Journal of Universal Computer Science, vol. 3, no. 5 (1997), 666-688
submitted: 20/12/96, accepted: 20/5/97, appeared: 28/5/97

Refining an ASM Specification of the Production Cell to C++ Code

Luca Mearelli
(Università di Pisa, Italy
luca@tex.odd.it)

Abstract: We present here the transformation to C++ code of the refined ASM model for the production cell developed in the paper "Integrating ASMs into the Software Development Life Cycle" (see this volume) which serves as program documentation. This implementation is a refinement step and produces code which has been validated through extensive experimentation with the production cell simulator of FZI Karlsruhe.
Key Words: Programming Techniques, Stepwise Refinement,Code Documentation, Code Inspection, Abstract State Machines.
Category: D.1, D.1.3, D.1.5, D.2.1, D.2.4, D.2.5

1 Introduction

In the paper "Integrating ASMs into the Software Development Life Cycle" (see this volume) we developed a formal specification and the basic design for a production cell controller. We complete this work here by translating the refined ASM model RefCELL into executable code, namely a C++ program, whose structure directly reflects the ASM functions and rules, besides handling the input of the sensor values and the output of the commands to the actuators.

Some additional code simulates a (piece inserting) operator, and processes the error messages coming from the simulation.

See [2] for a detailed description of the simulation environment.

2 The C++ code

2.1 The Main Program

The file control.cc contains the main loop of the controller program.

In the loop the function `AskNewStatus()` is called to ask the environment new sensor values which are read (in the order) as they are given by the simuation. Each agent reacts to these new sensor values and outputs the actuator commands. At last the error messages are collected and the (function simulating a) piece inserting operator is called. For the correctness of the underlying sequentialization of the distributed ASM we refer to section 5 of [3].

```
# include "control.h"

cElist        Errors;

cFeedBelt     FeedBelt;
cElevRotTable ElevatingRotaryTable;
```

```
cRobot        Robot;
cPress        Press;
cDepositBelt  DepositBelt;
cTravCrane    TravellingCrane;

int main()
{
  loop
    {
      AskNewStatus();
      cin>> Press
         >> Robot
         >> ElevatingRotaryTable
         >> TravellingCrane
         >> FeedBelt
         >> DepositBelt
         >> Errors;
      Oper();
    };

}
```

2.2 The Header File

Here is most of the code for the controller (e.g. the classes controlling the machines are defined here).

We have some include and a definition for the infinite loop to be used in the main program.

```
#include <iostream.h>
#include <String.h>
#include <SLList.h>
#define loop for(;;)
```

We define the following arrays of commands for the actuators, as defined for the simulation in [2]. Collecting the command strings here, and later accessing them only through the arrays, we can easily cope with possible variations (to affect all the program we would need only to change the string in the array definition).

```
// Feed Belt
const char* FB_com[] ={ "belt1_start", "belt1_stop" };

// Deposit Belt
const char* DB_com[] ={ "belt2_start", "belt2_stop" };

// Elevating Rotary Table
const char* ERT_rot[] =
            { "table_right", "table_left", "table_stop_h" };
```

```
const char* ERT_ver[] =
            { "table_upward", "table_downward", "table_stop_v" };

// Robot
const char* R_rot[] =
            { "robot_right", "robot_left", "robot_stop" };
const char* R_a1Ext[] =
            { "arm1_forward", "arm1_backward", "arm1_stop" };
const char* R_a2Ext[] =
            { "arm2_forward", "arm2_backward", "arm2_stop" };
const char* R_a1Mag[] =
            { "arm1_mag_on", "arm1_mag_off" };
const char* R_a2Mag[] =
            { "arm2_mag_on", "arm2_mag_off" };

// Travelling Crane
const char* TC_ver[] =
            { "crane_lift", "crane_lower", "crane_stop_v"};
const char* TC_hor[] =
            { "crane_to_belt2", "crane_to_belt1", "crane_stop_h"};
const char* TC_mag[] =
            { "crane_mag_on", "crane_mag_off" };

// Press
const char* P_com[] =
            { "press_upward", "press_downward", "press_stop" };
```

We define also some enumerated types for motor states, such that we can use them, instead of some index, to access the corresponding commands.

```
enum OnOff { on, off };
enum ExtMov { extend, retract, idlee };
enum VertMov { up, down, idlev };
enum RotMov { clockwise, counterclockwise, idler };
enum TChmov { toDepBelt, toFeedBelt, idleh };
```

We define here the constants for the model, they are significant values for the sensors and are defined in the documentation for the simulation.

We use for these constants the same names as in the ASM models.

```
// Elev. Rot. Table (table rotation) constants
const float minERTRot = 0.0;
const float maxERTRot = 50.0;

// Robot Arms Extensions
const float OverTable       = 0.5208;
const float Arm2IntoPress   = 0.7971;
const float OverDepBelt     = 0.5707;
const float Arm1IntoPress   = 0.6458;
```

```
const float retracted       = 0.0;

// Robot Base rotation angles
const float Arm1ToTable   = 50.0;
const float Arm2ToPress   = 35.0;
const float Arm2ToDepBelt = -45.0;
const float Arm1ToPress   = -90.0;

// Travelling Crane (magnet height) constants
const float OnDepositBelt = 0.9450;
const float OnFeedBelt    = 0.6593;
const float SafeDistanceFromFeedBelt    = 0.0;
```

Here are some interface functions used to transmit information among the modules, as they are defined in the specification paper.

They are global functions because we need to share them among the classes, but we are sure that no conflicts will arise in their use since we have proven the consistency of the updates to interface functions (see chap. 3 of [3]).

```
bool TableLoaded;
bool PressLoaded;
bool FeedBeltFree;
bool DepositBeltReadyForLoading;
bool PieceAtDepositBeltEnd;
bool TableInLoadPosition;
bool TableInUnloadPosition;
bool PressInUnloadPosition;
bool PressInLoadPosition;
bool TravellingCraneLoaded;
```

The AskNewStatus function is used to require new sensors readings (the `get_status` command is defined in [2]).

```
void AskNewStatus()
{
  cout<< "get_status"<< endl << flush;
};
```

The class cModule: ASM module abstract base class, gives the structure to all the classes implementing the modules from the ASM specification.

We will define a class for each ASM module, deriving them from cModule, they will define the function `Where()` to implement the ASM macro definition, and the function `Rules()` to implement the update rules of that module.

```
class cModule
{
 protected:
  virtual void Where() = 0;
  virtual void Rules() = 0;

};
```

Template classes are used to make the code of the controller look as similar as possible to that of the ASM model.

The cActuator class template embeds the general actuator abstraction. We overload the assignment operator such that, to trigger an action, we just need to "assign" the right value to the object representing it. The "==" and "()" operators are overloaded to allow us to check which command is executing the actuator (i.e. the last command issued) as we do in the ASM models.

```
template<class T>
class cActuator
{
  T          state;
  String*    commands;
  int        num;

 public:

   cActuator (const char* v[], const int n)
     { commands = new String[n];
       for (int i =0;i<n;i++) { commands[i] = v[i];};
       num = i;
     };

   ~cActuator ()
     { delete commands;
     };

   T operator() ()
    { return state; };

   cActuator& operator=( T st )
     { if ((int)st < num)
         { state = st;
            cout<< commands[st]<< endl<< flush;
         };
       return *this;
     };

   friend int operator==(const cActuator &ac, const T st )
     { return (ac.state == st);};
};
```

The cSensor class template embed the idea of sensor with its operations. The >> operator is overloaded to be used to retrieve the sensor values from an input stream. Changes in the way the values have to be read (e.g. in the characters separating them) would be reflected in changes to this class.

```
template<class T>
class cSensor
{
```

```
  T    value;

 public:

  cSensor();
  cSensor( T val ) {value = val;};

  friend istream& operator>> (istream& is, cSensor& s)
    { return  is>> s.value; };

  friend int operator== (const cSensor& s, const T val)
    { return (s.value == val);};

  T operator() ()
    { return value; };
};
```

The Oper function embeds the behaviour of an operator which tries to insert pieces into the system following the Insertion Priority Assumption.

```
void Oper()
{if (FeedBeltFree && (!TravellingCraneLoaded))
   { cout<<  "blank_add"<< endl <<flush;
     FeedBeltFree = false;
   };
}
```

The class cElist implements a list of errors represented as integers, as they are given by the simulation program.

```
class cElist : public SLList<int>
{
 public:

  cElist() : SLList<int>() {};

  friend istream& operator>> (istream& is, cElist& el)
    {
      char c;
      int  k;

      is>> c>> c;

      while (c != '}')
        { is.putback(c);
          is>> k;
          if (k) { el.append(k);};
          is>> c;
```

```
        };

      return is;
    };

  friend ostream& operator<< (ostream& os, cElist& el)
    {
      if (el.empty()) {cout<< "Empty list!"<< endl;}
      else
        { cout<< "List Content: ";
          Pix p;
          p = el.first();
          while (p)
             { cout<< el(p)<< " ";
               el.next(p);
             };
          cout<< endl;
        };
    };
```

Then we have the classes implementing various cell elements, i.e. implementing the ASM modules for the corresponding machine part. The constructor is used to initialize the module status functions, including the globals related to the corresponding machine. It is easy to recognize in the code all the elements of the RefCELL module (functions, macro definitions, rules). Therefore we refer to [3] for further comments.

2.2.1 The Feed Belt

```
class cFeedBelt : public cModule
{
  bool Delivering;
  bool NormalRun;
  bool CriticalRun;
  bool Stopped;
  bool TableReadyForLoading;

  // The Sensor
  cSensor<bool> PieceInFeedBeltLightBarrier;

  // The Actuator
  cActuator<OnOff> FeedBeltMot;

  void Where()
    { NormalRun = (FeedBeltMot == on)&&(!Delivering);
      CriticalRun = (FeedBeltMot == on)&&Delivering;
      Stopped = FeedBeltMot == off;
      TableReadyForLoading = TableInLoadPosition &&
                   (!TableLoaded);
    };
```

```
  void Rules()
    {
      if (NormalRun && PieceInFeedBeltLightBarrier())
        {FeedBeltFree = true;
         if (TableReadyForLoading)
            { Delivering = true;}
         else { FeedBeltMot = off;};};

      if (CriticalRun && (! PieceInFeedBeltLightBarrier()))
        { Delivering = false;
          TableLoaded = true;};

      if (Stopped && TableReadyForLoading)
        { FeedBeltMot = on;
          Delivering = true;};
    };

 public:

  cFeedBelt()
    : PieceInFeedBeltLightBarrier(false), FeedBeltMot(FB_com, 2)
    { FeedBeltFree = true;
      FeedBeltMot = on;
      Delivering = false;
      NormalRun = true;
      CriticalRun = false;
      Stopped = false;
      TableReadyForLoading = TableInLoadPosition &&
                                          (!TableLoaded);
    };

  friend istream& operator>>(istream& is, cFeedBelt& fb)
    { is>> fb.PieceInFeedBeltLightBarrier;

      fb.Where();
      fb.Rules();

      return is;
    };
};
```

2.2.2 The Elevating Rotary Table

```
class cElevRotTable : public cModule
{
  bool StoppedInLoadPosition;
  bool StoppedInUnloadPosition;
  bool MinRotation;
  bool MaxRotation;
```

```
bool NoMovement;

// The Sensors
cSensor<bool>  TopPosition;
cSensor<bool>  BottomPosition;
cSensor<float> TableRotation;

// The Actuator
cActuator<VertMov> TableElevationMot;
cActuator<RotMov>  TableRotationMot;

void Where()
  {
    MinRotation = (TableRotation == minERTRot);
    MaxRotation = (TableRotation == maxERTRot);
    NoMovement = (TableElevationMot == idlev) &&
                 (TableRotationMot == idler);
    StoppedInLoadPosition = BottomPosition() && MinRotation &&
                            NoMovement;
    TableInLoadPosition = StoppedInLoadPosition;
    StoppedInUnloadPosition = TopPosition() && MaxRotation &&
                              NoMovement;
    TableInUnloadPosition = StoppedInUnloadPosition;
  };

void Rules()
  {
    if ( StoppedInLoadPosition && TableLoaded )
      { TableElevationMot = up;
        TableRotationMot = clockwise;
      };

    if ((TableElevationMot == up) && TopPosition())
      { TableElevationMot = idlev; };

    if ((TableRotationMot == clockwise) && MaxRotation)
      { TableRotationMot = idler; };

    if ( StoppedInUnloadPosition && (!TableLoaded))
      { TableElevationMot = down;
        TableRotationMot = counterclockwise;
      };

    if ((TableElevationMot == down) && BottomPosition())
      { TableElevationMot = idlev; };

    if ((TableRotationMot == counterclockwise) && MinRotation)
      { TableRotationMot = idler; };
  }
```

```
public:

 cElevRotTable()
   : BottomPosition(true), TopPosition(false),
     TableRotation(0.0), TableElevationMot(ERT_ver, 3),
     TableRotationMot(ERT_rot, 3)
   {
     TableLoaded = false;
     TableElevationMot = idlev;
     TableRotationMot = idler;
     StoppedInLoadPosition = true;
     TableInLoadPosition = true;
     StoppedInUnloadPosition = false;
     TableInUnloadPosition = false;
     MinRotation = true;
     MaxRotation = false;
     NoMovement = true;
   };

 friend istream& operator>>(istream& is, cElevRotTable& ert)
   { is>>ert.BottomPosition
       >>ert.TopPosition
       >>ert.TableRotation;

     ert.Where();
     ert.Rules();

     return is;
   };
};
```

2.2.3 The Robot

The robot, as the simulation starts, is in a different position from that assumed in the RefCELL model so that we have to take care of the "preprocessing" steps needed to insert the first blank in the cell through the FeedBelt. This explains the additional lines of code for the robot with respect to the rules in RefCELL (see [3]). We leave as an easy exercise to write a simple extension to the ASMs describing the Robot "boot sequence", and to extend the proofs to the initial steps.

```
class cRobot : public cModule
{
  bool boot;

  bool ArmsRetracted;
  bool RobotIdle;

  bool WaitingInUnloadTablePos;
  bool WaitingInUnloadPressPos;
```

```
bool WaitingInLoadDepBeltPos;
bool WaitingInLoadPressPos;

bool ExtendingArm1ToTable;
bool ExtendingArm2ToPress;
bool ExtendingArm2ToDepBelt;
bool ExtendingArm1ToPress;

bool ExtendedOverTable;
bool ExtendedArm2IntoPress;
bool ExtendedOverDepBelt;
bool ExtendedArm1IntoPress;

bool RetractingArm1FromTable;
bool RetractingArm2FromPress;
bool RetractingArm2FromDepBelt;
bool RetractingArm1FromPress;

bool MovingToUnloadPressPos;
bool MovingToLoadDepBeltPos;
bool MovingToLoadPressPos;
bool MovingToUnloadTablePos;

bool MovingToInitialPosition;

bool UnloadTablePosReached;
bool UnloadPressPosReached;
bool LoadDepBeltPosReached;
bool LoadPressPosReached;

bool InitialPosReached;

bool TableReadyForUnloading;
bool PressReadyForUnloading;
bool PressReadyForLoading;


// The Sensors
cSensor<float> Arm1Ext;
cSensor<float> Arm2Ext;
cSensor<float> Angle;

// The Actuators
cActuator<RotMov> RotationMot;
cActuator<ExtMov> Arm1Mot;
cActuator<ExtMov> Arm2Mot;
cActuator<OnOff> Arm1Mag;
cActuator<OnOff> Arm2Mag;

void Where()
```

```
{
  ArmsRetracted =
     ((Arm1Ext == retracted)&&(Arm2Ext == retracted));

  RobotIdle = ((RotationMot == idler)&&(Arm1Mot == idlee)&&
               (Arm2Mot == idlee));

  WaitingInUnloadTablePos =
     ((Angle == Arm1ToTable) && ArmsRetracted &&
       RobotIdle &&
      (Arm1Mag == off) && (Arm2Mag == off));

  WaitingInUnloadPressPos =
     ((Angle == Arm2ToPress) && ArmsRetracted &&
       RobotIdle &&
      (Arm1Mag == on) && (Arm2Mag == off));

  WaitingInLoadDepBeltPos =
     ((Angle == Arm2ToDepBelt) && ArmsRetracted &&
       RobotIdle &&
      (Arm1Mag == on) && (Arm2Mag == on));

  WaitingInLoadPressPos =
     ((Angle == Arm1ToPress) && ArmsRetracted &&
       RobotIdle &&
      (Arm1Mag == on) && (Arm2Mag == off));

  ExtendingArm1ToTable =
    ((Angle == Arm1ToTable) && (Arm1Mot == extend));

  ExtendingArm2ToPress =
    ((Angle == Arm2ToPress) && (Arm2Mot == extend));

  ExtendingArm2ToDepBelt =
    ((Angle == Arm2ToDepBelt) && (Arm2Mot == extend));

  ExtendingArm1ToPress =
    ((Angle == Arm1ToPress) && (Arm1Mot == extend));

  ExtendedOverTable =
    ((Angle == Arm1ToTable) && (Arm1Ext == OverTable) &&
     RobotIdle);

  ExtendedArm2IntoPress =
    ((Angle == Arm2ToPress) && (Arm2Ext == Arm2IntoPress) &&
     RobotIdle);

  ExtendedOverDepBelt =
    ((Angle == Arm2ToDepBelt) && (Arm2Ext == OverDepBelt) &&
     RobotIdle);
```

```
ExtendedArm1IntoPress =
  ((Angle == Arm1ToPress) && (Arm1Ext == Arm1IntoPress) &&
   RobotIdle);

RetractingArm1FromTable =
  ((Angle == Arm1ToTable) && (Arm1Mot == retract));

RetractingArm2FromPress =
  ((Angle == Arm2ToPress) && (Arm2Mot == retract));

RetractingArm2FromDepBelt =
  ((Angle == Arm2ToDepBelt) && (Arm2Mot == retract));

RetractingArm1FromPress =
  ((Angle == Arm1ToPress) && (Arm1Mot == retract));

MovingToUnloadPressPos =
   (ArmsRetracted && (Arm1Mot == idlee) &&
   (Arm2Mot == idlee) &&
   (RotationMot == counterclockwise) &&
   (Arm1Mag == on) && (Arm2Mag == off));

MovingToLoadDepBeltPos =
   (ArmsRetracted && (Arm1Mot == idlee) &&
   (Arm2Mot == idlee) &&
   (RotationMot == counterclockwise) &&
   (Arm1Mag == on) && (Arm2Mag == on));

MovingToLoadPressPos =
   (ArmsRetracted && (Arm1Mot == idlee) &&
   (Arm2Mot == idlee) &&
   (RotationMot == counterclockwise) &&
   (Arm1Mag == on) && (Arm2Mag == off));

MovingToUnloadTablePos =
  (ArmsRetracted && (Arm1Mot == idlee) &&
  (Arm2Mot == idlee) && (RotationMot == clockwise) &&
  (Arm1Mag == off) && (Arm2Mag == off));

UnloadPressPosReached = (Angle == Arm2ToPress);

LoadDepBeltPosReached = (Angle == Arm2ToDepBelt);

LoadPressPosReached = (Angle == Arm1ToPress);

UnloadTablePosReached = (Angle == Arm1ToTable);

InitialPosReached = ((Angle == Arm1ToTable) &&
                     (ArmsRetracted) && (RobotIdle));
```

```
      TableReadyForUnloading =(TableInUnloadPosition &&
                               TableLoaded);

      PressReadyForUnloading =(PressInUnloadPosition &&
                               PressLoaded);

      PressReadyForLoading = (PressInLoadPosition &&
                              (!PressLoaded));

    };

  void Rules()
    {
      if (boot)
{
  if (MovingToInitialPosition)
            {
              if ( Angle == Arm1ToTable )
                { RotationMot = idler; };
              if ( Arm1Ext == retracted )
                { Arm1Mot = idlee; };
              if ( Arm2Ext == retracted )
                { Arm2Mot = idlee; };
              if ( InitialPosReached )
                { MovingToInitialPosition = false; };
    }
         else{

            if (WaitingInUnloadTablePos &&
                TableReadyForUnloading)
              { Arm1Mot = extend; };

            if (ExtendingArm1ToTable && (Arm1Ext == OverTable))
              { Arm1Mot = idlee;
                Arm1Mag = on; };

            if (ExtendedOverTable && (Arm1Mag == on))
              { Arm1Mot = retract; };

            if (RetractingArm1FromTable &&
                (Arm1Ext == retracted))
            { Arm1Mot = idlee;
              RotationMot = counterclockwise;
              TableLoaded = false; };

            if (MovingToLoadPressPos && LoadPressPosReached)
              { RotationMot = idler; };

            if (WaitingInLoadPressPos && PressReadyForLoading)
```

```
              { Arm1Mot = extend; };

            if (ExtendingArm1ToPress &&
                (Arm1Ext == Arm1IntoPress))
              { Arm1Mot = idlee;
                Arm1Mag = off; };

            if (ExtendedArm1IntoPress && (Arm1Mag == off))
              { Arm1Mot = retract; };

            if (RetractingArm1FromPress &&
                (Arm1Ext == retracted))
              { Arm1Mot = idlee;
                RotationMot = clockwise;
                PressLoaded = true; };

            if (MovingToUnloadTablePos && UnloadTablePosReached)
              { RotationMot = idler;
                boot = false; };
  };
}
      else
{
          if (WaitingInUnloadTablePos && TableReadyForUnloading)
            { Arm1Mot = extend; };

          if (ExtendingArm1ToTable && (Arm1Ext == OverTable))
            { Arm1Mot = idlee;
              Arm1Mag = on; };

          if (ExtendedOverTable && (Arm1Mag == on))
            { Arm1Mot = retract; };

          if (RetractingArm1FromTable && (Arm1Ext == retracted))
            { Arm1Mot = idlee;
              RotationMot = counterclockwise;
              TableLoaded = false; };

          if (MovingToUnloadPressPos && UnloadPressPosReached)
            { RotationMot = idler; };

          if (WaitingInUnloadPressPos && PressReadyForUnloading)
            { Arm2Mot = extend; };

          if (ExtendingArm2ToPress && (Arm2Ext == Arm2IntoPress))
            { Arm2Mot = idlee;
              Arm2Mag = on; };

          if (ExtendedArm2IntoPress && (Arm2Mag == on))
           { Arm2Mot = retract; };
```

```
            if (RetractingArm2FromPress && (Arm2Ext == retracted))
              { Arm2Mot = idlee;
                RotationMot = counterclockwise;
                PressLoaded = false; };

            if (MovingToLoadDepBeltPos && LoadDepBeltPosReached)
              { RotationMot = idler; };

            if (WaitingInLoadDepBeltPos &&DepositBeltReadyForLoading)
              { Arm2Mot = extend; };

            if (ExtendingArm2ToDepBelt && (Arm2Ext == OverDepBelt))
              { Arm2Mot = idlee;
                Arm2Mag = off; };

            if (ExtendedOverDepBelt && (Arm2Mag == off))
              { Arm2Mot = retract; };

            if (RetractingArm2FromDepBelt &&
                (Arm2Ext == retracted))
              { Arm2Mot = idlee;
                RotationMot = counterclockwise;
                DepositBeltReadyForLoading = false; };

            if (MovingToLoadPressPos && LoadPressPosReached)
              { RotationMot = idler; };

            if (WaitingInLoadPressPos && PressReadyForLoading)
              { Arm1Mot = extend; };

            if (ExtendingArm1ToPress && (Arm1Ext == Arm1IntoPress))
              { Arm1Mot = idlee;
                Arm1Mag = off; };

            if (ExtendedArm1IntoPress && (Arm1Mag == off))
              { Arm1Mot = retract; };

            if (RetractingArm1FromPress && (Arm1Ext == retracted))
              { Arm1Mot = idlee;
                RotationMot = clockwise;
                PressLoaded = true; };

            if (MovingToUnloadTablePos && UnloadTablePosReached)
              { RotationMot = idler; };
};
    };

 public:
```

```
cRobot()
  : Arm1Ext(1.0), Arm2Ext(1.0), Angle(0.0),
    RotationMot(R_rot, 3), Arm1Mot(R_a1Ext, 3),
    Arm2Mot(R_a2Ext, 3), Arm1Mag(R_a1Mag, 2),
    Arm2Mag(R_a2Mag, 2)
  {
    RotationMot = clockwise;
    Arm1Mot = retract;
    Arm2Mot = retract;
    Arm1Mag = off;
    Arm2Mag = off;

    ArmsRetracted = false;
    RobotIdle = true;
    WaitingInUnloadTablePos = false;
    WaitingInUnloadPressPos = false;
    WaitingInLoadDepBeltPos = false;
    WaitingInLoadPressPos = false;
    ExtendingArm1ToTable = false;
    ExtendingArm2ToPress = false;
    ExtendingArm2ToDepBelt = false;
    ExtendingArm1ToPress = false;
    ExtendedOverTable = false;
    ExtendedArm2IntoPress = false;
    ExtendedOverDepBelt = false;
    ExtendedArm1IntoPress = false;
    RetractingArm1FromTable = false;
    RetractingArm2FromPress = false;
    RetractingArm2FromDepBelt = false;
    RetractingArm1FromPress = false;
    MovingToUnloadPressPos = false;
    MovingToLoadDepBeltPos = false;
    MovingToLoadPressPos = false;
    MovingToUnloadTablePos = false;
    MovingToInitialPosition = true;
    UnloadPressPosReached = false;
    LoadDepBeltPosReached = false;
    LoadPressPosReached = false;
    UnloadTablePosReached = false;
    TableReadyForUnloading =
       (TableInUnloadPosition && TableLoaded);
    PressReadyForUnloading =
       (PressInUnloadPosition && PressLoaded);
    PressReadyForLoading =
       (PressInLoadPosition && (!PressLoaded));
    boot = true;
  };

friend istream& operator>>(istream& is, cRobot& r)
  { is >>r.Arm1Ext >> r.Arm2Ext>> r.Angle;
```

```
      r.Where();
      r.Rules();

      return is;
    };
};
```

2.2.4 The Press

The press has to start in the position where it is unloaded and ready to be loaded by the robot as imposed by the simulation (see section 5 of [3]).

```
class cPress : public cModule
{
  bool OpenForUnloading;
  bool MovingToMiddlePosition;
  bool OpenForLoading;
  bool MovingToTopPosition;
  bool ClosedForForging;
  bool MovingToBottomPosition;
  bool ForgingCompleted;

  // The Sensors
  cSensor<bool> BottomPosition;
  cSensor<bool> MiddlePosition;
  cSensor<bool> TopPosition;

  // The Actuator
  cActuator<VertMov> PressMot;

  void Where()
    {
      OpenForUnloading =
          (BottomPosition() && (PressMot == idlev));
      MovingToMiddlePosition =
          ((!PressLoaded) && (PressMot == up));
      OpenForLoading =
          (MiddlePosition() && (PressMot == idlev));
      MovingToTopPosition =
          (PressLoaded && (PressMot == up));
      ClosedForForging =
          (TopPosition() && (PressMot == idlev));
      MovingToBottomPosition =
          (PressMot == down);
      PressInUnloadPosition = OpenForUnloading;
      PressInLoadPosition = OpenForLoading;
    };

  void Rules()
```

```
    {
      if ( OpenForUnloading && (! PressLoaded) )
        { PressMot = up; };

      if ( MovingToMiddlePosition && MiddlePosition() )
        { PressMot = idlev; };

      if ( OpenForLoading && PressLoaded )
        { PressMot = up; };

      if ( MovingToTopPosition && TopPosition() )
        { PressMot = idlev; };

      if ( ClosedForForging && ForgingCompleted )
        { PressMot = down; };

      if ( MovingToBottomPosition && BottomPosition() )
        { PressMot = idlev; };

    };

 public:

  cPress()
    : BottomPosition(true), MiddlePosition(false),
      TopPosition(false), PressMot(P_com, 3)
    { PressLoaded = false;
      PressMot = idlev;
      ForgingCompleted = true;
      OpenForUnloading = false;
      MovingToMiddlePosition = false;
      OpenForLoading = true;
      MovingToTopPosition = false;
      ClosedForForging = false;
      MovingToBottomPosition = false;
      PressInUnloadPosition = false;
      PressInLoadPosition = true;
    };

  friend istream& operator>>(istream& is, cPress& p)
    { is>>p.BottomPosition >>p.MiddlePosition >>p.TopPosition;

      p.Where();
      p.Rules();

      return is;
    };
};
```

2.2.5 The Deposit Belt

```
class cDepositBelt : public cModule
{
  bool Critical;
  bool NormalRun;
  bool CriticalRun;
  bool Stopped;

  // The Sensor
  cSensor<bool> PieceInDepositBeltLightBarrier;

  // The Actuator
  cActuator<OnOff> DepBeltMot;

  void Where()
    { NormalRun = ((DepBeltMot == on)&&(!Critical));
      CriticalRun = ((DepBeltMot == on)&&Critical);
      Stopped = DepBeltMot == off;
    };

  void Rules()
    {
      if (NormalRun && PieceInDepositBeltLightBarrier())
        { Critical = true;};

      if (CriticalRun &&
         (!PieceInDepositBeltLightBarrier()))
        { Critical = false;
          PieceAtDepositBeltEnd = true;
          DepositBeltReadyForLoading = true;
          DepBeltMot = off;};

      if (Stopped && !PieceAtDepositBeltEnd)
        { DepBeltMot = on;};
    };

 public:

  cDepositBelt()
    : PieceInDepositBeltLightBarrier(false),
      DepBeltMot(DB_com, 2)
    { DepositBeltReadyForLoading = true;
      PieceAtDepositBeltEnd = false;
      DepBeltMot = on;
      Critical = false;
      NormalRun = true;
      CriticalRun = false;
      Stopped = false;
    };
```

```
  friend istream& operator>>(istream& is, cDepositBelt& db)
    { is>> db.PieceInDepositBeltLightBarrier;

      db.Where();
      db.Rules();

      return is;
    };
};
```

2.2.6 The Travelling Crane

```
class cTravCrane : public cModule
{
  bool WaitingToUnloadDepositBelt;
  bool WaitingToLoadFeedBelt;
  bool Stopped;

  // The Sensors
  cSensor<bool>  CraneOverFeedBelt;
  cSensor<bool>  CraneOverDepositBelt;
  cSensor<float> GripperVerticalPos;

  // The Actuators
  cActuator<TChmov> CraneHorizontalMot;
  cActuator<VertMov> CraneVerticalMot;
  cActuator<OnOff> CraneMagnet;

  void Where()
    {
      Stopped = (CraneHorizontalMot == idleh)&&
                 (CraneVerticalMot == idlev);
      WaitingToUnloadDepositBelt =
                (GripperVerticalPos == OnDepositBelt)&&
                (CraneOverDepositBelt()) && (Stopped)&&
                (CraneMagnet == off);
      WaitingToLoadFeedBelt =
                (GripperVerticalPos == SafeDistanceFromFeedBelt)&&
                (CraneOverFeedBelt()) && (Stopped) &&
                (CraneMagnet == on);
    };

  void Rules()
    {
      if ( WaitingToUnloadDepositBelt && PieceAtDepositBeltEnd )
        { CraneMagnet = on; };

      if ( (CraneVerticalMot == idlev) &&
           (GripperVerticalPos == OnDepositBelt) &&
```

```
          (CraneMagnet == on) && CraneOverDepositBelt())
        { CraneVerticalMot = up; };

      if ( (CraneVerticalMot == up) &&
           (GripperVerticalPos == SafeDistanceFromFeedBelt) &&
            CraneOverDepositBelt())
        { CraneVerticalMot = idlev;
          CraneHorizontalMot = toFeedBelt;
          PieceAtDepositBeltEnd = false;
          TravellingCraneLoaded = true; };

      if ( (CraneHorizontalMot == toFeedBelt) &&
           (CraneOverFeedBelt()) )
        { CraneHorizontalMot = idleh; };

      if ( WaitingToLoadFeedBelt && FeedBeltFree )
        { CraneVerticalMot = down; };

      if ( (CraneVerticalMot == down) &&
           (GripperVerticalPos == OnFeedBelt) &&
            CraneOverFeedBelt() )
        { CraneVerticalMot = idlev;
          CraneMagnet = off;
        }

       if ( (CraneVerticalMot == idlev) &&
            (GripperVerticalPos == OnFeedBelt) &&
             CraneOverFeedBelt() &&
            (CraneHorizontalMot == idleh) &&
            (CraneMagnet == off) )
         { FeedBeltFree = false;
           TravellingCraneLoaded = false;
           CraneHorizontalMot = toDepBelt; };

       if ( (CraneHorizontalMot == toDepBelt) &&
              CraneOverDepositBelt() )
         { CraneHorizontalMot = idleh;
           CraneVerticalMot = down; };

       if ( CraneOverDepositBelt() &&
           (CraneVerticalMot == down) &&
           (GripperVerticalPos == OnDepositBelt) )
         { CraneVerticalMot = idlev; };

   };

public:

 cTravCrane()
   : CraneOverFeedBelt(false), CraneOverDepositBelt(true),
```

```
      GripperVerticalPos(0.0), CraneHorizontalMot(TC_hor, 3),
      CraneVerticalMot(TC_ver, 3), CraneMagnet(TC_mag, 2)
    { TravellingCraneLoaded = false;
      CraneHorizontalMot = toDepBelt;
      CraneVerticalMot = idlev;
      CraneMagnet = off;
      Stopped = true;
      WaitingToUnloadDepositBelt = false;
      WaitingToLoadFeedBelt = false;
    };

  friend istream& operator>>(istream& is, cTravCrane& tc)
    { is>>tc.CraneOverDepositBelt
        >>tc.CraneOverFeedBelt
        >>tc.GripperVerticalPos;

      tc.Where();
      tc.Rules();

      return is;
    };
};
```

3 Conclusions

This code was extensively tested with the simulation environment. In all the tests the cell worked conforming to all the requirements, and achieving the maximum throughput of 7 pieces circulating in the system.

The full code and the simulation environment are freely available for experimentation at the Production Cell web site

http://www.fzi.de/prost/projects/production_cell/contributions/ASM.html.

References

1. T. Lindner. *Task Description* . in C. Lewerentz, T. Lindner (eds.), Formal Development Of Reactive Systems. Case Study "Production Cell". Springer LNCS-891. 1995.pp.9-21
2. A. Brauer, T. Lindner. *Simulation.* in C. Lewerentz, T. Lindner (eds.), Formal Development Of Reactive Systems. Case Study "Production Cell". Springer LNCS-891. 1995.pp.273-284
3. E. Börger, L. Mearelli. *Integrating ASMs into the Software Development Life Cycle* . This volume.