Presented at Spin Workshop, 16 Oct. 1995, Montreal, Canada.

Process Control Design Using SPIN

Thierry CATTEL[1]

[1]Laboratoire de Téléinformatique, Ecole Polytechnique Fédérale
CH-1015 Lausanne, Switzerland, cattel@di.epfl.ch

Abstract — This paper reports an experience with the modeling, verification and concurrent implementation of a medium-sized process control problem. The case study was proposed by Forschungszentrum Informatik, Karlsruhe in 1993 in order to promote the usage of formal methods in industry. It concerns an industrial robotics application that processes metal plates. A top-down design approach is followed where successive CCS and Promela specification levels of decreasing abstraction are considered, each layer little by little allows verification of parts of the security requirements thus providing a mean for coping with state explosion. The level refinements are checked with the Concurrency Workbench a CCS-based tool. Safety and liveness requirements are expressed in linear temporal logic and checked with SPIN. From the ultimate specification, two different implementations are derived. The first one is in Synchronous C++, a concurrent extension of C++ and the second in Regis/Darwin. This application shows that SPIN is also quite appropriate for developing control process problem from scratch and with requirements to be checked in mind. It appeared clearly that the specification phase was very important for obtaining a satisfactory specification from which a well behaved implementation was derived easily in a few days.

Keywords — application, process control, refinement approach, LTL properties verification, process equivalence, concurrent programming.

Introduction

Though SPIN[1 2 3] was specially designed for tackling protocols, it appeared that it was also quite suitable for addressing other problems such as distributed algorithms and multiprocessor operating systems [4]. Provided one is able to express problems as protocols it is quite possible to take advantage of SPIN's power for modeling and verifying them also. We show in this paper that process control systems may be seen as particular protocols and be verified as such. Some dedicated languages and tools such as LUSTRE[5] are certainly more powerful and efficient for expressing and verifying such systems, but it is not clear that the resulting specifications are more readable than the ones obtained with Promela. Furthermore there is no possibility, in particular with LUSTRE, to check liveness properties, whereas SPIN is well adapted for this purpose. When we started to design a controller for the Production Cell case study proposed in 1993 by Forschungszentrum Informatik, Karlsruhe, Germany[6] there already existed around 20 contributions[7]. Few of them proposed a complete solution to the problem in term of specification, verification and implementation, and no one really solved the liveness requirements checking. My main motivation was to contribute by firstly addressing the liveness concerns. Another goal was to derive a straightforward implementation of the detailed Promela specifications by translation into Synchronous C++[8], a concurrent extension of C++ developed in our Labs, that should be soon integrated in Gnu distribution. I have also derived an implementation of the production cell in Regis/Darwin the distributed framework of Imperial College[9], from the Promela specifications ; one of the advantages being the obtention of a clearer architecture that may be graphically built with a visual tool such as the Software Architect Assistant[10].
For being tractable the produced models need to be designed according to several levels of decreasing abstraction, thus a refinement approach was used. Unfortunately SPIN currently provides no way of verifying the consistency of such refinements since no support for checking process equivalencies is available. Some attempts exist [11] for extending SPIN in

that direction but only allows for restricted equivalencies (trace equivalence, trace inclusion). For this reason I also developed in parallel some CCS[12] models that correspond to the Promela specification.
In the following sections, I will first briefly present the production cell case study, the design approach used and some of the resulting models, the verification of the liveness requirements, the verification of some safety requirements and eventually some considerations related to the implementation.

1. The Production Cell Case Study

This case study is inspired of an actual industrial installation in a metal-processing plant in Karlsruhe. It is a realistic industry-oriented problem in which safety requirements are important. The production cell (see Fig. 1.1) processes metal blanks with a press. First the blanks are introduced on a feedbelt that leads them towards a rotary table that presents the blanks to a two-armed robot. The robot takes the blanks from the table, feeds the press and takes back the forged blank to deposit it on a second belt. This belt leads the blanks toward a traveling crane that brings them to the feedbelt again. The production cell is cyclical only for sake of the case study, in reality the blanks would be dropped from the traveling crane into some container. Up to 8 blanks may be processed in parallel by the system.

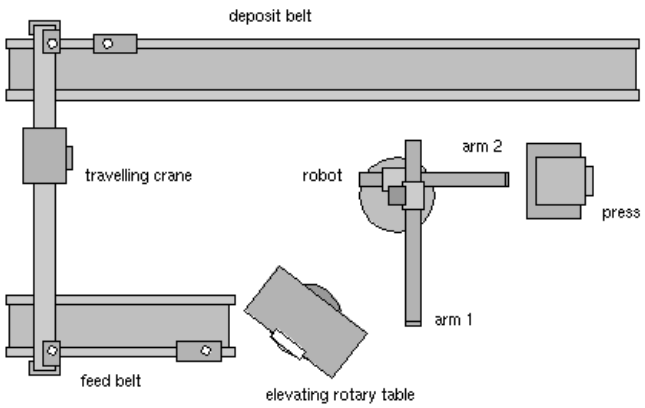

Fig.1.1 — The Production Cell

The task description document[6] explicits three kinds of requirements : safety properties, liveness properties and performance and general software engineering properties. There are four classes of safety properties. First, mobility restriction properties that specify that a physical element has to restrict its movements within certain limits so as not to damage itself. Second, machine collision avoidance properties : for instance the press must not move if one arm of the robot is engaged inside it. Third, blanks must not be dropped outside safe regions, and fourth, to being distinguished, blanks need to be kept sufficiently distant from each other. The stronger liveness requirement expresses that every unforged blank introduced into the system will eventually leave it forged. A weaker form of this liveness property expresses that for each of the cell elements, if a given blank enters it, it will eventually exit it. The performance requirements are irrelevant here since we only address an untimed model. The software engineering considerations are related to the maintainability and flexibility of the resulting control software.

2. Design Approach

First of all, I make the hypothesis that the controller I will design will be fast enough for controlling the plant, namely it will not loose any significant plant data and hence the corresponding models will be only qualitative and untimed. Second, I will comply with the

following guidelines : as commonly accepted for reactive systems, the specification of the controller and the environment it drives will be clearly separated, besides the controller will be object-oriented, namely the architecture of the controller will reflect the physical cell organization in the sense that to each physical cell element will correspond an (active) object. And third, for the purpose of liveness checking, I will consider two versions of the system : a closed version where the traveling crane takes back the blanks to the feedbelt and an opened one where the traveling crane drops the blanks into a container.

The complete approach involves 4 description levels of decreasing abstraction. The motivation for doing so is to take advantage of a stepwise refinement development. Indeed this framework powerfully copes with complexity in two ways. First, the specifications are built progressively adding details to a very abstract model, thus releaving the designer creation effort. The proof of the refinement correctness from one level to the next one consists in checking process equivalencies. It was carried out with the Concurrency Workbench[13] on CCS specifications. Second, the successive description levels allow for checking included subsets of the requirements, thus giving a mean for addressing the state explosion problem. More precisely the first description level only expresses abstract interactions (blank exchanges) through synchronous rendezvous between components of the controller, the second level integrates detailed interactions (adding for instance *"are you ready"* synchronization's before the effective blank passing), the third level is enriched with the movement orders submitted by the cells elementary controllers and the abstract specification of the movement realization. Finally the fourth level details the movement realization and includes the physical cell environment specification. It is from this last description, structured as three layers, of course without the environment specification, that a concurrent implementation will be derived in a straightforward way.

3. Modeling and Verification

The Promela models were written for levels 2 to 4. With level 2 it was possible to check that the closed production cell may process up to 8 blanks without deadlocking (it reaches a deadlock as soon as the 9th blank arrives), and to check the strong liveness requirements with a 100% coverage with supertrace on a 128M memory machine. If the number of blanks is limited (e.g. to 4), full search (without supertrace) may be done. Attempts to check these properties on models of further levels is possible but the coverage decreases dramatically from the second or third blank.
From level 3 the weak liveness requirement and the machine collision avoidance requirements may be verified as well as the properties expressing that the blanks are dropped only in safe regions. With level 4 it is possible to check that the blanks are kept sufficiently distant and that the element mobility restriction is insured. For all these, the full search is possible. Additionally it is verified that the specification does not contain any unreached code.

I now present some limited excerpts of the Promela models that will be used for showing the verification of some significant requirements.
First, follow the abstract models of the two ends of the opened production cell : the feedbelt and a container which would be placed after the traveling crane. The feedbelt is a one direction moving belt with a motor that may be set to *on* or *off*. At its end a sensor detects the incoming blanks. When a blank arrives at the sensor the belt stops and continues if the next element, the rotary table is ready for accepting it. A blank may be put at the entrance of the feedbelt if it is empty or as soon as the previous blank has reached the sensor, so this belt may contain two blanks at the most. It is modeled with two processes (Fig.3.1).
The production of the blanks is modeled by a process that sends them to the feedbelt ; they are materialized as two-fielded messages, the first field being a blank number modulo *MAX_BLANKS*, the second is a boolean telling if the blank is forged or not. The container accepts all the incoming blanks, it is modeled by process of Fig.3.2.

```
proctype FeedBelt1(chan in1,out){
    byte blanknum;
    do
#if !LIVENESS
    :: in1?blank(blanknum,recvblankforged);
        d_step{
        prevrecvblank=recvblank;
            recvblank=blanknum;
            assert(   recvblank==(prevrecvblank+1)%MAX_BLANKS )};
        out!blank(recvblank,0);
#else
    :: in1?blank(recvblank,recvblankforged);
        out!blank(recvblank,0);
        recvblank=UNDEF;
#endif
    od}

proctype FeedBelt2(chan in,out){
    byte blanknum;bit forg;
    do
    :: in?blank(blanknum,forg);
        out!ready(0,0);
        out!blank(blanknum,forg);
    od}
```

Fig.3.1 — The FeedBelt

```
proctype Container(chan in){
    do
#if !LIVENESS
    :: in?ready(0,0);
        prevworkblank=workblank;
        in?blank(workblank,workblankforged);
        assert(workblank==(prevworkblank+1)%MAX_BLANKS);
#else
    :: in?ready(0,0);
        in?blank(workblank,workblankforged);
#endif
    od}
```

Fig.3.2 — The Container

The verification of the strong liveness requirement, may be achieved thanks to two safety properties and a progress property expressed in LTL[14]. The first safety property means that there must not be any blank duplication nor blank loss, the second one means that the sequence of number of the sent blanks need to be preserved when they are received by the container. These two properties may be easily captured with the two assertions appearing in Fig.3.1 and 3.2. The progress requirement is verified with the following :

$\Box(P(0)\Rightarrow\Diamond Q(0)) \wedge \Box(P(1)\Rightarrow\Diamond Q(1)) \wedge \ldots \wedge \Box(P(m)\Rightarrow\Diamond Q(m))$ or equivalently

$\Box((P(0)\Rightarrow\Diamond Q(0)) \wedge (P(1)\Rightarrow\Diamond Q(1)) \wedge \ldots \wedge (P(m)\Rightarrow\Diamond Q(m)))$

where

$P(b) = ((recvblank=b) \wedge \neg recvblankforged)$

$Q(b) = ((workblank=b) \wedge workblankforged)$

It is enough that *m* equals 7 if *MAX_BLANKS* is equal to 8. Fig.3.3 shows the related Promela definitions.

```
#define P(b) ((recvblank==b) && !recvblankforged)
#define Q(b) ((workblank==b) && workblankforged)
/* ![](P(0)-><>Q(0))/\((P(1)-><>Q(1))/\ ... /\ (P(7)-><>Q(7)))) */
never{
    do
    :: skip
    :: (P(0)&&!Q(0)) -> goto accept0
    ...
    :: (P(7)&&!Q(7)) -> goto accept7
    od;
accept0:
    do
    :: !Q(0)
    od;
...
accept7:
    do
    :: !Q(7)
    od}
```

Fig.3.3 — The Strong Liveness Requirement

I now make explicit the safety requirement related to the absence of collision between the press and the robot. A scenario is built with the robot and the press detailed models, and two processes that very abstractly emulate the output of the table to the robot and the input of the deposit belt from the robot. The safety property is verified with the LTL property :

¤ *(pressing ⇒ (¬arm1_in_press ∧ ¬arm2_in_press))*

Where *pressing*, *arm1_in_press* and *arm2_in_press* are variables set respectively by the press and the robot in appropriate situations.

```
/* specification */
proctype TableH(chan in,out)
    { do  :: in?go_right;out!at_right;  :: in?go_left;out!at_left; od}

/* detailed model */
proctype TableH(chan in,out){
    byte command, ack;
    do
    ::  in?command;
        d_step{
            A7=(command==go_right -> ON : REV);
            if /*.....................Environment.....................*/
            :: (S9==TA_LEFT && A7==ON)   -> S9=TA_RIGHT;
            :: (S9==TA_RIGHT && A7==REV) -> S9=TA_LEFT;
               /* mobility restriction */
            :: (S9==TA_LEFT && A7==REV)  -> assert(FALSE);
            :: (S9==TA_RIGHT && A7==ON)  -> assert(FALSE);
            fi; /*.....................................................*/
            if
            :: (S9==TA_LEFT)  -> A7=STOP;ack=at_left;
            :: (S9==TA_RIGHT) -> A7=STOP;ack=at_right;
            fi};
        out!ack;
    od}
```

Fig.3.5 — Table Horizontal Movements

I conclude this section with the process in charge of the execution of the table horizontal movements. It will show how the safety requirements regarding machine restriction mobility are defined and how the hypothesis of the controller being fast enough is actually modeled. The table may move horizontally in 2 directions thanks to a motor (*A7*) that may be set to *on*, *rev* or *off*. Two positions are significant : *left* and *right* given by a sensor (*S9*). The process *TableH* accepts commands *go_right, go_left* on channel *in* and reports their completion on channel *out*. Fig.3.5 shows the trivial specification of *TableH* and its implementation : when a command is received the motor is set in the appropriate direction, then the part of the environment related to *TableH* is given the control and evolves according to possible physical changes. For instance when the Table is at *left* and the motor is *on*, then the Table will reach the right position. Then *TableH* gets back the control and reports the command completion. The attempts to exceed the physical limits are naturally expressed in the environment evolution possibilities, for instance if the table is at *left* and the motor is *rev*. One may notice how the environment specification is separated from that the controller's and we will see in the implementation section how its suppression will lead to the actual controller. The fact that the controller and the corresponding environment are merged in a single process may surprise at first. This is just a commodity for reducing the verification complexity because they could have been put in separated processes communicating through rendezvous for exchanging the motor orders and the sensors values (this is actually the way it is expressed in the CCS models).
In both cases what is important is that the environment is constrained to evolve only under the supervision of the controller. We will see in the implementation section that this corresponds to the synchronous option for driving the graphical simulation of the production cell, thus preserving the hypothesis that the controller is fast enough with regard to the plant.

4. Implementation

I now outline how the implementations were derived from the Promela models.
The physical production cell in reduction (Fischer Technik) or the graphical simulation written by FZI in Tcl/Tk[15] can be driven with the same protocol. On UNIX, commands are sent via *stdout*, and sensor values are read from *stdin* after having sent the command *"get_status"*. In this implementation a particular process called *Sampler* regularly samples the environment and forwards the interesting values of the sensors to the concerned controllers (e.g. *S9right*, *S9left*). The use of the special command *"react"* and the option *"-snc"* for running the simulation in synchronous mode, insures that whatever the speed of the controller it will not loose any sensor values.

Synchronous C++, is an extension of C++ that is also very similar to Ada to given extents. Fig.4.1 shows the implementation of TableH. A process template is declared as an active class and instances are created with *new* as instances of usual C++ classes. Synchronous rendezvous are declared as methods of an active class. They correspond to Ada task entries and may be used within a *select* statement for awaiting multiple events with the *accept* clause. This may be achieved on a local rendezvous but also with a call to a rendezvous of an other active object, in that sense is Synchronous C++ more symmetrical than Ada.
In Synchronous C++ it is possible to structure the program as a network of nested active objects but there is no dedicated support to set up connexions between active objects ; the consequence is that the programs tend to have a flat structure and one has to add extra statements for initializing the objects so that they know of each other if needed. This is sometimes tedious and leads to mixing up the program architecture with its behavior.

Darwin allows for expressing the architecture of a system as an interconnected network of component each of which may be itself a network of components. The interface of each component is a collection of input/output ports. Only for the leaves of such a system does one need to express the behavior. This is done with Regis which is C++ with some predefined classes for managing the communication ports. Regis globally allows the same features as Synchronous C++, in particular it possesses an equivalent to the *select* statement but in a more hand-coded way than in Synchronous C++, besides it is possible to await for multiple events

in reception but not in emission, whereas both are allowed in Synchronous C++. The implementation of TableH in Regis is similar to the one in Synchronous C++. Fig.4.2 shows the Darwin interface of TableH and the way TableH is interconnected to the whole controller.

```
active class TableH{
    Environment   *env;
    Table              *ta;
    command       com;
    @TableH(){
        position ack;
        ...
        for(;;){
            accept Command;
            switch(com){
                case go_right:
                    env->Putcommand("table_right");
                break;
                case go_left:
                    env->Putcommand("table_left");
                    break;
            };
            select{
                accept S9right;
                env->Putcommand("table_stop_h");
                tack=at_right;
             ||
                accept S9left;
                env->Putcommand("table_stop_h");
                tack=at_left;
            };
            ta->AckFromTAH(ack);
        };
    };
public:
    void Command(command com) {this->com=com};
    void S9right() {};
    void S9left() {}; …
};
```

Fig.4.1 — TableH Implementation in Synchronous C++

Conclusions

This work shows that there are great benefit in using SPIN for process control with some property to be verified in mind, and that it fits well within a top-down design approach. There are lots of similarities between the verification of this case study and that of a protocol. First the problem was structured in layers of decreasing abstraction and second the strong liveness requirement verification was inspired from that of a sliding window protocol studied previously. Since the cell is considered in its normal functioning mode (no element breakdown or transmission failure), the problem was further simplified and no fairness consideration had to be taken into account.

The whole specification was written with only synchronous communications and its translation into a concurrent programming language was quite easy, the distance between both being very small. As forecast, no unforseen events occurred when running the implementation

since many design inconsistencies such as deadlocks, safety violations or cycles had been discarded during the specification phase. The specification and implementation are quite readable and the whole approach should be accessible to any engineer. The obtained models are easily modifiable and generic enough to be reused in similar problems. Regarding the requirements, they were almost all specified in LTL. SPIN being now doted with a LTL translator, it is possible to remain at this high specification level. The new debugging facilities of Xspin (message charts, hypertext correspondence, breakpoints, ...) where of great aid and I hope that SPIN will keep on growing, in particular toward verification of process equivalencies.

Some more details about this case study (reports, models, demos and code) may be found by FTP at *ltidec1.epfl.ch /pub* or by WWW at *http://ltiwww.epfl.ch/~cattel/prodcell.html*.

```
/* TableH.dw */
component TableH {
    provide
    in <port command>;
    S9left <port int>;
    S9right  <port int>;
    require
    out  <port position>;
    com <port action>;
}
/* Controller.dw */
#include "Table.dw"
#include "TableH.dw"
…
component Controller (int blanks) {
    require
    com <port action>;
    provide
    S9left <port int>;
    S9right  <port int>; …
    inst
    ta : Table;
    tah : TableH; …
    bind
    S9left    -- tah.S9left;
    S9right   -- tah.S9right;
    ta.Hin    -- tah.in;
    tah.out   -- ta.Hout;
    tah.com   -- com; …
}
```

Fig.4.2 — TableH Implementation Architecture in Darwin

Acknowledgments

I thank Adil Mesbahi of IIE, Evry who started this case study in the context of his engineer diploma at LTI and I am very grateful to Claude Petitpierre and Christoph Sprenger of LTI for their useful ideas and comments.

References

1. Holzmann G.J., *What's new in SPIN version 2*, AT&T Bell Laboratories, May 1995.
2 Holzmann G.J., Design and Validation of Computer Protocols, 512 pgs, ISBN 0-13-539925-4, Publ. Prentice Hall, (c) 1991 AT&T Bell Laboratories.
3. Holzmann G.J., *Design and validation of protocols : a tutorial,* Computer Networks, 25(9), April 93, pp. 981-1017.
4. Cattel, T. *Modelling and verification of a multiprocessor realtime OS kernel*. Proceedings of the Seventh International Conference on Formal Description Techniques, Berne, Switzerland. Oct. 4-7, 1994. In press. Nat. Research Council of Canada 38309.
5. Halbwachs N. *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.
6. Lindner T., *Production Cell Case study, Task description*, ZFI, Karlsruhe, 1993.
7. Lewerentz C., Lndner T., *Formal Development of Reactive Systems, Case Study Production Cell,* Lecture Notes in Computer Sciences 891, Spinger Verlag, 1994.
8. G. Caal, A. Divin, C. Petitpierre, *Active Objects: a Paradygm forCommunications and Event Driven Systems*, Globecom'94, San Francisco.
9. Magee J., Dulay, N., Eisenbach S, Kramer J., *Specifying Distributed Software Architectures*, to appear at the 5th Software Engineering Conference, ESEC'95, Barcelona, September 1995.
10. Keng N., Kramer J., Magee J., Dulay N., *The Software Architect's Assistant - A visual Environment for Distributed Programming*, Proc. of Hawaii International Conf. on System Sciences, January 1995.
11. Erdogmus H., Johnston R., Cleary C., Compositional verification based on relation checking in SPIN, submitted at Forte 95, Montreal October 1995.
12. Milner R. : Communication and Concurrency, Prentice Hall International, 1989.
13. Cleaveland R., Parrow J., Steffen B., *TheConcurrency Workbench : A semantics-Based Tool for the Verification of Concurrent Systems*, ACM TOPLAS, Vol. 5, No 1., January 1993, pp. 36-72.
14. Manna Z., Pnueli A., The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer-Verlag, 1992.
15. Ousterhout J., *Tcl and the Tk toolkit*, Addison-Wesley, 1994.