A MODEL FOR THE EXECUTION OF TASK LEVEL SPECIFICATIONS FOR INTELLIGENT AND FLEXIBLE MANUFACTURING SYSTEMS

Peter Loborg, Per Holmbom*, Martin Sköld and Anders Törne

Department of Computer and Information Science
*Department of Physics and Measurement Technology
Linköping University, Sweden

ABSTRACT

We introduce here a software architecture for the control of a sensor based manufacturing system consisting of a number of machines and peripheral equipment. The architecture divides the programming effort into two levels, task level programming and control level programming. The task level programming is based on the programming of a discrete model of the world, the World Model (WM). The WM provides a symbolic representation of the world state and isolates the task programs from the control level algorithms. Programming the control level amounts to modelling the manufacturing equipment as components with 'behaviour' using object oriented techniques. Each component specifies how it should react to changes in the WM, i.e. selection and specification of the control algorithms to be executed. Programming at both levels can be done incrementally and control algorithms may be changed dynamically in the real-time kernel.

INTRODUCTION

This report presents a model for the execution of task level specifications in a manufacturing system. The presented work is carried out within the ARAMIS-project[1] and is a part of a cooperation project between the Dept. of Computer Science (IDA) and the Dept. of Physics and Measurement Technology (IFM) at Linköping University.

The motivation behind this project has been to provide control engineers with adequate tools to design and specify intelligent behaviours, i.e. behaviours which from the engineers point of view are robust to changes in the environment and performs a wanted high level function without additional detailed low level control programming. By control engineers we mean any engineer or operator who is specifying or diagnosing the behaviour of a physical system, with one or several embedded computers/control systems. More specifically, we focus our work on manufacturing systems consisting of a number of machines and peripheral equipment. This kind of environment is characterized by high demands on the precision and accuracy of movements, complicated and interdependent sequential actions and different user categories performing different tasks with respect to the equipment. The users must therefore be provided with means to specify accurate control algorithms as well as task level programs. These programming tasks are also normally performed by different user categories.

The function of an embedded control system is to provide the engineer with means to execute actions in the environment - automatically by preprogramming or manually by interactive command. We are using 'programming' to denote not just the actual programming of the computer, but also the specification of work descriptions and operation lists. The engineer specifies in the program a "sequence" of actions to perform. This program is very often cyclic, e.g. a feedback algorithm for fast motoric actions or a manufacturing cycle in a machine cell. The former is an example of an action which traditionally is specified at a low level, close to the controlled process and involving explicit reference to I/O on the embedded computer. The latter, on the other hand, is an example of a program which is originally specified at a higher level involving basic operations of larger time granularity. More importantly, the latter type of program is specified without reference to the embedded control systems, e.g. as operations lists or work descriptions. The translation of such specifications to executable programs in the embedded comput-

1. The ARAMIS-project at CAELAB, Department of Computer Science, Linköping University, Sweden is the continuation of a joint research project between the Department of Computer Science, ABB Corporate Research and ABB Robotics in Västerås, Sweden, during the years 1985 -1988.

ers is today done manually as a specific programming task.

Our aim is to equip the embedded computer system with an ability to execute “task programs”, like operations lists and work descriptions, directly, i.e. behaving intelligently. The different user categories and abilities, the different primitive elements in the specification languages and the different types of abstract machines for task programs and control algorithms implies some sort of multi-level programming and executing environment. For the moment we are investigating the possibilities of just three levels, which should give enough evidence for the useability of the ideas. The levels are - the task level, the control level and the physical level. Three levels might also be a just compromise between functionality and complexity in the programming and run-time environments.

First we present a general view on sensing and sensor fusion which motivates the concept of sensor integration and the ARAMIS-model of the environment and task program execution. Next the ARAMIS-model is described in more detail. After this, a presentation is made about the specification and encapsulation of control algorithms, together with an example. Finally we discuss the current status of the project and some future work.

We will not discuss the task level language and the higher level functionality in this paper. The general ideas regarding the task level are presented in [14] and the graphical task level language for specifying intelligent behaviours is presented in [7].

SENSOR INTEGRATION AND PROGRAMMING IN PHYSICAL ENVIRONMENTS

Sensor fusion is characterized by Clark and Yuille [2] as concerned with methods for combining raw sensory data to obtain information about the world. They classify the methods into weakly and strongly coupled, where the weakly coupled assume independence of data sources. Dependence means that the algorithm and its validity constraints for calculating one world (fused sensor-) data is dependent on the output of another, separately calculated, world data, e.g. feature matching stereo algorithms which are dependent on the depth information calculated from other sources.

Our concept of sensor integration extends this dependence to also comprise the dependence of algorithms on implicit knowledge about the world state and not only on measurable, sensed knowledge. This implicit knowledge is derived from the knowledge of the executing context of the actions performed in the environment.

Example 1:

Consider a program which can switch the light ON and OFF in the room. No sensory device exists for light detection and the operation of the switch is failproof (within the validity of the system specification). If there is a position determination task in the program, it might have different fusion algorithms for darkness and light. This information is implicit in the sense that it is determined by the last switching command and is non-local to the position sensing device. The program using the ON/OFF information is independent of whether this information is explicitly measured or implicitly calculated from execution context information.

We stress the point that the programming environment should support a unified view on explicitly sensed and implicitly known information in the programming of the (intelligent) behaviour of the complete system. To clarify this point, consider the example again. An operator or an engineer which is interested in using the position function in the description of the behaviour, would like the program to be independent of whether there exists a sensor for light or of the type of sensor used. This abstraction barrier is realized by the concept of world model programming explained below. We are convinced that such abstraction barriers are desirable in industrially used control equipment, where several different user categories are cooperating in the supervision and programming.

Our model has similarities with current implementations of different integrated environments for programming (“controlling”) computers where the machine is an abstract computing device designed for the portability of programs. However, our case differs in some respects [15]. The hardware (i.e. the machine environment) is not static even for a single ‘executing’ environment. The machinery and sensory equipment may change over time or differ slightly between sites, although the functionality for the end user is identical. A changing abstract machine means that the engineers must change the behavioural, control level specification to reflect the new (or different) environment. The task for the machine is constant, however, and should be portable to the new executing machine. Therefore a good programming environment must be provided not only for the task level programming, but also for the control level programming.

THE ARAMIS MODEL

Layered architectures and ARAMIS

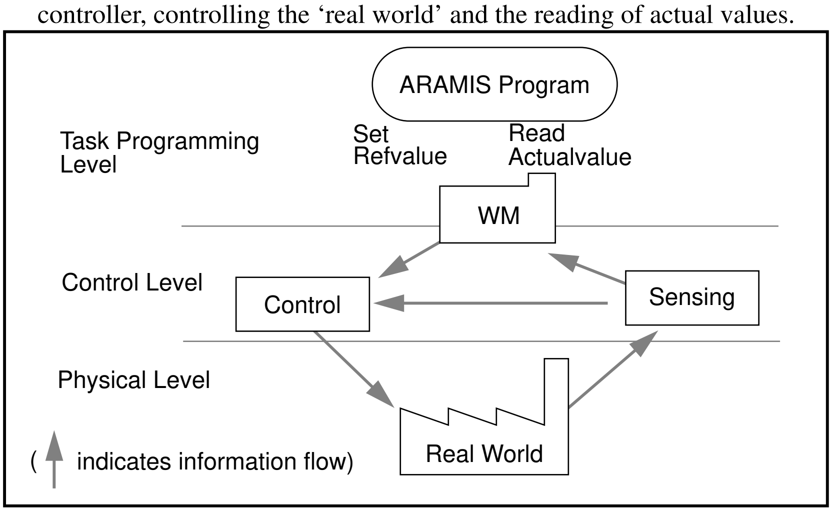
The system has been designed with a layered architecture, based on different levels of abstraction, it consists of three different levels; the task programming level, the control level and the physical level (Figure 1). At the task level the operator specifies what operations should be performed in the physical environment and under what conditions, using a graphical hybrid, rule based language [7]. The task program executes by setting reference values for the objects in the world model. The control level is responsible for keeping the real world in a state represented in the model of the world (WM), i.e. a servomechanism, as the WM is changed by task program execution. The programming at this level is typically done by control engineers. The physical level is the actual connection to the real world, where explicit I/O is performed with sensors and actuators.

Other principles for layered architectures in this area have been proposed, e.g. Brooks [1] defined "behavioural decomposition" as a design criterion for a layered architecture for sensor/actuator control systems, to make them robust and flexible. The different levels then reflects different levels of "competence". Our approach differs from this in that our "modules", the WM-objects (explained below), are not ordered in a simple subsumption hierarchy. Instead each module or object has a responsibility for controlling the behaviour of a part of the system. The interaction between the objects is orchestrated by the task level program. Each goal (following Brooks) is represented as a "worker" (or process) on the task level.

In [4] a three-layer architecture consisting of an analysis layer, a rule layer and a process layer, is presented. In the analysis layer, requested goals or tasks, expressed in temporal logic, are translated into plans for the rule layer to execute. The rule layer transforms the plans into sets of rules which are passed to the process layer. The process layer uses the rules to control the execution of the actions in the plan. Our task level corresponds to the rule layer, but also incorporates the ability for context dependent plan selection. We furthermore model the world as a set of objects which introduces structure into the (process-) state representation.

CHIMERA II [13] is a programming environment and operating system designed to reduce the development time for sensor-based control applications. This design is influenced by the NASREM Model for Telerobot Control System Architecture. Chimera II is a RT-OS which supports multiprocessor environments, but does not provide any specific modelling tools other than light-weight tasks, locally shared memory, local semaphores etc. Similar approaches but incorporating object oriented paradigms are presented in [10] and [11]. Our work does not focus on the OS support, but rather on the specification tools. The approaches above normally

Figure 1. The abstract machine/world model viewed as a reference value that is set, used by a servo controller, controlling the 'real world' and the reading of actual values.



assume application programming in C or C++, which correspond to the control level in our model. Since our work started with investigating the desirable properties of the task level language, the stress on full object-orientation in the control level has been decreased.

The task level programming problem is thoroughly investigated, e.g. [3][6][9][12]. In our opinion these approaches handle the problem as a conventional computer programming issue. Our model takes into consideration that the executing machine is not discrete and is changing (sometimes during execution) and has a partially unknown state. Even if theoretical work has investigated these issues, the combination of control algorithms and symbolic plans is fairly uninvestigated experimentally.

Generally, we aim at providing "adequate tools to design and specify intelligent behaviours" for a sensor-based manufacturing system. The assumption is that this kind of processes are well known by the operators and that the process should be deterministic from a global perspective. Therefore we do not need, or even wish, "exploring" or other non-deterministic behaviour embedded in the system, as in [1]. Another issue is that we aim at different user categories which includes non-programmers for the application programming. This means, for example, that a rule layer as in [4], which does not give a good overview of the task plans, is unsuitable. As plans somehow have to be specified for the system, we prefer a graphical language for representing them.

The world model

The task program operates, for an external observer, by executing actions in the physical environment. The state of the environment is represented in a world model (WM). Each primitive action corresponds to a change of model state. Each machine or part is viewed abstractly as a modified deterministic finite state automata (DFA), and is referred to as a world object (WO). The WO state is from the task program view a mapping from WO state variable names to other WO's or to simple data types like integers or symbols (an ordered sequence of symbols is called a symbolic state vector).

Each WO state is identified by the set of {state-variable,value} pairs of the WO. At any time, the actual WO state might be fully or partially known. Partially means that the value of some variables are unknown and impossible to determine by sensory action. A set of WO's correspond to the abstract machine mentioned in the beginning and a primitive operation by the task program corresponds to changing one (or several, simultaneously) of the state variables of one WO, called a WO transition.

The state of a WO is considered constant unless a transition is taking place. A transition occurs in finite, non zero time. During the transition the value of the state variables which are requested to change are unknown, (in fact known to be changing from an earlier to a later value, unless the variable is explicitly characterized as a sensed variable, see below). Thus, the time-line for the actual WO state consists of intervals of fully determined constant state, interspersed with time intervals of partially known states during transitions.

Modelling of objects

In the control level there is a numerical model of object behaviour, the control algorithm, which calculates output signals from given input. This numerical model is constant as long as the WO state of the object is constant. If the WO state changes then the model might or might not change. Let this be an injective mapping from WO state to a numerical model. This means that the set of state variable pairs for the WO denotes a numerical model in the control level and that the model is constant between transitions. Some WO-states have identical control models but with different parameters. Other WO-states differ also in model structure and may therefore be thought of as different modes in the object control model.

Each state variable is represented by two values, one called the reference value and the other called the actual value. Each time the task program orders a WO-transition, the reference value(s) is(are) changed and a request is made to the control level to change the state of the environment to the corresponding state. When this change has been performed, it is acknowledged by the control level.

If the value of a state variable is requested by the task program (possibly in another concurrent sequence of events) the actual value is returned. Normally this value is simply equal to the reference value or unknown. The normal action in this case would be to wait for the completion of the WO-transition, but other options may be possible[2]. However, if the variable is specified at the control level as a sensed variable, it can be calculated (from real sensor data) at any time, even during transitions ordered by the task program. For the task program there is no semantic difference between reading a sensed or an unsensed state variable. This effectively makes it possible to isolate the task program from the configuration, e.g. to selectively simulate sensors or to change the sensor configuration. The world model is discussed more fully in [8].

<u>The control level</u>

The control level must perform actions on the real world so that the difference between the reference value and the actual value of state variables in the world model disappears. This can be done in several ways - a single serial message output on a communication link, setting bits on output ports or by executing a control algorithm which reads sensor data and outputs control signals in a cycle. Such an execution (a transition, servoing algorithm) will bring the real object into the state desired by the task program. On the other hand, sometimes the new state is not an equilibrium state, thus there is a need to execute maintaining control algorithms (these would correspond to control modes for the object in question). The transition algorithms are parametrized by the start and end value of the state variable(s) in the transition and the maintaining algorithms are, of course, parametrized by the end value of the last transition.

2. The resource allocation problem has not been addressed so far, and must presently be taken care of explicitly by the user/programmer.

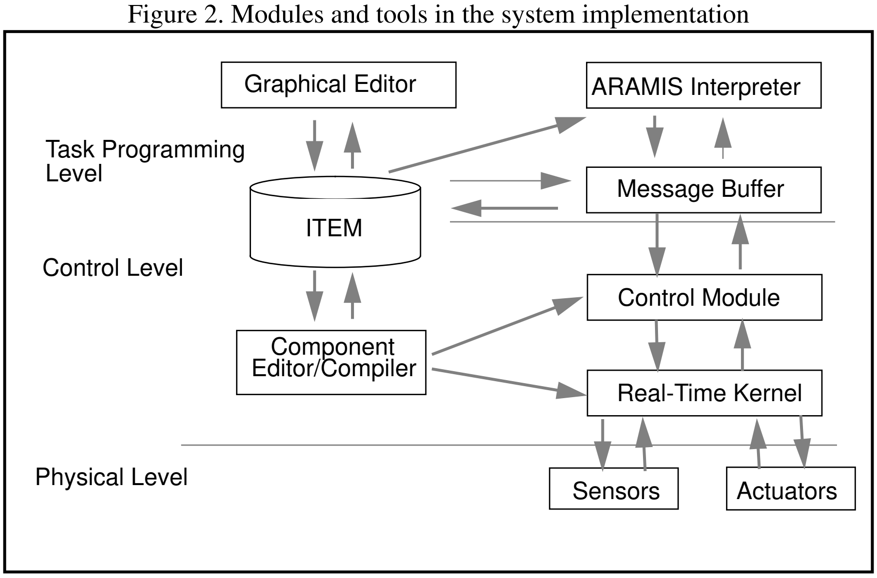
The control level must also transfer sensed information back to the world model. This is, however, only necessary in those cases where the intention is for the task program to be able to specify a feedback loop. For mixed sensors/actuators, i.e. a sensor which has some controllable feature (like the direction of a supervision video camera), there must never be any possibility for interaction between feedback loops defined in the task program and the transition and maintaining algorithms defined for the corresponding WO. This cannot be guaranteed in the present model, but is the responsibility of the programmer of the control level.

THE IMPLEMENTATION OF THE CONTROL LEVEL

The ARAMIS system consists of three levels, the task programming level, the control level and the physical level (Figure 2). In the task programming level the system operator can write task programs. The ARAMIS programs are specified in a graphical editor and are stored in a memory resident database, ITEM. When modelling objects of the manufacturing system in the control level, which we hereafter call *active components*, an object oriented view is used. Component code (control algorithms) is specified in a special programming language that is compiled and downloaded to a real-time kernel. Component behaviour is specified by DFA[3]-diagrams, for each component, that specifies the

3. Deterministic Finite Automation

Figure 2. Modules and tools in the system implementation

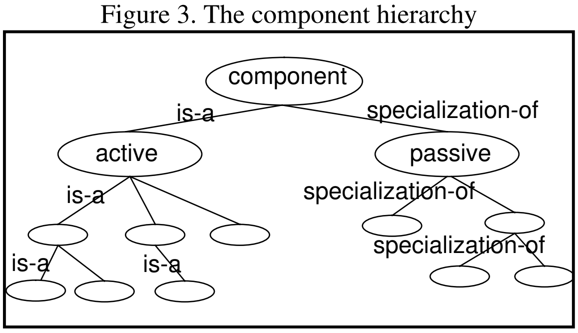


algorithms to execute in case of a change in the WM. Component behaviour and component code are specified in an integrated component editor/compiler tool. The real-time kernel is implemented in a multi-tasking Forth system executing on a separate real-time processor which directly communicates with the actual sensors and actuators.

Modelling the world as components

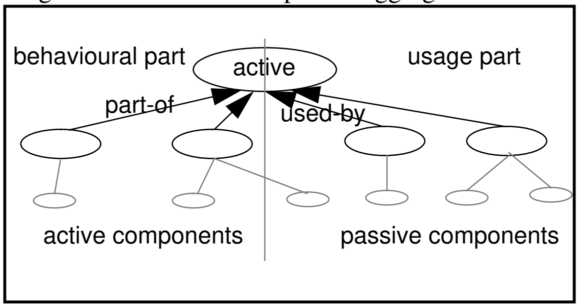
Components are divided into two main hierarchies, active and passive components (Figure 3). An active component is a model of a physical object in the world (e.g. a sensor or an actuator) or a virtual object (e.g. an abstract sensor using sensor fusion of several physical sensors). Active components have a local state associated with the control algorithms and are represented as concurrent processes. The hierarchy of active components specifies the levels of parentage or kinship (e.g. an asynchronous- and a synchronous motor can both be seen as instances of an electrical motor). The inheritance from a parent component includes both the static code and the dynamic behaviour of the parent. All objects in the WM are modelled as active components, but not all active components are WM-objects.

Figure 3. The component hierarchy



Active components (Figure 4) may be structured to consist of other active components to model complex behaviour and may use passive components to structure and share code with other components. Active components which are used by others are not possible to reference from the task level. Active components which are not used by other components are called top components and are possible to reference from the task program.

Figure 4. The active component aggregation model



A passive component is a package (in a software module sense) which contains related code (e.g. a package containing mathematical functions). Passive components have no local state and are represented as a collection of data and code. The hierarchy of passive components specifies the levels of abstraction (e.g. packages for path-following algorithms with and without sensory feedback can both be seen as specializations of an abstract path following package). By having the same external interface in an abstract passive component, no changes have to be made in an active component that uses the passive component, when one algorithm is exchanged with another (procedural abstraction).

Calls to active components are implemented as implicit message passing (i.e. syntactically as procedure calls, but semantically as interprocess calls) and calls to passive components are implemented as regular procedure calls (both syntactically and semantically). Passive components can use other passive components. Passive components can not define or reference local state variables since this would violate the integrity (OO encapsulation) of active components, according to the ARAMIS model.

The component description language

The component description language consists of two parts, an algorithm specification and a behaviour specification. Algorithms are written as functions and procedures in a typed, imperative language. The language includes primitive types like booleans, integers and reals together with traditional logic and arithmetic operators. The language includes traditional control constructs such as *if-then-else* and *case* constructs and iterative constructs such as *for*, *repeat* and *while* loops, and recursion. The language also includes primitive I/O for communication with sensors and actuators.

The behaviour specification is a state transition graph (or a state machine), where the states denotes a set of legal valuesets of the WO state variables and the transitions represent legal changes of the WO state variables. The state transitions denotes an initial and a final state, an algorithm (in the algorithm specification) which will execute the state change and a time constraint specifying a possible time out for the state change. Maintenance algorithms can also be specified for each state. The state transition graph is made to cover all remaining (illegal) subsets of the value domain by adding one or several error states.

Component definitions are compiled and downloaded to the real-time kernel in two phases. In the first phase (the instantiation phase) the algorithm specification of the component is compiled to target code. In the second phase (the installation phase) the behaviour specification is used to install the active component as an executable unit, by creating a state table in the control module reacting to changes in the WM and a task in the real-time kernel which can execute the changes in the real world.

To be more specific, an instantiated active component consists of two main parts:

- A data base part, stored in a memory resident data base, consisting of one static and one dynamic part: The static part includes the complete description, i.e. both the algorithm specifications and the behaviour specifications. The dynamic part includes state variables, accessible from the task program through the memory resident data base, and a state table generated at the instantiation from the behaviour specification (top components only). State variables are represented as a tuple with an actual value and a reference value. When a state variable of an active component is changed by the task program, the reference value is set. The state table is then indexed by the current state of the component, returning a function that takes the reference state vector (the values of all the state variables of the component) and returns the next state, the transition algorithm to be called and the maintenance algorithm of the next state. When the state transition is acknowledged from the real-time kernel, i.e. when the transition algorithm has terminated successfully, the actual value of the state variable is updated. When the task program reads a state variable the new actual value is returned. The actual value might also be changed due to asynchronous changes in the real-time kernel induced by nondeterministic events in the environment. In this case the actual value is changed directly. The first class of state variables are called implicitly determined and the latter are explicitly determined (sensed).
- A real-time kernel part consisting of:
  One main process representing the active top component:
    - A communication channel.
    - A table of constants and state variables.
    - A table of algorithms.
    - A table of subcomponents represented as processes defined as above.

Installed components are registered in the control level and are activated in a system initialization phase. When an active component is declared to consist of other active components the subcomponents are also instantiated and installed before the main component. An active component class can be instantiated both as a stand-alone component and as being part of another component (i.e. as two different instances), but only the stand-alone component will have a dynamic part installed in the data base and thus be visible from the task level.

A COMPONENT EXAMPLE

We illustrate the modelling of component behaviour with an example - a conveyer belt. The model consists of an active component consisting of a motor, a speed sensor and the actual conveyer belt. The main component may be modelled as follows. The conveyer belt can be in three different states, halted, moving forward or moving backwards (Figure 5). The component has two WO state variables, direction and speed, which can be directly manipulated by the task level. The halted state is the initial state. The initial value of direction in the halted state is unknown. The algorithm change_direction will be called in the real-time kernel to change the direction. The algorithm change_speed is called to set the speed to either halted, low, medium or high and, if the algorithm succeeds, a state change will occur. In the states forward and backward the maintenance algorithm maintain_speed is executed continuously to make sure the speed is kept. When the speed is changed, the maintenance algorithm is interrupted and the algorithm change_speed is called. If the speed is changed to halted , the state changes to halted. If the speed is changed to some other value than halted , the current state will be kept and the maintenance algorithm will be called again after the speed change. If either a transition algorithm or a maintenance algorithm fails, the task level is notified by an error message. In case an inconsistent state occurs, i.e. if the state variables are set to values not consistent with any of the defined states, a transition to an error state will occur and the task level will be notified. In the example, this is the case if direction is unknown when setting speed to some other value than halted .

How the values of the WO state variables, seen by the task level, correspond to the values in the real-time kernel has to be defined in a mapping between the values of the variables in the WM and the "real" values of the variables in the real-time kernel. This mapping defines the accuracy by which the maintenance algorithms should maintain the variable values and it also defines the intervals by which asynchronous changes of values (from sensory data) should be reported to the WM.

CURRENT STATUS

Hardware and Software Platforms

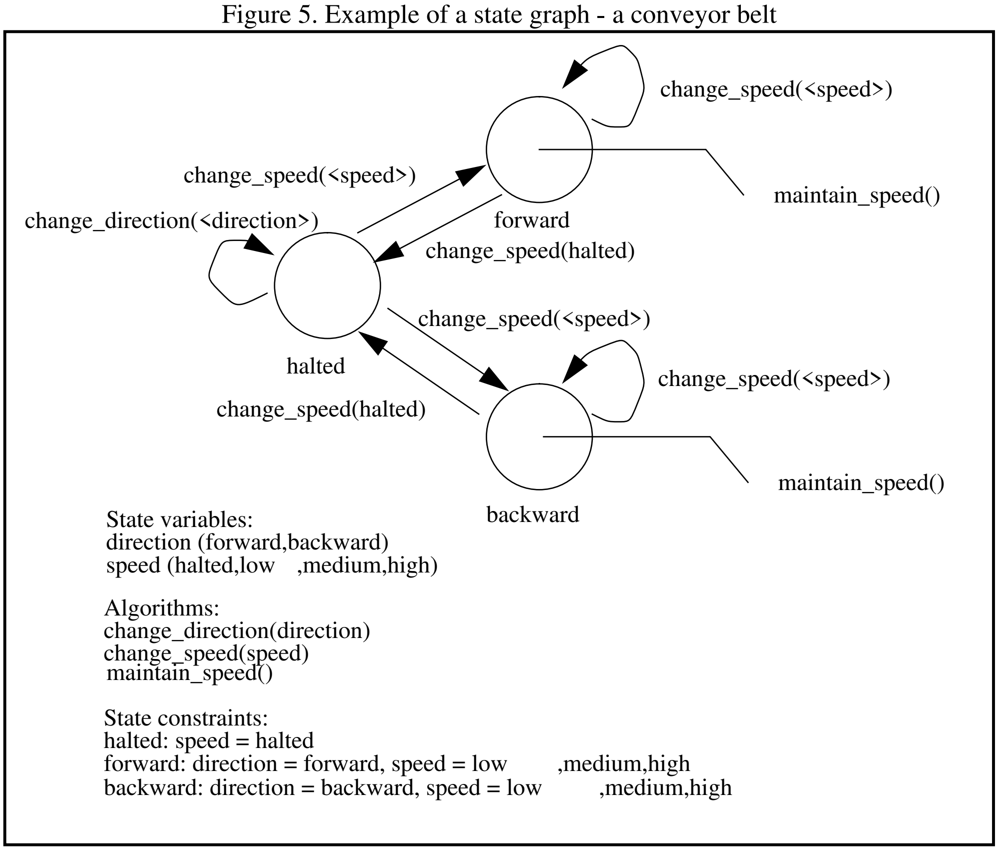
The hardware for the task level and the control level programming is a SUN SPARCstation 1. For the real-time environment a VME-bus based Motorola 68020 is used. The software for the SPARC is the XEROX EnVõs Common Lisp environment and for the Motorola, the TILE multi-tasking Forth environment[4].

The Real-Time Environment

The real-time environment consists presently of a robot with a three-finger gripper and some various sensors, controlled by the Motorola 68020. The robot is a conventional 6-axis PUMA 560. Some additions have been made to the control system to provide extended

4. M. Patel, Dept Computer Science, Linköping University.

Figure 5. Example of a state graph - a conveyor belt



external control. The gripper is specially designed to be more flexible and controllable than conventional grippers, and is also prepared for mounting sensors onto it. A CCD camera is used as vision sensor, giving 256x256 pixels resolution and placed as a scene camera. The vision system is situated on a separate computer, communicating with the VME computer. Tactile sensor arrays are mounted on the gripper's fingers, giving imprint patterns. An ultra-sonic sensor on the gripper can be used as range/proximity sensor. A strain-gauge sensor on the robot arm can be used as force sensor. A similar sensor is also intended to be mounted in the gripper, to be able to measure the grip force.

For accessing and controlling sensors and actuators from the Forth environment, some extensions have been made. These includes code for reading and writing from analogue and digital I/O boards and communicating with the robot control system and the vision computer. The communication uses a simple package based protocol that is embedded in the communication code, thus making it transparent to the user.

FUTURE WORK

Many aspects of the architecture described in this paper needs to be expanded further. Some of the urgent extensions includes fault handling and the scheduling of processes in the real-time kernel. Simple fault handling is supported at the present, but this does not include propagation and translation of error information between the two levels in the architecture. Error information at the control level needs to be translated using the context of the task being executed at the task level. This involves translating from how the error presents itself to what the logical cause of the error might be. By extending control level programming to include components as first-class objects, e.g. by moving closer to an object oriented programming language, it will be possible to implement generic control algorithms. Today every executable algorithm must be encapsulated within the corresponding component. Other interesting areas that need further work includes support for software requirement analysis to determine the consistency of a component behaviour specification and support for the distribution of data and programs on distributed hardware.

ACKNOWLEDGEMENTS

We would like to thank NUTEK, the Swedish National Board for Industrial and Technical Development, and CENIIT at Linköping University, for funding this project. We would also like to thank other members of the Measurement Technology group at IFM for providing us with a real-time environment and for teaching us more about sensor technology.

REFERENCES

[1] R.A. Brooks, "A Robust Layered Control System for a Mobile Robot", in *IEEE Journal of Robotics and Automation* vol. RA-2 no. 1, March 1986

[2] J.J. Clark and A. L. Yuille, "Data Fusion for Sensory Information Processing Systems", Kluwer Acad. Publ.

[3] G.C. Gini, M.L. Gini, "Dealing with World--Model-Based Programs", *ACM Transactions on Programming Languages and Systems*,7,2,(1985).

[4] J. Hultman, A. Nyberg, M. Svensson, "A Software Architecture for Autonomous Systems", *6th Int'l Symposium on Unmanned, Untethered Submersible Technology*, Elicott City, Maryland, 1989

[5] S. Levi and A.K. Agrawala, "Real-Time System Design", chap. 7, McGraw-Hill 1990

[6] L.I. Lieberman and M.A.Wesley, "AUTOPASS: an automatic programming system for computer controlled mechanical assembly", *IBM J. Research & Development*, 21,4,(1977).

[7] P.Loborg and A.Törne, "A Hybrid Language for the Control of Multimachine Environments", in *Proc. of 4th Int. Conf Ind. & Eng. Appl. of AI & Experts.*, Koloa, Hawaii, 1991, by University of Tennessee Space Institute.

[8] P. Loborg, M. Sköld, A. Törne, "A Hierarchical Software Architecture for Control of Industrial Robots and Manufacturing Equipment", presented at 1st National Symposium on Real-Time Systems, Proc. in Tech.Rep. no30, Dept of Computer Systems, Uppsala University, june 1991. CAELAB Memo 92-01.

[9] T.Lozano-Perez, P.H.Winston, "LAMA: a language for automatic mechanical assembly", *Proc. 5th IJCAI*, (1977).

[10] C.W. Mercer, H. Tokuda, “The ARTS Real-Time Object Model”, in *Proceedings of 11th IEEE Real-Time Systems Symposium*. December, 1990.

[11] D.J. Miller and R.C.Lennox, “An object-oriented Environment for Robot System Architectures”, in *Proc. IEEE Int. Conf. on Robotics and Automation in Ohio 1990*, vol. 1, p. 352 -.

[12] B.Shepherd, “Task-level programming of a robot using an intelligent human-robot interface”, *Proc. 2nd Int’l Conf. on IEA-AIE*, ACM, (1989).

[13] D. B. Stewart, D. E. Schmitz, P. K. Khosla, “Implementing Real-Time Robotic Systems Using CHIMERA II”, in *Proc. IEEE Int. Conf. on Robotics and Automation in Ohio 1990*, vol. 1, p. 598 -.

[14] A. Törne, “The Instruction and Control of Multi-Machine Environments”, in *Applications of Artificial Intelligence in Engineering V*, vol 2, p. 137, proc. of the 5th Int'l Conf in Boston july 90, Springer-Verlag.

[15] H. Van Dyke Parunak, J. Kindrick, B.W. Irish, “Viewing the factory as a Computer: A Cognitive Approach to Materials Handling”, in *Artificial Intelligence - Manufacturing Theory and Practice*, eds. S.T. Kumara, R. L. Kashyap, A.L. Soyster, Inst. Ind. Engineers, Industrial and Management Press, 1989, p225 - 264.