Formal Verification of a Coordinated Atomic Action Based Design[1]

E. Canver
Abteilung Künstliche Intelligenz
Universität Ulm

Ulmer Informatik-Berichte
Nr. 98-05
March 27, 1998

Abstract

Coordinated atomic actions (CAAs) have been used in a semi-formal way for the design of the production cell case study. This paper presents a formal specification and verification of the production cell building on this design. However, this report is not intended to present yet another formalization of the production cell case study but rather as an approach to formalizing a CAA based system design in order to formally verify its properties.

Each CAA is modeled as an atomic state transition characterized by its pre- and postconditions. In order for such transitions to become enabled, conditions are formalized requiring all associated roles to be activated. Activation of roles is performed by controllers, which are again modeled in terms of state transitions. The state space of the production cell can be viewed as being finite; hence, the production cell is specified as a finite state transition system and the formal verification of the CAA-design is carried out using model-checking.

[1]This work has partly been funded by the Esprit Long Term Research Project 20072 "Design for Validation"

Contents

Chapter 1

Introduction

Coordinated atomic actions (CAA) [RRS+97, XRR+95] have been proposed as a concept for structuring complex concurrent activities in safety-critical systems. The principles of applying CAAs to the design of critical systems have been demonstrated in [ZRX+97] by developing a semi-formal solution to the production cell case study [LL95]; the design presented in [ZRX+97] is referred to in the remainder of this paper as the "CAA-PC design".

The production cell is a model of an industrial processing unit that consists of several devices: feed belt, table, robot, press, deposit belt, and crane. In addition, an operator, who has the task of loading the initial plates into the production cell, is included as a device in the CAA-PC design. The activities of the devices need to be coordinated specifically at points where they interact with each other. This kind of coordination can be expressed very elegantly by CAAs. A CAA coordinates the activities of roles (threads of control) that are activated from outside the CAA. This simple but elegant and powerful concept is used in the CAA-PC design.

The CAAs are characterized by their respective pre- and postconditions. They appear to be atomic transactions when viewed from the environment. The pre- and postconditions are designed to be compatible with the intended order on the execution of the CAAs: the postcondition of a CAA should imply the precondition of the next CAA to be executed. The activities of the devices are represented by controller processes for the devices, which have the primary task of activating the next appropriate role at each step.

This paper presents a formalization of the CAA-PC design as a finite state transition system. The formalization is based on a previous version of [ZRX+97], which did not contain fault tolerance aspects; also some details have changed since. In our formalization, safety and liveness properties are specified in terms of CTL [Eme90] and the model-checker SMV [McM92, McM93] is used for verifying the asserted properties. Note that a state transition system may exhibit many different behaviours; when model-checking is applied, all possible behaviours are considered. For a brief summary of SMV see appendix A of this paper.

Several approaches to the production cell problem based on model checking have been reported in [LL95]; in fact, these turned out to be more effective than most of the other approaches considered. The purpose of this paper, however, is not to add yet another such approach, but to investigate how CAA-based designs and model checking can be combined, using the production cell as a test case. Further work would be necessary to assess the benefits or drawbacks of CAA-based designs for model-checking as compared to other reported approaches based on model-checking towards formalizing the production cell.

Chapter 2

Overview of the CAA-PC Design

In the CAA-PC design the devices are controlled by processes which use CAAs as building blocks for coordinating their activities. Each CAA is characterized by its pre- and post-conditions and provides roles that can be activated by the controller process. For example the controller for the table device repeatedly iterates activating its roles in the two CAAs for loading and then unloading the table. Controller processes are designed for each of the devices including the operator.

Ten CAAs have been developed, as presented in the following list. Each of them models one step of plate processing and most CAAs involve passing a plate from one device to the next.

LoadPlate for passing a plate from the operator to the feed belt.

LoadTable for passing a plate from the feed belt to the table.

UnloadTable for passing a plate from the table to the robot.

LoadPress for passing a plate from the robot to the press.

ForgePlate for forging a plate in the press[1].

UnloadPress for passing a plate from the press to the robot.

LoadDepositBelt for passing a plate from the robot to the deposit belt.

TransportPlate for moving a plate from the beginning of the deposit belt to its end.

UnloadDepositBelt for passing a plate from the end of the deposit belt to the crane.

ReloadPlate for passing a plate from the crane back to the feed belt again, thus enabling the system to proceed infinitely.

The requirements for the production cell case study include safety and liveness properties. Liveness requirements have not been addressed explicitly in [ZRX+97]. In order to ensure the safety requirements of avoiding collisions, the CAA-PC design has been developed in such a way that at any time, each plate is processed by at most one CAA and each CAA processes at most one plate.

Each CAA has a rather complex task to perform, including control over the devices involved, like for example moving the table up or down until some required position is reached.

[1] The pre- and postconditions of this CAA have been changed in the current CAA-PC design; the formalization presented here is based on the previous version, where a non-forged plate is expected and a forged plate is produced.

At this level, other safety requirements (machine mobility) need to be considered. Thus, in addition to the top level view of the production cell, which includes the external interfaces (roles, pre- and postconditions) of the CAAs and the design of the device controllers, also a refinement of the CAAs is presented in [ZRX+97], where the state transition caused by a CAA is broken down into smaller steps and intermediate states. This also involves the development of additional devices (sensors) and CAAs.

The formal specification and analysis presented in this paper is concerned so far only with the top level view. Hence, the lower level CAAs and sensors are not considered here and intermediate states are not observable. Also the fault tolerance aspects are not yet included.

Chapter 3

Formal Specification of the CAA-PC Design

The formal specification is intended to resemble very closely the top level description of the CAA-PC design. The state of the devices, e.g. the table position, which are to be maintained by the controller processes are represented by state-variables covering an appropriate range of values, like `top` and `bottom` for table positions. The controller processes perform their tasks mainly by activating roles in the appropriate CAAs.

In this paper, the main emphasis is on the formalization of the CAAs. Each CAA is characterized by its pre- and postconditions and its roles. These pre- and postconditions are expressed in terms of the states of the devices and plates involved. The inspection of all pre- and postconditions reveals that the given CAA-PC design can be formalized in terms of a finite state machine since all values of state-variables that can be seen in the top level specification are explicitly named, even for variables whose range of values is a-priori non-finite. If other values are left anonymous but still are considered to be important, they can be modeled by adding a specific value, e.g. called `other`, to cover all other values.

In the SMV formalization of the design, all controllers and CAAs are modeled by separate SMV processes. The formalization presented below also contains initializations of state components, like for example the initial position of the table. Such initializations are necessary to prove that certain requirement are satisfied. Most of the necessary initialization of state components has been neglected in [ZRX+97].

Some details in the formalization presented further below are due to the notation required by the SMV system; specifically, there is no support for procedure calls and for passing parameters. This has to be encoded using appropriate static structures. For further details on the language of SMV see [McM92, McM93].

3.1 Formalizing Plates

Plates to be processed by the production cell are represented as records of two entries, a unique identifier name for identifying the plate and a state component expressing whether a plate has already been forged[1]. At the beginning the plate is initialized to be not forged. Individual plates are defined by instantiating the generic module `Plate` with an appropriate identifier for the plate.

[1] This entry is derived from the pre- and postcondition of CAA ForgePlate

```
MODULE Plate(name)
  DEFINE
    present := !(id = void);
  VAR
    id :        { void, id1, id2, id3, id4, id5, id6, id7, id8 };
    state :     { plain, forged };
  ASSIGN
    init(id) := name;
    init(state) := plain;
```

Module `Plate` allows a production cell with at most eight plates being processed to be modeled. In the CAA-PC design, passing a plate from one device to the next is modeled by "moving" the plate from an input parameter to an output parameter of roles of the same CAA. These parameters are modeled using the structure presented in module `Plate`. The pre- and postconditions of the CAAs contain conditions for checking whether a plate is present at some device. For the purpose of encoding that no plate is present a specific identifier `void` is provided. Additional modules have been developed to model forging a plate (module `Forge`) and to express moving a plate (module `MovePlate`) from one location, e.g. the input parameter, to another location, e.g. the output parameter; see Appendix B for a listing.

3.2 Formalizing CAAs

The key to formalizing the CAA-PC design is the formalization of CAAs. The interface provided by a CAA to its environment, which is mainly the collection of the device controller processes, is characterized by its roles and its effect. The environment can make use of a CAA by activating its roles. The CAA will be executed if and only if all roles are activated. The execution of a CAA corresponds to an atomic state transition and leads to a state change which is specified by its effect. The coordination of the different roles takes place within the CAA and involves intermediate steps, which are not visible in our formalization except for their common outcome which is specified by the effect of the CAA. The actions of a CAA are intended to be interleaved with the actions of other CAAs and of its environment. Thus, modules specifying CAAs are intended to be used/instantiated as processes.

A role of a CAA is activated by a controller process with a mechanism that resembles a procedure call: if the role is not active, then the controller initializes the inputs for the CAA and changes the state of a role to be activated. After activating the role the controller waits for the role to be finished[2]. Then it obtains the available outputs and takes the role back to a non-active state. Thus a role can be in one of three states:

`free:` means role is not activated; this is the initial state

`occ:` means role is occupied or activated

`ret:` means role is returning

The state of a CAA, which is visible to the environment is represented by the collection of its role's states. The environment takes a role from state `free` to state `occ` and from state `ret` to state `free`. The CAA itself only takes the role from state `occ` to state `ret`.

The effect of a CAA is expressed in terms of its inputs and outputs. Variables representing the inputs and outputs are not part of the internal state of a CAA. When a CAA is executed the input and output variables are modified according to the CAA's effect and the roles are changed to state `ret` signaling to the environment that the execution of the CAA is

[2]Before a role can finish, it is required that all roles had been activated by some other controllers; upon finishing it is ensured that all other roles are finished, too.

finished. The formalization of a CAA `foo` with two roles is illustrated (in SMV notation) by the following example.

```
MODULE foo(InOut)
  VAR role1, role2 : { free, occ, ret };
  ASSIGN
    init(role1) := free;
    init(role2) := free;

    next(role1) := case
                   role1 = occ & role2 = occ: ret;
                   1 : role1;
                   esac;
    next(role2) := case
                   role1 = occ & role2 = occ: ret;
                   1 : role2;
                   esac;
    next(InOut) := case
                   role1 = occ & role2 = occ: Effect(InOut);
                   1 : InOut;
                   esac;
```

Initially, roles are not active. If all roles are occupied, a state change occurs and in the next state, the IO variables of the CAA are assigned values satisfying its intended effect and the role's states are set to be returning. The changes to IO variables and the role's state occur simultaneously. If not all roles are occupied, then no state change occurs, as specified by the default outcomes of the case constructs.

Since all roles are treated similarly, the initialization and next-state assignment for roles can be formalized in a separate SMV module `Role`, as listed below. Then the CAA can be modeled by using instances of module `Role` as synchronized components of the module specifying the CAA.

A module for specifying a role is parameterized with an enabling condition, which is to be instantiated by the CAA the role belongs to. The according enabling condition in module `foo` is `role1 = occ & role2 = occ`

```
MODULE Role(enabled)
  VAR
    state:      { free, occ, ret };
  DEFINE
    nonactive := state = free;
    activated := state = occ;
    returning := state = ret;
  ASSIGN
    init(state) := free;
    next(state) :=
        case
        enabled : ret;
        1 : state;
        esac;
```

Initialization and next state assignment are equivalent to those contained in `foo`. In order to simplify the application of module `Role` some useful definitions are provided. Now CAA `foo` can be rewritten as a structured specification:

```
MODULE foo_struct(InOut)
  DEFINE
    enabled := role1.activated & role2.activated;
  VAR
    role1 : Role(enabled);
    role2 : Role(enabled);
  ASSIGN
    next(InOut) := case
                   enabled : Effect(InOut);
                   1 : InOut;
                   esac;
```

3.3 Formalizing CAAs of the CAA-PC Design

In [ZRX+97] CAAs are described by their respective roles with pre- and postconditions expressing their effects. For a CAA to become enabled, all of its roles must be activated. In this formalization, also the precondition is modeled as an enabling condition. The above mentioned input and output parameters of the roles are modeled as parameters of the CAA.

The formalization is exemplified here with the CAA `LoadTable`. The other CAAs described in the CAA-PC design are formalized similarly. The CAA-PC design of `LoadTable` defines roles `FeedBelt` and `Table` for the feed belt controller and the table controller to participate in. There are two more roles, which, however, are only used in the implementation of `LoadTable`; they are therefore ignored here.

The pre- and postconditions for `LoadTable` are defined in the CAA-PC design in terms of the state of the devices maintained by the controllers and in terms of the role's input/output parameters for passing a plate from the feed belt to the table.

preconditions	postconditions
feed belt off	feed belt off
plate on the feed belt	no plate on the feed belt
no plate on table	plate on table
table on top position	table on bottom position
table angle 50°	table angle 0°

"plate on table" and "plate on the feed belt" correspond to the input and output parameters of roles in [ZRX+97]. The other entries represent conditions on state components of the devices involved. The formalization of `LoadTable` is parameterized with all entries appearing in the pre- and the postcondition.

```
MODULE LoadTable(feed_belt, plate_on_feed_belt, plate_on_table,
                 table_position, table_angle)
  DEFINE
    pre :=      feed_belt = off &
                plate_on_feed_belt.present &
                !plate_on_table.present &
                table_position = top &
                table_angle = deg50;
    next_table_position := bottom;
    next_table_angle := deg0;
    enabled :=  FeedBelt.activated &
                Table.activated &
                pre;
  VAR
    FeedBelt :  Role(enabled);
```

```
  Table :     Role(enabled);
  mvplate :   MovePlate(enabled, plate_on_feed_belt, plate_on_table);
ASSIGN
  next(feed_belt) := feed_belt;
  next(table_position) :=
      case
      enabled : next_table_position;
      1 : table_position;
      esac;
  next(table_angle) :=
      case
      enabled : next_table_angle;
      1 : table_angle;
      esac;
FAIRNESS running
```

The action of passing a plate from the feed belt to the table is defined by a (synchronized) instance of `MovePlate(cond,source,destination)`, which moves the contents of `source` over to `destination` when `cond` is true; the synchronization mechanism is analogous to the use of instances of module `Role`. The `ASSIGN` section defines the coordinated state transition of the two controllers for the feed belt and the table. The assignments are designed to ensure the postcondition of the CAA `LoadTable`; notably the state of the feed-belt is the same in the pre- and the postcondition. The module also contains a fairness assumption to specify that the state transition associated with this module is selected infinitely often.

3.4 Formalizing Controllers

The main activities of the production cell are defined by the CAAs. But, in order for a CAA to perform a state transition, all of its roles need to be activated. Activation of roles within CAAs is performed by controller processes, which take the role from state `free` to state `occ`. After activating a role, a controller waits for the role to return and then resets the state of the role to `free`. These activities are synchronized with the actions of the controller for setting the inputs and obtaining the outputs of the CAA. The controller may perform other (internal) activities that do not affect calling a role. The actions of a controller can therefore be divided into three groups:

if	*all roles are free and controller wants to call some role*	then	*CAA inputs are set and role becomes activated*
if	*role is returning*	then	*CAA outputs are obtained and the role becomes non active*
if	*all roles are free and controller does not want to call a role*	then	*the internal actions of the controller are executed*

The first two actions on the state of the role are formalized in the module `CallRole`, which is parameterized with the role to be activated and a condition `at_call` expressing whether the controller wants to call this role. The default action in the case expression causing no state change is intended to be synchronized with the internal actions of the controller.

```
MODULE CallRole(role,at_call)
  DEFINE
    returning := role.returning;                -- returning condition
  ASSIGN
    next(role.state) :=
        case
        at_call & role.nonactive : occ;         -- activating role
        returning : free;                       -- returning from role
        1 : role.state;
        esac;
```

The CAA-PC design describes controllers for each of the devices involved, including the operator. Each controller is formalized as a process that maintains the state of the respective device represented by state variables. Each controller is parameterized with the CAAs it participates in. This is illustrated here for the table controller process.

```
MODULE TableController(loadtable, unloadtable)
  VAR
    table_angle :        deg0, deg50 ;
    table_position :     bottom, top ;
    plate_on_table :      Plate(void);
    lt: CallRole(loadtable.Table, !plate_on_table.present
                                  & unloadtable.Table.nonactive
                                  );
    ut: CallRole(unloadtable.Table, plate_on_table.present
                                  & loadtable.Table.nonactive
                                  );
  ASSIGN
    init(table_angle) := deg50;
    init(table_position) := top;
  FAIRNESS running
```

The controller maintains the state of the table, which is expressed by the state variables `table_angle` and `table_position`. These components are initialized according to the `ASSIGN` section. `TableController` may participate in the CAAs `loadtable` and `unloadtable` for loading and unloading the table. The table controller also maintains a local variable for encoding which plate, if any, is on the table. Initially, there is no plate on the table. This variable corresponds to the output parameter for CAA `loadtable` and to the input parameter for CAA `unloadtable`[3] as described in [ZRX+97]. The table controller activates its role in the CAA for loading the table, when there is no plate on the table and when the other role is not active, and it activates its role in the CAA for unloading the table, when a plate is on the table and its loading role is not active. There are no internal actions and additional activity for setting inputs or obtaining outputs is not necessary.

3.5 The Main Module

The modules for CAAs and controllers are composed and connected with each other in a module called `main`. It contains separate processes for each of the CAAs and each controller, connected with each other by appropriate instantiation of their parameters.

[3] Due to the syntactic structure of the SMV language, this connection is expressed only globally in `MODULE main`.

```
MODULE main
  DEFINE
    N := PLATES_IN_PRODCELL;          -- number of plates (max. 8)
  VAR
    -- CAA as processes
    lpl : process LoadPlate(...);
    ltb : process LoadTable(fbc.feed_belt, fbc.plate_on_feed_belt,
                tbc.plate_on_table, tbc.table_position, tbc.table_angle);
    utb : process UnloadTable(...);
    lpr : process LoadPress(...);
    fpl : process ForgePlate(...);
    upr : process UnloadPress(...);
    ldb : process LoadDepositBelt(...);
    tpl : process TransportPlate(...);
    udb : process UnloadDepositBelt(...);
    rpl : process ReloadPlate(...);
    -- Controller Processes
    opc : process OperatorController(...);
    fbc : process FeedBeltController(...);
    tbc : process TableController(ltb, utb);
    dbc : process DepositBeltController(...);
    crc : process CraneController(...);
    prc : process PressController(...);
    rbc : process RobotController(...);
```

`ltb` is an instance of CAA `LoadTable`. It is connected with the controllers for the feed belt and the table and has access to some of the local state components of the device controllers through the parameters. `fbc.plate_on_feed_belt` instantiates the CAA's input parameter for referencing the current plate on the feed belt. This plate is moved by the actions within `ltb` to the CAA's output parameter which is instantiated with `tbc.plate_on_table` for referencing the plate currently on the table. `tbc` is the table controller process instantiated with the two CAAs for loading (`ltb`) and unloading (`utb`) the table[4].

In order to analyze the formal model of the CAA design of the production cell, the constant `PLATES_IN_PRODCELL` has to be instantiated (substituted) with some number in the range from 1 to 8. Each such instance has to be created and analyzed on its own; SMV cannot deal with generic systems although the system parameter might be restricted to a finite range.

[4]SMV allows forward references to module names

Chapter 4

Proof-Obligations for Verifying the CAA-Design

Verification is performed with different purposes in mind. One goal is to develop a formal specification that conforms to the CAA-PC description of the design. This is done by trying to prove simple assumptions that are expected to hold in the (formal) model. A failed proof might point to a bug in the formalization. Another goal is to verify that certain properties are satisfied by the design under the assumption that it has been formalized correctly. Specifically in the case of safety critical systems, these properties include the safety requirements. The functionality expected from the design often includes liveness requirements.

As mentioned earlier, properties to be verified are stated as CTL formulae in the module section introduced by the keyword `SPEC`.

4.1 Plausibility Assumptions

As is the case with programs, formal specifications tend to be initially erroneous. One approach to eliminate such mistakes is to perform plausibility checks. A rather simple proof-obligation (stated in the main module) for checking plausibility is

```
-- at any point in the computation there exists some next state
SPEC AG EX TRUE
```

The purpose of this proof-obligation is to ascertain that the system makes some sort of progress and that no deadlock has been introduced into the system, for instance by inappropriate use of the SMV specification constructs.

Plausibility checks can be used to assess that there is some progress within specific modules. For example, it is expected that, if there is a plate on the table, then eventually the plate will be (re-)moved from the table. This goal is expressed in the `SPEC` section presented below.

```
MODULE TableController(loadtable, unloadtable)
  ...
  -- goals for gaining confidence in formalization
  SPEC AG (plate_on_table.present
           -> AF !plate_on_table.present)
```

One may also state the expectation that if there is no plate on the table, then eventually one will arrive:

```
  SPEC AG (!plate_on_table.present
           -> AF plate_on_table.present)
```

The task of verifying these goals also helped to reveal minor flaws in the previous version of the CAA-PC design, which is no longer present in the current version; this analysis is described in section 5.

4.2 Postconditions

Each CAA is formalized with an `ASSIGN` section that is intended to ensure its postcondition. Since each CAA has a postcondition with a fairly simple form, it is easy to see that its assignment really conforms with its postcondition. However, in general it is necessary to verify that the assignment produces a next state in which the postcondition is fulfilled. In order to accomplish this verification task, for each CAA a suitable goal has to be formulated which has then to be verified. An informal description of such a goal for some CAA is

If	in some state	all roles of the CAA are activated and its precondition is satisfied,
then	if	the next state results from executing this CAA,
	then	the postcondition is satisfied.

In the context of SMV, such a goal is written in terms of a CTL formula. This is illustrated here for the CAA `LoadTable`. Recall that the pre- and postconditions are given by

preconditions	postconditions
feed belt off	feed belt off
plate on the feed belt	no plate on the feed belt
no plate on table	plate on table
table on top position	table on bottom position
table angle 50°	table angle 0°

The module `LoadTable` presented earlier is extended by a definition for post and a `SPEC` section containing a CTL formula for this goal.

```
MODULE LoadTable(...)
  DEFINE
    ...
    enabled :=  FeedBelt.activated &
                Table.activated &
                pre;
    post :=     feed_belt = off &
                !plate_on_feed_belt.present &
                plate_on_table.present &
                table_position = bottom &
                table_angle = deg0;
  VAR
    ...
  ASSIGN
    ...
  -- postcondition is satisfied - safety property
  SPEC AG (enabled -> AX (FeedBelt.returning & Table.returning -> post))
```

This CTL formula corresponds to the informal description of the goal given above. Whether a next state results from executing CAA `LoadTable` is encoded by checking whether in a next state all its roles are in a returning state. Such a specification would also be satisfied by a system, which does not execute and finish CAA `LoadTable`. Therefore, a liveness property has been added:

```
-- postcondition is satisfied - CAA termination (liveness) property
SPEC AG (enabled -> A[enabled U FeedBelt.returning & Table.returning & post])
```

This CTL formula expresses that, if enabled, the CAA will eventually return and then the postcondition will hold (expressed by means of the temporal operator *until*).

4.3 Safety Requirements

Several safety requirements are stated in the informal description of the production cell case study [LL95]. One requirement states that plates may not be dropped outside safe areas. In terms of the top level model of the production cell this can be described by the requirement that a plate, once introduced into the system, stays in the system. A formalization of this requirement for plate "`id1`" is given below. For a complete analysis, such a goal would have to be stated and verified for each plate.

```
SPEC
  AG (fbc.plate_on_feed_belt.id = id1 ->
      AG (  fbc.plate_on_feed_belt.id = id1
          | opc.plate_with_operator.id = id1
          | fbc.plate_on_feed_belt.id = id1
          | tbc.plate_on_table.id = id1
          | rbc.plate_on_arm1.id = id1
          | rbc.plate_on_arm2.id = id1
          | prc.plate_in_press.id = id1
          | dbc.plate_on_beg_deposit_belt.id = id1
          | dbc.plate_on_end_deposit_belt.id = id1
          | crc.plate_on_crane.id = id1))
```

The formula states that if once plate "`id1`" is on the feed belt (the first device) then in all states from then on it will be on (at least) one of the devices, which ensures that the plate stays in the system.

It is argued in [ZRX+97] that the system safety requirements for ensuring that neither plates nor devices can collide are satisfied because:

1. only one plate can be in an action
2. a plate cannot be involved in more than one action
3. a device can participate in only one action

These requirements on the relation between plates and CAAs can be considered as new (transformed) safety requirements which are sufficient for ensuring the original non-collision requirement. The first requirement can be demonstrated by proving for each CAA a goal that states that if the CAA is active then at most one plate is in that action. This is shown here for CAA `LoadTable`:

```
MODULE LoadTable(...)
  ...
  -- only one plate can be in a CAA
  SPEC AG (!FeedBelt.nonactive & !Table.nonactive ->
             !plate_on_feed_belt.present | !plate_on_table.present)
```

The formula describes that if no role of the CAA `LoadTable` is inactive (i.e. `LoadTable` is active), then at most one of the both plate locations accessible by `LoadTable` (i.e. feed-belt and table) contains a plate.

The second requirement stating that a plate is involved in at most one CAA at a time is divided into two properties: first, a plate is present on at most one device and, second, at

any time at most one of the CAAs utilized by a device controller can be active. The second property is identical to the third requirement. By assuming the second property it is here sufficient to formalize the first property only. The property that a plate being processed by one device cannot be present at any other device is here exemplified for plate location `plate_on_table`.

```
MODULE main
  ...
  SPEC
    AG (tbc.plate_on_table.present ->
          !(tbc.plate_on_table.id = opc.plate_with_operator.id)
        & !(tbc.plate_on_table.id = fbc.plate_on_feed_belt.id)
        & !(tbc.plate_on_table.id = rbc.plate_on_arm1.id)
        & !(tbc.plate_on_table.id = rbc.plate_on_arm2.id)
        & !(tbc.plate_on_table.id = prc.plate_in_press.id)
        & !(tbc.plate_on_table.id = dbc.plate_on_beg_deposit_belt.id)
        & !(tbc.plate_on_table.id = dbc.plate_on_end_deposit_belt.id)
        & !(tbc.plate_on_table.id = crc.plate_on_crane.id))
```

The third requirement states that at most one of the CAAs utilized by a device controller is active. This is exemplified here for the table controller: if a plate is on the table then one of the two CAAs being used there must be inactive. This is expressed with the following formula in Module `TableController`.

```
  SPEC AG (plate_on_table.present ->
           unloadtable.Table.nonactive | loadtable.Table.nonactive)
```

The formalization of the safety requirements presented here also makes use of some structural assumptions: The formalization of the first requirement is based on the assumption that a CAA may gain access to only some of the plate locations, like for example CAA `LoadTable` may only access plates on the feed-belt and plates on the table. The formalization of the third requirement is based on the assumption that a device controller may gain access to only some of the CAAs, like for example `TableController` may only participate in the CAAs `UnloadTable` and `LoadTable`.

4.4 Liveness Requirements

The functionality of the production cell is specified in terms of a liveness property, which requires that a plate introduced into the system via the feed belt will eventually have been forged and dropped by the crane onto the feed belt. The formalization of this requirement is presented here for plate "`id1`"; again, for a complete analysis this requirement has to be expressed for each plate.

```
  SPEC                          -- plate will arrive forged on the crane
    AG (fbc.plate_on_feed_belt.id = id1 ->
        AF (crc.plate_on_crane.id = id1 &
            crc.plate_on_crane.state = forged))
  SPEC                          -- plate will be reintroduced into system
    AG (crc.plate_on_crane.id = id1 -> AF fbc.plate_on_feed_belt.id = id1)
```

Chapter 5

Results from the Verification Process

There are two kinds of results from performing the verification process. On one side, there are results on the benefits from checking certain proof obligations for improving or correcting the formal specification of the specific application (production cell case study) to be developed. On the other side, there are also results on the applicability and usefulness of the particular formal method chosen to approach the given problem. In the following, both issues are discussed.

5.1 Results on the Verification of the Production Cell

The SMV system checks whether a given CTL formula is valid in the specified state transition system. If it is not valid, an execution path is constructed and displayed which serves as a counterexample. This is very useful for debugging a specification. Specifically the plausibility checks reveal mistakes resulting from slight oversight when writing the specification; these mistakes are comparable to "typos", like using a wrong logical connective, missing negations in formulas, etc.

These plausibility checks also helped in finding a minor design flaw and some incompleteness in the CAA-PC design. One incompleteness is that hardly any initialization of state components are mentioned in [ZRX+97]. Proper initialization, however, is necessary for a formal analysis. Otherwise the system might just start in an undesired state, or devices might be initialized such that a deadlock can occur in the system.

Another incompleteness results from a missing postcondition for CAA UnloadDepositBelt. The pre- and postconditions of the CAAs should be such that the postcondition of a CAA implies the precondition of the next CAA to be executed. This is not the case for CAA UnloadDepositBelt since the next action would be ReloadPlate, which requires the crane's lower switch to be off. This is not implied by the postcondition of CAA UnloadDepositBelt. A proof obligation in module `CraneController`, which expresses that if a plate is present on the crane then it will eventually be removed from there, reveals this incompleteness:

```
SPEC AG (plate_on_crane.present -> AF !plate_on_crane.present)
```

is false when `crane_lower_switch = off` is missing in the postcondition of CAA UnloadDepositBel. This condition has been added to the postcondition. Symmetrically, although not necessary for correctness, the precondition has been extended with the condition `crane_lower_switch = on`. The SMV system generates a counter example execution path which helps in identifying the reason for failed proofs; see appendix C.

For similar reasons also the postcondition of CAA ForgePlate is found to be incomplete. The formula

```
SPEC AG (plate_in_press.present -> AF !plate_in_press.present)
```

is false due to the missing condition `press_position = top` in the postcondition of CAA ForgePlate. This has been added to the postcondition and, for the sake of symmetry, the precondition has been extended with `press_position = middle`.

The minor design flaw, that was present in the previous version of the CAA-PC design, is connected to forging plates. On one hand, plates are reintroduced into the system after they have been forged; on the other hand, CAA ForgePlate expects a non-forged plate for processing, as stated in the precondition. When checking the formula

```
SPEC AG AF pre
```

(stated in Module `ForgePlate`) it is shown to be false and a counterexample is generated that shows that when a plate arrives at the press for the second time the precondition will not be true. This design flaw can be eliminated by different means. One would be to modify the pre- and postconditions of the CAA ForgePlate. Another option would be to "un-forge" plates before they are reintroduced onto the feed belt. Since the crane is an artifact with the sole purpose of keeping the system running infinitely, the second option has been chosen here: when a plate is reloaded from the crane to the feed belt, it is also un-forged. Counting up the number of forge operations, as mentioned in the previous version and adopted in the current version of [ZRX+97], would not be possible in this formalism, since for SMV a finite state system is required. The verification of postconditions does not exhibit any further errors. This is partly due to the very simple form of the pre- and postconditions.

The other proof obligations deriving from safety and liveness requirements were shown to be satisfied by the state transition system model of the production cell.

5.2 Results on Applying SMV Model-Checking to the CAA-Design

Although the production cell is a rather small system, the CAA design adds considerable complexity by increasing the state space. Each instance (with one up to eight plates) of the production cell needs to be analyzed separately. Whereas the instance with a single plate can be analyzed very quickly, other instances consume considerably more time for being analyzed and the presented formalization drives SMV at its limits. Hence, special steps need to be taken in order to be able to successfully apply model-checking.

- The specification actually being analyzed has been augmented with additional conditions for enabling certain state transitions in order to reduce the overall reachable state space; see the appendix for a listing of relevant modules in the SMV formalization of the production cell.
- The SMV model checker is based on binary decision diagrams (BDD). A very important aspect of BDD representations is, that their size and hence the efficiency of their analysis strongly depends on the ordering of variables that the BDD representation is based upon. One simple heuristic for ordering variables is to place the ones that are strongly related to each other, when the system is executed, also close to each other in the ordering. Using this heuristic, an initial variable ordering was generated (manually) which immediately gave better run-time results; however, for practical application and experimentation with the model-checking approach, the run-times still were not acceptable. SMV provides an option for dynamically re-ordering variables. This was used for generating new variable orderings. Two approaches were taken: in the first approach a new ordering was generated starting from the manual ordering (see above). In the second approach a new ordering was generated from scratch, i.e. without initializing SMV with a specific ordering. The first approach yielded an

ordering which results again in better run-time performance. The second approach also lead to an ordering with better run-time performance than starting the analysis from scratch; however, this ordering was worse than the manual ordering. This process of re-ordering variables was applied to a single plate system to obtain a new ordering; intriguingly, the new ordering could also be applied to decrease considerably the run-time for analyzing other instances of the production cell. Whether this process can be generalized to other examples needs to be examined further and in a different context.

All instances of the production cell with one up to eight plates has been analyzed. Note that model-checking is performed with respect to all possible behaviours, for example the second plate can be put on the feed-belt in different states: while the first plate is on the table, in the press, etc.; all these behaviours are considered by model-checking. The eight plates system can be shown to deadlock[1], while the others are proved to be correct. The following table summarizes the run-times[2] reported by SMV:

Plates	Platform	Run-Time
1	Sparc Ultra I	330.3 s
2	Sparc Ultra I	5692.1 s
3	Sparc Ultra I	22379.7 s
4	Sparc Ultra I	62263.7 s
5	Sparc Ultra II	84265.8 s
6	Sparc Ultra II	76511.9 s
7	Sparc Ultra II	66885.8 s
8	Sparc Ultra II	$\approx$ 20400.0 s

While the one-plate instance can be checked within reasonable time, the run-times for checking larger instances increase tremendously. Even with the improved ordering on variables, model checking of the two-plate system takes roughly 1,5 hours, the three-plates system takes roughly 6 hours, and the "hardest" instance with 5 plates takes almost 24 hours on "Sun Sparc Ultra" "I" and "II" platforms. The pure run-times for checking all system instances sums up to roughly 94 hours (or 4 days)

[1] When checking the 8-plates instance, SMV crashes, probably due to memory allocation problems, after displaying the first counter example path.

[2] For the 8-plates instance we have measured the time until the crash occurs.

Chapter 6

Conclusion and Outlook

The work presented here has been done in order to evaluate the feasibility of model-checking for the analysis of systems designed using CAAs. A generic concept has been presented for formally specifying both, CAAs and according mechanisms for activating their roles.

Some aspects of CAAs are rather cumbersome and non-intuitive for encoding in the restricted language of the SMV system. Specifically parameters for calling procedures (methods, roles) have to be encoded explicitly, since the SMV language does not allow dynamic instantiation, but is restricted to static instantiation of modules. In the encoding presented here, these parameters are "hard-wired" to specific locations. Despite these particular peculiarities, a CAA-design can be formalized appropriately with SMV. However, certain properties (safety requirements) could be formalized based on assumptions that are induced by the CAA structure.

The structure imposed by CAAs increases the state space of a system. The state explosion problem is present although advanced analysis techniques (model-checking based on BDDs) have been applied. Thus, the CAA design did not contribute directly to a simplification of the verification process, specifically when compared to reports on other model-checking approaches applied to the production cell case study [LL95]. One approach to deal with the state explosion problem might be a combination of theorem proving and model-checking. This issue still needs further investigation.

Refinement issues that are also presented in [ZRX+97] have not been considered here, but it is in principle possible to formalize the implementation of CAAs expressed in terms of other (low level) controllers and CAAs similar to the formalization presented in this paper.

Fault tolerance mechanisms have not been considered yet; it still needs to be investigated whether the approach presented here is suitable for representing a fault tolerant CAA-designs. It is even more important to investigate, whether model-checking can be applied to the CAA design of a fault-tolerant system since one may expect an increase in state space when additional (exceptional) outcomes are defined for CAAs.

Despite the problems associated with the state explosion it is noteworthy that the errors in the formalization and the incompleteness and flaws in the design could already be detected in the single plate instance of the production cell. Safety properties were specified by directly encoding safety requirements and by encoding (transformed) safety requirements for the CAA-PC design. Furthermore, the liveness requirements, which were not assessed in [ZRX+97], could be checked and proved correct. Thus, model-checking can be considered a useful technique to supplement other analysis methods for assessing properties of a safety critical system based on its CAA-design.

A benefit of the CAA concept is, that certain properties, like atomicity, are provided by CAAs and need not to be implemented by a programmer. Thus, the refinement of a CAA only needs to respect its pre- and postconditions, which may entail erroneous outcomes when faults occur and according fault-tolerance mechanisms and strategies may be used when refining a CAA; it is not necessary to repeat the verification of the refinement with

respect to the complete system; only a verification with respect to its pre- and postconditions, which is expected to be much easier, needs to be carried out. Thus, the higher effort needed for validating a CAA-based design may pay off.

Bibliography

[Eme90] E. Allen Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier Science Publishers B.V., 1990. Formal Models and Semantics.

[LL95] Claus Lewerentz and Thomas Lindner, editors. *Formal Development of Reactive Systems: Case Study Production Cell*, volume 891 of *LNCS*, Berlin, 1995. Springer.

[McM92] K.L. McMillan. *The SMV system.* Carnegie Mellon University, draft edition, February 1992.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Carnegie Mellon University, 1993. Revised version of PhD thesis.

[RRS+97] B. Randell, A. Romanovsky, R.J. Stroud, J. Xu, and A.F. Zorzo. Coordinated Atomic Actions: from Concept to Implementation. Submitted to Special Issue of IEEE Transactions on Computers, 1997.

[XRR+95] 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 *Proceedings of the 25th Int. Symp. on Fault-Tolerant Computing*, pages 450–457, Pasadena, USA, 1995. IEEE CS Press.

[ZRX+97] A. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. Stroud, and I. Welch. Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study. ESPRIT LTR Project 20072 - DeVa Technical Report 37, Department of Computing Science, University of Newcastle upon Tyne, UK, 1997. See http://www.newcastle.research.ec.org/deva/trs/index.html.

Appendix A

Brief Summary of SMV

SMV [McM92, McM93] is a system for checking finite state systems against specifications in the temporal logic CTL [Eme90]. The SMV language describes the Kripke model and the specification of a system. Its input language is thus designed to describe state transition systems. Descriptions can be structured hierarchically into several modules, each describing some part of the finite state transition system. The state transitions of each part can be defined to occur synchronously or asynchronously with the state transitions defined in other modules. In the following, the example of a three-bit counter (taken from [McM93]) serves to illustrate the concepts; the specified system increments the counter in each step.

```
MODULE main
  -- A three bit counter
  VAR bit0 : cell(1);
      bit1 : cell(bit0.carry_out)
      bit2 : cell(bit1.carry_out)
  FAIRNESS running
  SPEC AG AF bit2.carry_out
```

```
MODULE cell(carry_in)
  DEFINE carry_out := val & carry_in;
  VAR val : boolean;
  ASSIGN
    init(val) := 0;
    next(val) := (val & !carry_in)
               | (!val & carry_in)
```

MODULE is followed by `main` or a user-defined name for the module and optional parameters. Module `main` is considered to be the root of the module hierarchy. The SMV language is only weakly typed: e.g. parameters are not typed. A module consists of several sections, each introduced by a key-word; these are explained below. A part of a system is represented by an instance of a module. Instances of modules are created in the VAR section.

DEFINE is used for defining shortcuts for complex expressions, comparable to macros without parameters. Forward references to names that will be defined later are permitted.

VAR has two purposes. One is to declare the state variables. The state of the Kripke model consists of the collection of all state variables; the variables may only range over a finite set of values expressed by a finite type (boolean, enumerated type, or integer subtype). The second purpose is to define instances of modules. Module instances are defined by declaring a name for the module instance and by instantiating the module parameters accordingly. The arguments for instantiating module parameters may contain forward references to names that are not yet declared. All module instances are distinct from each other and can be uniquely identified by the instance name. The state transition of each module instance can be specified to be synchronous, i.e. occurring simultaneously, with the state transition of the module in which the instance is created, or it can be specified to be asynchronous, i.e. being interleaved. In the example given above, the state transitions of `bit0`, `bit1` and `bit2` are synchronized. An asynchronous instance is considered to be a separate process and it is created using a notation of the form

`instance: process ModuleName(arguments)`. Entities defined within instances can be accessed by prefixing the name declared within the module with its instance name, like `bit0.carry_out` in the example above.

ASSIGN is used for initializing state variables and for defining the next-state relation. State variables can be initialized using the form `init(var):=expr`. The next-state relation is defined by assigning to state variables their value in the next state. The syntactic form used here is `next(var):=expr`. `expr` is built from constants[1], variables, boolean, integer, and set operations, and case expressions. SMV provides the boolean operators `!` (negation), `&` (conjunction), and `|` (disjunction). The next-state expression in the example given above encodes "`val xor carry_in`". A case expression of the form "`case cond1:expr1; cond2:expr2;...; 1:expr; esac`" is regarded as being equivalent to "if `cond1` then `expr1` elsif `cond2` then `expr2`...else `expr`". All next-state assignments within the same assignment section occur simultaneously (synchronized).

FAIRNESS is used to declare a fairness constraint, i.e. a formula that is assumed to be true infinitely often in a fair execution path. The predefined boolean expression `running` is true if the instance of a process module containing this expression is active. `running` is very useful for specifying processes that are assumed to be executed in a fair execution path infinitely often. When SMV evaluates specifications, only fair execution paths are considered.

SPEC is used to specify, by means of a CTL formula, the system property to be checked. A CTL formula is built from boolean expressions, the boolean operators, the CTL path quantifiers `A` (for all paths) and `E` (exists a path), and the CTL time quantifiers `X` (next time), `F` (eventually), `G` (globally), and `U` (until), with the restriction that every path quantifier must be followed by a time quantifier. For example, the specification "`AG AF bit2.carry_out`" states that along each execution path `bit2.carry_out` will infinitely often be set true.

[1]The boolean values false and true are represented by 0 and 1 respectively

Appendix B

Listing of the SMV Code

This appendix contains listings from SMV modules that are referenced in the main text of this paper. The complete source file of the SMV specification is available online from `http://www.informatik.uni-ulm.de/abt/ki/publications.html` together with an online copy of this report.

B.1 Module MovePlate

`MovePlate` specifies the state transition associated with moving a plate from one location to another. This state transition is enabled if parameter `cond` is true.

```
MODULE MovePlate(cond, from, to)
  ASSIGN
    next(to.id) :=
        case
        cond : from.id;
        1 : to.id;
        esac;
    next(to.state) :=
        case
        cond : from.state;
        1 : to.state;
        esac;
    next(from.id) :=
        case
        cond : void;
        1 : from.id;
        esac;
    next(from.state) :=
            case
            cond : plain;
            1 : from.state;
            esac;
```

B.2 Module Forge

`Forge` specifies the state transition associated with forging a `plate`. This state transition is enabled if parameter `cond` is true.

```
MODULE Forge(cond, plate)
  ASSIGN
    next(plate.id) := plate.id;
```

```
    next(plate.state) :=
        case
        cond : forged;
        1 : plate.state;
        esac;
```

B.3 Module UnloadDepositBelt

The design of CAA `UnloadDepositBelt` was originally incomplete. The condition requiring that the crane's lower switch is turned off was missing from the postcondition. Therefore a plate once unloaded from the deposit belt onto the crane was never passed further on.

```
MODULE UnloadDepositBelt(plate_on_crane, plate_on_end_deposit_belt,
                         deposit_belt, crane_upper_switch,
                         crane_lower_switch, crane_height)
  DEFINE
    pre :=      !plate_on_crane.present &
                plate_on_end_deposit_belt.present &
                deposit_belt = off &
                crane_upper_switch = off &
                crane_lower_switch = on &
                crane_height = ht6593;
    post :=     plate_on_crane.present &
                !plate_on_end_deposit_belt.present &
                deposit_belt = off &
                crane_upper_switch = on &
                crane_lower_switch = off &      -- was originally missing
                crane_height = ht6593;
    next_crane_upper_switch := on;
    next_crane_lower_switch := off;             -- was originally missing
    enabled :=  DepositBelt.activated &
                Crane.activated &
                pre;
  VAR
    DepositBelt:Role(enabled);
    Crane :     Role(enabled);
    mvplate :   MovePlate(enabled, plate_on_end_deposit_belt, plate_on_crane);
  ASSIGN
    next(deposit_belt) := deposit_belt;
    next(crane_upper_switch) :=
        case
        enabled : next_crane_upper_switch;
        1 : crane_upper_switch;
        esac;
    next(crane_lower_switch) :=                 -- was originally missing
        case
        enabled : next_crane_lower_switch;
        1 : crane_lower_switch;
        esac;
    next(crane_height) := crane_height;
  FAIRNESS running
```

B.4 Module ReloadPlate

CAA ReloadPlate requires the crane's lower switch to be off as a precondition. The precondition is part of the enabling condition for this CAA.

```
MODULE ReloadPlate(feed_belt, plate_on_feed_belt, plate_on_crane,
                   crane_upper_switch, crane_lower_switch, crane_height)
  DEFINE
    pre :=      feed_belt = off &
                !plate_on_feed_belt.present &
                plate_on_crane.present &
                crane_upper_switch = on &
                crane_lower_switch = off &
                crane_height = ht6593;
    post :=     feed_belt = off &
                plate_on_feed_belt.present &
                !plate_on_crane.present &
                crane_upper_switch = off &
                crane_lower_switch = on &
                crane_height = ht6593;
    next_crane_upper_switch := off;
    next_crane_lower_switch := on;
    enabled :=  FeedBelt.activated &
                Crane.activated &
                pre;
  VAR
    FeedBelt :  Role(enabled);
    Crane :     Role(enabled);
    mvplate :   MoveAndUnforge(enabled, plate_on_crane, plate_on_feed_belt);
  ASSIGN
    next(feed_belt) := feed_belt;
    next(crane_upper_switch) :=
        case
        enabled : next_crane_upper_switch;
        1 : crane_upper_switch;
        esac;
    next(crane_lower_switch) :=
        case
        enabled : next_crane_lower_switch;
        1 : crane_lower_switch;
        esac;
    next(crane_height) := crane_height;
  FAIRNESS running
```

B.5 Module FeedBeltController

The specification of the feed-belt controller contains additional conditions in order to reduce the state space of the considered system; the operator must load all plates onto the feed-belt before a plate can be re-loaded from the crane. This is encoded as an additional enabling condition when instantiating module `CallRole`: A variable, pc, is used to encode whether the operator still has some plates ($pc \neq 0$). In that case CAA LoadPlate is enabled. When no plates are left with the operator ($pc = 0$), then plates can be re-loaded from the crane.

```
MODULE FeedBeltController(loadplate, reloadplate, loadtable,
                          plate_with_operator, plate_on_crane, plate_on_table)
  VAR
    feed_belt :                 { on, off };
    pc :                        0 .. PLATES_IN_PRODCELL;
    plate_on_feed_belt :        Plate(void);
    lp: CallRole(loadplate.FeedBelt, !plate_on_feed_belt.present & !(pc=0)
                                    & reloadplate.FeedBelt.nonactive
                                    & loadtable.FeedBelt.nonactive
```

```
                                  & plate_with_operator
                                  );
    rp: CallRole(reloadplate.FeedBelt, !plate_on_feed_belt.present & (pc=0)
                                    & loadplate.FeedBelt.nonactive
                                    & loadtable.FeedBelt.nonactive
                                    & plate_on_crane
                                    );
    lt: CallRole(loadtable.FeedBelt, plate_on_feed_belt.present
                                  & loadplate.FeedBelt.nonactive
                                  & reloadplate.FeedBelt.nonactive
                                  & !plate_on_table
                                  );
  ASSIGN
    init(feed_belt) :=  off;
    init(pc) := 1;
    next(pc) :=
        case
        lp.returning & pc >= PLATES_IN_PRODCELL : 0;    -- all plates loaded
        lp.returning : pc + 1;
        1 : pc;
        esac;
  FAIRNESS running
  SPEC EF plate_on_feed_belt.present
  SPEC AF plate_on_feed_belt.present
  SPEC AG (plate_on_feed_belt.present
           -> AF !plate_on_feed_belt.present)
  SPEC EG EF plate_on_feed_belt.present
```

B.6 Module CraneController

The description of the crane controller specifies the property which revealed an incompleteness of CAA `UnloadDepositBelt`. This property requires that if a plate arrives on the crane then it will eventually be removed from the crane, i.e. passed on to the next device.

```
MODULE CraneController(unloaddepositbelt, reloadplate,
                       plate_on_end_deposit_belt,
                       plate_on_feed_belt)
  VAR
    crane_height :              { ht6593, other };
    crane_lower_switch :        { on, off };
    crane_upper_switch :        { on, off };
    plate_on_crane :            Plate(void);
    ud: CallRole(unloaddepositbelt.Crane, !plate_on_crane.present
                                      & reloadplate.Crane.nonactive
                                      & plate_on_end_deposit_belt
                                      );
    rp: CallRole(reloadplate.Crane, plate_on_crane.present
                                  & unloaddepositbelt.Crane.nonactive
                                  & !plate_on_feed_belt
                                  );
  ASSIGN
    init(crane_height) := ht6593;
    init(crane_lower_switch) := on;
    init(crane_upper_switch) := off;
  FAIRNESS running
  -- a plate will eventually be passed on from the crane onto the next device
  SPEC AG (plate_on_crane.present -> AF !plate_on_crane.present)
```

Appendix C

Using Counter Examples for Debugging Specifications

This appendix describes an example from the verification process and explains how counter examples produced by SMV have been used to identify the reasons for proofs to fail.

Incomplete Postcondition of CAA `UnloadDepositBelt`

The incompleteness of the postcondition of CAA `UnloadDepositBelt` can be observed from the counter example execution path generated by SMV when trying to prove the specification `AG (plate_on_crane.present -> AF (!plate_on_crane.present))`. State 1.1 below[1] shows the initial state of the counter example execution path.

```
state 1.1:
rpl.enabled = 0
rpl.pre = 0
rpl.FeedBelt.activated = 0
rpl.Crane.activated = 0
fbc.feed_belt = off
fbc.plate_on_feed_belt.present = 0
crc.crane_height = ht6593
crc.crane_lower_switch = on
crc.crane_upper_switch = off
crc.plate_on_crane.present = 0
```

Variables starting with `rpl` are defined in module `ReloadPlate`, variables starting with `fbc` are defined in module `FeedBeltController`, and variables starting with `crc` are defined in module `CraneController`. Recall that `rpl.enabled` is defined by

```
rpl.enabled :=  rpl.FeedBelt.activated &
                rpl.Crane.activated &
                rpl.pre;
```

and `rpl.pre` is defined by

```
rpl.pre :=      fbc.feed_belt = off &
                !fbc.plate_on_feed_belt.present &
                crc.plate_on_crane.present &
                crc.crane_lower_switch = off &
```

[1] The SMV output is here restricted to the information relevant to the proof obligation

```
crc.crane_upper_switch = on &
crc.crane_height = ht6593;
```

The modifications to these variables that are displayed in the execution path generated by SMV are listed below; note, that variables are only listed in the SMV output if their values are modified.

```
...
state 1.6:
fbc.plate_on_feed_belt.present = 1
fbc.plate_on_feed_belt.id = id1
...
state 1.10:
fbc.plate_on_feed_belt.present = 0
fbc.plate_on_feed_belt.id = void
...
state 1.36:
[executing process udb]
...
state 1.37:
crc.crane_upper_switch = on
crc.plate_on_crane.present = 1
...
state 1.83:
rpl.FeedBelt.activated = 1
...
state 1.86:
rpl.Crane.activated = 1
...
-- loop starts here --
state 1.88:
...
```

Note also, that this listing of the execution path is truncated and restricted to show only modifications to the relevant variables. SMV displays the message *"loop starts here"* close to the end of the execution path and after that point no modifications to the variables of the system are shown.

The counter example can be used to identify the reason why the proof of the specification fails: The specification can only become true, if CAA `ReloadPlate` is executed; for this the appropriate condition `rpl.enabled` must become true, which requires `rpl.pre` to be true. This however is not the case, since `crc.crane_lower_switch` is at no state changed to `off`. We would expect that this happens in CAA UnloadDepositBelt (`udp` changing the state from 1.36 to 1.37), since also variable `crc.crane_upper_switch` is set to value `on` there. Therefore the conclusion is, that the postcondition of CAA `UnloadDepositBelt` is incomplete and needs to be extended with the additional condition `crc.crane_lower_switch = off`.