by a human. To be effective, the model must be general enough to allow for different views and positioning of the object and be able to differentiate the object from other, entangling, objects. Furthermore, the algorithms effectively apply the model to the given image.

Suetens, Fua, and Hanson[7] propose a classification scheme for understanding the approaches to the object recognition problem. These relate features of the model itself (simple or complex) and features of the image (simple or complex, reliable or noisy). Different computational strategies are appropriate for each case.

For example, if the object to be recognized is adequately characterized in the given image by a set of specific features (e.g., colour, texture), then a simple strategy that scans the image for those features is appropriate. On the other hand, if the object requires a more complex representation and the image may contain other objects with similar representations, then more sophisticated strategies are needed.

The authors describe four basic strategies: feature vector search (for simple models and simple image data); fitting models to the photometry, that is, digitized image data (for simple models and complex image data); fitting models to symbolic structures (for complex models and simple image data); and combined strategies (for complex models and complex image data).

Fitting models to symbolic structures assumes that a set of features has been reliably extracted from the image data by some pre-processing operation. These features are usually found by a local statistics-based operator, without using shape information or contextual scene knowledge. This process is often referrred to as segmentation. The features, however, may also be the raw pixel intensities or even labels produced by a method such as template matching or gradient descent, thereby yielding a hybrid strategy. In these cases, the subsequent matching processes use the symbolic output of the initial process without referring back to the image. The major categories are as follows:

- *Graph Matching.* Objects are modelled as a relational structure or graph of primitives. The nodes are components of the object or scene, whereas the arcs denote relationships. Labels are assigned by searching for the optimal match between the model graphs and the graph derived from the image data. Heuristics can be used to prune the search tree and reduce the computation time at the possible risk of finding a nonoptimal solution.

https://doi.org/10.1017/S026357479700012X Published online by Cambridge University Press

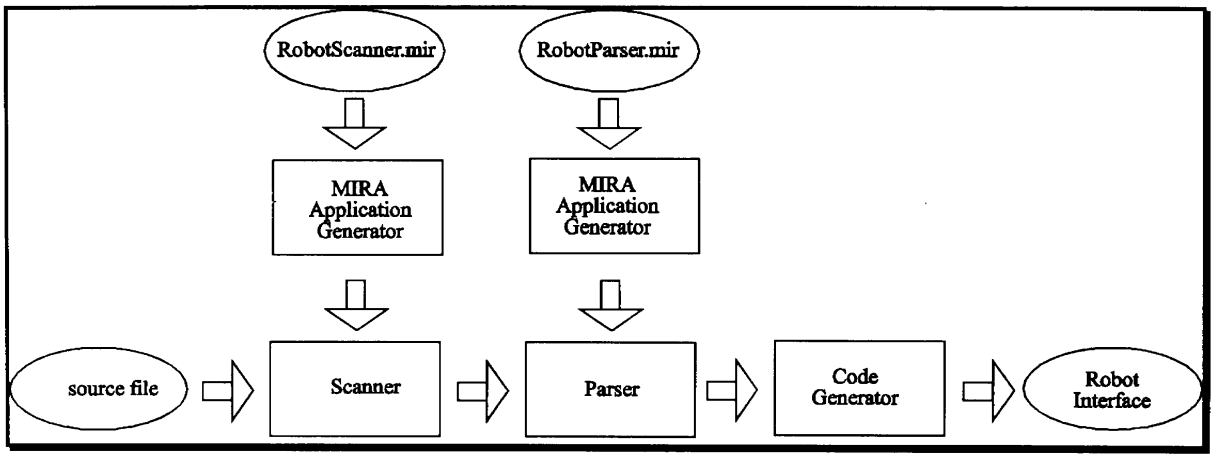


Fig. 1. Structure of the MIRA generation tool for the robot interface.

- *Composite* (*Hierarchical*) *Model Fitting.* A reduction of the search space is obtained by working hierarchically, that is, by finding partial matches using a hierarchy of intermediate models and then refining them.

In this latter method, hierarchical modelling techniques are used to limit the search; a sequence of intermediate and progressively more complete models is used to find and refine partial matches. The intermediate states in the computational process have an obvious semantic meaning and create a context for the subsequent analysis. For example, if the problem is to recognize yellow cars in an aerial image, we might first look for all yellow patches in the image, then see which of the patches had the characteristics of a car. Solving this problem relies on a large amount of domain-specific knowledge. The designer's knowledge about a complex domain needs to be represented in a form that offers flexibility. Knowledge-based systems attempt to achieve this goal by using declarative languages, as in rule-based systems. Rules are to be considered as small pieces of domain knowledge, and their activation produces intermediate states with an obvious semantic meaning. Such systems therefore typically have a high degree of human understandability. This information can be described in a domain specific language such as "CDL" (Configuration Description Language). The description in CDL is used to generate configuration files that are used by the software. The information can be entered at a high level of abstraction. Before generating the configuration files, a set of semantic controls are executed to ensure the consistency of the data. After analyzing the application domain in the *domain analysis* step, the results are used to implement reusable components and the actual application within the domain. The MIRA environment is used for the definition of domain-specific languages. The user can generate an ELL(1) parser (lexical as well as syntax analyser) starting from a BNF-like specification of the domain language.

Figure 2 shows a typical structure of an application built with MIRA. The two most important components, i.e. the lexical and syntax analyzers are both generated from a BNF description of the syntax of the domain language. After compilation of the generated code, we

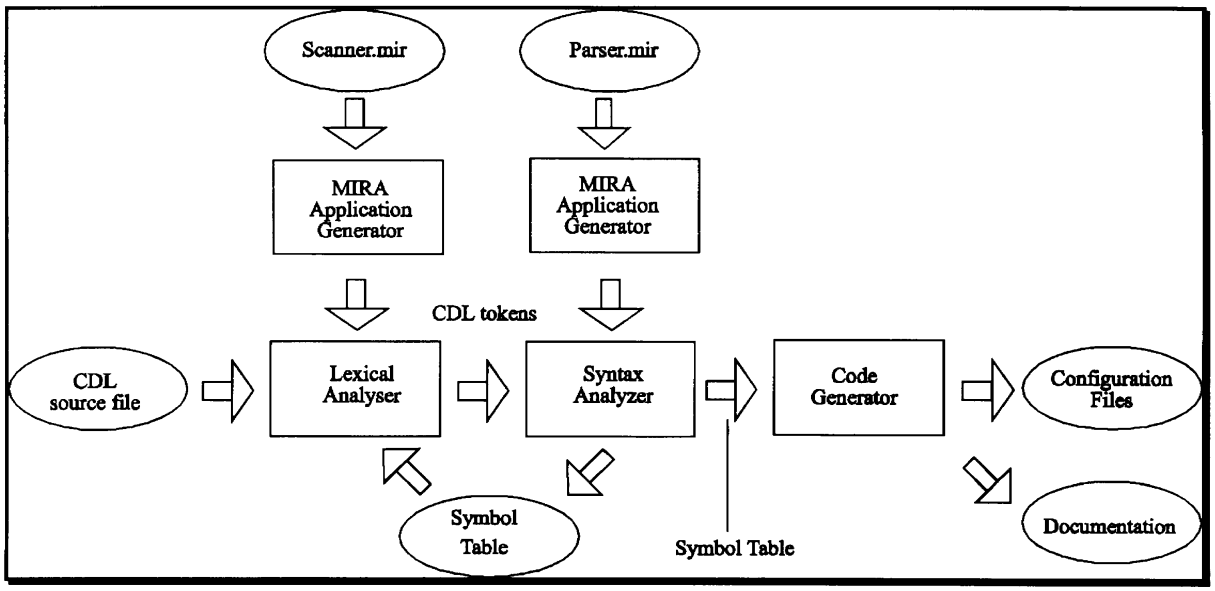


Fig. 2. Use of an application generator.

https://doi.org/10.1017/S026357479700012X Published online by Cambridge University Press

get an application which reads a configuration specified in the domain language, checks it for completeness and consistency and generates an equivalent description on a lower level of abstraction (e.g. binary configuration files).

It is clear that these methods should be restricted to situations with moderately complex photometry combined with semantic complexity. Therefore, in controlled environments such as an automated factory, where we can depend upon low-level operators to extract relevant features.

4. MOTION CONTROL

Consider a highly automated factory where mobile robots pick up parts and deliver them to assembly robots. The robots must find their way to parts, pick them up, and move to the assembly stations. All these motions have to be executed without colliding with objects and other robots. Paths of the robots have to be short, and the pickup operations should not include unnecessary movements. Without a motion planner for the robots and arms, human operators have to constantly specify the motions. An automatic motion planner will relieve the operators from this tedious job and enable them to control at a supervisory level. In turn, this increases efficiency by eliminating human errors. The need for collision avoidance and efficient motion leads to the problem of motion planning (MP). Motion planning can be broadly classified as either gross- or fine-motion planning. Gross-motion planning is concerned with the problems involving free space much wider than the objects' sizes plus the positional error of the robot. This ensures that positional error will not cause unexpected collisions while executing the collision-free paths generated by gross motion planners. Fine-motion planning deals with the problem of moving objects when the space is so narrow that the required accuracy of motion exceeds a robot's positional accuracy. Each problem requires a different approach to plan motion.

Hwang and Ahuja[7] present a survey of recent developments in gross-motion planning. It summarizes the past work, mostly in computational geometry and robotics, and discusses possible directions for future research. It is directed toward people interested in motion-planning research or in implementing an algorithm on real robots. There are several different classes of motion-planning problems. Motion planning can be static or dynamic, depending on whether the information on the robot's environment is fixed or updated. If obstacles are stationary (moving), it is called time invariant (time varying). If robots can move some objects, it is called the movable-object problem. If motions of more than one robot are planned, it is called the multi-movers problem. If the robot can change its shape it is conformable. Motion planning is either constrained or unconstrained, depending on whether there are constraints on a robot's motion other than the geometric interference. These include bounds on a robot's velocity and acceleration and constraints on the curvature of a robot's paths.

A basic issue in motion planning, is the model of objects in its environment a robot has to have before it can plan a collision-free motion. In an unknown environment or a dynamic environment, the robot has to sense objects at an appropriate time interval. Typically used are visual sensors, e.g. stereo cameras, or range sensors based on sonar, infrared, or laser light. The system collects video frames and compares them with still pictures from a data base. The functionality of such a system is primarily determined by the system's interactions with its environment. In *Specification and Design of Embedded Hardware-Software System.*[8] Gajski and Vahid present a methodology for the specification and design of today's embedded-system product development requirements. They propose the use of a formal specification language, which allows creation of a precise specification that can be simulated, thus helping detect and correct functional errors at an early stage and reducing overall design time. Capturing a precise specification of a complex system in a formal language is a complex task. It is not a simple process of ''writing down'' a well-understood functionality; rather, it is the process of learning, understanding, organizing, and defining a functionality. Specification capture consists of three subtasks: model creation, description generation, and simulation. We usually must iterate these tasks several times before we obtain a complete and correct functional specification.

The choice of a model is the most important influence on our ability to understand and define system functionality during specification. Once we have chosen the appropriate model, we must capture system functionality in a functional specification, using one of many different languages. A functional specification is easy to generate if there is a one-to-one correspondence between model characteristics and language constructs. If a language construct does not exist for a particular characteristic, we must try to find a set of constructs that describes that characteristic. This leads to a less readable description, possibly with more functional errors. If the system does not have to perform complex computations, but must respond to complex sequences of external events, such as a control-dominated system of a robot, the finite-state machine (FMS) may be the most appropriate model. It represents the system as a finite set of states and a set of arcs that indicate transition of the system from one state to another when certain events occur.

The descriptions of these finite-state diagrams are made in the domain-specific STRUKT language[9] and are typically between 20,000 and 200,000 lines long. The STRUKT language allows to model a process as a hierarchy of subprocesses. Every process consists of a number of states, transitions and conditional transitions. While a process is in a particular state, multiple tasks are executed and state transitions can be induced. The STRUKT syntax-oriented editor allows to enter and update a STRUKT test interactively. At every moment, the software checks whether an entry is syntactically correct and whether the modifications are syntactically

https://doi.org/10.1017/S026357479700012X Published online by Cambridge University Press

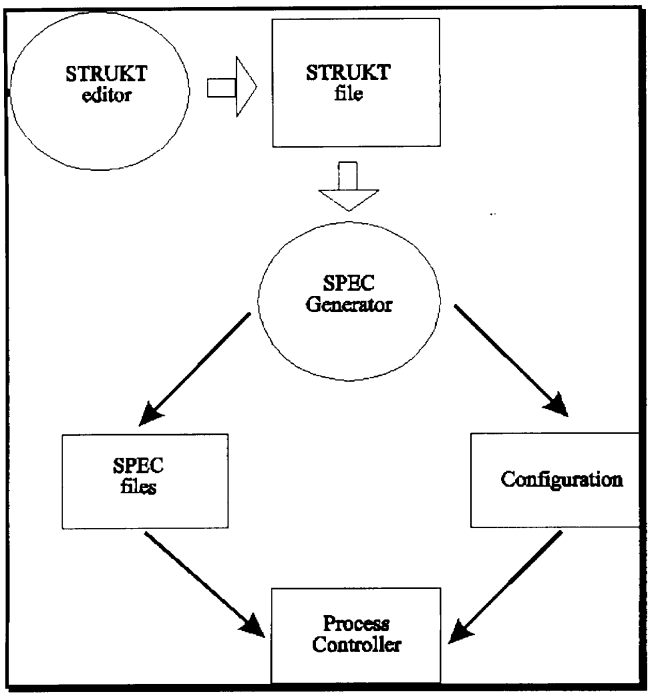


Fig. 3. The STRUKT and SPEC environment.

allowed. In this way, an edited STRUKT description is continually free of syntax errors.

The STRUKT editor offers a number of filters which allow to view a process specification from multiple viewpoints depending on the level of detail that is desired. Moreover, the editor formats the entered text, which standardizes the layout of STRUKT descriptions. This increases the readability and facilitates the reuse and transfer of STRUKT texts. The syntax-oriented editor internally builds an internal data structure (a so-called abstract syntax tree) instead of a normal flat text structure. This allows to easily perform interactive consistency checks and generate documentation. After a process definition is entered, the STRUKT editor writes a STRUKT file. Besides the STRUKT editor, a SPEC generator is developed, that reads the STRUKT file and transforms it into a number of SPEC files. The SPEC files are the input for a number of other tools which transform them into a format that can be read by the process computers (see Figure 3). STRUKT and SPEC are application generators that have been described as ELL(1) specifications and were generated by MIRA.

5. CONCLUSION

In large software projects, the construction of programs of high quality is a difficult and tedious task. This task becomes manageable by introducing a methodology that promotes simplicity, reliability and adaptability in the first place, and portality and efficiency in the second place. Such a methodology can be tuned towards a given application area, such as robot controllers. This application-oriented methodology is in fact the accumulation of an amount of experiences and know-how about the application, gathered by many people and made accessible for any potential user working in that area. If a methodology has been established in a given application area, the next logical step is to design a software tool to support that methodology. This tool must be such that the user is encouraged and even forced to follow the underlying methodology. With the given examples, we have shown that one is able to use ELL(1) descriptions and MIRA to produce robot controllers meeting the requirements of simplicity, reliability, adaptability, portability and efficiency.

References

1. R.C. Arkin, "Intelligent Robotic Systems" *IEEE Expert* **10,** 6–8 (1995).
2. H.R. Hartson and D. Hix, "Human-Computer interface development: concepts and systems for its management" *ACM Computing Surveys* **21,** 5–92 (1989).
3. J. Lewi, K. de Vlaminck, E. Steegmans and I. Van Horebeek, *Software Developent by LL*(1) *Syntax Description* (J. Wiley & Sons, New York, 1992).
4. A. Hoogewijs, "Language specification and automated programming" *CC-AI* **12,** 343–364 (1995).
5. M. Huybrechts, "Visual Mira" *CC-AI, CC-AI* **12,** 365–381 (1995).
6. P. Suetens, P. Fua and A.J. Hanson, "Computational strategies for object recognition" *ACM Computing Surveys* **24,** 5–61 (1992).
7. Y.K. Hwang and N. Ahuja "Gross Motion Planning – A survey" *ACM Computing Surveys* **24,** 219–291 (1992).
8. D.D. Gajski and F. Vahid, "Specification and Design of Embedded Hardware-Software Systems" *IEEE Design and Test of Computers* **12,** 53–67 (1995).
9. G. Pauwels, "Process control and domain-specific languages" *CC-AI* **12,** 425–434 (1995).

https://doi.org/10.1017/S026357479700012X Published online by Cambridge University Press