Towards Error Recovery in Sequential Control Applications

Peter Loborg **Anders Törne**

Dept. of Computer and Information Science, Linköping University, Sweden

ABSTRACT

In this paper we describe how to generate a large set of restart points for a sequential control program, thus enabeling fast restart of a plant after a fault and subsequent repair. This is work in progress, and will recapitulate the main idea as developped in the Aramis project, and discuss its applicability to the programming languages defined by IEC1131-3.

KEYWORDS: error recovery, sequential control, task-level programming

INTRODUCTION

The languages used to instruct the production equipment at a shop floor are primarily designed for describing the normal flow of activities — sequential control in combination with continuous control. The handling of an abnormal or unforeseen situation is generally not supported. If such situations must be dealt with, this must be explicitly encoded using the same language elements as for the normal control.

There exist several attempts to use different form of planning formalisms to synthesise the program, given a specification of goals or intended behaviour. In these formalisms, the problem of exception due to simple, manageable faults such as misaligned parts are handled either by generating special code for catching the problem and retry the operation [1], to plan for a set of actions to overcome the problem and then return to the original program [3][5], or to perform a replanning step and extend the original program with this new plan [11][12].

We are focusing on the situation when a piece of equipment is to be restarted after a failure and subsequent repair. The machinery is almost guaranteed to be in a state in the middle of its activities (it seldom breaks in its 'home state'), and the state of the controlling computer/PLC is often inconsistent with respect to the controlled machinery. The task is to resynchronise and restart with minimal loss (in time and material) from an almost arbitrary state. The resulting restart state may be a state in the same sequence of states as was passed when the error occurred, or it might be a state in an alternative sequence of states.

The tool we use for this is a world model, containing an abstract view of the state space and additional information about possible operations on each device of the equipment. At compile time, we analyse the program and construct a database with information of all possible restart points. The problem of exponential blowup of the state space is avoided by exploiting the parallelism inherent in the application and by using the abstraction of the world model. If the machinery to restart is not in a precomputed restart point, a graph search can be adopted to find such a point 'close' to the current state.

6:th int. Symp. on Robotics in Manufacturing
(ISRAM96), Montpellier, France, May 27-30 1996

As a byproduct, the analysis phase will also reveal programming errors such as an attempt to apply an operation or start an activity in a state where it is not applicable, and if timing information for all primitive operations is provided, it is also possible to obtain cycle time estimates at compile time, as well as to reschedule the default behaviour of the system to reduce the cycle time.

THE PROBLEM OF RESYNCHRONISATION

To resynchronise the controller with the plant, we need to have more knowledge about the plant available in the controller than what is customary. The task of resynchronisation can informally be described as *given a current observable state of the plant, find a controller state such that if reached during normal execution it would have produced the observable plant state*.

By *observed plant state* we mean a set of values for all sensors and actuators of the plant (not the actual plant state). By *controller state* we mean an observed plant state *and* the internal description of the program counter, the stack and all argument bindings of the controller, at some suitable level of abstraction.

Using this definition, it is clear that there can be several controller states matching an observed plant state, since a program can pass the same observed plant state multiple times but with a different trace of observable plant states each time, or since there can be several programs passing this observed plant state. It is also clear that we prior to the execution need a translation table that given an observed plant state returns a set of possible controller states. If such information is available, the task of resynchronisation is reduced to the task of picking the proper controller state and continue execution. In current practice, this information is generally not available, although a small subset of the information can be hand coded, i.e. implementing a set of restart points in the program. Although it is not realistic or even possible to hand code all possible restart points, it is possible to generate them as a part of the compilation of a program — provided there exists a more detailed description of the data structures and devices used in the program as compared to the current practice.

EXTENDED DESCRIPTION OF A DEVICE

In ordinary computers such as IBM-PC's, Unix machines etc., a device interface consists of a set of functions and procedures (interface routines) designed to control the device. For each device there is also a set of rules describing the intended usage of the interface routines, often presented to the programmer as a manual for the device in question.

As an example, consider the device *read-only file*. This device is controlled using three functions and a procedure:

- *open(filename)* will try to open a file identified by *filename*, and if it succeeds it will return a reference to the open file, a descriptor *file*
- *read(file)* will read and return the next data item of *file* if there is one
- *eof(file)* is a predicate function for testing if there exist more data items to read or not
- *close(file)* is a procedure that will close and forget the object referenced by descriptor *file*

The rules for when these interface routines are applicable can be described using a state transition graph (Fig. 1).

If this description is available to the compiler, the compiler can employ abstract interpretation to analyse the program, and it will be able to catch most of the errors resulting from

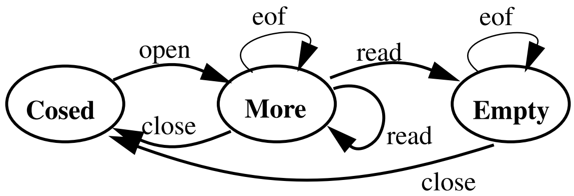

Figure 1. A state transition diagram showing the possible states for a read-only file and allowed operations in each state — normally not available to the compiler, only to the programmer.

miss-usage of the device.

In a programmable controller (for controlling a plant) implementing the standard IEC 1131 [4], the information available about each device or component in the plant is represented by a set of variables of a known, finite domain. Some of the variables represent sensors, and other variables represent actuators. The controller operates by reading input variables and changing output variables, and if the change is consistent with the intended usage of the device the execution will proceed. If not, the device will not respond as expected. This situation should not be misconceived with the situation when the change of values for the output variables is correct but the device fails to deliver the expected response due to other circumstances such as a broken actuator or sensor. In the latter case the problem can be foreseen, and an exception handler can be implemented to look for and react to such situations.

If all devices in the plant where modelled by both the variables and a state transition diagram showing the intended usage of the device (Fig. 2), the problem of misusage of the device can be caught by the compiler.

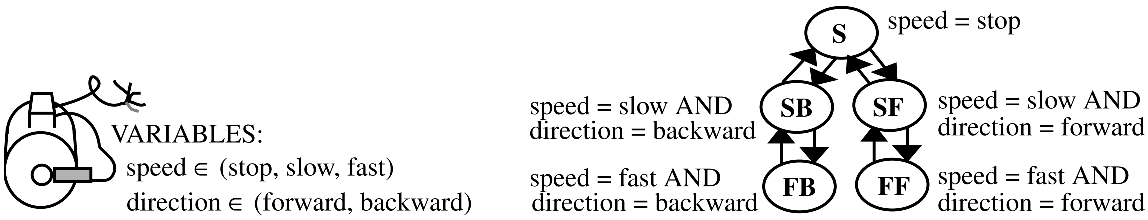

Figure 2. An electrical, bidirectional two speed engine is modeled by two variables, direction and speed (left hand part). The behaviour of this device is modeled by a state transition diagram with five abstract states, each representing a combination of variable values according to its constraint. The arcs between the states describe legal transitions, rouling out the posibility to reverse the engine without requesting it to stop in between.

The analysis technique to be used, abstract interpretation [2], operates by exploiting all possible execution paths of a program in parallel. All variables in the program are assumed to have any value consistent with their domain when the algorithm starts, and the set of possible values is reduced and modified during the analysis. The algorithm continues until it reaches a fix point, i.e. the set of possible values for the variables does not change any more. During the analysis the algorithm constructs a mapping between different (control) states of the program and possible values for the set of variables in the program.

Our goal is to construct a reversed mapping, i.e. for each state (as defined by the variables) a set of controller states are stored.

ARAMIS

The device model presented above has been developed in the scope of Aramis — A Robot And Manufacturing Instruction System [7][8][9][10][13]. In this system, the device interface is a part of a world model, which acts as an abstraction barrier between the task programming level on top of the world model and the control level below the world model. A program executed at the task level operates by reading and writing values from and into the variables defined by each device. The change of an input variable of a device is regarded as a request to the device, a request to change the internal state *iff* this change is consistent with the state transition diagram for the device. The device will then execute a control algorithm at the control level, an algorithm responsible for changing the plant into a state that is consistent with the requested state of the device model. When the control algorithm succeeds, the request is completed and the calling task level program returns from the call. Using this architecture we achieve the following characteristics and functionality:

- a well-defined abstraction of the actual state space at the task level, defined by the actuators and senors available in the plant
- a decoupling of task specification and device implementation
- a possibility to execute a task level program on a simulated system, i.e. to experiment with tasks and task level programs during the construction or reconstruction of the plant
- a guarantee against violations of the implemented abstraction layer during the life cycle of the plant
- reusable, hardware independent task level programs

We will return to these characteristics and compare them with a system based on the standard IEC1131 in the next section. In the remainder of this section we will describe the analysis algorithm in more detail.

Analysis of a task level program in Aramis

The algorithm for analysing a task level program is not a trivial algorithm, and in the general case using an arbitrary task level language it will either have an unacceptable complexity or will not compute at all. The following is a list of the prerequisites needed for the algorithm to be useful:

- the task level program must be compileable into a safe petri net, implying that it cannot include a goto statement or equivalent. If constructs such as procedures exist, they must also be compileable into safe petri nets.
- only top level programs or processes may have global references to devices and other data objects, procedures or other subprograms must only reference objects passed to them as parameters
- iterations in the application must be bounded and the bound must be known at compile time. This is normally not a problem in the manufacturing context, since the only unbounded iteration is the repeated production of a product, and we restrict the analysis to one such cycle.
- in the definition of the state transition diagram in a device, the set of nodes of that diagram (possibly extended with an implicit error node) must form an equivalence relation over the state space defined by the device variables. The rationale is that each possible combination of variable values in a device must match at most one node in the diagram.

Given these prerequisites, a task level program can be analysed. The algorithm consists of the following basic steps:

- Translate each named program entity, process or procedure, into a safe petri net. This translation can be done at definition time, i.e. each time a process or procedure is edited and saved. The symbol table generated for that entity should also be saved along with the petri net.
- For the petri net under analysis (PNuA): translate each condition in the petri net to the smallest set of abstract device states that will satisfy the condition. An abstract device state is a node in the state transition diagram of the device. This static translation will fail in cases where run time values are needed, in which case the translation is postponed until needed. However, all conditions consisting of a boolean combination of expressions on the form $v \rho c$ (where v is a variable, c is a constant in the domain of v and ρ is a relation defined over that domain) can be translated initially.
- based on the symbol table information of the top level process, all devices and other objects referenced are known. Form an *initial state set* (ss_i) for the PNuA as a set of possible device states, where a possible device state in turn is a set of all abstract device states for that device. As an example, consider a process referencing a single electrical engine (fig X) and a single file object (fig Y), the initial state set will be the set: {{FB, SB, S, SF, FF}, {Closed, More, Empty}}
- using the initial state set, perform an abstract interpretation of the PNuA by incrementally generating the graph of reachable and valid markings of the PNuA. A marking is valid *iff* all elements of the *current state set* (ss_c) for that marking are non-empty. For each transition with a condition, let the ss_c for the marking after that transition be the intersection of the previous ss_c and the state set representing the condition. For each place representing a state changing request for a device d into a new abstract state a, verify that the request is applicable, i.e. that the element d of the ss_c (ss_{cd}) is a subset of the set of abstract states in d from which a is reachable with one transition. If not, an error due to missusage of the device is found. Otherwise, the new ss_c is formed by replacing ss_{cd} with $\{a\}$.
- If a place in the PNuA represents a call to a sub-program, that place is replaced by the petri net representing that sub-program after substituting all local variables with conflicting names with a set of new unique variables. Thereafter the algorithm continues as described above.

As a result of this process, all reachable and valid markings are found, and for each such marking there is one or several state sets associated. Thus it is possible to construct a mapping from state sets to markings, to be used if there should be a need for resynchronisation of the controller and the plant.

The complexity of this algorithm is estimated as in the same size as the complexity of generating all reachable markings for a safe petri net. The fact that the conditions on transitions are used to restrict the size of the reachability graph is favourable. However, a marking can be reached more then once and with different current state sets, and thus a sub tree of the reachability graph can be searched multiple times. This indicates a slightly worse complexity. However, since the algorithm will be used at compile time, i.e. offline, the complexity is not a major issue.

THE LANGUAGES IN IEC 1131-3

In the international standard IEC1131-3 a set of five different languages for programming a controller is defined, namely: *instruction language* (IL), *ladder diagram* (LD), *function* *block diagram* (FBD), *sequential function chart* (SFC) and *structured text* (ST). Two of the languages are textual languages (IL and ST), and the other three are graphical languages.

Two of the more important improvements made by the standard is the introduction of a rigorous type system and the specification of how programs written in these languages can interact with each other. Any new device or program, regardless of implementation language, which can be provided in the controller using a function block as interface is an allowed extension of the controller.

Applicability of the analysis algorithm to IEC 1131-3 programs

Since all programs (POU — *program organisation unit* in the standard) can be viewed as either a mathematical, side-effect free function or as a function block, the following discussion will only cover these language elements.

In order for the analysis algorithm to operate there is a need for a layer of device descriptors, a layer that defines a border between a higher and a lower level of the system. Such a layer *can* be implemented using the FBD language, using a set of function blocks with the implementation part described as an SFC. However, the petri net notation is primarily a graph of activities, while a state transition diagram is primarily a graphical notation for a state changing system where the state is defined by the variables in the system*. There is no support for making it mandatory to use the intermediate device layer. Each programmer is free to call other parts in the lower layer directly from within the upper layer. This problem can, however, be handled by introducing these restrictions in the editors used for writing programs in the system.

Another requirement is that it should be possible to compile each program into a safe petri net. Although this is also the recommended practice when writing control programs [6], this is not the case for an arbitrary IEC 1131 program since a goto statement or equivalent can be expressed in all languages except FBD. Thus, this requirement imposes additional restrictions on the languages to be verified at compile time.

In Aramis, the possibility to select a suitable data abstraction is used extensively to reduce the state space. The new type system introduced in the standard provides the possibility for a similar data abstraction in the IEC1131 languages, but again, it is not mandatory to define an abstract version of a type used at the lower level of the system in order to use it in the upper level.

Concluding from the arguments above, it seems as if the analysis algorithm would apply given that a set of programming rules where strictly followed. There are, however, more details in the IEC 1131 languages that must be investigated in order to relax the restrictions imposed as much as possible, and thereby retaining a flexible programming system.

CONCLUSIONS

We have presented a method for generating all possible restart points of a program, given an extended representation of the devices used in the program. The essential properties of this extended representation and the programming language used has been discussed, and the applicability of this method on a program written in one of the IEC 1131-3 languages has been investigated. We have found that this method is applicable, given that a set of

* In the context of languages, by the set of strings consumed or generated up to the current node.

programming rules are followed. This is a preliminary result, and the details are yet to be investigated.

REFERENCES

[1] Cao, T. and Sanderson, A.C. “Sensor-based Error Recovery for Robotic Task Sequences Using Fuzzy Petri Nets”. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 1063-9, 1992.

[2] Cousot, P and Cousot, R. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints* Conf. Record of Fourth ACM Symposium on POPL, pp. 238-252, Los Angeles, 1977

[3] Gini, M. “Recovering from Failures: A New Chalenge for Industrial Robotics”. In *Proceedings of the 25’th IEEE Computer Society International Conference (COMPCON-83)*. pages 220-227, Arlington 1983.

[4] IEC 1131-3 International Standard, Programmable Controllers — part 3

[5] Lee, M. H., Barnes, D. P. and Hardy, N. W. “Knowledge Based Error Recovery in Industrial Robots”. *In Proceedings of the International Joint Conference on Artificial Intelligence.* pages 824-826, 1983

[6] Lewis, R.W. *Programming industrial control systes using IEC 1131-3*. IEE Control Engineering Series 50. The Institution of Electrical Engineers, London (1995)

[7] P. Loborg and A. Törne. “A Layered Architecture for Real-Time Applications”. *The 7:th Euromicro Workshop on Real-Time Systems*, Odense, Denamrk 1995

[8] P. Loborg. *Error Recovery Support in Manufacturing Control Systems*. Licentiate theses no. 440, Linköping University, Sweden, 1994.

[9] Loborg, P., Holmbom, P., Sköld, M. and Törne, A. “A Model for the Execution of Task Specifikations for Intelligent and Flexible Manufacturing Systems.” *Integrated Computer-Aided Engineering*, p.185-194, no. 1(3), 1994

[10] Loborg, P. and Törne, A. “A Hybrid Language for the Control of Multimachine Environments”. *Proceedings of EIA/AIE-91*, Hawaii, June 1991.

[11] Noreils, F. R. “Integrating error recovery in a mobile robot control system”. *Proceedings 1990 IEEE International Conference on Robotics and Automation*, pages 396-401 vol.1, 1990.

[12] Noreils, F.R. and Chatila, R. G. “Control of mobile robot actions”. *Proceedings of 1989 IEEE International Conference on Robotics and Automation*. pages 701-7 vol.2, 1989

[13] Törne, A. “The Instruction and Control of Multi-Machine Environments”. *Applications of Artificial Intelligence in Engineering V, vol. 2, pp. 137-152, proc. of the 5th Int. Conf.* in Boston July 90, Springer-Verlag, 1990.