Long Term Scheduler for Real Time Industrial Installations

A. F. Zorzo, L. A. Cassol, A. L. Nodari, L. A. Oliveira, L. R. Morais

Faculdade de Informática
Pontifícia Universidade Católica do RS
Porto Alegre, Brazil, 90619-900
{zorzo, cassol, laugusto}@inf.pucrs.br, andrenodari@hotmail.com, lierson@tutopia.com.br

Abstract. This paper presents the use of a multiparty interaction mechanism, which provides means for dealing with concurrent exceptions, in the design of a controlling software for a real time production cell. In this paper we propose a new approach for scheduling the multiparty interactions, i.e. we deal with the scheduling problem as a planning problem, and use a planning algorithm used in problems of Artificial Intelligence. This algorithm is used for the long term scheduling, while a short term scheduler is used to inform all participates of the interaction about the next action that must be performed.

Keywords: Multiparty Interactions, Real Time Production Cell, Planning Algorithm, Long Term Scheduling.

1 Introduction

The use of computer systems has increased in the past few years in almost every field of science. Hence, the increase of the quality of the results these computer systems have to provide. One of such type of computer system is industrial control systems, which are responsible for managing devices that interact with the real world. The interaction with real world events brings these systems to deal with real time, i.e. the system cannot only produce correct results, but have to generate these results at the right time.

Realistic real time safety-critical systems often involve complex concurrent activities. In some cases these concurrent activities may be working together, i.e. *cooperating*, to solve a given problem; in other cases the activities can be completely *independent* or may be essentially independent though needing to *compete* for shared common system resources. In practice, different kinds of concurrency might co-exist in a complex application which thus will require a general supporting mechanism for controlling and coordinating complex concurrent activities.

In this paper, we show how to design and implement real time safety-critical applications. This mechanism is called *Dependable Multiparty Interaction* [1], which provides means for dealing with concurrent exceptions. In addition, the DMI mechanism allows for checking the consistency of the results upon termination of the interaction, not allowing the participants of the interaction to leave before all participants have terminated their activities.

To show how our approach can deal with real time properties in an industrial installation, we use a real time production cell case study developed in the Forschungszentrum Informatik (FZI), Karlsruhe, Germany [2].

This paper is organized as follows. Section 2 describes the production cell case study. Section 3 shows the multiparty interaction mechanism used to design and implement the control system for the production cell described in Section 2. Section 4 describes the design and implementation of the control software for the real time production cell. Section 5 presents the scheduling strategy used in the production cell control system. Section 6 draws some conclusions.

2 FZI RT Production Cell

Industrial installations which have several pieces of equipment controlled by software systems require that interactions between this equipment are executed in a safe and fault-tolerant way, and respect real time properties. In this paper we use a real time (RT) production cell case study to explore how interactions that happen between devices can be enclosed by DMIs to guarantee safety and fault tolerance properties. Furthermore, we explore the use of long term and short term scheduling algorithms to guarantee real time properties of the case study.

The RT production cell is composed of two conveyor belts – a feed belt and a deposit belt, a barcode reader, four processing units and two cranes (see Figure 1). The state of devices is reflected by sensors that provide information about their position. Each device has a set of actuators and sensors that are used by a control program to change the state of the device.

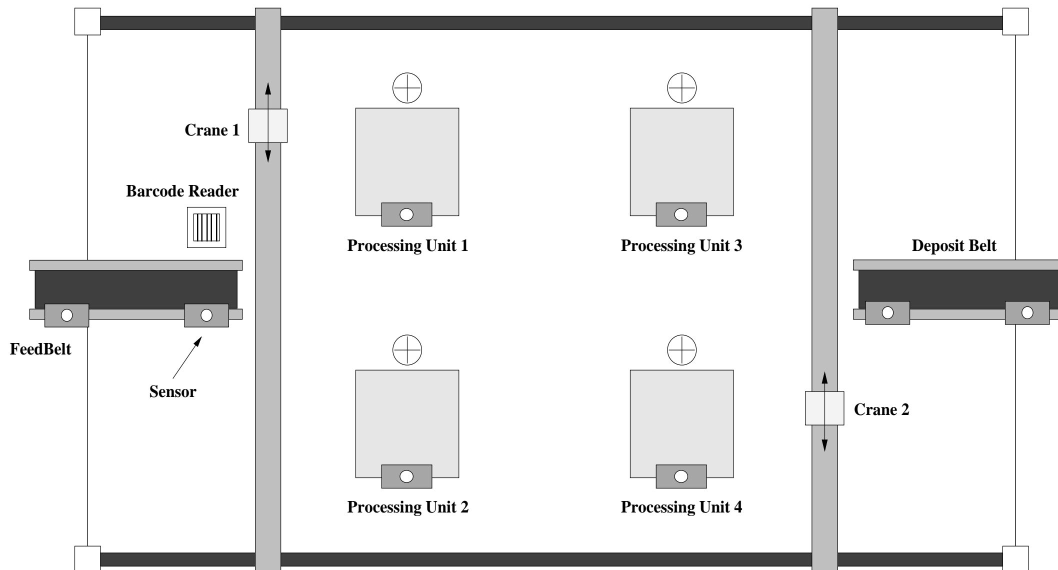

Fig. 1. FZI Real Time Production Cell

The processing of a metal plate takes the following steps. Metal plates are conveyed from the beginning of the feed belt to the front of the barcode reader. When the metal

plate is positioned in front of the barcode reader, then the feed belt is stopped and the barcode reader can read information about the processing units that plate must be taken to. The barcode reader send this information to the controlling software. The first crane of the RT production cell has then to take the metal plate to a processing unit read from the barcode. When the processing unit has finished processing the metal plate, the same crane, or the second crane, takes the metal plate to another processing unit. The metal plate is taken by the second crane to the deposit belt after being processed by all units read from the barcode. The deposit belt is controlled by the environment.

The barcode contains the following information: A) processing units that must be used to process the metal plate; B) time the metal plant can stay in the cell; C) minimum and maximum time the plate can stay in the each processing unit; D) processing order, i.e. the metal plate can be processed in any unit order, or has to follow the unit order read from the barcode.

The RT production cell contains two types of processing unit: A) drill or press – devices that have to be switched on and off by the controlling software; B) oven – devices that are always on, so the metal plates are processed while inside the devices.

3 Dependable Multiparty Interaction

Usually, multiparty interaction mechanisms [3] [4] do not provide features for dealing with possible faults that may happen during the execution of the interaction. Typically, the underlying system that is executing those multiparty interactions will simply stop the system in response to a fault. In DisCo [5], for instance, if an assertion inside an action is false, then the run-time system is assumed to stop the whole application. This is unacceptable in many situations, e.g. a flying aircraft.

In this section, we describe an augmented multiparty interaction mechanism, i.e. the *dependable multiparty interaction* (DMI). Specifically, a DMI provides facilities for:

- HANDLING CONCURRENT EXCEPTIONS: when an exception occurs in one of the bodies of a participant, and is not dealt with by that participant, the exception must be propagated to all participants of the interaction [6]. A DMI must also provide a way of dealing with exceptions that can be raised by one or more participants. Finally, if several different exceptions are raised concurrently, the DMI mechanism has to decide which exception will be raised in all participants.
  With respect to how the participants of a DMI will be involved in the exception resolution and exception handling, there are two possible schemes: synchronous or asynchronous. In synchronous schemes, each participant has to either come to the action end or to raise an exception; it is only afterwards that it is ready to participate in any kind of exception handling; this means that the participant's execution cannot be pre-empted if another participant raises an exception. In asynchronous schemes, participants do not wait until they finish their execution or raise an exception to participate in the exception handling; once an exception is raised in any participant of the DMI, all other participants are interrupted and handle the raised exceptions together. Although

implementing synchronous schemes is easier than asynchronous, because all participants are ready to execute the exception handling, the synchronous scheme can bring the undesirable risk of deadlock. Therefore the asynchronous scheme is adopted.

– ASSURING CONSISTENCY UPON EXIT: participants can only leave the interaction when all of them have finished their roles and the external objects are in a consistent state. This property guarantees that if something goes wrong in the activity executed by one of the participants, then all participants have an opportunity to recover from possible errors.

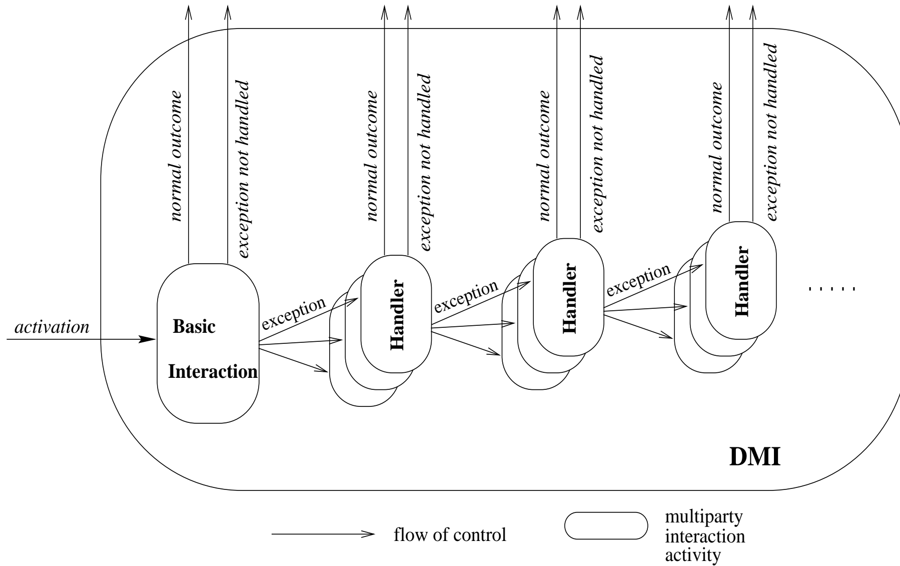

Fig. 2. Dependable Multiparty Interaction

The key idea for handling exceptions is to build DMIs out of not necessarily reliable multiparty interactions by chaining them together, where each multiparty interaction in the chain is the exception handler for the previous multiparty interaction in the chain. Figure 2 shows how a basic multiparty interaction and exception handling multiparty interactions are chained together to form a composite multiparty interaction, in fact what we term a DMI, by handling possible exceptions that are raised during the execution of the DMI. As shown in the figure, the basic multiparty interaction can terminate normally, raise exceptions that are handled by exception handling multiparty interactions, or raise exceptions that are not handled in the DMI. If the basic multiparty interaction terminates normally, the control flow is passed to the callers of the DMI. If an exception is raised, then there are two possible execution paths to be followed: *i*) if there is an exception handling multiparty interaction to handle this exception, then it is activated by all roles in the DMI; *ii*) if there is no exception handling multiparty interaction to handle the raised exception, then this exception is signaled to the invokers of the DMI. The whole set of basic multiparty interaction and exception handling multiparty interactions form a single entity: they are

isolated from the outside so that internal activities (e.g., the raising of an exception) are not visible to the enclosing environment.

The exceptions that are raised by the basic multiparty interaction or by a handler, should be the same for all participants in the DMI. If several participants raise different concurrent exceptions, the DMI mechanism activates an exception resolution algorithm based on [6] to decide which common exception will be raised and handled.

4 Control System: Design and Implementation

Our design for the RT Production Cell is based on [7] and addresses the safety, functionality, and efficiency requirements by separating the design into a set of DMIs that control the interactions between devices, a set of device controllers that execute DMIs, and a scheduler that selects the next DMI to be executed. Safety requirements are satisfied at the level of DMIs, while the other requirements are met by the device controllers and the scheduler (real time requirements). Both the scheduler and the set of device controllers can be programmed in several ways.

Using this approach, the controlling software for the whole RT Production Cell consists simply of the set of DMIs, a set of device controllers, and a DMI scheduler. Our priority was to provide a clear system design that was simple in structure and met the case study requirements. Our intention was to structure all critical system activities, i.e. *safety* properties, out of DMIs and leave the real time properties and *liveness* properties to be dealt with by a special scheduler.

In this work, the main design decisions were as follows. To meet the safety requirements, boundaries of DMIs were chosen in such a way that neither metal plates nor devices could collide. Only one metal plate can be in a DMI, a metal plate cannot be involved in more than one DMI, and a device can participate in only one DMI at a given time. All device movements are performed within DMIs and the devices involved in a DMI are switched off before leaving that DMI. Thus, all devices are stationary when not under the control of a DMI. To meet the real time and liveness properties, a special scheduler communicates with the device controllers in order to receive information about the devices state, and to inform a device controller about the next DMI can be executed by that controller.

We have designed 21 DMIs to control the interactions between devices for our system. Each DMI controls one step of the metal plate processing and typically involves passing a metal plate between two devices. Figure 3 shows three DMIs from the controlling software. Each DMI is represented by a dashed rectangle, and includes the devices that are involved in that DMI. If two DMIs are overlapped, then those two DMIs cannot be executed in parallel because the controller of that device can execute only one of them at one time. For example, DMI LoadCrane1 and LoadUnit1 have a common activity that has to be executed by the crane controller, therefore they cannot be executed in parallel.

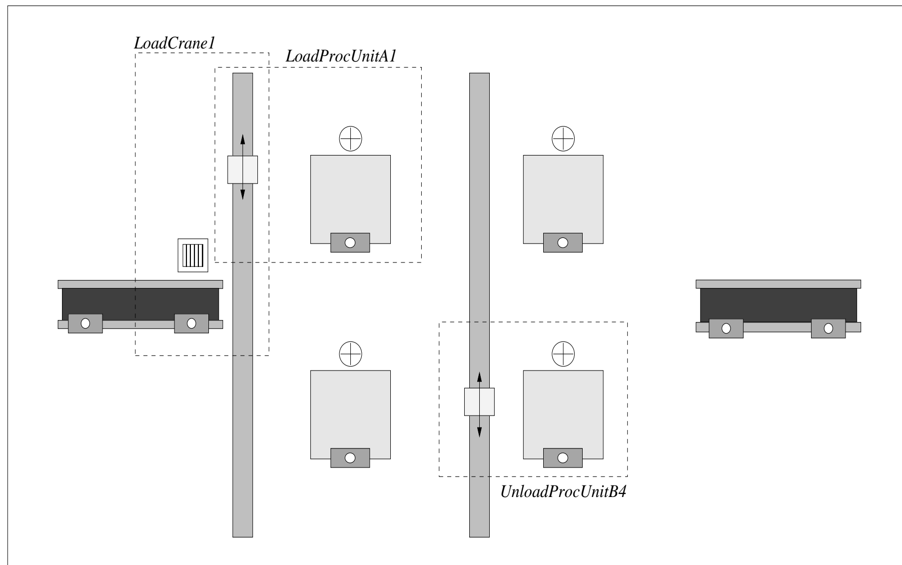

Fig. 3. Three DMIs in the RT Production Cell

5 Real Time Scheduling

In several real time systems, we have to meet the deadline not for only one activity, but for several activities that are being performed in parallel (or concurrently). Therefore, the order in which these activities are executed is very important (if not crucial), i.e. changing the order of two activities can bring the two of them to meet their deadlines or not.

In the controlling software for the RT production cell presented in the previous section, we have implemented two different schedulers: one short term scheduler – decide which activity has to be executed next (it is very fast); one long term scheduler – based on the load of the system to decide which activity can be executed next. Our first scheduler has shown to be very difficult to be implemented for the controlling software of the RT production cell. It is very hard to program a scheduler that can decide very quickly the next action to be executed, because there are several variables to be considered. We have implemented an *ad hoc* scheduler but it could deal only with two metal plates in the system, otherwise several deadlock situations could happen in the system depending on the units a new plate had to use.

Our second approach was to use a long term scheduler that would arrange the set of activities, which have to be executed, in such a way that all metal plates meet their deadlines. A new metal plate will not be introduced into the system if this new metal plate can cause any plate to miss its deadline. The long term scheduler arrange the activities in a queue, which is used by a short term scheduler that inform the controllers about the DMI they have to execute. Depending on the state of the system, another DMI can be sent to a different set of device controllers.

In the next section we describe a partial-order planning algorithm [8] implemented in our long term scheduler.

5.1 Scheduler algorithm

Before we start describing the way a partial-order planning algorithm has been applied in our long term scheduler, we reduce our problem to a two belt, two units, one crane production cell (this is done only to demonstrate the algorithm). Additionally, consider the following state for the RT production cell: crane over processing unit 1; one processed metal plate in the processing unit 1 (B1); and, a metal plate over the feed belt (B2).

Consider also the following set of actions that can be executed on the reduced production cell described in the previous paragraph: A) `CONSUME(plate)` – take the plate to the deposit belt; B) `MOVE(plate,begin,end)` – move a metal plate from position begin to position end (these are valid positions in the cell, i.e. belts or processing units); C) `PROCESS(plate,unit)` – process the plate in that processing unit. The final state of the cell, after executing all activities in the cell, for the two plates is the following: metal plate B1 has been processed by unit 1 and metal plate B2 has been processed by unit 2, and both metal plates leave the cell via the deposit belt.

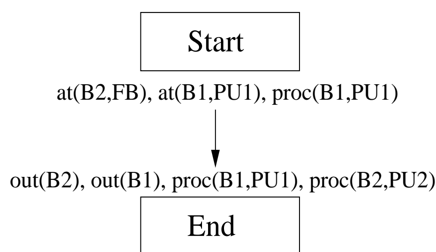

Fig. 4. Minimum plan.

Further to the above actions, we have defined the following set of predicates: A) `at(X,Y)`: states that metal plate, or crane, X is over device Y (belt or processing unit); B)`out(X)`: states that metal plate X has left the production cell, i.e. has been processed and left the system via the deposit belt; C) `proc(X,Y)`: states that metal plate X has been processed by processing unit Y.

Based on the above predicates, we have the following initial pre-conditions for the described production cell described above:

`at(B1,PU1), at(B2,FB), proc(B1,PU1)`

The goals we have (post-conditions) are the following:

`out(B1), out(B2), proc(B1,PU1), proc(B2, PU2)`

Figure 4 shows the minimum plan for our system, i.e. contains the initial state of the system and the goal for that system. Steps are represented as rectangles, where the predicates over the rectangle represent the pre-conditions, and the predicates under the rectangle represent the effects of the step. An arrow indicates the order relation between steps.

The first step taken by the algorithm (planner) is to test whether the plan is a solution or not. This guarantees that nothing is executed when the pre-condition is also the goal.

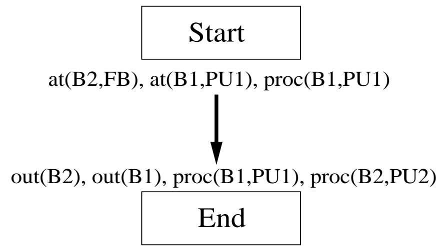

Fig. 5. Causal relation between plans.

The next step chooses a sub-goal, i.e. a pre-condition of one of the steps that has not been satisfied. For example, pre-condition `proc(B1,PU1)` from step `End`. Although step `Start` has this predicate as a post-condition, the planner does not have this knowledge. Therefore, the planner has to choose a step. In this example, the planner chooses step `Start`, which has predicate `proc(B1,PU1)` as a post-condition. Then the planner sets a causal relation between steps `Start` and `End`. Figure 5 shows this relation through a thicker arrow. It is important to note that whenever there is a causal relation between steps, this implies in an order relation as well.

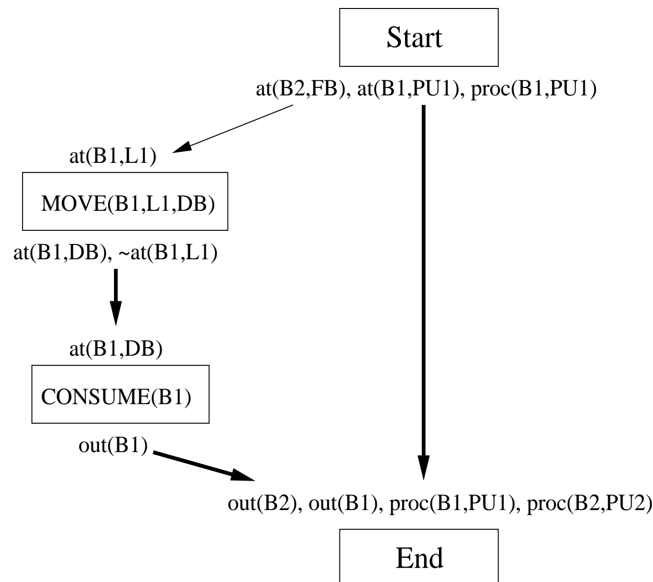

Fig. 6. Step CONSUME added to satisfy predicate `out(B1)`.

At this point, we do not have any threat (we explain this later), therefore we can execute the next iteration of the planner, i.e. we choose a new pre-condition as sub-goal, e.g. `out(B1)`. The planner does not find any step that has this predicate as post-condition, so it chooses a new step to be inserted into the plan. In our example, `CONSUME(B1)` is chosen. This step has `at(B1,DB)` as pre-condition and `out(B1)` as post-condition. Now the

planner has to find a step that has `at(B1,DB)` as a post-condition. The step that satisfy that is `MOVE(B1,L1,DB)` (note that `L1` represents any valid place over which the metal plate is stored). The pre-condition of this last added step is satisfied by step `Start`. This completes a plan for one of the metal plates we have in the cell.

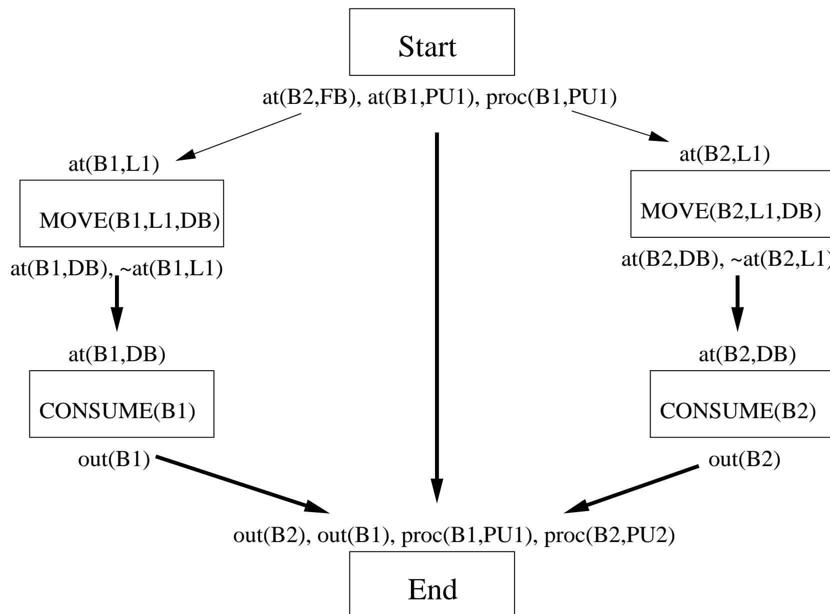

Fig. 7. Predicate `out(B1), out(B2) and proc(B1,PU1)` satisfied.

Figure 7 shows part of a plan for the second metal plate B2. The steps for this second metal plate are added to the plan in the same way the planner added steps to metal plate B1. Note that we still have not satisfied all pre-conditions of our goal, i.e. pre-condition `proc(B2,PU2)`. To achieve that, the planner includes a new step into the plan: `PROCESS(B2,PU2)`. The pre-condition for this new step is `at(B2,UP2)`, and this pre-condition can be satisfied by step `MOVE(B2,L1,PU2)`. After setting the causal and order relation between these new steps, the new plan is shown in Figure 8.

Figure 8 shows a threat in the plan. A threat is a situation one step can invalidate a pre-condition of another step in the plan. For example, in Figure 8 two steps `MOVE` are being executed over the same metal plate threating the pre-conditions `at(B2,PU2)` of step `PROCESS(B2,PU2)`, and `at(B2,DB)` of step `CONSUME(B2)`. To solve this threat, the planner set an order for these two conflicting steps. For example, the planner take step `MOVE(B2,L1, DB)`, which is threating `PROCESS(B2,PU2)`. The planner can demote this step (move the step to before `PROCESS`), or promote it (move the step to after `PROCESS`). When the planner tries to demote step `MOVE`, it detects that the plan will become inconsistent, and backtracks. It tries now to promote `MOVE`, and verifies that the plan is consistent.

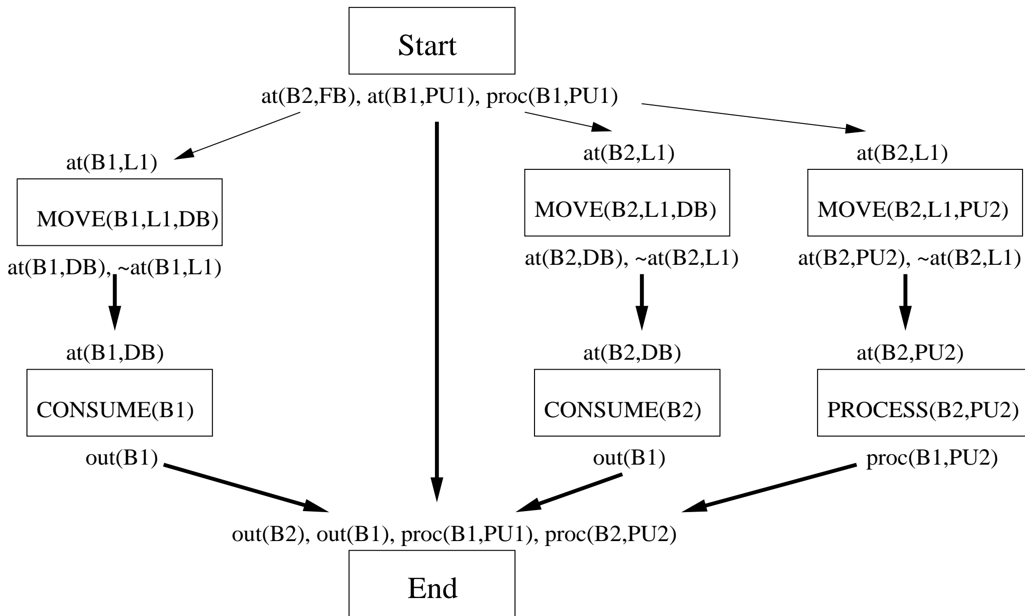

Fig. 8. Threat between PROCESS(B2,PU2) and MOVE(B2,L1,DB).

At this point, the planner realizes that the plan is a solution, i.e. all goals are met. The plan is complete and consistent, therefore the planner can stop. Figure 9 shows the full plan generated by the algorithm.

6 Conclusion

The work presented in this paper shows how a planning algorithm can be used to schedule a set of activities in a real time production cell. The design of the controlling software for the RT production cell made use of a fault tolerant mechanism called dependable multiparty interaction. This mechanism provides mean for handling concurrent exception handling during a process of interaction between several entities.

Unlike two other case studies, similar to the one presented in this paper, i.e. Production Cell I [9] and Production Cell II [10], the RT Production Cell had to deal with complications as concurrent timing and value faults. A previous work has shown how this could be dealt with in real time distributed object oriented systems [11]. This work, however, shows how real time attributes can be added to a multiparty interaction abstraction called Coordinated Atomic action [12], but does not describe how those activities have to be scheduled. Other solutions to schedule multiparty interactions have been presented in [7] [13] [14], but all of them leave the scheduling problem to be solved by the set of device controllers.

The system we described here has been used to drive a simulator provided by FZI. We have faced several performance problems, but solved them using a distributed solution, in which the long scheduler is executed in a separate computer. The long term scheduler communicates with the short term scheduler via remote procedure calls.

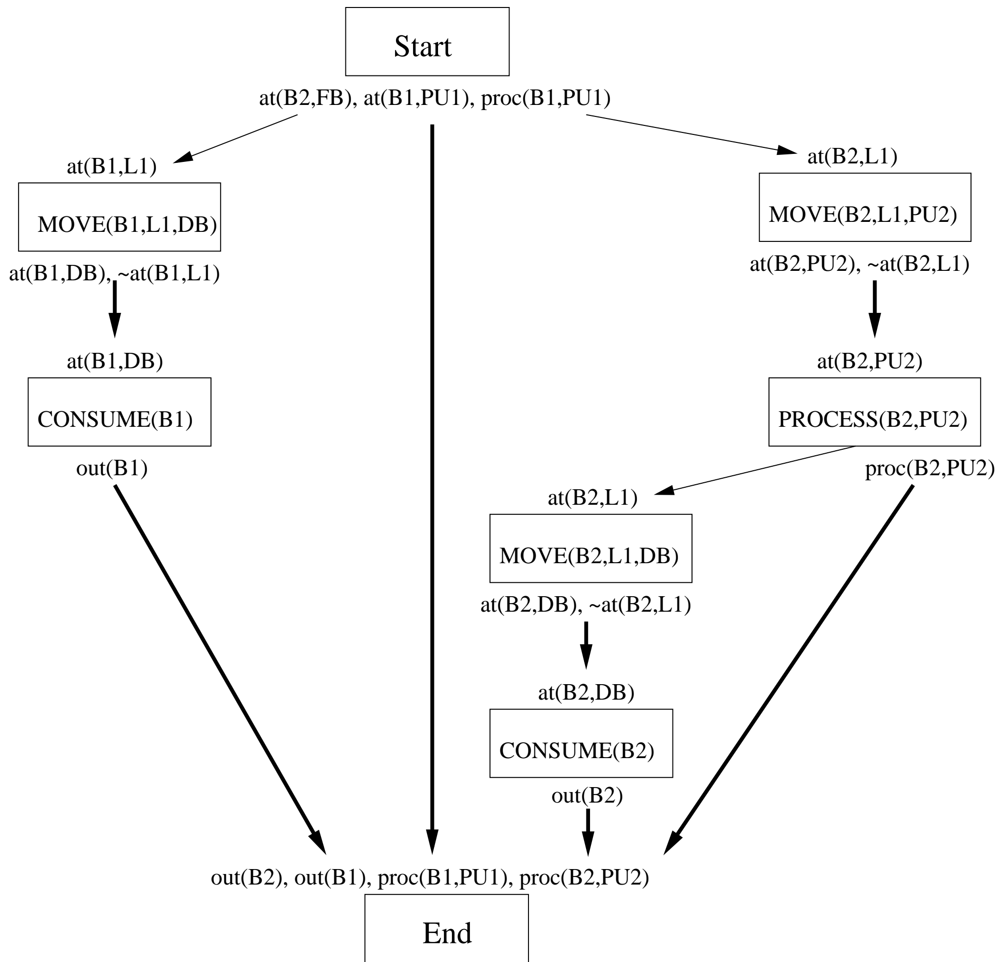

Fig. 9. Full plan.

The next step of this work is to implement the scheduler proposed here using *Field-Programmable Gate Arrays* (FPGAs) [15]. This implementation is realized direct into hardware, reducing, therefore, the performance problems we had.

Acknowledgments

We would like to thank people from the University of Newcastle upon Tyne, England, that helped at the beginning of this work (during ESPRIT LTR Project DeVa - Design for Validation). We also thank the Brazilian agencies CNPq and FAPERGS for their financial support in this project. L. Cassol is financed by Hewlett-Packard Brazil.

References

1. A. F. Zorzo. *Multiparty Interactions in Dependable Distributed Systems.* PhD thesis, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, 1999.

2. A. Lötzbeyer and R. Muhlfeld. Task description of a flexible production cell with real time properties. Technical report, Forschungszentrum Informatik, Karsruhe, Germany, 1996. http://www.fzi.de/divisions/prost/projects/korsys/korsys.html.
3. Y.-J. Joung and S. A. Smolka. A comprehensive study of the complexity of multiparty interaction. *Journal of ACM*, 43(1):75–115, 1996.
4. I. Forman and F. Nissen. *Interacting Processes - A multiparty approach to coordinated distributed programming.* ACM Publishers, 1996.
5. H.-M. Jårvinen and R. Kurki-Suonio. Disco specification language: Marriage of actions and objects. In *11th International Conference on Distributed Computing Systems*, pages 142–151. IEEE CS Press, 1991.
6. R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, 1986.
7. A. F. Zorzo, A. Romanovsky, B. Randell J. Xu, R. J. Stroud, and I. S. Welch. Using coordinated atomic actions to design safety-critical systems: A production cell case study. *Software: Practice and Experience*, 29(8):677–697, 1999.
8. S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Inc., 1995.
9. C. Lewerentz and T. Lindner, editors. *Formal development of reactive systems: Case study production cell*, volume 891 of *Lecture Notes in Computing Science*. Springer Verlag, Berlin, Germany, 1995.
10. A. Lötzbeyer and R. Muhlfeld. Task description of a fault-tolerant production cell. Technical report, Forschungszentrum Informatik, Karsruhe, Germany, 1996. http://www.fzi.de/divisions/prost/projects/korsys/korsys.html.
11. A. Romanovsky, J. Xu, and B. Randell. Exception handling and resolution in distributed object-oriented systems. In *16th IEEE International Conference on Distributed Computing Systems*, pages 545–552. IEEE Computer Society Press, 1996.
12. J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *25th International Symposium on Fault-Tolerant Computing*, pages 450–457. IEEE Computer Society Press, 1995.
13. J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, E. Canver A. F. Zorzo, and F. von Henke. Rigorous development of a safety-critical system based on coordinated atomic actions. In *29th International Symposium on Fault-Tolerant Computing*, pages 68–75. IEEE CS Press, 1999.
14. J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Transactions on Computers*, 2002.
15. S. D. Brown, R. J. Francis, J. Rose, and Z. G.Vranesic. *Field-Programmable Gate Arrays.* Kluwer Academic Publ., 1992.