

Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions

J. Xu*, B. Randell, A. Romanovsky, R.J. Stroud, and A.F. Zorzo

University of Newcastle upon Tyne, NE1 7RU, UK

E. Canver and F. von Henke

University of Ulm, D-89069 Ulm, Germany

Abstract

This paper describes our experience using coordinated atomic (CA) actions as a system structuring tool to design and validate a sophisticated control system for a complex industrial application that has high reliability and safety requirements. Our study is based on the “Fault-Tolerant Production Cell”, which represents a manufacturing process involving redundant mechanical devices (provided in order to enable continued production in the presence of machine faults). The challenge posed by the model specification is to design a control system that maintains specified safety and liveness properties even in the presence of a large number and variety of device and sensor failures. We discuss in this paper: i) a design for a control program that uses CA actions to deal with both safety-related and fault tolerance concerns, and ii) the formal verification of this design based on the use of model-checking. We found that CA action structuring facilitated both the design and verification tasks by enabling the various safety problems (e.g. clashes of moving machinery) to be treated independently. The formal verification activity was performed in parallel with the design activity: the interaction between them resulted in a combined exercise in “design for validation”.

Key Words — Concurrency, coordinated atomic (CA) actions, exception handling, object orientation, formal verification, model checking, reactive systems, reliability and safety.

1: Introduction

The goal of this work is to investigate a rigorous approach to the development of safety-critical applications, in particular to examine the feasibility of using coordinated atomic (CA) actions [11] as a structuring tool to design a realistically-detailed fault-tolerant control system, and then to use model-checking to debug, improve, and verify the design formally.

A production cell model, based on a metal-processing plant in Karlsruhe, Germany, was first created by the FZI (Forschungszentrum Informatik) in 1993 [4] in order to evaluate different formal methods and to explore their practicability for industrial applications. Since then, this

original case study, Production Cell I, has attracted wide attention and has been investigated by over 35 different research groups. In 1996, the FZI presented the specification of an extended model, called the “Fault-Tolerant Production Cell” or Production Cell II [6]. This second model exposes more and richer issues related to failures and fault tolerance. Because devices, sensors and actuators can fail, the required control program is necessarily much more complex than the program for the original, non-fault-tolerant production cell.

This paper is organized as follows. Following a brief description of the CA action concept and the Fault-Tolerant Production Cell, Section 3 presents an analysis of possible failures. Section 4 describes a design of a control system. Sections 5 and 6 describe the formal validation of this design. Section 7 concludes the paper.

2: CA Actions and Production Cell II

Real-world applications often give rise to complex concurrent and interacting activities. An effective mechanism is required for controlling and coordinating such activities. Due to their complexity, concurrent and distributed systems are also very prone to faults and errors. The CA action scheme [11] is motivated by the need to deal with general and complicated fault situations that occur in many real-world applications.

A CA action is a mechanism for coordinating multi-threaded interactions and ensuring consistent access to objects in the presence of concurrency and potential faults. CA actions can be regarded as providing a programming discipline for nested multi-threaded transactions [2] that in addition provides very general exception handling provisions. They augment any fault tolerance that is provided by the underlying transaction system by providing means for dealing with i) unmasked hardware and software faults that have been reported to the application level to deal with, and/or ii) application-level failure situations that have to be responded to.

The concurrent execution threads participating in a given CA action enter and leave the action synchronously. Within the CA action, operations on objects can be performed cooperatively by *roles* executing in parallel. To cooperate in a CA action a group of concurrent threads must come together and agree to perform each role of the

* Present address: Dept. of Computer Science, University of Durham.

action, with each thread undertaking a different role. Inside a CA action, some or of all its roles can be involved in further (nested) CA actions. If an error is detected within a CA action, appropriate recovery measures must be invoked cooperatively, by all the roles, in order to reach some mutually consistent conclusion.

Figure 1 shows an example in which two concurrent threads enter a CA action in order to play the corresponding roles. Within the CA action the two concurrent roles communicate with each other and manipulate the external objects cooperatively in pursuit of some common goal. However, during the execution of the CA action, an exception e is raised by one of the roles. The other role is then informed of the exception and both roles transfer control to their respective exception handlers $H1$ and $H2$ for this particular exception, which then attempt to perform forward error recovery. The effects of erroneous operations on external objects are repaired, if possible, by putting the objects into new correct states so that the CA action is able to exit with an acceptable outcome. The two threads leave the CA action synchronously at the end of the action.

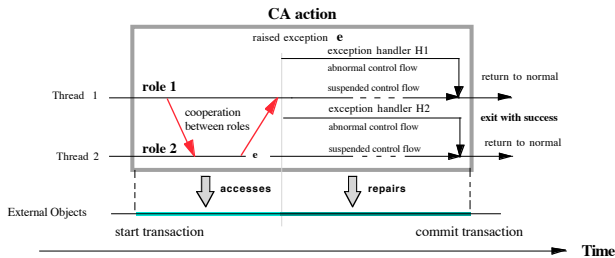


Figure 1 Example of a CA Action.

In general, the desired effect of performing a CA action is specified by an *acceptance test*. The effect only becomes visible if the test is passed. The test allows both a normal outcome and one or more exceptional (or degraded) outcomes, with each exceptional outcome signalling a specified exception to the surrounding environment. The CA action is considered to have failed if the action failed to pass the test, or roles of the action failed to agree about the outcome. In this case, it is necessary to try to undo potential effects of the CA action and signal an `abort` exception to the environment. If the CA action is unable to satisfy the “all-or-nothing” property (e.g. because the undo fails), then a *failure* exception must be signalled to the surrounding environment. Ideally, the execution of a CA action will only produce one of the following outputs: a normal outcome, an exceptional outcome, an `abort` exception, or a *failure* exception.

Production Cell II consists of six devices: two conveyor belts (a feed belt and a deposit belt), an elevating rotary table, two presses and a rotary robot that has two orthogonal extendible arms equipped with electromagnets (see Figure 2). These devices are associated with a set of sensors that provide useful information to a “controller”,

and a set of actuators via which the controller can exercise control over the whole system. The task of the cell is to get a metal blank from its “environment” via the feed belt, transform it into a forged plate by using a press, and then return it to the environment via the deposit belt. More precisely, the production cycle for each blank is: i) if the traffic light for insertion shows green, a blank may be added, e.g. by the blank supplier, to the feed belt, ii) the feed belt conveys the blank to the table, iii) the table rotates and rises to the position where the magnets of the robot are able to grip the blank, iv) arm 1 of the robot picks the blank up and places it into an unoccupied press, v) the chosen press forges the blank, vi) arm 2 of the robot removes the forged plate from the press and places it on the deposit belt, and vii) if the traffic light is green, the plate may be carried to the environment.

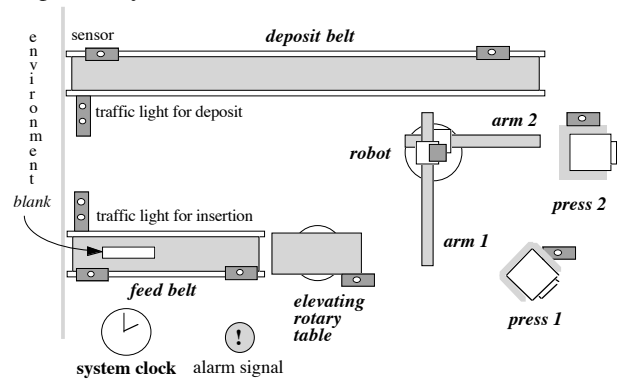


Figure 2 Fault-Tolerant Production Cell (Top View).

Basic System Requirements

A correct controller or control program must satisfy certain requirements specified by the Fault-Tolerant Production Cell model, namely:

Safety: i) device mobility must be restricted, ii) device collisions must be prevented, iii) blanks must not be dropped outside safe areas (i.e. feed belt, table, press, and deposit belt,) and iv) sufficient distance must be maintained between blanks.

Liveness: Any blank put into the cell via the feed belt must eventually leave the cell via the deposit belt and must have been forged by one of the presses; this property must still hold if one of the two presses fails.

Failure Detection and Continuous Service: When any of a large number of defined failures occurs, it must be detected and unless it just concerns one of the presses the system must be stopped in a safe state. After recovery from the failure, which typically would require action by the user of the cell, the system should be able to resume operations starting from this safe state. Similarly, after a failed press has been repaired, it should be able to resume its contributions to the production process. (Certain safety requirements can no longer be met if some special failures occur, e.g. a blank is dropped outside safe areas, but other safety properties must still be guaranteed.)

Clock, Stop Watches and Alarm Signals

The Fault-Tolerant Production Cell model provides a global system clock that gives the current time at any instant. Based on this system clock, a control program can implement several stop watches supervising individual processes, e.g. the movement of the feed belt. The model also provides an alarm signal mechanism for reporting component failures to the user of the cell. The control program is required to switch on the alarm whenever a failure is detected — it is switched off by the user when the failed device has been repaired.

3: Failure Definitions and Analysis

The major assumptions made in the Fault-Tolerant Production Cell model, as defined by FZI, are:

- 1) The system clock, two traffic lights, and the alarm signal mechanism are fault-free and do not fail.
- 2) Values of sensors, actuators and clocks are always transmitted correctly without any loss or error.
- 3) No failure can cause devices to exceed certain limiting positions; in the worst case devices are stopped automatically.
- 4) All sensor failures are indicated by sensor values.
- 5) All actuator failures will cause devices to stop.

For a given device, we classify possible failures into: i) sensor failures, ii) actuator failures, and iii) lost or stuck blanks. We also show how a failure can be detected by sensors, actuators, stopwatches, singly or in combination. In many cases certain different types of failure cannot be distinguished using just the on-line information available. We therefore discuss failure detection only, and assume that fault diagnosis and subsequent device repair are performed off-line. Due to limitations of space, we illustrate the analysis that we performed by treating just the case of a single failure of a press; for a complete treatment, see [12].

Sensor Failures: There are four sensors associated with each press, one reporting whether a blank is in the press (called *blank sensor*), and others reporting press positions. A failure of the blank sensor can be detected by checking whether a robot arm has transferred a blank to or from the press. The failure of a sensor that reports press positions can be detected by using a stop watch to measure the moving time of the press and by checking other sensor values on press positions.

Actuator Failures: Failure modes for the actuators that move the lower part of a press include: no response (i.e. cannot move), and a moving press unexpected stopping, which can be detected by checking values of the press position sensors and values of stop watches.

Stuck or Lost Blank: This failure can be detected only by checking the value of the sensor that reports whether a blank is in a press.

In order to detect various failures of sensors and actuators as well as lost blanks, appropriate detection measures must be incorporated into the control software. Assertion statements are a common form of failure detection measure. For example, after the control program has sent a control command to the robot and asked the robot to drop a blank into press 1, the value of the sensor that reports a blank in the press must be checked by an assertion. If the sensor returns 0, indicating that no blank is in press 1, then an appropriate exception must be raised.

There are several possibilities that could have caused this exception: i) the blank might have been lost, ii) arm 1 of the robot might have failed to drop the blank, and iii) the sensor of press 1 might have failed to report that the blank has been dropped into the press. Unfortunately, our analysis showed that distinguishing these failures from each other at run-time is extremely difficult. In most cases, if a failure occurs and thus an exception is raised, the cell will simply have to be stopped in a safe state, if at all possible, for the user to deal with.

Failures of sensors that report press positions and failures of the press actuator can be detected by assertion statements and identified unambiguously with the aid of stopwatches. Such failures must be reported to the user through the alarm. However, because the Fault-Tolerant Production Cell has two presses, normal operations can be maintained using a single press, albeit with some performance degradation.

A device or sensor failure should not affect normal operations of other devices. For example, when a failure of the robot occurs and is handled by the control program, the deposit belt should still deliver an already forged blank, if there is one, to the blank consumer. In the following, we will demonstrate how CA actions can confine damage and failures effectively, and minimize the impact of component failures on the entire cell.

4: Design of a Control Program

The main characteristics of our design are the way it separates safety, functionality, and efficiency concerns among a set of CA actions, which thus can be designed, and validated, independently of each other, and of the set of device/sensor-controllers that dynamically determine the order in which the CA actions are executed at run-time. In particular, the safety requirements are satisfied at the level of CA actions, while the other requirements are met by the device/sensor-controllers. There is a detailed discussion in [13] as to how these design decisions were made and why we used actions to enclose the interaction between certain devices in our control program developed for Production Cell I. Our design for Production Cell II follows a similar strategy. It includes 12 main CA actions; each action controls one step of the blank processing and typically involves passing a blank between

two devices. Any device can move only within a CA action.

There are six concurrent execution threads in the control program, corresponding to the six devices, each of which threads basically performs a simple endless loop. All device movements are performed within CA actions, and the devices involved in each action are switched off before the action is left, so that when not under the control of an action each device is stationary. Two additional threads model activities in the environment: *BlankSupplier*, and *BlankConsumer*. Note that *FeedBelt* is responsible for controlling the traffic light that indicates when another blank can be inserted, while *BlankConsumer* is responsible for controlling the light for deposit. A blank is designed as an external object with respect to the top-level CA actions. Usually, one role of a CA action takes the blank as an input argument, and the device corresponding to this role passes it to another role which returns it as an output argument. Figure 3 portrays the 12 related CA actions as overlays on the FZI simulator diagram [6].

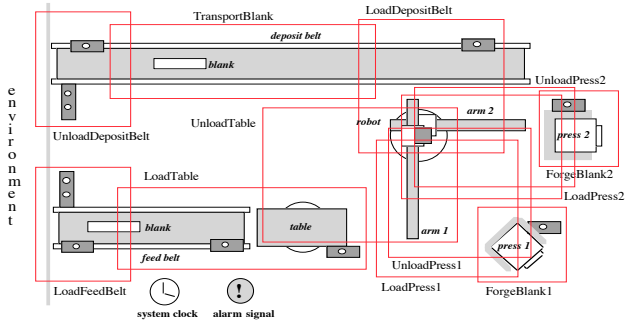


Figure 3 CA Actions That Control Production Cell II.

An intersection between CA actions in Figure 3, e.g. between *TransportBlank* and *LoadDepositBelt*, represents the fact that those CA actions cannot be executed in parallel. The mutual exclusion feature of CA actions guarantees that a blank or a device cannot be involved in more than one action at a time so that neither blanks nor devices can collide. Even if the actions that devices participate in are invoked in the wrong order, the result will be at worst a safe deadlock. Each device-controller (i.e. an execution thread) is responsible for dynamically specifying the sequence of actions that the device will participate in. For example, the *robot* thread can skip all the CA actions related to one of the presses if this press has failed.

4.1: Design of CA Actions

Our design assumes that an action will begin only if its pre-conditions are valid, and that if no exception is raised during the execution of an action then its post-conditions will hold (though this could, if so wished, be checked using an acceptance test). For a given action, these conditions are used to ensure that the execution of that action will not violate in any way the system

requirements given in Section 3, especially those related to safety and fault tolerance. Due to limitations of space, we take just the action *LoadPress1* as an example.

CA action *LoadPress1*

pre-conditions	post-conditions
robot off	Robot off
Blank on arm 1	No blank on arm 1
Both arms retracted	Both arms retracted
Robot at one of the defined angles	Robot angle: arm 1 towards press 1
press 1 off	Press 1 off
no blank in press 1	Blank in press 1
press 1 in bottom position	press 1 in middle position

Values of the related sensors or states of the related actuators that can be used to check these conditions are identified in our detailed design to facilitate the actual implementation of a control program (see [12]).

Figure 4 illustrates the interactions (themselves involving nested CA actions) between the participating threads within the *LoadPress1* action. This action has four roles: *Robot*, *Press1*, *RobotSensor*, and *Press1Sensor*, and represents the co-operation that arranges for arm 1 of the robot to drop a blank into press 1.

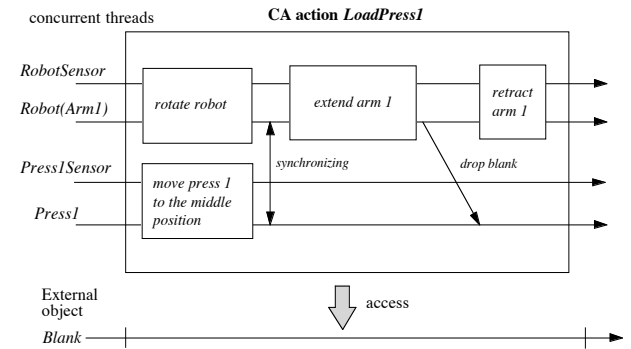


Figure 4 CA Action *LoadPress1*.

Action *LoadPress1* is described below using the COALA notation, which was developed for the formal specification of CA actions [10]. Our Java implementation of the control program is based on a set of pre-defined templates for CA actions that can be used to implement CA action designs specified in COALA.

```

CAA LoadPress1;
Interface
Use
    MetalBlank;
Roles
    Robot: blankType, robotActuator;
    Press1: blankType, press1Actuator;
    RobotSensor: arm1ExtensionSensor, ...;
    Press1Sensor: blankSensor, ...;
Exceptions
    Press1Failure, Arm1Failure1, ...;
signal
    ;exceptions to
Body
    Use CAA
        ;;specify nested actions
    RotateRobot, MovePress1toMiddle, ...;

```

```

Object
  robotPress1Channel: Channel;
                        ;;shared local objects
Exceptions
  press1_failure, blank_sensor_failure, ...;
                        ;;internal exceptions
Handlers
  press1_handler, blank_sensor_handler, ...;
Resolution
  press1_failure -> press1_handler, ...;
                  ;;exception      resolution
graph
  Role Robot(...);
  Role Press1(...);
  ... ..
End LoadPress1;

```

The exceptions declared in the **Interface** part of an action are those that can be signalled to the enclosing action. The roles of an action can signal an exception directly but must guarantee that the exception that is signalled has been agreed by all the roles of that action. In the case of abortion or failure, the CA action support mechanism will enforce the abortion and signal the appropriate exception, either `abort` or `failure`, to the enclosing action. When multiple exceptions are raised within an action, the CA action support mechanism controls the execution of a resolution algorithm based on an exception resolution graph declared in the **Resolution** part. After a resolving exception is identified, the corresponding handler declared in the **Handlers** part will be invoked (see Section 4.3).

An exception handler will attempt to bring the system back to normal. If it is successful, the CA action will end with a normal outcome. However, in most situations the handler can only provide some degraded service, i.e. an exceptional outcome, and must signal the corresponding exception. Again, in the case of abortion or failure, the CA action support mechanism will take control. If a further exception is raised during the execution of an exception handler, control is transferred to the CA action support mechanism immediately and the action must either abort or signal a `failure` exception.

4.2: Dealing with Component Failures

We first investigate situations involving single faults, i.e. we assume that *only one component failure can occur before the system is brought if necessary to a safe stop, and the component is repaired*. During the execution of a CA action, if a failure (of a component involved in this CA action) occurs and is detected by an assertion statement or an acceptance test, a corresponding exception will be raised within the action by one of its roles. The exception is propagated immediately to the other roles of the action and all roles then transfer control to their exception handlers for this exception so that they can attempt to perform appropriate error recovery. In most cases when a component failure takes place in the cell, it is not possible to recover completely from the error and the *normal* post-conditions of the action can no longer be satisfied. Thus, exceptional post-conditions with respect to various given failures must be defined to specify the exceptional outcomes of an action.

By way of example, we outline the basic requirements for the handlers of two different exceptions:

Handler for the Press 1 Failure: The `LoadPress1` action performs forward error recovery by moving the robot to an appropriate position so that it will be able to put the unforged blank, which is still on arm 1, into press 2 once the press is available.

Handler for the Rotary Sensor or Motor Failure: (In this case, action `LoadPress1` fails to rotate the robot to the intended position.) The action will simply use backward error recovery to attempt to move the robot back to its initial position and rotate it again. If the failure persists, the action will produce an exceptional outcome as defined below.

For the `LoadPress1` action, we identify seven exceptional outcomes and corresponding exceptional post-conditions (see [12]). By way of example, the following table illustrates the exceptional outcome when press 1 fails. Different exceptional outcomes may lead to different states of the production cell. For example, the exceptional outcome caused by just a press 1 failure corresponds to the situation where the production cell continues with only one operational press. On the other hand, since the blank sensor is a redundant component of the cell, if both presses are still operational its failure merely requires a report to be made to the user of the cell. However, the other five outcomes will have to stop the entire cell in a safe state.

<i>exception to signal</i>	<i>Exceptional post-conditions</i>
press 1 failure	robot off
	Blank on arm 1
	Both arms retracted
	Robot angle: arm 1 towards press 2
	press 1 off
	no blank in press 1

By means of such analyses, given the way in which CA actions enable the different failure situations to be treated independently of each other, the design of the actual set of handlers for the various exceptional outcomes of each of the 12 top-level CA actions becomes rather straightforward – full details can be found in [12].

4.3: Dealing with Concurrent Failures

In the interests of simplicity, we assume that *only two failures may occur within the same time interval before the system is stopped and the related components repaired*. Some concurrent failures can be covered implicitly by the corresponding single failure situation. Others may need different handling and require separate post conditions. The following table shows post-conditions for an example pair of concurrent failures:

<i>exception to signal</i>	<i>Exceptional post-conditions</i>
(rotary sensor or motor failure) & press 1 failure	robot off
	Blank on arm 1
	Both arms retracted
	press 1 off
	No blank in press 1

```

UnloadPress1.Press(plate)
-- activate action UnloadPress1 }

```

5: Validating Properties of the Cell

The failure of the robot's rotary sensor or motor can be detected automatically and indicated by a special sensor value. However, the returned sensor value does not indicate which component, i.e. the sensor or the motor, actually failed. This causes difficulty in performing effective error recovery. In such circumstances the control program is designed simply to bring the system to a stop in a safe state, so that off-line diagnosis can be performed.

For each (enclosing or nested) action, various exceptions are defined based on failure analysis and an exception graph for resolving concurrent exceptions is defined. For example, the `LoadPress1` action may give rise to exceptions such as `pr1_failure` (press 1 failure), `b_sensor_failure` (blank sensor failure), `arm1_failure1` (blank lost), `arm1_failure2` (cannot drop the blank), etc.

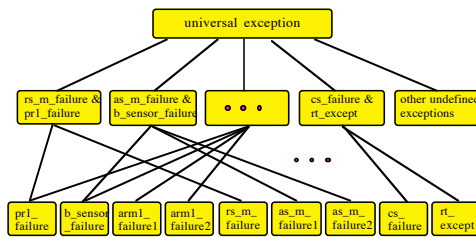


Figure 5 Exception Graph for CA Action `LoadPress1`.

An exception graph for this action is shown in Figure 5. For example, if both press 1 and the robot rotation motor fail simultaneously, this exception graph will be searched and the resolving exception `rs_m_failure & pr1_failure` will be raised instead of the individual exceptions `rs_m_failure` and `pr1_failure`, so that a suitable handler for this particular situation can be invoked. Any undefined exception pairs will not be resolved and will simply lead to the raising of the `universal` exception. (The handler for the `universal` exception is responsible for stopping the system and leaving the cell in a pre-defined safe state, if possible.)

4.4: Design of Device-Controllers

Device/sensor-controllers are used to determine dynamically the order in which the CA actions are executed. Eight controllers are designed: `FeedBelt`, `Table`, `Robot`, `Press1`, `Press2`, `DepositBelt`, `Supplier`, and `Consumer`. Two queue objects are defined in order to improve the flexibility of operations of both the robot and the deposit belt: `robotQueue` and `depositBeltQueue`. The `Press1` controller is shown below as a simple example:

```

Press1Controller:
loop forever {
robotQueue.put(PRESS1_FREE)
-- put message in robotQueue
LoadPress1.Press(plate)
-- activate action LoadPress1
ForgeBlank1.Press(plate)
-- activate action ForgeBlank1
robotQueue.put(FORGED_PLATE_IN_PRESS1)
-- put message in robotQueue
}

```

We had earlier developed a general scheme for formalizing CA action-based designs of finite systems as state transition systems specifically for the purpose of checking system properties such as liveness, safety and fault tolerance [1]. This general approach assumes that a set of controlling processes is defined together with a set of CA actions that are utilized by the controllers, and enables the system behaviour to be formalized in terms of its operations on the global objects in the system that are external to all CA actions.

The state transition system corresponding to a CA action-based design is characterized by its (global) state-space, a set of initial states and a `next-state` relation. The global state-space is composed from the global objects and the state-spaces of the CA actions, representing the kind of outcome – normal or exceptional – produced by each CA action and encoding whether its roles are idle or activated. The initial states are supposed to satisfy two kinds of properties: i) any application specific requirements that need to be considered, and ii) the requirement that initially all roles should be idle and no exception should have been signalled.

The `next-state` relation defines the computation paths that are possible in the system. This corresponds essentially to four kinds of activities that may occur in the system: i) a controlling process may call and thus invoke a role from a CA action, thereby activating it, ii) if all roles of a CA action have been activated, the CA action may be executed according to its interface specification given in terms of pre- and post-conditions for both normal and exceptional outcomes, iii) after a CA action has been executed, a return is issued from its roles to the corresponding controlling processes that called them, and iv) a controlling process may execute an (internal) action in which no CA action is involved.

Due to the atomicity of CA actions and since internal actions of controlling processes are independent from each other, it is sufficient to view only interleaving occurrences of state transitions. Thus we have modelled the `next-state` relation to encode the interleaving semantics. We have used SMV [8] to represent the state transition system so obtained. (The technical details of representing a CA action-based design in SMV and the properties of a system in CTL were described in [1].)

Model-checking is a technique for analyzing whether a given set of behavioural properties is satisfied by a given model. It is similar to exhaustive testing of a finite instance, and it produces a counterexample whenever a property is violated. We have model-checked a significant proportion of the safety, liveness, and fault-tolerance requirements for the Production Cell II case study. The properties are expressed in terms of CTL formulae over the transition system for the CA action-based design

formalized in SMV. CTL allows several temporal modalities to be used for expressing properties over the behaviour of a system; we have mainly used the AG (“henceforth”) operator for expressing properties that are to hold in all reachable states and the AF (“eventually”) operator for expressing properties that are expected to eventually hold in some reachable state.

We are mainly concerned with fault-tolerance requirements which express properties over the behaviour of a system despite the occurrence of a failure. These may include safety and liveness properties. If `tolerable` is a formula describing states where there are no faults, or only those faults that are supposed to be tolerated by the system, and if `P` expresses some desired property, then formula

```
AG (tolerable -> P)
```

expresses that along each execution path property `P` is valid if and faults that occur are tolerable ones, i.e. `P` is treated as a (conditional) safety property. Similarly for liveness: the formula

```
AF (tolerable -> P)
```

expresses that along each execution path either a state satisfying `P` will be reached or a non-tolerable fault will occur. This means that `P` will eventually become true along each path where at most tolerable faults occur, i.e. `P` is treated as a (conditional) liveness property.

If we set `tolerable` equal to true then we express that any (modelled) failure should be tolerable. In this case the conditional safety and liveness properties reduce to the non-conditional forms `AG P` and `AF P`. Properties that are only expected to hold in the case that no fault occurs now need to be written in the conditional form where formula `tolerable` characterises the fault-free cases.

We illustrate this scheme for formalizing properties with the main fault-tolerance requirement, i.e. the “continuous service requirement” which states that the system will continue to operate in a degraded manner, even if one of the presses fails. Such failures are signalled by the CA action `LoadPress1` as `press1_failure` or `b_sensor_failure`. The property encoding the continuous service requirement expresses that if a `press1_failure` or a `b_sensor_failure` occurs, then any blank in other devices of the Production Cell or blanks inserted afterwards will be processed and arrive on the deposit belt unless another failure occurs later. This is formalized here for a blank with name `id1` on the feed belt:

```

“If a press1 related failure occurs ...”
AG (loadpress1.signal in
  { press1_failure, b_sensor_failure } ->
  “... then a blank (named id1) on the feed belt ...”
AG (blank_on_feed_belt.id = id1 ->
  “... will eventually, if only tolerable failures occur,
...”
AF ((loadpress1.signal in { normal,
  press1_failure, b_sensor_failure
}))
“... arrive on the deposit belt”

```

```
-> blank_on_end_deposit_belt.id = id1)))
```

We have created two SMV models and actually performed model-checking on a Sun Sparc Ultra-II. The general model contains 1251 lines of SMV code, while the simplified model has 1079 lines of the code. The potential state space of these models is about $10^{26} - 10^{34}$. The models were instantiated with the number of blanks. When the general model produces no result for two or more blanks after one week of computation, the simplified model responds in all possible situations within at most 52824.1 seconds.

6: Design for Validation

The analysis of properties of Production Cell II was carried out in parallel with the development of its CA action based design. Model-checking helped us to find several flaws in early versions of the design. By analyzing the causes for failed proofs of the required properties, we have been able to derive corresponding solutions. The flaws we found affected both the fault tolerance and the coordination aspects of the CA action-design of the cell. The results from the formal analysis have directly contributed to refining and improving our design.

To take just one example, we identified a problem that affected the order in which the robot interacts with the devices around it. The problem does not occur in the single blank instance of our model and thus it is hard to detect by just reviewing the specification text. If two blanks are in the system then the robot could manoeuvre itself into a situation from which no further activities were possible. Such “critical” sequences of actions can be derived from counter-example paths generated by the model-checker. The counter-example also helps in finding solutions to the detected problem: we dealt with this particular problem by enabling the occurrence of the next appropriate actions after such critical sequences. This was done by appropriately weakening the preconditions of the actions to be executed next

7: Conclusions

Unlike the first Production Cell model, in the “Fault-Tolerant Production Cell” failures of electro-mechanical components are of major concern. This requires a control program that is much more complex than the program developed for the original cell, though it follows the same general strategy, i.e. using CA actions where there are safety-critical interactions. In order to develop the required control program, we have conducted an analysis of possible component failures and identified the various ways of detecting these failures. We have used the results of this analysis to guide the design of a system employing what is in fact a very sophisticated exception handling scheme, capable of dealing appropriately even with concurrent occurrences of any of the wide variety of possible failures defined in the FZI.

We have implemented our control system using a Java implementation of a distributed CA action support

scheme [12]. (This scheme makes use of the nested multi-threaded transaction facilities provided by the Arjuna transaction support system [9].) During the testing phase and the demonstration of our implementation, all injected device or sensor failures were caught successfully and handled immediately by our control program. Even a previously unknown software bug in the original FZI simulator was also detected by the acceptance test of a CA action and recovered by the retry operation associated with the action. We are now in the stage of collecting experimental data for further dependability and performance-related evaluation.

As a result of the experience we have gained during the process of formalizing and designing this control software, we feel that we now have a much fuller understanding of CA actions and the design issues involved in their implementation. It was very pleasing to confirm that the much more complex requirements of Production Cell II could be satisfied by what was in fact a straightforward though large extension of the approach we had used in Production Cell I [13]. This again enabled all the dependability (and especially the safety) related aspects of the problem to be solved very directly using just the CA action mechanism, despite the need to add very extensive exception handling strategies. It was also pleasing to confirm that the CA action structuring greatly aided not just the design but also the validation of the control program, in this case by model-checking.

In light of the fact that the original Production Cell was the subject of extensive studies using various formal approaches, we should emphasize that to the best of our knowledge our work represents the first and so far only complete formal analysis and validation of a design for the much more complex and realistic Production Cell II. The work in [7] describes a system design for Production Cell II that focuses just on a dynamic and transparent reconfiguration scheme that preserves safety properties. Our design is essentially different, and focuses mainly on cooperation between devices during both normal execution and the process of exception handling. A Formal Risk Analysis approach was developed in [5] for analyzing the run-time behaviour of Production Cell II, and studying how various sensor and actuator faults could affect both system reliability and safety. However, their analysis is not complete, and only uses the elevating rotary table of the Production Cell as an example. In contrast, our analysis is much more comprehensive and complete, including the classification of various failures and the identification of possible failures related to every device in the cell. This analysis leads further to the design of a complete control system and an actual, workable implementation.

The design style we have been using was one that we arrived at through very specific consideration of the problems raised by the Production Cell examples. We now realize that a more methodical and general means of

arriving at the design of CA action-based programs is possible, as well as being highly desirable [3].

Acknowledgements

This work was supported by the ESPRIT Long Term Research Project 20072 on "Design for Validation" (DeVa).

References

- [1] E. Canver, D. Schwier, A. Romanovsky and J. Xu, "Formal Verification of CAA-based Designs: The Fault-Tolerant Production Cell," 3rd Year Report, *ESPRIT Research Project on Design for Validation*, pp.229-258, Nov. 1998.
- [2] S.J. Caughey, M.C. Little and S.K. Shrivastava, "Checked Transactions in an Asynchronous Message Passing Environment," in *1st Intl. Symp. OO RT Distr. Computing*, Kyoto, pp. 222-229, April 1998.
- [3] R. DeLemos and A. Romanovsky, "Co-ordinated Atomic Actions in Modelling Object Co-operation," in *1st Intl. Symp. OO RT Distr. Computing*, Kyoto, pp.152-161, April 1998.
- [4] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study "Production Cell"*, Springer, 1995.
- [5] P. Liggesmeyer and M. Rothfelder, "Improving System Reliability with Automatic Fault Tree Generation," in *28th Int. Symp. Fault-Tolerant Computing*, Germany, pp.90-99, June 1998.
- [6] A. Lötzbeyer, "Task Description of a Fault-Tolerant Production Cell," version 1.6, available from <http://www.fzi.de/prost/projects/korsys/korsys.html>, 1996.
- [7] G. Matos and E. White, "Application of Dynamic Reconfiguration in the Design of Fault-Tolerant Production Cell," in *4th Intl. Conf. Configurable Distr. Systems*, Maryland, USA, pp.2-9, 1998.
- [8] K.L. McMillan, "Symbolic Model Checking," PhD thesis, Carnegie Mellon University, Kluwer Academic Publishers, 1993.
- [9] G.D. Parrington, S.K. Shrivastava, S.M. Wheeler and M.C. Little, "The Design and Implementation of Arjuna," *USENIX Computing Systems Journal*, vol.8, no.3, 1995.
- [10] J. Vachon, D. Buchs, M. Buffo, G.D.M. Serugendo, B. Randell, A. Romanovsky, R.J. Stroud, and J. Xu, "COALA - A Formal Language for Coordinated Atomic Actions," 3rd Year Report, *ESPRIT Research Project on Design for Validation*, Oct. 1998.
- [11] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R.J. Stroud and Z. Wu, "Fault Tolerance in Concurrent Object-Oriented Software through Co-ordinated Error Recovery," In *25th Int. Symp. Fault-Tolerant Computing*, Pasadena, pp.499-508, June 1995.
- [12] J. Xu, A. Romanovsky, A. Zorzo, B. Randell, R.J. Stroud and E. Canver, "Developing Control Software for Production Cell II: Failure Analysis and System Design Using CA Actions," 3rd Year Report, *ESPRIT Research Project on Design for Validation*, pp.167-188, Nov. 1998.
- [13] A.F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R.J. Stroud, and I.S. Welch, "Using Co-ordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study," 3rd Year Report, *ESPRIT Research Project on Design for Validation*, pp.139-166, Nov. 1998 (also to appear in *Software - Practice & Experience*.)