A natural language interface for a flexible assembly cell

Afke van Rijn
Delft University of Technology
P.O.Box 356, 2600 AJ Delft, The Netherlands
Email: afke@dutirt2.tudelft.nl

Abstract

When a program for the construction of a product by a flexible assembly cell is automatically derived from a CAD drawing, this may give rise to problems. To help the programming process additional input by an expert will be very practical. In this paper a description is given of how to handle the input of additional information with a user interface. This user interface consists of two parts: an instruction interpreter and a dialogue system. With the instruction interpreter the expert can give advice on (details of the) the cell program; with the dialogue system the programming system itself can ask the expert for solutions of problems it encounters. When the expert uses the instruction interpreter or communicates with the dialogue system, he may use a subset of English, which facilitates the communication.

1 Introduction

At the Delft University of Technology a project concerning the design of a flexible assembly cell is being carried out. The cell, consisting of two robots, is to produce small series of varying kinds of products. Programming this cell will proceed off-line, since for small series production on-line programming is not cost–effective.

The way a product is to be assembled, is determined based on a CAD drawing and data about the capabilities of the cell. To obtain a program for the cell, this information is processed by several modules, viz. the assembly modeller, the planning module, the scheduler and the dispatcher respectively. But during this process several problems may occur. First of all there are problems like ambiguity when the system cannot make a choice between several possibilities, or incompleteness/unsolvability then the system needs additional information to be able to continue the programming process. But another important problem is that in a product the number of choices the system has, grows exponentially with the number of parts. Because of these problems it would be practical to have a possibility for additional user input to help and optimize the programming process.

To this effect a user interface has been constructed that consists of two parts. First there is an instruction interpreter that accepts assembly instructions. In this way the programmer can give additional information to the system concerning problems that he recognizes beforehand. He can also use this instruction to give certain adjustment data for the product (like the specification of a torque). Secondly, there is a dialogue system where the system itself can ask for additional information when this is needed. A global overview of the system is given in figure 1.

In this paper both the instruction interpreter and the dialogue system are described. This is followed by a brief overview on the knowledge representation used for both modules. The final section contains some concluding remarks.

2 The instruction interpreter

The instruction interpreter is meant to give the programmer of the flexible assembly cell the opportunity to give certain information to the programming system of the cell, like adjustment data, data on product or assembly details or boundary conditions to the assembly process. To this effect, the instruction interpreter obtains a text in a certain "programming language". Since it is not cost–effective to have to make complex programs for small series production, the language in which the text is written, should be easy. To this effect a subset of English was chosen as the "programming language". A text in this sublanguage resembles assembly instructions as they come with model construction boxes. These assembly instructions are either contained in the CAD drawing or in a separate piece of text.

To program the flexible assembly cell for small series

production, existing robot programming languages are unsuitable since, first of all, these cannot be used directly to program a cell consisting of more than one robot and secondly to construct a program in most of the languages is rather complex, which, as was said before, is not cost–effective for small series production (for a comparison between fourteen existing robot programming languages see [Bonner, 1982]). In the literature some natural language interfaces for robots can be found as well, see for example [Winograd, 1972], [Evrard, 1983], [Selfridge, 1986] and [Kratchanov, 1987]. But these are for on–line programming of a single robot, which is not practical when small series production is concerned, since it requires taking the robot out of production. Besides, working with a single robot sets different boundary conditions to the programming process than working with more than one robot in parallel.

Working with natural language, one is relatively free in one's choice of words and grammatical constructions. But on the other hand working with a subset of language implies a restricted vocabulary (with less ambiguities) and a restricted grammar. The subset of language used in assembly instructions contains several sentence types.

The most important sentence type is an assembly task. An assembly task contains information on the part that has to be assembled and possibly the parts to which the part has to be assembled or the tool to do the assembly with. It must be possible to supplement the tool or parts with some arguments (like a torque for a screwdriver or a male–female connection between two parts). An example of an assembly task is "Screw the screw on the plate using a Phillips screwdriver and a torque of 0.5 Nm" or "The screw has to be screwed on the plate with a torque of 0.5 Nm".

Other sentence types are:

- a conditional construction that may either indicate conditions that denote when a task can or cannot be executed (for example "Assemble the lid to the box only if the ball is already assembled"), or make it possible to use one instruction for several types of a product (for example "If the motorbike is of type r, paint it red");
- the definition of a subassembly that contains the name of the subassembly and the tasks to be executed in order to construct the subassembly; this information will be used to restrict the number of assembly sequences (for example "To make a wheel perform the following actions ..."");
- an iteration which indicates that tasks must be executed more than once (for example "Assemble the bolts and tighten them to a torque of 0.5 Nm");
- a test instruction to specify a test whether a specific state of a part or assembly has been attained, it can be followed by tasks to be executed when the test fails (for example "Test the crankshaft for freedom of rotation");
- a warning instruction to indicate that some specific assembly tasks have to be executed with extra conditions to avoid a normally acceptable but for the tasks unacceptable side effect (for example "Gently insert the camshaft into the crankcase taking care not to damage the camshaft bearings with the cams");
- an instruction to indicate an ordering between the assembly tasks which can be used to restrict the number of possible assembly sequences (for example "Execute the following instructions in the given order" or "Assemble the ball before the lid is assembled").

Since the instruction interpreter is implemented in Prolog, the sentences are represented with Prolog clauses. An assembly task for example is transformed into the Prolog clause :

```
assemble( part1(Conc-P1, Actent-P1, [P1arg]),
          [part2(Conc-P2, Actent-P2, [P2arg],
                 [Carg], [Garg])],
          [tool(Conc-T, Actent-T, [Targ])] )
```

This clause indicates the part that has to be assembled (part 1) and arguments for this part (P1arg-list). A part in a product is identified by the concept (Conc-P) and by a unique identification, called an "actual entity" (Actent-P). By using actual entities it is possible to distinguish between similar occurrences of a part. For example a 'screw' may denote many actual 'screw'-s in the product. The actual entity indicates the specific 'screw' of the product it concerns. Besides this, part(s) to which the part has to be assembled may be indicated in the list of Part2-s; this list may also contain information on arguments of the part2 (P2arg), information on the connection between part1 and part2 (Carg) or information on the geometry between part 1 and part2 (Garg). The connection and geometry arguments always relate to two parts: they are binary relations. This is required by the product data model that only contains those binary relations between two parts that are automatically derived from the CAD drawing. The product data model is a structural description of the product. To integrate the information from the assembly instruction in the product data model, the relations derived from the instruction must be binary as well. As a last item in the clause the tool to perform the assembly with may be indicated and also arguments (Targ) for this tool.

Most of the information of the assembly task is integrated in the product data model derived from the CAD drawing (connections, geometry, arguments of tools), the rest (tools) is communicated directly to the planning module. Of course the information contained in the product data model is accessible to the planning module as well. All information derived from the other sentence types, except the iteration instruction, goes directly to the planning module. This information is represented in clauses and relations between such clauses similar to the "assemble"–clause. The iteration instruction is expanded by the instruction interpreter in such a way that the

data on every part or subassembly contained in the iteration is connected to that part or subassembly. Only this expansion is communicated to the other modules.

3 The dialogue system

The dialogue system serves to solve problems occurring in the modules of the programming system of the assembly cell. Several of the classes of problems that occur were already mentioned; for controlling the process these were ambiguity, incompleteness and unsolvability and for optimizing the process they were requests for advice for example concerning assembly sequences. But another problem may arise, namely contradiction; most of the time this will be caused by user input that contradicts data that is derived automatically by the system. Solving problems with dialogue will also be very useful for the exception handler. Exception handling is done on–line, which implies that on–line dialogue may be conducted as well.

When a problem in a module occurs, a request for dialogue stating the problem and information required to be able to solve the problem, has to be "sent" to the dialogue system by that module. The dialogue system translates the problem into a question to the user. The dialogue may be supported with a graphical interface. The dialogue may only be initiated by the system and the user answering the question should be an expert in the problem domain and should also be helpful to the system. Of course being engaged in a dialogue, the initiative may change. This means that the user will be able to ask additional information concerning the problem from the system before answering the question. In the sequel only the control of this dialogue will be discussed and not the analysis and generation of language.

When dialogue systems are studied many of them use one or more aspects from speech act theory [Searle, 1969]. In speech act theory a distinction is made in the forces of speech acts:

1. locutionary force: the utterance of words
2. propositional force: the content of the speech act
3. illocutionary force: the form of the speech act (question, statement, ...)
4. perlocutionary force: the effect of the speech act on the listener

The locutionary force is the least interesting one. The perlocutionary force is the hardest to model, since it depends on the situation of the speaker and listener. Both are not considered further in the dialogue system. But in the dialogue system a distinction will be made similar to the distinction between the illocutionary force and propositional force. The illocutionary force is used in structuring the dialogue. The propositional force represents the meaning of the speech acts.

When dialogue is studied in an assembly environment, a set of speech acts that are used can be constructed, as for example greeting, question, answer, ... To determine the speech act represented in a sentence, a set of features of the sentence is constructed, that enables the system to recognize the speech act. This causes some problems when indirect speech acts are used. Indirect speech acts are sentences that literally mean one thing but are used in another way. For example, "Can you tell me the time" is a yes/no question when interpreted literally, but in most contexts this sentence is a request for the right time. Interpretation of indirect speech acts has been a subject of much research in speech act theory; it will not be considered further in this paper.

To structure the dialogue, a transition network may be used, as [Bruce, 1986] does. When a transition network is used, the system speech act is determined, based on previous speech acts, while the user speech act must be recognized based on previous speech acts. The user is in principle free to perform any speech act he wants, as long as he does not change the topic of the discourse. To control this dialogue the system performs the speech act of which the illocutionary force is indicated at the transition of the network. When a user performs a speech act, the system has to proceed to the next state in the network according to the illocutionary force of the speech act performed by the user. Thus, the structural function of a speech act in the dialogue, consists of two parts:

1. the illocutionary force of the speech act
2. the position of the speech act in the dialogue (structured by a transition network)

But, when questions and answers are considered, this mechanism is not satisfactory. With these question and answer speech acts, the meaning of the speech act is of much importance and must be processed by the dialogue control system. The meaning of other speech acts (like greeting) is of less importance and will not be remembered by the dialogue system. The only way these other speech acts influence the dialogue system is that they give rise to a transition in the transition network.

To be able to control the question and answer speech acts, a subdivision has been made in the kinds of questions that occur. These are:

1. multiple choice question: in a multiple choice question it is asked which of a distinct set of possible answers is the right answer;
2. open question: in an open question new information is wanted.

To represent the meaning of these speech acts a logical language has been defined, in which both of these question speech acts are representable. Both the open question and the multiple choice question will be represented with a lambda

formula, the multiple choice question needs an additional select–function. Some examples of the representation of a question with this language, are:

$(\lambda x : screw)[attach(x,plate1)]$
that represents the question "Which screw must be attached to plate1";

$(\lambda x : screw)(\lambda y : attach)[y(x,plate1)]$
that represents the question "Which screw must be attached to plate1 with what kind of connection";

$(\lambda x : screw)[attach(x,plate1) \wedge$
$x = select(screw1,screw2)]$
that represents the question "Must screw1 or screw2 be attached to plate1?".

The appropriate answers are represented similarly.

As was said before, to structure the dialogue for questions and answers, the transition network is not enough. A sequent system has been defined, which is able to handle the processing of the meanings of questions and answers. This sequent system is an augmentation of the transition network: the network can (and will) be easily represented in the sequent system. A state in the transition network will become a sequent and a state transition will become a sequent transition. In this way the structure of the dialogue will be represented with the sequent system. To be able to handle the question and answer speech acts, an additional mechanism is constructed consisting of two stacks, one for the questions of the user and one for the questions of the system.

The sequent system is a four-tuple $<\Sigma,\Pi,A,Y>$, in which Σ and Y are the stack of the system questions and the user questions respectively; Π is the party whose turn it is and A is the speech act of the most recent party. A is of the form $A(\phi)$ in which A is the illocutionary force of the speech act and ϕ is the meaning of the speech act if A is a question or an answer. Whenever a question is asked, this question is put on top of the appropriate stack, when an answer is given the stacks are changed according to the answer given. Usually this will only change the question on top of the appropriate stack, but it is possible to change other questions as well. Processing the answer may give rise to new questions if a question is not completely answered, otherwise the question is removed from the stack. The dialogue is finished when both the system stack and user stack are empty.

To describe the working of the sequent system, a set of sequent transitions is defined. These describe when a transition (that may be conditional) from sequent to sequent is allowed. In this way the dialogue is structured and the stacks are changed according to the meanings of questions and answers. Some examples of sequents and sequent transitions are:

1. The sequent that represents the first question (always asked by the system) will be:
   $s_i = <[\phi],user,Q(\phi),\varnothing>$

2. The sequent transition representing the processing of an answer of the user on the stacks will be:
   $<[\phi \mid \Sigma],user,Q(\phi),[\Theta \mid Y]> \rightarrow$
   $<[\phi' \mid \Sigma'],system,A(\psi),[\Theta' \mid Y']>$
   As can be seen both stacks may be changed according to the answer of the user.

3. A conditional sequent transition representing the termination of the question and answering dialogue when the stack of user questions is empty and the stack of system questions only contains a totally answered question, will be:
   $\text{t-type}(\phi) \Rightarrow (<[\phi],\Pi,A,\varnothing> \rightarrow s_f)$
   in which t-type is a function that determines if the question ϕ is totally instantiated (of type t), and s_f denotes a sequent that starts the terminating dialogue.

A formal description of both the logical language and the sequent system can be found in [van der Leeuw, 1990].

4 The internal representation

For the internal representation of both the sentences analyzed by the instruction interpreter and the sentences analyzed and generated by the dialogue system, conceptual dependency graphs [van Rijn, 1989b] are used. "Conceptual dependency graphs" is a formalized and generalized version of conceptual dependency theory [Schank, 1975]. Conceptual dependency graphs are very similar to conceptual graphs [Sowa, 1984]. For a comparison see [van Rijn, 1989a]. The knowledge representation obtained is a high level knowledge representation independent of the actually chosen concepts. The question whether this representation corresponds in any sense with human usage and similar psychological issues is of no concern to us.

This representation was chosen, because the programming system uses different knowledge representations for the various programming modules. The representation that is used for the dialogue system in the assembly environment, should function as a kind of interlingua between the user and the various modules of the cell, each with its own knowledge representation. Each of these knowledge representations is based on concepts and relations between concepts. The user interface needs a knowledge representation that can handle both the assembly knowledge contained in the modules it communicates with and "linguistic" knowledge to communicate with the user.

Also, when the representation is used to represent the assembly instruction, it should store from that instruction the concepts and relations between concepts.

The objects needed in the knowledge representation are concepts and relations between concepts, the possibility to introduce new concepts in the world (necessary for the definition of subassemblies), the use of actual entities (to denote

a specific part in a product) and the possibility to establish relations between networks of concepts (to be able to represent the more complex sentence types). Since most of these objects were present in the conceptual dependency theory, this theory was chosen as a basis for the representation "conceptual dependency graphs". It was changed according to the needs for use of the representation in an assembly environment. But, conceptual dependency graphs is a high level representation independent of the concepts chosen. The representation can be filled in in the way Schank does or in a way suitable for the assembly application. The representation can be used easily for diverse application areas.

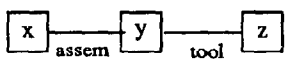
In the knowledge representation, concepts are ordered in unlabelled, acyclic, directed graphs. A concept is an instantiation of another concept if there exists a directed path between the concepts. Concepts may have actual entities, that are objects unique in the world under consideration. Conceptual categories form a partition on the set of concepts. Furthermore between conceptual categories a restricted set of conceptual relations may exist. This whole is called the concept structure.

Example 1 *When TOOL is considered as a conceptual category consisting of the set of tools {normal screwdriver, Phillips screwdriver, screwdriver, hammer, fastener, gripper}. A concept ordering for this conceptual category may be:*

{normal screwdriver, Phillips screwdriver}=screwdriver
{screwdriver, hammer}=fastener
(with {a, b}= c meaning that a ∈ c and b ∈ c).
Thus, a normal screwdriver is an instantiation of a fastener.

Simple conceptual dependency graphs are directed graphs with nodes that carry three labels, namely a conceptual category, a finite set of concepts belonging to this category and an actual entity belonging to each of these concepts and with labelled arcs relating two or more nodes. Simple conceptual dependency graphs represent simple sentences.

Example 2 *To represent an assembly instruction for a specific product, the main conceptual categories are ACT, PART and TOOL. The most important ACT for assembly is 'MOUNT' representing the primitive assembly task of the robot. The conceptual categories PART and TOOL contain the different parts and tools. Tools or parts may be instantiated with an actual entity. The simple conceptual dependency graph presented below, represents the sentence "Assemble the Phillipsscrew".*



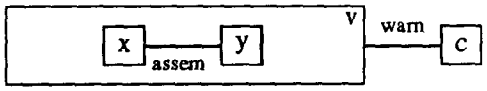
x f_1=PART y f_1=ACT z f_1=TOOL
f_2={Phill.screw} f_2={MOUNT} f_2={sdriver}
f_3= item005 f_3=impossible f_3=tool5

The nodes are connected to one another with labelled arcs, representing the "assemble" relation (between node x *and node* y*) and the "tool" relation (between node* y *and node* z*). Each of the nodes has three labels:*

1. *f_1 represents the conceptual category to which the node belongs;*
2. *f_2 represents the finite set of concepts;*
3. *f_3 represents the actual entity; it has the value "impossible" if the conceptual category given by f_1 does not allow actual entities.*

Two simple conceptual dependency graphs or a simple conceptual dependency graph and a concept may be related to form a conceptual dependency graph. Two conceptual dependency graphs or a conceptual dependency graph and a concept may again be related to form a more complex conceptual dependency graph. Conceptual dependency graphs are labelled similarly to simple conceptual dependency graphs. In this way it is possible to introduce new concepts in the world, which is necessary in assembly when a subassembly is defined.

Example 3 *An example of a conceptual dependency graph representing the sentence "Assemble the egg cautiously" is*



v F_1=CDG x f_1=PART y f_1=ACT
F_2= f_2={egg} f_2={MOUNT}
F_3= f_3=item01 f_3=impossible

c f_1= DA
f_2={cautious}
f_3=impossible

Node v *has three labels similar to nodes of simple conceptual dependency graphs, but only F_1 has a value: the dummy value "CDG". This labeling of conceptual dependency graphs is done to be able to introduce new concepts in the world. As a new conceptual category (node* c*) DA is used denoting a default argument.*

To be able to distinguish between conceptual dependency graphs that may have a meaning in the world and those that have not, a subclass of conceptual dependency graphs is introduced, called the well–formed conceptual dependency graphs. Well–formed conceptual dependency graphs are those conceptual dependency graphs that may be derived from the lexicon. To this purpose each lexicon entry of a word contains besides the transformation of the word into a concept a required context of the concept. This required context is specified as a (simple) conceptual dependency graph. Well–formed conceptual dependency graphs are obtained by unification of required contexts contained in lexicon entries.

To use this high level representation in an assembly environment, the concept structure must be filled in and the lexicon must be constructed according to the application. Assembly knowledge is contained in the lexicon entries. For example the lexicon entry of a normal screw could contain the information that it should be assembled with a normal screwdriver (and thus not with a Phillips screwdriver!) or the entry of a Phillips screw size 5 that it should be assembled with a screwdriver size 5. The appealing thing of the use of unification to obtain well–formed conceptual dependency graphs is that the representation of a sentence like "The normal screw should be assembled with a screwdriver size 5" contains the information that it concerns the assembly of a normal screw size 5 with a normal screwdriver size 5.

5 Conclusion

It is very practical to use a user interface to support the automatic derivation of a cell program for a specific product from a CAD drawing. In the system described this user interface consists of two parts: first an instruction interpreter to give the programmer of the cell the opportunity to help and optimize the programming process on problems that he foresees in advance and second a dialogue system to let the programmer solve problems that the system encounters during the programming process. A parser for the instruction interpreter has been constructed. This parser is implemented in Prolog, since Prolog is very suitable where unification is concerned. Due to the limited grammar, the parser accepts only sentences with one main verb and possibly one or more auxiliary verbs. At this moment sentences with subclauses are not accepted. The lexicon is still rather small (about a hundred words). Both the lexicon and the grammar will be extended in the future to enable the instruction interpreter to accept a larger number of sentences.

At this moment, only the assembly tasks, the subassembly indication and the order instruction can be handled by the programming system. The planning module is not yet capable to process the other sentence types.

Currently we are working on the dialogue system. The dialogue control of questions and answers has already been implemented. The formalism which is used in handling these speech acts can be easily extended to control the other dialogue structures likewise. The rest of the dialogue system (including the analysis and the generation of natural language) is still under construction. A small prototype of a generator of language from conceptual dependency graphs has been constructed.

The use of conceptual dependency graphs seems to be very practical for an assembly environment. The graphs are very suitable to represent texts of a restricted domain. The representation can be filled with the concepts, conceptual categories and conceptual relations that one needs for any specific domain.

References

[Bonner, 1982] S. Bonner, K.G. Shin; *A comparative study of robot languages*; IEEE Computer, pp. 82-96, 1982.

[Bruce, 1986] B.C. Bruce; Generation as a social act; in: B.J. Grosz, K. Sparck Jones, B.L. Webber; *Readings in natural language processing*; Morgan Kaufman Publishers, 1986.

[Evrard, 1983] F. Evrard, H. Farreny, H. Prade; A flexible interface for understanding task–oriented unconstrained natural language; *Comp. and Art. Intell., vol. 2, nr. 6, pp. 497-51*, 1983.

[Kratchanov, 1987] K. Kratchanov, I. Stanev; A rule–based system for fuzzy natural language control; In: I. Plander; *Artificial intelligence and information control systems of robots*; Elsevier, 1987.

[van der Leeuw, 1990] J. van der Leeuw, A.M.C. van Rijn, R. Sommerhalder; *Dialogue handling*; T.U. Delft, Reports of the Faculty of Techn. Mathematics and Informatics, forthcoming.

[van Rijn, 1989a] A.M.C. van Rijn; *Conceptual dependency graphs*; Proceedings of the Fourth Annual Workshop on Conceptual Structures; Detroit, August 1989.

[van Rijn, 1989b] A.M.C. van Rijn; *Conceptual dependency theory and robot programming*; T.U. Delft, Reports of the Faculty of Techn. Mathematics and Informatics, nr. 89-20.

[Schank, 1975] R.C. Schank; *Conceptual information processing*; North Holland, 1975.

[Searle, 1969] J.R. Searle; *Speech acts*; Cambridge, 1969.

[Selfridge, 1986] M. Selfridge, W. Vannoy; A natural language interface to a robot assembly system; *IEEE Journal of robotics and automation, vol. RA-2, nr. 3, pp.167-171.*

[Sowa, 1984] J.F. Sowa; *Conceptual structures*; Addison Wesley, 1984.

[Winograd, 1972] T. Winograd; *Understanding natural language*; Academic Press, 1972.

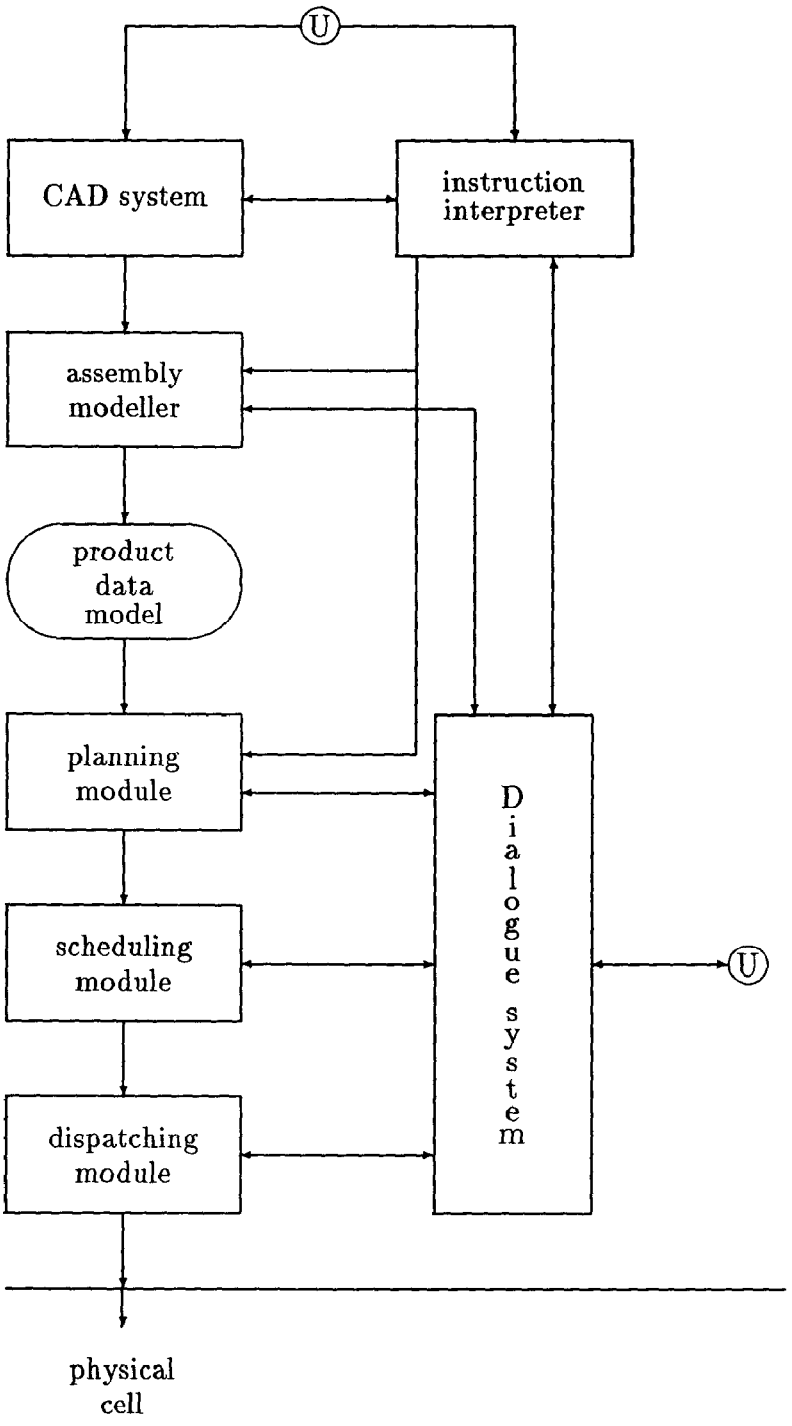


Figure 1: Assembly cell programming system

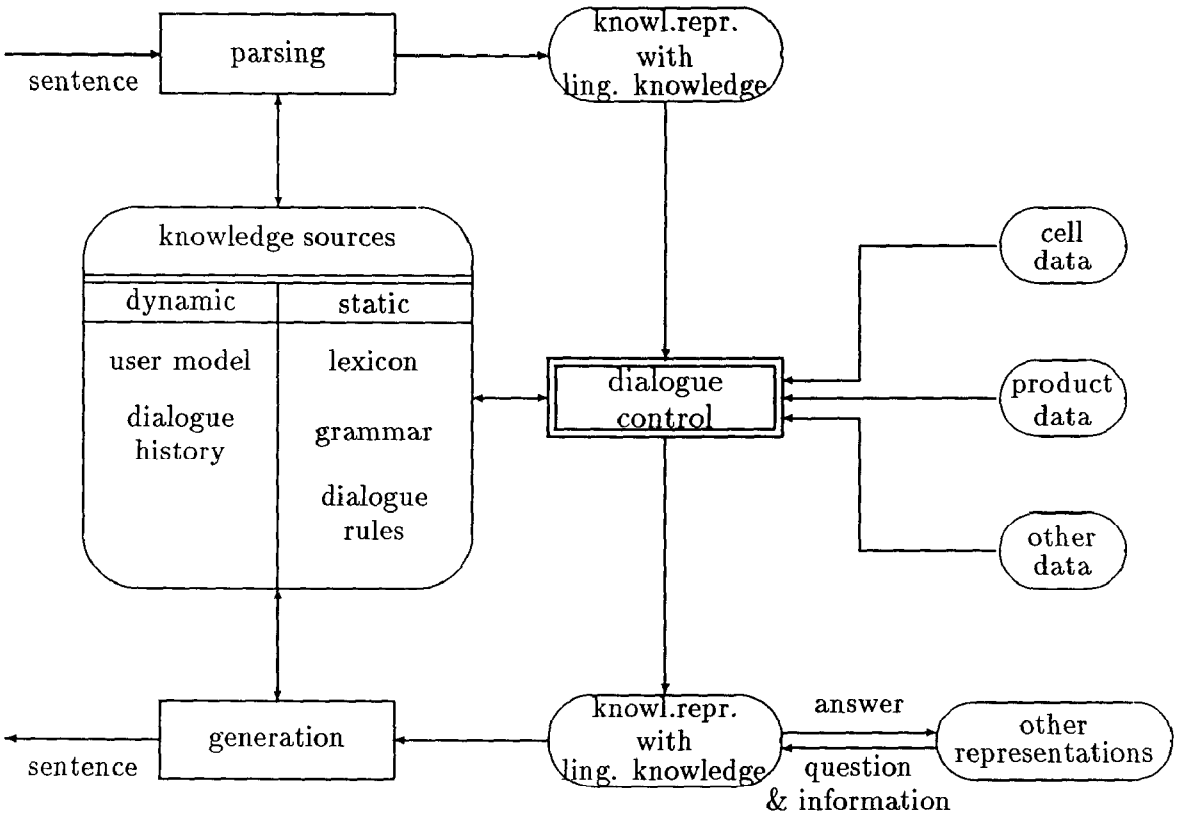


Figure 2: Dialogue system