The software development process for an intelligent robot

by Derek W. Seward and Alastair Garman

The key problems in producing software to control intelligent robots are identified, the principal one being the difficulty of producing a detailed requirements specification. A series of process steps is defined and described in relation to the development of an intelligent robot excavator. The importance of a rational decomposition of the system into modules is stressed, and one particular component of the robot excavator is discussed in some detail—the activities manager. It is shown how techniques such as a 'production system' and a 'blackboard' were implemented in Ada to produce a flexible and easily maintainable system. Two other components are described—the low-level controller and the safety manger. A brief description of the hardware platforms used is included.

There is currently a great deal of research and development going on throughout the world in the area of intelligent robots, particularly for use in hazardous environments such as nuclear decommissioning, toxic waste tips and subsea. In order to carry out useful tasks in unstructured environments, robots inevitably require a richer sensory system than conventional industrial robots, and much increased intelligence to make sense of their more complex world. They also need the intelligence to carry out work sequences that involve subtle and adaptive operations without constant reference to human operators. This means that the proportion of system costs occupied by software will continue to increase for the foreseeable future, and probably dominate hardware costs.

There are specific problems in developing software for intelligent robots, and these problems must be faced and overcome if such robots are to escape from the research laboratories and become viable commercial products. This article sets out to investigate the problems of software development for intelligent robots by reference to a research project to develop a robot excavator. The lessons learned from this project are formulated into a series of steps that constitute a suitable process model for software development.

Background

For the last five years, staff and students at Lancaster University have been involved in the development of an autonomous robot excavator, and the end result of this work is LUCIE—the *L*ancaster *U*niversity *C*omputerised *I*ntelligent *E*xcavator.[1] An excavator provides a good opportunity for development, as it is basically a highly efficient and well developed four-degree-of-freedom manipulator arm, but with the complete absence of automation or intelligence. The aim of the project is to add autonomy in order to produce a robot excavator with the following characteristics:

- It should concentrate on the task of *trenching*, and be able to produce a good quality and accurate smooth-bottomed trench.
- It should adapt to different soil types without human intervention.
- It should cope with obstructions, such as boulders in the trench.

The project needed to be organised in such a way that:

- It is capable of incremental development with different design teams completing a stage of the work throughout an academic year.
- It is highly modular to provide for the possibility of adding, as yet, undeveloped technologies, such as a navigation system and a system for the detection of hidden service pipes.

A working prototype has been produced and the above aims achieved (see Fig. 1). A hardware platform was provided by the JCB excavator company in the form of a JCB 801 tracked mini-excavator. Many lessons have been learned, particularly in the field of software engineering, which are relevant to the generic class of intelligent robots.

The nature of the problem

The conventional opinion in software engineering emphasises the need for a complete and unambiguous definition of the system requirements before moving to the design stage (although it is now accepted that this is usually unrealistic, and some iteration between requirements and design is required). Herein lies the major difficulty with this class of intelligent robot. In our case the task of digging a trench is a subtle manual task that has never been properly defined. Indeed it is clear from the observation of skilled human drivers that their tactics are constantly being modified to maintain efficient operation as ground conditions vary. They make wide use of many sensory feedback loops (e.g. listening to the sound of the engine) to optimise the path of the excavator bucket. Even if the human approach was fully understood, it would still leave a significant problem in knowing how closely an automated method should mimic it. What is needed is a development system that allows tactical experimentation. Furthermore, this experimentation requires at least a prototype *hardware* system (not just the software). Thus we require a 'whole system' development process.

A modern excavator has a very high power-to-weight ratio—about ten times that of a conventional industrial robot. In fact an excavator arm can apply a force which is sufficient to turn itself over. This, combined with mobility, means that safety is an important issue.

The problems of system development can therefore be summarised as follows:

- It is not possible to write the complete system requirements in advance, and hence they must be further refined throughout the development process.
- The undefined nature of the task means that there is a need to be able to tune the system during operation. This points to the necessity for a 'training' facility, where tactics can be refined in the field without the need for constant re-compiling of the software code.
- A working hardware prototype is required for the refinement of the requirements. Consideration was given to software simulation, but this was discounted as the primary tool on the grounds that the simulation of soil with realistic non-homogenous properties is too difficult.
- Large systems will usually consist of many off-the-shelf components and so a top-level architecture which stresses high modularity is vital for the flexible integration of subsystems. An example of such a component is a satellite global positioning system which contains both hardware and software components. The architecture of the system must cope with software components implemented in different computer languages.
- It must be possible to validate the safety of the complete system.

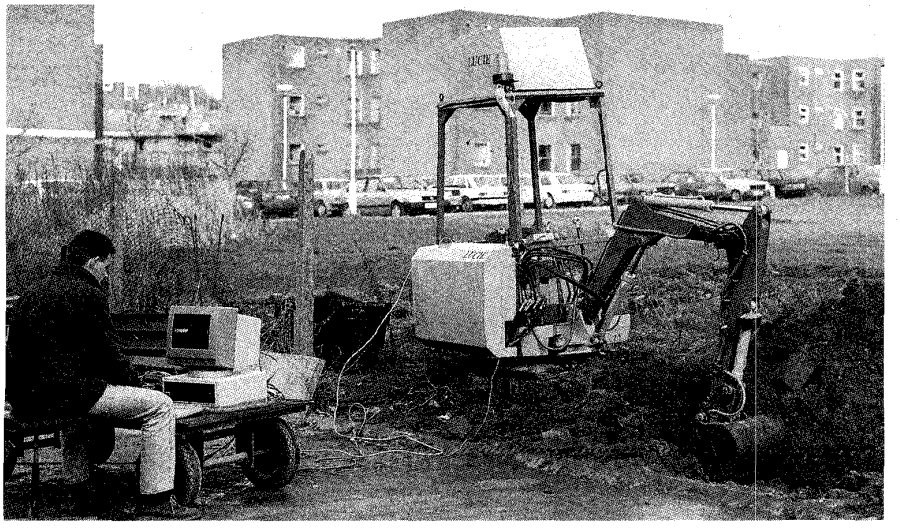

Fig. 1 LUCIE the robot excavator being trained

Stages of development

Of the established software development models, that which comes closest to providing a framework for this development is the Boehm Spiral Model.[2] This encourages the reduction of development risks by incorporating prototypes, simulations and models. However, the suggested steps given below are based more on a critical review of the experience gained on the project than strict adherence to the model:

1 Problem definition and scope

This is to define the principle objectives of the robotic system, and also to indicate estimates of available resources in terms of equipment, finance, manpower and time.

2 Knowledge acquisition

This stage may involve observation and interrogation of human operators so that the robot's operational process can be understood and the functionality and target performance of the system defined.[3] In the particular case of the robot excavator this stage also included a theoretical study of 'soil cutting', and an analysis of excavator kinematics and forces.

3 Preliminary system requirements definition and feasibility study

A high-level description of what is required from the robotic system is produced in written form. This is then compared with available technical resources to see if it is feasible.

4 Global system decomposition

The overall system is broken down into manageable modules which can be relatively independently developed. A rational decomposition is one of the most vital steps in system development, and is dealt with in more detail below.

Fig. 2 Fifth-scale model of excavator arm

5 Detailed requirements specifications for subsystems using a rapid prototype

The risks associated with developing each module should be considered, and as a result of this, the most appropriate process model(s) adopted. If modules can be tightly specified at an early stage, then this should be carried out using a specification technique such as DeMarco's Structured Data Flow Diagrams.[4]

A rapid prototype of both the hardware and software is required for further refinement of the requirements, for those modules which cannot be adequately defined. For large robots it may be advantageous to also build a software or hardware scale model. This is discussed in more detail below. In general the rapid prototype will use off-the-shelf hardware components, which will probably bear little resemblance to a finished production system, and will be far from optimum in terms of cost, robustness, compactness and performance.

6 Design and implementation of a development prototype followed by experimental tuning and production of a detailed requirements specification

This stage will build on the experience gained from stage 5 and the aim is to build a robot which has the full functionality of the final system, but also has additional capabilities for tuning and adjustments so that the detailed requirements can be finalised. It would be expected that many of the hardware and software components would be re-usable in the production version.

7 Design and implementation of the production version to meet the detailed requirements defined in stage 6.

The final production version is likely to contain more highly optimised but less adaptable software, but supported by a professional user interface.

With simpler robots it should be possible to merge steps 5 and 6. The success of the design of the high-level system architecture at stage 5 can be measured by how similar it is to the final product at the end of stage 7. For one-off or low volume robots the development would cease at stage 6. Some specific points from the above steps will now be considered in more detail.

The use of robot models

There are strong arguments for including models throughout the development. These can take the form of mathematical models, graphical computer simulations or physical scale models.

Mathematical models are required for the production of effective control algorithms, particularly when the robot involves complex kinematics or fast moving parts, which add a dynamic component to the problem. This was not

an important issue with LUCIE as the excavator arm is well damped by the soil when actually digging, and when moving in air does not need to be accurately positioned. Graphical computer simulations are useful for defining basic geometry, evaluating working strategies and, as part of the safety analysis, to avoid collisions. Software tools of various complexity are now available for both the above, but are beyond the scope of this article.

Hardware scale models are of particular advantage when developing large robots, and they should, where possible, be designed to be driven by the same software interface as the full-sized prototype. The key benefits are:

- Control software can be tested under laboratory conditions.
- In the case of the excavator, trial holes can be easily dug and refilled without the need to hire a driver.
- There are clear safety advantages compared to testing new control software on large and powerful robots.
- They provide a good demonstrator to motivate staff, provide confidence and encourage further funding.
- They can often be quicker and cheaper to develop than equivalent software simulations.

Fig. 2 shows a fifth-scale model of an excavator arm that was used to test control software and develop digging strategies in a sand box. This model had a similar electro-hydraulic system to the full-scale excavator and was controlled through an identical software interface. It is important that such models are easily scaleable. In this case the same software was successfully used to drive both the fifth-scale model and the full-sized excavator—it was only required to change the geometry and a few control parameters.

Decomposition of system components

A logical top-down decomposition of system components is the key to achieving a flexible and easily maintainable system. (An exception to this can occur when using off-the-shelf components which may dictate some bottom-up considerations to allow for specific interfacing restrictions.) An important software engineering concept that can help in obtaining a coherent decomposition is *coupling*.[5] The degree of coupling is a measure of the amount and range of data-flow that occurs between system modules. For ease of long-term maintenance coupling should be minimised. A useful aid for identifying the high-level modules is to consider where a human would intervene in the system assuming hardware or software faults at various stages in the system. These interventions normally occur at points where coupling is minimised. Fig. 3 illustrates the top-level decomposition for the LUCIE excavator together with the obvious human interventions in the event of system failure. The safety manager occupies a unique position in relation to the other modules and so is not shown connected at this stage.

Two of the modules in Fig. 3—the low-level arm controller and the activities manager—have been fully implemented and the safety manager is currently under development. These will be discussed in the following sections.

Low-level arm controller

The purpose of the low-level arm controller is to take movement commands from the activities manager and send the appropriate control signals to the electro-hydraulic valves. Experiments with the fifth-scale model in step 5 revealed the desirability of a dual control strategy which is closely reflected in the human approach to digging:

- When moving in air (i.e. tipping the spoil and positioning the bucket teeth) a *positional controller* is required. The commands from the activities manager thus instruct the bucket to move to specific *x,y,z* co-

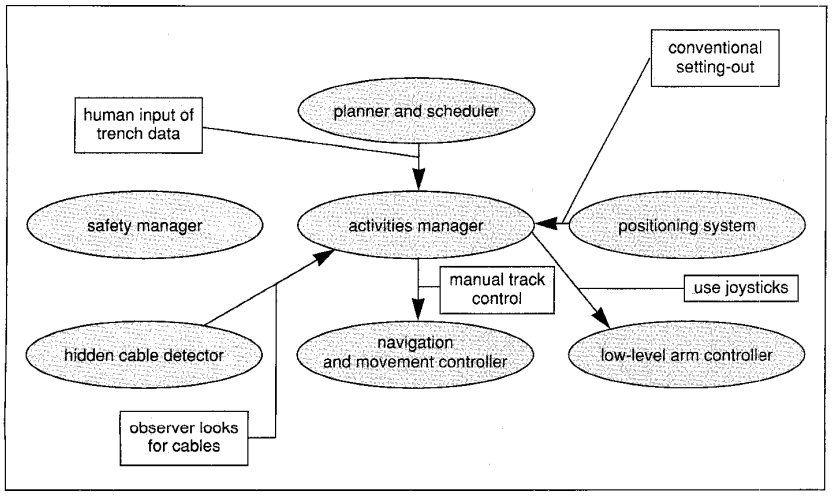

Fig. 3 Top-level decomposition with points of human intervention

ordinates in space. Angle sensors were placed on the arm joints to provide closed-loop control.

- When moving in soil a *velocity controller* is required. The commands from the activities manager instruct the tip of the bucket teeth to comply with a particular velocity vector (i.e. speed and direction). This strategy accepts the fact that movement in the ground needs to be highly adaptive as ground conditions change, and that there is little likelihood of reaching a specific point via a predetermined path. Error feedback is used by the activities manager to modify the velocity command in order to optimise performance. Thus if the excavator cannot achieve the demanded velocity because the ground is too hard, the activities manager will direct the low-level controller to attempt a shallower dig where the ground is expected to be softer. This approach has proved very effective in providing pseudo-force feedback without the need of additional force sensors.

The low-level arm controller is currently implemented in 'C'.

Development prototype of the activities manager

Of the above high-level modules, it is most difficult to provide an early detailed requirements specification for the *activities manager*. The activities manager is the module that directs the digging process and has knowledge of the tactics required for efficient operation. The knowledge required by the developers was obtained by observing human operators, theoretical studies and trial and error experimentation. This section describes how the module was tackled at the development prototype stage (step 6).

Because the requirements for the activities manager are not completely defined it is necessary to prototype the control software. The prototyping process helps to identify and clarify the requirements specification of the control software.

To help the prototyping of the activities manager a design platform concept was used. The design platform allows the developers to try out and modify ideas, as well as reacting swiftly to requirements changes in other system components.

Design platform concept

The aim of the design platform is to provide maximum flexibility without compromising on maintainability. Maintainability is essential not only because of the potentially fast and possibly radical prototyping process, but also because of the unstable nature of developing the system using students. The purpose is to produce a detailed and static specification of the activities manager module. This specification is then used to produce an optimised and well engineered software solution.

In order to construct a design platform, it is necessary to have at least a basic understanding of the robotic system and the high-level goals of the control software. Most useful intelligent robots will be *finite state machines*. These are systems which are in one or other particular state of activity depending on the stimuli received. These stimuli can be as a result of signals from sensors, timers, switches or work instructions from a higher level program. The stimuli trigger the switch from one state to another. Fig. 4 shows a state transition diagram for 'digging within reach'. The words inside the boxes describe particular states and the words in italics outside the boxes indicate the stimuli that triggers the transition from one state to another.

The digger can only be in one state at any one time and the process is performed by moving from a completed state to the next state. If implemented with care, the finite state machine approach provides both flexibility and maintainability. Flexibility comes from the ease of adding new states to the finite state machine. The maintainability comes from the inherent modular structure of the finite state machine. Running one state at a time improves the performance of the whole system.

An interesting question arises as to 'where is the intelligence in such a system?'. Briefly, it lies in the algorithms that control the particular states, the sequence of states and the rules that control the transition between the states. This combination enables the system to adopt appropriate behaviour in an unstructured environment.

The next step is to identify key design features, around which to build the design platform. In the case of the activities manager these were: a real-time component, a non-real-time component, a means of applying knowledge and a control structure to model the phases of the digging process. These design features have to be merged with flexibility and maintainability characteristics to produce a design platform.

Features of the activities manager's design platform

The use of a high-level programming language is important, and the language used to implement the activities manager is Ada. This provides a modular approach, information hiding and meaningful data structures, which are all necessary for a maintainable system.

The activities manager involves a real-time component which was one of the critical design issues. A trade off exists between on the one hand maintainability and flexibility, and on the other hand real-time performance. The strict real-time option in Ada is called *tasking*, but this was rejected primarily because its complexity makes it unsuitable for a flexible, rapid prototyping process. Instead, simplicity, efficiency and a fast processor were used to provide an 'as fast as possible' solution. This solution does not compromise the high flexibility and maintainability characteristics of the system.

The next problem was how to embed knowledge in the activities manager. A well known technique is to use a

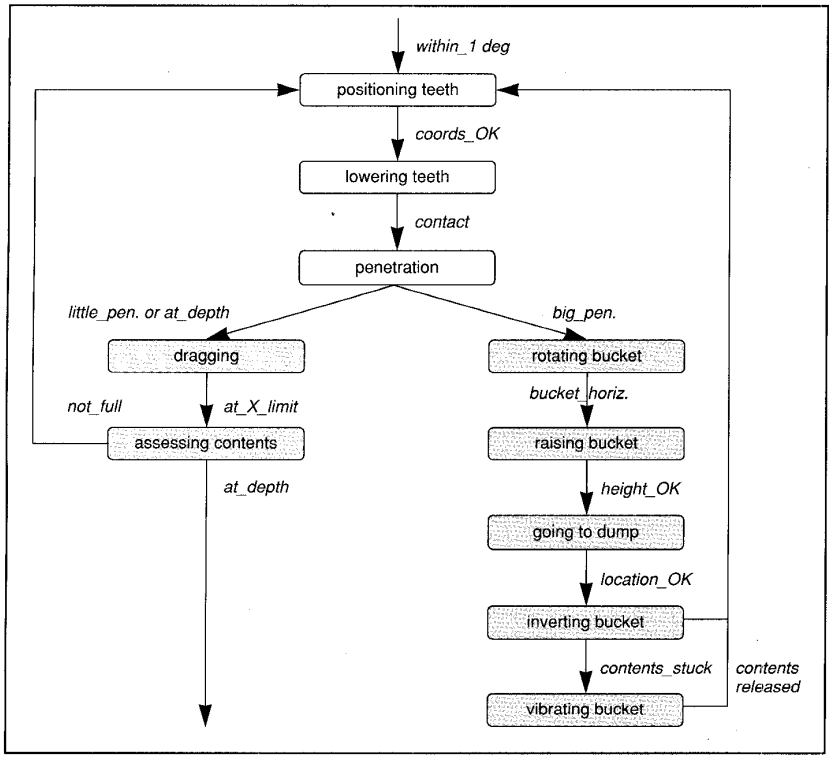

Fig. 4 State transition diagram for 'digging within reach'

production system. This consists of three components:

- A list of rules of the form: IF (condition true) THEN (perform action). Currently about 30 rules are used for a single digging cycle.
- A working memory containing the current value of all system data. Much of this data emanates from the low-level arm controller and can be considered to be displayed on a *blackboard* for use by any of the programme modules. The use of a blackboard is a well known technique for communicating between asynchronous processes which may be running on different processors and perhaps written in different computer languages. A particular section of memory can be allocated to hold the blackboard data.
- An inference engine which cycles through the rules, checks the conditions and instigates actions.

With complex systems this can result in having to check hundreds of rules at each cycle, and this can be a penalty in real-time systems. The solution was to place a separate production system in each of the states. In order to maintain performance, the production systems were kept small and efficient. If a production system became too big, the state was split into two or more substates. Each state is implemented as an Ada procedure with a 'while' loop forming the basis of the inference engine. When a transition state is reached the next desired state is transmitted to the working memory. If the most important and most likely rules are given the opportunity to fire first, then each iteration of the production system rules is performance optimised. The addition, modification and deletion of the rule is simple and therefore provides a flexible and maintainable knowledge base.

The production systems need data in order to work. The data structures required must be flexible and meaningful, in order to be maintainable. The data structures provided by high-level languages such as Ada are ideal for this purpose, as they provide flexibility, information hiding, logical structuring and meaningful naming. Two examples of abstract data types used are firstly that which contains the current state information as displayed on the blackboard:

```
type StateTableType is
   record
      TiltAngle:integer:=0;
      BucketAngle:integer:=0;
      xError:integer:=0;
      yError:integer:=0;
      VirtualPosition:Location:=(0,0,0);
      RealPosition:Location:=(0,0,0);
   end record;
```

Secondly the following data structure contains the commands which are issued to the low level controller:

```
type CommandType is
  record
      Velocity:Vector:=(0,0);
      Position:Location:=(0,0,0);
      BucketAngle:integer:=0;
  end record;
```

Another feature of the design platform was the removal of all constants from the source code. Constants are inflexible and were therefore replaced by variables which can be changed, as required by the prototyping process, without having to recompile the source code. The 'variable constants' can be changed by using a user interface, to edit external files. These configuration files contain the values that the 'variable constants' should have. They are read by the activities manager and placed into internal data structures. This method provides a certain amount of flexibility without having to recompile the source code, which is a great advantage in a rapid prototyping process.

The design platform can also be used as a test harness by simulating the input and analysing the output. This is again achieved by reading input in from external files and writing output out to external files. This enabled the activities manager to be tested without the digger and also before the digger was finished.

The safety manager

The concept of a safety manager is currently being investigated under the Safe-SAM project which is supported by the DTI/EPSRC safety-critical systems programme.[6] Mainstream thinking in safety-related systems is to develop a safety case based on making systems as deterministic as possible. Rigorous testing, static and dynamic analysis and/or formal methods can be used in an attempt to verify that a system performs safely to specification. For the robot excavator, which uses a knowledge base to produce adaptive control in an unstructured environment, such an approach is not realistic and could not be justified on economic grounds. For this reason the concept of a safety manager is being developed. The safety manager's role is to be aware of objects in the working environment, and to sanction only safe behaviour. It will be resident on a separate processor and, in biological terms, play the role of a 'conscience' to the system. Another analogy is a nuclear protection system, which is not concerned with the functional process of the plant, but has the power to intervene to prevent hazards arising. The safety manager is being implemented in Pascal using a validated compiler.

Brief comments on hardware

Although not the prime purpose of this article, a few comments on the hardware environment may be of interest. The initial rapid prototype used a single powerful RTX2000 processor to run both the activities manager and the low-level controller. Both programs were written in the language FORTH for which this processor is optimised. Off-the-shelf eurocards communicating via an STE bus were used for the processor and supporting sensor and electro-hydraulic valve interface cards. These were rack mounted on the roof of the excavator, as shown in Fig. 1. Performance was satisfactory but the system was bulky, relatively expensive and lacked robustness in a high-vibration environment.

For the later development prototype a more conventional route is being followed with the use of multiple 486 PCs in the ultra-compact PC104 format. So far, one PC is being used for navigation and track control, one for the activities manager and low-level controller and one for the safety manager. Communications between the cards is via CAN (controller area network) bus. It is the long-term aim to provide all on-board sensors with intelligent CAN bus interfaces which will significantly reduce wiring.

Summary and conclusions

Software development for intelligent robots is difficult because it is usually not possible to define the software requirements in sufficient detail at the start of the project. Prototyping provides a means of refining requirements, but this requires a hardware as well as a software prototype. Scale models are useful, particularly with large robots, and they should be scaleable and capable of being driven by the same software as the full-sized robot.

The development process for the generic class of intelligent robots can be described in a series of seven steps, which may be reduced to six for simple or one-off systems.

The design platform concept provides the flexibility for those modules that need to be changed at the prototype stage. This should be written in a high-level language such as Ada, so that it can be easily changed and maintained. The use of well known techniques such as finite state modelling, production systems and blackboard architectures can be combined to form an efficient and coherent intelligent knowledge-based controller.

References

1 SEWARD, D. W.: 'LUCIE—the autonomous robot excavator', *Industrial Robot*, 1992, **19**, (1) (MCB University Press)
2 BOEHM, B. W.: 'A spiral model of software development and enhancement', *IEEE Computer*, 1988, **20**, (9), pp.43–58
3 GREEN, P., SEWARD, D. W., and BRADLEY, D. A.: 'Knowledge acquisition for a robot excavator', 7th Int. Symp. on Robotics in Construction, Bristol, 1990, pp.351–357
4 DeMARCO, T.: 'Structured analysis and system specification' (Yourdon Press, New York, 1978)
5 YOURDON, E., and CONSTANTINE, L. L.: 'Structured design' (Prentice-Hall, 1979)
6 'Safe system architectures for large mobile robots', EPSRC, ref. GR/J18064

Derek Seward is with the Department of Engineering and Alastair Garman is with the Department of Computing, Lancaster University, Lancaster LA1 4YR, UK