From the 7:th Euromicro Conference on Real-Time Systems, Odense, Denmark, June 14-16, 1995.

A Layered Architecture for Real Time Applications

Peter Loborg (petlo@ida.liu.se), Anders Törne (andto@ida.liu.se)
Dept of Computer and Information Science
Linköping University, S-581 83 Sweden
Phone: +46 13 28{2494 | 2365}, Fax: +46 13 284020

Abstract

This paper argues that real-time control systems should be structured in abstraction layers in provide adequate instruction formalisms for both task specification and for low level control, and that the design information used when specifying the system should be included in the encoding of the abstraction barrier. When this design information is available, describing the semantics of the abstraction, the problems with interdependencies between real-time control and task level operation can be handled.

An example of a system encompassing this principle — Aramis — is presented. The paper discusses how this principle can be utilised to analyse the system and for error recovery support.

1 Introduction

Many of the commercially available real-time systems of today uses a singe instruction formalism to describe or encode all parts of the system, from low-level control loops, to processes and process interaction up to mission related tasks. During the specification of a system a formalism different from the implementation language is often used to describe various requirements like timing restrictions. When the system is encoded, high level properties are translated into constructs and algorithms described in a programming language, for example, fast response requirements are translated into priorities between processes. Much effort has been spent on developing packages and algorithms to support this encoding, e.g., advanced priority-based scheduling algorithms with predictable behaviour.

When the system is to be validated there exists basically two methods; intensive testing of the system vis-a-vis the requirements, or formally validating a model of the system. Both methods have their drawbacks. Although testing is a well established method to validate a program, it is also known that testing can only pinpoint bugs, not guarantee the absence of them. The method of formally validating a model of the system can guarantee the absence of implementation errors provided the model is correct.

Lately, the use of object oriented methods have increased, not only in design but also in the implementation of real-time systems. Although the object oriented paradigm may offer advantages compared with traditional imperative languages, there are drawbacks. Problems addressed ranges from aspects concerning dynamic binding of, e.g., overloaded operators, to how to provide a more complete interface specification of what objects actually do and with whom they interact.

An alternative approach to the one described above is to synthesise the system from a specification. Formalisms such as Estelle, Lotos and SDL are used by the telecom industry to specify switching applications. However, their usability in general, embedded real-time applications is limited. One of the reasons for this is that the mechanisms for abstraction corresponds to that of procedural abstraction, i.e., there is no abstraction level above in which a different instruction formalism more suitable for task instruction or mission control is to be used.

In a project called Aramis (A Robot and Manufacturing Instruction System) we address these problems and have proposed an architecture which uses a world model as an intermediate store and interface between the real-time control kernel and the task level of the system. Apart from storing an abstract view of the current state in the world model, we also retain a description of the semantics of the code executing in the kernel. This description enables a loose coupling between the task level and the kernel.

Using this semantic description and the timing characteristics of the kernel, it is possible to predict task level execution time. Furthermore, there is ongoing work indicating that error recovery — or rather — restart from an almost arbitrary state during task execution is achievable using this architecture.

In the following section we will present an overview of the Aramis architecture, concentrating on the control level and the interface to the task level. Thereafter we will discuss how to analyse a given implementation and how recovery from errors can be supported in this architecture.

2 Aramis — An Overview

The Aramis 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 (Fig.1). It uses a *world model* (WM) as interface between the task level and the control level.

At the task level the operator specifies what operations or actions should be performed upon the WM and under what conditions, using a graphical hybrid rule based language [7]. The physical environment is modelled as a set of objects in the WM, where each object has a set of state variables. The task program executes by setting reference values for the objects (their state variables), thereby requesting the objects to change their state. The control level acts as a servomechanism and is responsible for keeping the real world in a state represented in the WM, as the WM reference values are changed by task program execution.

The programming at the control level is typically done by control engineers, and consists mainly of specifying how devices in the real world should be viewed in the WM, and the control algorithms needed to implement the servo mechanism.

The physical level is the actual connection to the real world, where explicit I/O, requested by control level algorithms, is performed with sensors and actuators.

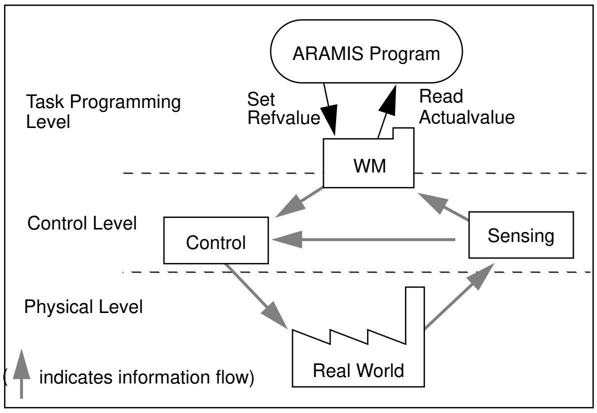

Figure 1: The world model is viewed as a set of reference values which is set by the Aramis program and used by a servo controller (sensing/control) to invoke correct control algorithms in the ‘real world’. The WM also support the reading of actual values.

2.1 World model based programming

In Aramis, we adopt an object centered view of the world, where each object is responsible for its own internal state and its consistency. The task level program is responsible for coordinating the operation of all objects involved.

Each object contain an internal *state*[1] and information about its proper usage. This will prevent the task level programmer from abusing the object unnoticed - an exception will occur. This principle is best explained through the following example, where a two speed bidirectional electrical engine is modelled (fig 2). This device is represented in the WM by an object containing two state variables, named `speed` and `direction`. The domain of `speed` is a set of the three symbols `stop`, `low` and `high`. The domain of `direction` consists of `forward` and `backward`. This results in an object with six possible states. We use a set of *constraints* to group the state space into a set of *interesting* states (e.g. when the value of `speed` equals `stop`, the value of `direction` is uninteresting, and thus five states are adequate to describe the engine). A *finite state machine* (FSM) defined over the set of interesting states is used to describe how these variable values may be combined and changed in this object. To each transition defined in the FSM there might be an associated control algorithm responsible for achieving the operation in the real world which corresponds to the state transition of the object. This functionality is part of the control level and will be detailed in next subsection.

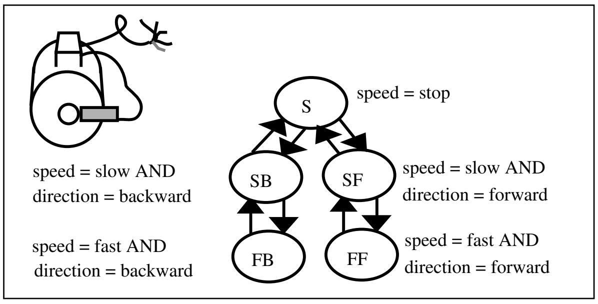

Figure 2: An electrical, bidirectional two speed engine is modeled by two variables, direction and speed. Its behaviour is modeled by a FSM with five interesting states. The arcs between the states describe legal transitions, rouling out the posibility to reverse the engine without requesting it to stop in between.

The task level program controls the object by requesting changes of its variable values. The object will respond upon a request with one of the following:

- error - there is no direct path from the current state to

1. We use the word *state* to denote a specific combination of values for a fixed set of variables. Each variable take its values from a finite (and thus discrete) domain.

any state representing the requested values, i.e., a state where the requested values will satisfy the constraint (task level programming error).
Or: there are several direct paths to states corresponding to requested variable values (object modelling error - should never occur)

- failed - the object did not succeed in reaching the requested variable values. This may be due to an engine overload situation, some fault in the engine or the electrical hardware from the I/O board and onwards, or a programming error in the algorithm responsible for achieving the state. This reply also indicates that any built in recovery actions have failed.
- succeed - the new state is reached which corresponds to the requested state.

After issuing the request, the task level process will be suspended until the object returns the result, in which case the execution of the requesting process continues as normal or an exception/error is raised.

This model of an object is well suited for specifying aspects of real time behaviour, such as *transition completion time*[1] and *algorithm periodicity.* This information is vital when designing controllers for any continuous physical process, and is a part of the description of a transition in the FSM.

2.2 Control layer - implements objects

The control layer hardware consists of a set of computing devices and peripheral hardware interconnected by communication channels. This layer should provide real time computational power and I/O to the world model.

The task of guaranteeing real-time properties in a distributed environment is in the general case hard. Although, major contributions have been made in the areas of real time communication systems and protocols (e.g. [15]), scheduling algorithms applicable both to communication and process scheduling (e.g. [4][12][14][1][13]) as well as distribution of time with bounded accuracy (e.g. [3][2]). The problem of allocating processor nodes for different sets of processes depending on both their computational and communicational need and the need for processor specific services (such as an I/O device) has also been investigated (e.g. [9][17]).

Our concern when developing Aramis has been to find a suitable specification for the computational needs of the objects/processes that support the usage of algorithms and analytical methods referenced above, and to find a good abstraction/interface to a high level task level language.

1. The maximal amount of time the transition will use once it is actually started.

2.3 Task level programming

A program written at the task level operates by coordinating object activities. The most primitive way to do this is to request a single object to change the value of one or several of its WM variables. This is a *primitive action*. A *user defined action* is a collection of calls to primitive and/or to user defined actions, and a *partial temporal order* (PTO) over them. The PTO consists of a set of *temporal restrictions* over pairs of these action calls. A temporal restriction between two action calls specifies that they must execute in sequence and which of them that should execute first. When a user defined action executes, the default is to execute all its parts in parallel, except when the PTO prohibits this. Whether parallel actions actually will be executed in parallel or not, depends on resource management. The control level may not be powerful enough to execute all requested algorithms in parallel, or some object may be occupied by other activities and the requested action will have to wait.

A set of processes (called *workers*) repeatedly call actions to solve their tasks. These workers are activated and deactivated by a human operator, acting as a foreman, deciding on what tasks to perform.

The description above is not complete. There exist constructs used within an action to define alternatives, and recursive actions are used to express iterations. There are constructs to express data flow between actions and transformations of data through the use of functions. For a complete description, se [16], [7], [6] and [5].

3 Validation of an application

This section describes how to translate a program written in a non-trivial programming language such as Aramis into a Modified Timed Petri Net (MTPN).

In Aramis, the specification is encoded as a part of the system, and is used as a backbone when building the system. Thus, verification of an Aramis program can be viewed as a consistency check, verifying:

- the absence of illegal task level operations, i.e. that operations are used only in situations for which they are defined,
- timing requirements on cycle times and actions
- that resource utilisation is “within bounds”, e.g., that computational resources are not exhausted.

The cycle time analysis is performed by translating the application into a Modified Timed Petri Net (MTPN), which can be analysed ([10][11]) and, if needed, provide information about what to alter in the Aramis program to satisfy the requirements.

We will focus this presentation on the translation of an application into a MTPN, since the result of the translation as well as the intermediate information computed are needed for the other consistency checks. The assumed application is a work cell on a shop floor.

3.1 Prerequisites for a timing analysis

There is a set of prerequisites, some imposed by the language and others by the application, that will allow us to perform timing analysis. These restrictions are:

- workers constitute non-parameterized blocks of code, operating over a well-defined and finite state space.
- the application is cyclic by nature; it will always start and end within a small set of possible states.
- the principle of recursion available in Aramis describes — correctly used — finite sub-iterations during one cycle of the main worker. It will be used to specify operations such as "insert five parts" or "fill pallet with parts" where the number of parts on a pallet is bounded — and known in advance.

With these restrictions, an Aramis program and the objects it operates upon can be rewritten as a MTPN.

3.2 Short overview of MPTN

A Modified Timed Petri net (MTPN) is a formalism in which both the controlling sub-system and the controlled sub-system are modelled. The controlling sub-system is modelled using a *timed Petri net*, and the controlled sub-system is modelled as an *activity diagram*.

An activity diagram is a graph of nodes, called states. Directed arcs between states model all possible transitions between states — the activities in the modelled system. For each activity there is also information about the time required to perform the activity.

A timed Petri net is a Petri net in which the minimum and maximum time a token will reside in a place is given. Since a token in a place in the timed Petri net is associated with the execution of a set of activities in the activity diagram, the time information for the place may be derived from the activity diagram.

3.3 Constructing the activity diagram

An activity diagram models the possible states and activities of a set of devices on the shop floor, at a suitable abstraction level. This can be compared with objects in the Aramis world model, where an object is defined as a set of state variables (of finite domain) and a set of transitions between a subset of the states for the object. An activity diagram can be seen as a partial view of the Aramis objects, only showing the object states, the object transitions and the transition time associated with each transition. For each transition there is a control algorithm defined, and information about the requested frequency of execution for the control algorithm and the amount of execution time needed in each iteration.

3.4 Constructing the timed Petri net

Given an instantiated Aramis program, i.e., a worker, the translation into a timed Petri net consists of the following six basic steps:

1. For each construct of an Aramis definition body (rules, action calls and temporal restrictions — in that order) generate a Petri-net template and link them together.
2. Associate each place representing a non-primitive action call with the corresponding definition body. Make a symbolic evaluation of the actual parameters in the call. Substitute the formal parameters of the body with the set of actual object references.
3. Perform 1. and 2. recursively for every definition body
4. Substitute each place with the corresponding generated Petri net recursively. The resulting Petri net is safe by construction.

   This will result in a Petri net with both action places (corresponding to primitive action calls) and temporary (synchronization) places, with no associated activity.
5. Reduce the number of temporary places by removing places which does not contribute to the synchronization or timing restrictions.
6. A primitive action call changes the world state according to the state variable assignments in the call. Since the state in which the action call is used is not known at translation time, the generated action place corresponds to a set of *possible* activity diagram arcs. Label the action place in the Petri net with the set of possible activity net arcs.
7. In the MTPN model, all activity diagram arcs for a place are executed in parallel. Since the set of possible activation diagram arcs for a place (as generated above) all apply to the same object, a resource conflict would occur if all were to be executed in parallel. Thus, the set of possible activation diagram arcs must be reduced to one arc for each action place. This is done by a symbolical execution of the PN, given an initial marking and an initial state of the activity diagram. For each action place, all arcs (in the set of possible activity diagram arcs) that are not applicable are removed[1].
8. After this pruning pass the resulting labelling of action places is one of the following three:

   a) the set of possible activity diagram arcs is empty, i.e., there will never be a situation in which the correspond-

1. A full presentation [8] will include a complete description of the abstract syntax of Aramis programs as well as their semantics, together with a description of the translation and all data structures needed during that process.

ing action call will be applicable — it will always generate an error,

b) the set of arcs contains one arc,

c) the set of arcs contains several arcs. Split the place into alternative places with guarding conditions on the incoming transitions to obtain one place per transition.

The result is a timed Petri net, where each place is either a temporary place (with no activity and zero time) or labelled with a single arc in the activation graph.

3.5 Validation analysis

During the translation into a MTPN an applicability control were made, revealing programming errors normally detected in runtime.

Using the MTPN, a scheduling of the execution can be performed to minimize the cycle time and resolve any resource conflict. As a result of this schedule, the minimum and maximum cycle time is known.

Another result is that the schedule shows what activities can actually be performed in parallel, and what activities can not. Thus, using the information about the associated control algorithm (requested frequency and the execution time for one iteration) an upper bound for the utilisation of real-time CPU-time can be estimated.

3.6 Error recovery support

We would like to emphasis the difference between exception handling and error recovery. In traditional programming languages (such as C or ADA) an exception is a well-defined but seldom occurring situation, for which there exists predefined corrective actions to perform — an exception handler.

By error recovery we mean the process of recovery from abnormal, unforeseen situations, often resulting from unforeseen events or events so seldomly occurring that enumerating them is not practical.

Providing automatic recovery from such events in the manufacturing context is often not possible, since the recovery typically include repair of, e.g., broken machinery. As an alternative, we have in [5] presented a method in which an abstract version of the system state is logged. The logged information about the normal operation of the system is used to support the user in the task of restarting the machinery after an error, minimizing the time needed or the material wasted during the restart. Since logged information is used, this approach is only useful in a system that has been in use for some time.

However, during the translation of an Aramis program into a MTPN, the information previously acquired by logging the execution is now available as a result of the translation process described above, provided that the Petri net and the information calculated during the symbolic evaluation is saved.

During restart of the machinery, the current state is matched against the places in the MTPN and the saved information. The resulting set of places are grouped into consistent sub-groups, where two places belong to the same sub-group if they are part of parallel branches of the Petri net. These subgroups define possible markings of the Petri net suitable for continued execution.

If this process results in several possible markings, this means that there are several possible start points in the production process with identical state — according to the observable part of the world. We suggest that the user, the operator of the machinery, is asked for confirmation or selection of restart point.

If there are no matching places, this indicates that the current state can not be passed during normal operation.

4 Conclusion

In this paper we have presented a layered design of a real-time control system for the design and instruction of manufacturing applications. Real-time properties and task level cycle time are validated in pre-runtime. The types of programming errors that are detected is extended to include applicability tests. Furthermore, the information calculated during the validation process can be reused to support restart from an almost arbitrary state.

5 References

[1] G. Fohler. Realizing Changes of Operational Modes with Pre Run-Time Scheduled Hard Real-Time Systems. In *Proceedings of the Second International Workshop on Responsive Computer Systems*, Saitama, Japan, October 1992.

[2] H. Kopetz. Sparse Time versus Dense Time in Distributed Real-Time Systems. *Proc. 12th Int. Conference on Distributed Computing Systems*, Japan, June 1992.

[3] H. Kopetz, W. Ochsenreiter. *Clocksynchronization in Distributed Real-Time Systems*, IEEE Transactions on Computers, Vol C36, No. 8, pp. 933-940, Aug. 1987.

[4] D. W. Leinbaugh and M. Yamini. Guaranteed response time in a distributed hard real-time environment, *Proc. Real-time systems symposium*, Dec. 1982.

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

[6] P. Loborg, P. Holmbom, M. Sköld and A. Törne. 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.

[7] P. Loborg and A. Törne. A Hybrid Language for the Control of Multimachine Environments, in Proceed*ings of EIA/AIE-91*, Hawaii, June 1991.

[8] ARAMIS — A Layered Architecture. (Technical report, under preparation)

[9] P. R. Ma, E. Y. S. Lee and M. Tsuchiya. A Task Allocation Model for Distributed Computing Systems, *Hard Real Time Systems*, Eds. Sytankovich/Ramamritham, pg.249-255, IEEE Comp.Soc.Press, 1988.

[10] Z. Peng and A. Törne, A Petri net based modelling and synthesis technique for real-time systems. *Proc. of the 5th Euromicro Workshop on Real Time Syetems*. p.204-209, Oulu, Finland, 1993.

[11] Z. Peng and A. Törne, Timing Analysis and Conditional Scheduling in a Real-Time System Design Environment. *Proc. of the 6th Euromicro Workshop on Real Time Syetems*. p.214-219, Västerås, Sweden, 1994.

[12] R. Rajkumar, L. Sha, J. P. Lehoczky, Real-Time Synchronization protocols for multi-processors. *Proc. IEEE Real-Time systems symp.*, CS Press, Los Alamitos, Calif., pp. 259-269, 1988.

[13] L. Sha and S. Sathaye. Distributed Real-Time System Design using Generalized Rate Monotonic theory, *Second Int. Conf. on Automation, Robotics and Computer Vission*, SINGAPORE, 1992.

[14] J. A. Stankovic, K. Ramamritham, and P.-F. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 184-194, April 1990.

[15] J.-P. Thomesse, P. Lorenz, J. P. Bardinet and T.Valentin, *Factory Instrumentation Protocol: Model, Products and Tools*, Control Engineering, pp65-67, Sept 1991.

[16] A. Törne. The Instruction and Control of Multi-Machine Environments, in *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.

[17] J. P. C. Verhoosel, E. J. Luit, D. K. Hammer, E. Jansen. A Static Scheduling Alghorithm for Distributed Real-Time Systems, *Journal of Real-Time Systems*, 3, pp.227-246, Kluwer Acad. Pub. 1991.

[18] C. Eriksson. An Object-Oriented Framework for the Design of Hard Real-time Systems — A Study focused on RealTimeTalk. Licentiate thesis KTH/MAE/R--94/1--SE, Royal Institute of Technology, Sweden, 1994.