Rigorous Development of an Embedded Fault-Tolerant System Based on Coordinated Atomic Actions

Jie Xu, Brian Randell, Alexander Romanovsky, Robert J. Stroud, Avelino F. Zorzo, Ercument Canver, and Friedrich von Henke, *Member*, *IEEE Computer Society*

Abstract—This paper describes our experience using coordinated atomic (CA) actions as a system structuring tool to design and validate a sophisticated and embedded control system for a complex industrial application that has high reliability and safety requirements. Our study is based on an extended production cell model, the specification and simulator for which were defined and developed by FZI (Forschungszentrum Informatik, Germany). This "Fault-Tolerant Production Cell" 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. Based on an analysis of such failures, we provide in this paper details of: 1) a design for a control program that uses CA actions to deal with both safety-related and fault tolerance concerns and 2) 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 (involving possible clashes of moving machinery) to be treated independently. Even complex situations involving the concurrent occurrence of any pairs of the many possible mechanical and sensor failures can be handled simply yet appropriately. The formal verification activity was performed in parallel with the design activity and the interaction between them resulted in a combined exercise in "design for validation"; formal verification was very valuable in identifying some very subtle residual bugs in early versions of our design which would have been difficult to detect otherwise.

Index Terms—Concurrency, coordinated atomic (CA) actions, embedded fault-tolerant systems, exception handling, object orientation, formal verification, model checking, reliability, safety.

1 Introduction

Many embedded systems today are designed in an ad hoc fashion. They are tuned for specific operating environments and validated through extensive testing. Research on the systematic design and development of embedded systems has traditionally focused on functionality, performance, and hardware reliability. However, the software and systems are becoming more and more complex and they often involve concurrent tasks such as low-level feedback control, supervision logic, data logging, and interprocess communications. Controlling such system complexity and ensuring system consistency and completeness with respect to various quality requirements are becoming a great challenge. We need better technologies—adequate solutions to the technical complexity of building a dependable embedded system that can be relied upon, even in the occurrence of failures and user's errors and when the system environment and user's requirements change. The goal of our work is, therefore, to investigate a rigorous approach to the development of embedded fault-tolerant systems for critical applications. In particular, we will examine the feasibility of using coordinated atomic (CA) actions [19] as a structuring tool to manage the system complexity and to design a fault-tolerant control program for a realistically detailed model of an industrial production cell that contains some redundant devices and sensors. We will then use model-checking to debug, improve, and verify the design formally.

An industrial production cell model, based on a metal-processing plant in Karlsruhe, Germany, was first created by the FZI in 1993 [8], within the German Korso Project, in order to evaluate and compare 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 version of the original production cell, called the "Fault-Tolerant Production Cell" or Production Cell II [10]. This second model, which has an additional press, extra sensors, and warning light systems to facilitate fault detection and fault tolerance, is much more complex and realistic than Production Cell I. Unlike the first model, failures of electro-mechanical components and sensors in Production Cell II are of major concern. In

- *J. Xu is with the University of Durham, DH1 3LE, UK. E-mail: Jie.Xu@durham.ac.uk.*
- *B. Randell, A. Romanovsky, and R.J. Stroud are with the University of Newcastle upon Tyne, NE1 7RU, UK.*
- *A.F. Zorzo is with the Faculdade de Informatica, PUCRS, 90619-900, Brazil.*
- *E. Canver and F. von Henke are with the University of Ulm, D-89069 Ulm, Germany.*

Manuscript received 15 Oct. 2000; revised 16 May 2001; accepted 19 June 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114413.

particular, the cell is intended to be used to provide continued service even if one of the two presses is out of order. However, to the best of our knowledge, very little work has been done on Production Cell II, especially regarding the development and formal treatment of different control programs.

The original, rather simplistic, production cell model assumes no device or sensor failures occur. Under such assumptions, we used the CA action concept to organize and design a control program and implemented it in Java [24]. The control program that we developed was then applied to an FZI-provided Tcl/Tk simulator, demonstrating how functional and safety-related requirements can be satisfied separately by controlled multithreaded cooperation and the strict enclosure of all interactions between cooperating devices within CA actions.

The Fault-Tolerant Production Cell exposes more and richer issues related to failures and fault tolerance and it is therefore a valuable case study for investigating and developing embedded fault-tolerant computer systems. Because devices, sensors, and actuators can fail, the required control program is necessarily much more complex than the program that we developed 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 model, Section 3 presents an analysis of the possible failures of the various devices and sensors as defined by FZI. Section 4 describes a design of a control program that uses CA actions both as a basic structuring tool and as a unified framework for handling exceptional situations. Sections 5, 6, and 7 study the formal treatment of CA action-based designs, formalize important properties of Production Cell II, and examine those properties by model-checking. Sections 8 and 9 discuss an implementation of the control program and then conclude the paper.

2 Preliminaries: CA Actions and Production Cell II

An embedded computer system is a system that is "built-in" or "embedded" within its environment. It has close interactions with its environment by partly or wholly controlling the functionality and the operation of the environment. The system receives signals from its environment, collected by the various sensors or by one or more communication channels, and sends control signals to its environment, usually directed to actuators. An effective supporting mechanism is often required for controlling and coordinating these concurrent and interacting activities. Also, due in no small measure to their complexity, embedded computer systems are very prone to faults and errors. Various fault tolerance techniques for coping with hardware and software faults can provide a practical way of improving the dependability of such systems. These typically use fault masking or backward error recovery. However, because faults can have an impact on, or arise from, the environment of a computer system [1], some forms of error recovery may require stepping outside the boundaries of a computer system (i.e., considering the computer system and its environment recursively as an entire distributed system at a higher level of abstraction), in which case, backward error recovery by the computer system will normally not suffice and fault masking is infeasible.

In current practice, however, the majority of fault-tolerant computing systems do not attempt to tolerate software faults or to facilitate provision of means of recovering from errors that affect both the computer system and its environment—rather, they concentrate on the problems that arise from operational faults (typically hardware faults). For example, many software systems that use the concept of atomic (trans)actions to construct fault-tolerant distributed applications generally assume that erroneous outputs can be detected before transaction commitment occurs and that user programs are correct. The CA action scheme is motivated by the need to deal with the more general and complicated fault situations that occur in many real-world applications.

2.1 Coordinated Atomic Actions: Overview and Example

A CA action is a mechanism for coordinating multithreaded 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 multithreaded transactions [3] 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 1) unmasked hardware and software faults that have been reported to the application level to deal with and/or 2) application-level failure situations, including failures in the environment of a system, that have to be responded to. (In fact, CA actions also provide a convenient structuring mechanism for using software fault tolerance by means of software design diversity at the application level, but this use of them is not considered further in this paper.)

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 action, with each thread undertaking a different role. Inside a CA action, some or all of its roles can be involved in further (nested) CA actions. If an error is detected inside a CA action, appropriate forward and/or backward recovery measures must be invoked cooperatively, by all the roles, in order to reach some mutually consistent conclusion. To support backward error recovery, a CA action must provide a recovery line that coordinates the recovery points of the objects and threads participating in the action so as to avoid the *domino effect* [15]. To support forward error recovery, a CA action must provide an effective means of coordinating the use of exception handlers. An *acceptance test* can and ideally should be provided in order to determine whether the outcome of the CA action is successful. Error recovery for participating threads of a CA action generally requires the use of explicit error coordination mechanisms, i.e., exception handling or backward error recovery within the

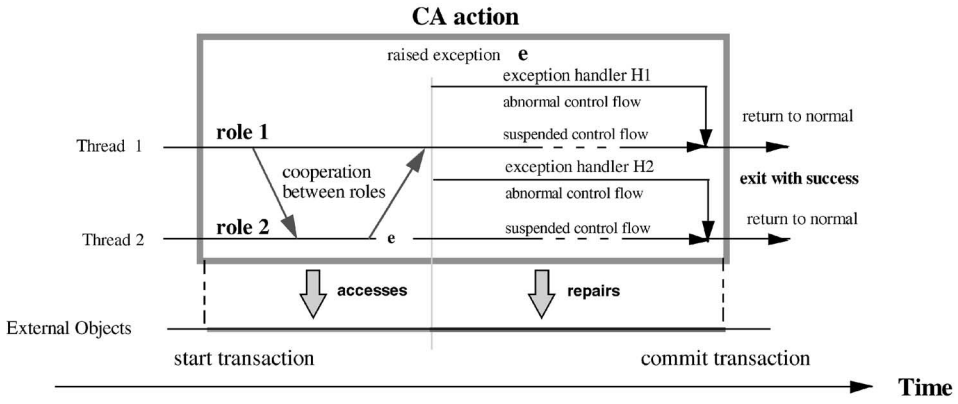


Fig. 1. Example of a CA action.

CA action; objects that are external to the CA action and so can be shared with other actions and threads must provide their own error coordination mechanisms. These external objects, which are in effect being competed for, must behave atomically with respect to other CA actions and threads so that they cannot be used as an implicit means of "smuggling" information [7] into or out of a CA action.

Fig. 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. (When multiple exceptions are raised within an action, a resolution algorithm based on an exception resolution graph [1], [20] is used to identify the appropriate exception and, hence, the set of exception handlers to be used.) 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. (As an alternative to performing forward error recovery, the two participating threads could undo the effects of operations on the external objects, roll back, and then try again. This would, in general, require the use of diversely designed software alternates if the aim was to tolerate residual design faults, though a simple "retry" strategy can be effective when the fault is in effect transient [6].)

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 acceptance test allows both a normal outcome and one or more exceptional (or degraded) outcomes, with each exceptional outcome signaling 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 the potentially visible effects of the CA action and signal an `abort` exception to the surrounding 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 signaled to the surrounding environment (in general, an enclosing CA action). This `failure` exception indicates that the CA action has not passed its acceptance test and its effects have not been undone so that the system has probably been left in an erroneous state, which it is now the responsibility of the environment to deal with. Thus, ideally, execution of a CA action will only produce one of the following four forms of outputs: a normal outcome, an exceptional outcome, an `abort` exception, or a `failure` exception.

If an exception is raised during the normal execution of a CA action, then control is passed to the corresponding exception handler for each role. If two or more exceptions are raised at the same time, then a process of exception resolution must take place first [23]. However, if an exception is raised during the execution of an exception handler, then the underlying CA action support mechanism will attempt to abort the action or else signal a `failure` exception to the enclosing action. Once exception handling begins within a CA action, it is not possible to resume normal execution of the CA action, but it is possible for the exception handlers to terminate normally or exceptionally, depending on the extent to which error recovery is successful. Again, all roles must still agree about the outcome.

A role may signal `abort` or `failure` at any time. For the purposes of determining the outcome, `failure` takes precedence over `abort`, which takes precedence over every other exception that can be raised internally. In fact, for a given action, exception handlers can only be provided for exceptions that are raised internally within that action. Exceptions that are signaled by an action, including `abort` and `failure`, will be handled at the level of the enclosing action.

2.2 The Fault-Tolerant Production Cell

The Fault-Tolerant Production Cell 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 Fig. 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

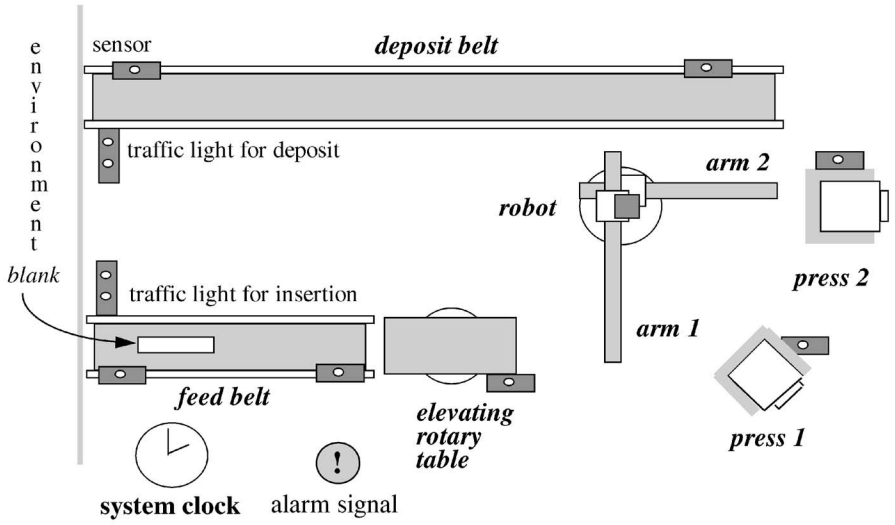


Fig. 2. The fault-tolerant production cell (top view).

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:

1. If the traffic light for insertion shows green, a blank may be added, e.g., by the blank supplier, to the feed belt from the environment,
2. The feed belt conveys the blank to the table,
3. The table rotates and rises to the position where the magnets of the robot are able to grip the blank,
4. Arm 1 of the robot picks the blank up and places it into an unoccupied press, either press 1 or press 2,
5. The chosen press forges the blank,
6. Arm 2 of the robot removes the forged plate from the press and places it on the deposit belt, and
7. If the traffic light for deposit is green, the plate may be forwarded further and carried to the environment where a container may be used, e.g., by the blank consumer, to store the forged pieces.

(Normally, both presses are used and a certain amount of interleaving of two such production cycles, one for each press, is possible.)

Note that the controller can be implemented in hardware and/or software. In the following, we will investigate a software-implemented controller only. Our design and control program will support a varying, adaptive operating sequence of robot actions in order to achieve high flexibility.

2.2.1 Basic System Requirements

A correct control program must satisfy certain requirements specified by the Production Cell II model, namely:

Safety:

1. Device mobility must be restricted,
2. Device collisions must be prevented,
3. Blanks must not be dropped outside safe areas (i.e., feed belt, table, press, and deposit belt), and
4. 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. In addition, 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 production 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, e.g., restricted device mobility.)

Other requirements, such as flexibility and efficiency, may be taken into account, but must not conflict with the above requirements.

2.2.2 System Clock, Stop Watches, and Alarm Signals

The Production Cell II 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 Production Cell II model also provides an alarm signal mechanism for reporting component failures to the user of the production cell. The control program is required to switch on the alarm signal whenever a failure is detected—it is switched off by the operator when the failed device has been repaired.

3 Failure Definitions and Analysis

Before defining and analyzing various possible failures, we state the major assumptions made in the Fault-Tolerant Production Cell model, as defined by FZI [8], [10]:

Assumption 1: The system clock, two traffic lights, and the alarm signal mechanism are fault-free and do not fail.

Assumption 2: Values of sensors, actuators, and clocks are always transmitted correctly without any loss or error.

Assumption 3: No failure can cause devices to exceed certain limiting positions; in the worst case, devices are stopped automatically.

Assumption 4: All sensor failures are indicated by sensor values. Boolean sensors return a zero value and enumeration type sensors return a specified value that represents a failure.

Assumption 5: All actuator failures cause devices to stop.

Now we can define and analyze various failures with respect to each of the six devices in the cell. For a given device, we classify possible failures into: 1) sensor failures, 2) actuator failures, and 3) lost or stuck blanks. We also show how a given failure can be detected by sensors, actuators, and stopwatches, singly or in combination. It is important to note that, in many cases, certain different types of failure cannot be distinguished using just the online information available. We therefore discuss failure detection only and assume that fault diagnosis and subsequent device repair are performed offline. Due to limitations of space, our discussion just treats the case of a single failure of either the robot or a press; for a complete treatment, see [21].

3.1 Failures of the Robot

Sensor Failures: There are three sensors associated with the robot—each sensor returns one of several predefined values about the position of one of the robot's arms or the robot's rotary position. Three electric motors are responsible for rotating the robot or extending/retracting its arms. Sensor failure or electric motor failure is indicated automatically by a special sensor value, but these two types of failure cannot be distinguished using this value alone.

Actuator Failures: There are three kinds of actuator associated with the robot and each has its own failure modes: 1) Failure modes of actuators that retract an arm of the robot include: no response (i.e., cannot move) and unexpected stopping of a moving arm, which can be detected by checking values of robot sensors; 2) failure modes of actuators that switch an arm magnet on or off include: no response (e.g., the arm cannot pick up or cannot drop a blank) and unexpected picking or dropping, which can be detected only by checking values of other devices interacting with the robot; and 3) failure modes of the actuator that rotates the robot that include: no response (i.e., cannot rotate) and unexpected stopping of the rotating robot, which can be detected immediately by checking values of the sensor that indicates the robot's rotary positions.

Lost Blank: This type of failure can be detected only by checking a group of sensor values from various devices interacting with the robot.

3.2 Failures of a Press

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. The 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 unexpectedly 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.

3.3 Failure Detection Measures

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 statement. 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: 1) The blank might have been lost, 2) arm 1 of the robot might have failed to drop the blank, and 3) the sensor of press 1 might have failed to report that the blank has been dropped into the press. If a powerful online diagnosis algorithm could identify this failure as the sensor failure, exception handling and error recovery would be quite straightforward—just report the exception to the user and continue normal operations of the cell. However, our analysis shows that distinguishing these failures from each other at runtime is extremely difficult, if not impossible. 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. (Certain safety requirements cannot be met if a blank is dropped outside the safe areas, but the others must still be maintained.)

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 fault-tolerant program should have the ability to confine damage and failures. For the production cycle of the cell, 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.

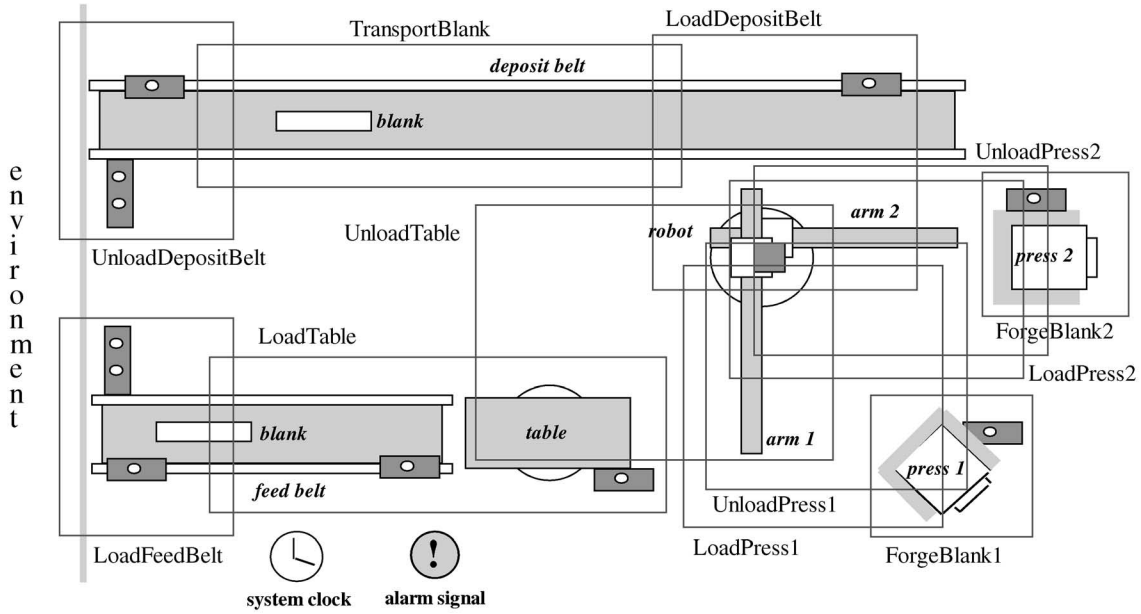


Fig. 3. CA actions that control Production Cell II.

4 Design of a Control Program Using CA Actions

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 runtime. 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 [24] as to how these design decisions were made and why we used certain actions to enclose the interaction between certain devices in our control program 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. (An action can contain further nested actions—see Fig. 4 for an example.)

There are six concurrent execution threads in the control program, corresponding to the six devices: `FeedBelt`, `Table`, `Robot`, `Press1`, `Press2`, and `DepositBelt`, each of which essentially performs a simple endless loop. (Details of the controllers are given in Section 4.4.) 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 that indicates when a processed blank can be deposited. 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. Fig. 3 portrays the 12 related CA actions as overlays on the FZI simulator diagram [10].

The control program interacts with the environment through two CA actions: `LoadFeedBelt` and `UnloadDepositBelt`. Within these CA actions, the environment plays the roles of supplier and consumer, respectively. It is the environment that adds a blank to or takes a blank from the production cell when the need arises. Note that an intersection between CA actions in Fig. 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. Furthermore, even if the actions that devices participate in are invoked in the wrong order because of a control program design fault, then the result will be at worst a safe deadlock.

As mentioned previously, each hardware device is associated with a device/sensor-controller (i.e., an execution thread) which is responsible for dynamically specifying the sequence of actions in which the device will participate. For example, without compromising safety and functionality requirements, the robot thread can skip all the CA actions related to one of the presses if this press has failed and, so, tolerate this fault.

4.1 Design of CA Actions

Our design assumes that an action will begin only if its preconditions are valid and that, if no exception is raised during the execution of an action, then its postconditions will hold (though this could, if we so wished, be checked using an acceptance test). In this section, we first address the normal pre and postconditions for actions that control the entire cell. For a given action, these conditions are used to ensure that the execution of that action will not violate, in

TABLE 1
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

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 (see Table 1).

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 [21]). For example, to check whether the robot is off, we can check that all three related actuators are in the stop state. To make sure that arm 1 and arm 2 are retracted (a safe state), we can check values from the sensors that report arm positions.

The robot has six defined rotary positions or angles, so the robot-related CA actions could specify a defined angle as one of their preconditions. However, this would affect the flexibility of the robot and limit the possible execution sequences of CA actions. The weaker precondition "robot at one of the defined angles" permits more possible execution sequences, thereby improving system performance.

We will now show how CA actions can deal with various types of failures in a well-controlled manner by specifying the exceptional postconditions for a given action. Consider the action `LoadPress1` again. Fig. 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 cooperation that arranges for arm 1 of the robot to drop a blank into press 1.

Action `LoadPress1` is described below using the COALA notation, which was developed for the formal specification of CA actions [18]. Our Java implementation of the control program is based on a set of predefined 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,
  robotAngleSensor;
  Press1Sensor: blankSensor,
  lowPositionSensor, midPositionSensor;
 Exceptions
  Press1Failure, Arm1Failure1, ...;
                        ;;exceptions to signal
Body
 Use CAA                ;;specify nested actions
  RotateRobot, MovePress1toMiddle,
  ExtendArm1, RetractArm1;
 Object
  robotPress1Channel: Channel;
                        ;;shared local objects
```

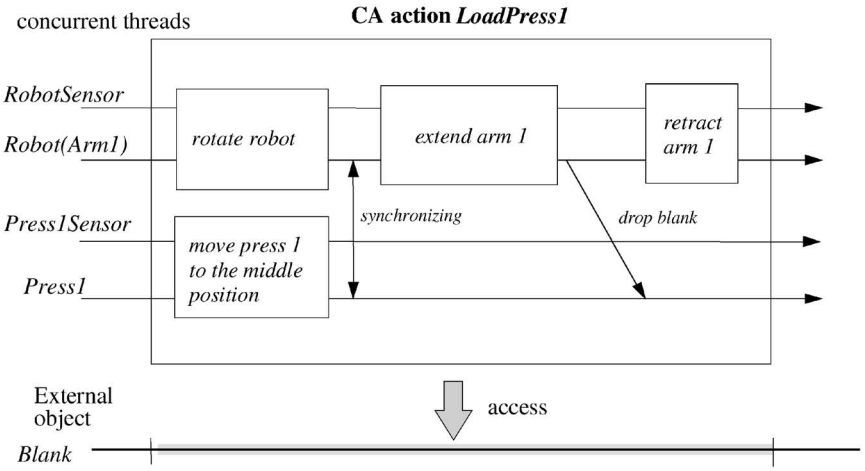


Fig. 4. CA action `LoadPress1`.

TABLE 2
Exceptional Outcome when Press 1 Fails

Exception to signal	*Exceptional post-conditions*
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

```
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 signaled to the enclosing action. The roles of an action can signal an exception directly, but must guarantee that the exception that is signaled has been agreed upon by all the roles of that action. In the case of abortion or failure, the CA action support mechanism (which can be assumed by the application programmer to be fault-free) will enforce the abortion and signal the appropriate exception, either `abort` or `failure`, to the enclosing action. Exceptions declared within the **Body** of a CA action can be raised by roles. 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* postconditions of the action can no longer be satisfied. Thus, exceptional postconditions 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 postconditions [21]. By way of example, the Table 2 illustrates the exceptional outcome when press 1 fails. It is important to notice that different exceptional outcomes may lead to different states of the production cell. For example, the exceptional outcome caused by just the 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.

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 [21].

TABLE 3
Postconditions for a Pair of Concurrent Failures

Exception to signal	*exceptional post-conditions*
(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

4.3 Dealing with Concurrent Failures

Now, let us address the problem of possible 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 postconditions. Table 3 shows postconditions for an example pair of concurrent failures.

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. Very often, despite a failure having been detected, it is not possible to determine from the available sensor readings which of several possible failures has actually occurred. In such circumstances, the control program is designed simply to bring the system to a stop in a safe state so that offline 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 [23]. 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), `rs_m_failure` (rotary sensor or motor failure), `as_m_failure1` (arm 1 sensor or motor failure while the blank on arm 1), `as_m_failure2` (arm 1 sensor or motor failure while the blank is in press 1), `cs_failure` (control software failure(s)), and `rt_except` (runtime exceptions such as overflow).

An exception graph for this action is shown in Fig. 5, again assuming that no more than two exceptions are raised concurrently. 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 production cell in a predefined safe state, if possible.)

4.4 Design of Device-Controllers

Given a set of CA actions to control the interaction of devices in the production cell, 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
```

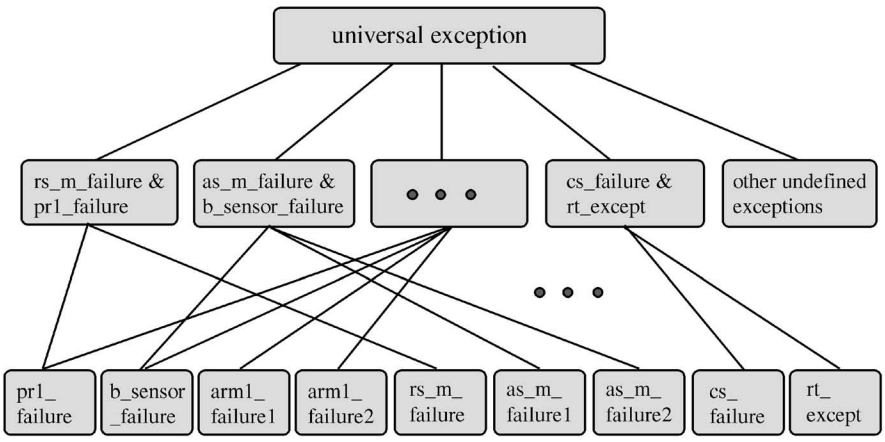


Fig. 5. Exception graph for CA action `LoadPress1`.

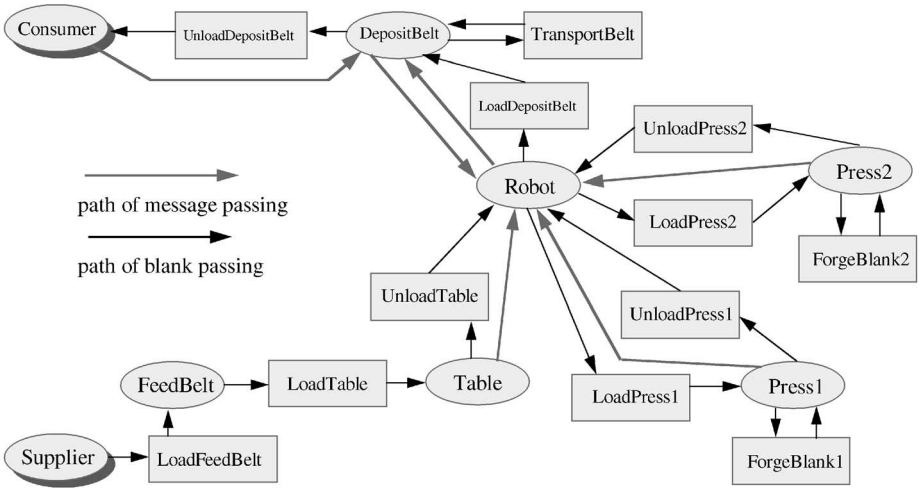


Fig. 6. Interaction between controllers and CA actions.

```
LoadPress1.Press(plate)
– activate action LoadPress1
ForgeBlank1.Press(plate)
– activate action ForgeBlank1
robotQueue.put(FORGED_PLATE_IN_PRESS1)
– put message in robotQueue
UnloadPress1.Press(plate)
– activate action UnloadPress1
}
```

Fig. 6 shows the interactions between the controllers and CA actions, where boxes represent CA actions and ovals represent controllers. A gray line indicates message passing between controllers, while a black line connects an action to a controller or vice versa and implies that the controller plays a role in that action.

5 Formalization of CA Action-Based Designs

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 [2]. 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 behavior 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: 1) any application specific requirements that need to be considered and 2) the requirement that, initially, all roles should be idle and no exception should have been signaled.

The `next-state` relation defines the computation paths that are possible in the system. This essentially corresponds to four kinds of activities that may occur in the system:

1. A controlling process may call and thus invoke a role from a CA action, thereby activating it,
2. 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 postconditions for both normal and exceptional outcomes,
3. After a CA action has been executed, a return is issued from its roles to the corresponding controlling processes that called them, and
4. 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 modeled the `next-state` relation to encode the interleaving semantics. The state transition system obtained from a CA action-based design of a finite system can be used to analyze its properties by model-checking. We have found SMV [13] particularly useful for this purpose: The state-transition system can be expressed in SMV and the properties of the system to be analyzed can be expressed in CTL [5]. The technical details of representing a CA action-based design in SMV and the properties of a system in CTL were described in [2]; in the following, we will illustrate this formalization and the formal analysis of its properties using representative parts of our Fault-Tolerant Production Cell design.

The CA actions that control the production cell are described according to their interfaces, which consist of the set of roles they provide, the external objects they access (as expressed by the parameters of the roles), and their pre and postconditions. They are formalized, using SMV derived from our COALA design, according to the general scheme outlined above, which is exemplified here for the `LoadPress1` action. (We use `An` and `Sn` to represent `Actuator n` and `Sensor n`.)

```
MODULE LoadPress1(...)
 DEFINE
  pre := A6 = stop & A7 = stop & A10 = stop &
          – robot off
         A8 = on &
          – blank on arm 1
         S15 = pos_ret & S16 = pos_ret &
          – arms retracted
         (S17 = R2 | S17 = R5) &
          – robot rotation
         A4 = stop &
          – press1 off
         (!S21 -> !S7) &
          – no blank in press1
         S8 & !S9 & !S10;
          – press1 at lower position
  enabled := roleRobot = activated & rolePress1
  = activated & pre;
  mv_cond := enabled & next(signal) in { normal,
  ... };
  drop_cond := enabled & next(signal) in
 { lost_blank };
 VAR
  roleRobot : { nonactive, activated,
  returning };
  rolePress1: ...
  signal : { normal, press1_failure,
  lost_blank, ... };
  mvblank : MoveBlank(mv_cond, drop_cond,
  blank_on_arm1, blank_in_press1);
 ASSIGN
  init(roleRobot) := nonactive;
  next(roleRobot) :=  case
                        enabled : returning;
                        1 : state;
                      esac;
  ...
  init(signal) := normal;
  next(signal) :=case
                  enabled: { normal,
                  press1_failure, ... };
                  OTHERWISE: signal;
                  esac;
  ...
  next(A8) :=  case
               enabled & next(signal) =
               normal: off;
               enabled & next(signal) =
               press1_failure: on;
               ... { values for other
               exceptional postconditions }
               OTHERWISE: A8;
               esac;
  ...
```

The SMV module of the CA action is parameterized with the external objects that it can access. The pre part defines the precondition for the CA action; the CA action is enabled when all of its roles are activated and the precondition is true. The roles are represented by their states (being nonactive, activated, or returning). Initially, roles are nonactive. When the CA action is enabled, it may be executed and, in this case, the role changes its state to `returning`. The main participants in this particular CA action are the controllers of press 1 and the robot. Thus, `LoadPress1` provides roles for both of them.

The `signal` variable represents the exception that is signaled by the CA action `LoadPress1`. It is initialized to `normal`, encoding to indicate that no exception has been signaled. When the CA action is executed, then any of the normal or exceptional outcomes is possible; this is expressed with the nondeterministic choice in the `next-state` assignment for the `signal` variable.

The `next-state` assignments for the external objects are obtained from the postcondition of the CA action `LoadPress1`:

```
post_normal == ... & A8 = off & ...
– no blank on arm 1
post_press1_failure == ... & A8 = on & ...
– blank on arm 1
...
post == (signal = normal & post_normal)
or (signal = press1_failure &
post_press1_failure)
or ...
– other except. outcomes
```

The SMV formalization of `LoadPress1` illustrates this for actuator `A8` (i.e., arm 1's magnet). When the CA action is executed, then, in normal conditions and also in certain exceptional conditions, a blank is moved from arm 1 to press 1. This is expressed with the condition `mv_cond` and the corresponding instance of the SMV-module `MoveBlank` which encodes in SMV the actions for moving the blank; for details, see [2].

The pre and postconditions of the CA actions are designed to be compatible with the intended ordering on the execution of the CA actions: After a CA action is executed, the precondition of the next appropriate CA action should be satisfied. Activating the next appropriate CA action is the main task of the device-controllers. The activation of a CA action is performed by calling its roles. Calling a role consists of two parts: invoking the role and finishing the call on termination of the CA action. The formalization of controllers in SMV is exemplified below using the `Press1Controller`:

```
MODULE Press1Controller(loadpress1,
forgeblank1, unloadpress1)
 DEFINE
  continue :=  loadpress1.signal =
               b_sensor_failure
               – blank sensor failure
             | loadpress1.signal = normal;
               – no failure
  at_load :=  !blank_in_press1.present
           & forgeblank1.Press1.nonactive &
             unloadpress1.Press1.nonactive
           & continue;
```

```
  at_forge :=  blank_in_press1.present &
               blank_in_press1.state = plain
           &   loadpress1.Press1.nonactive &
               unloadpress1.Press1.nonactive
           &   continue;
  at_unload:=  blank_in_press1.present &
               blank_in_press1.state = forged
           &   loadpress1.Press1.nonactive &
               forgeblank1.Press1.nonactive
           &   continue;
VAR
 A4 :  { up, down, stop }; – A4: move press1
 S7 :  boolean;         – S7: blank in press1
 S8 :  boolean;         – S8: in lower position
 S9 :  boolean;         – S9: in middle position
 S10 : boolean;         – S10: in upper position
 blank_in_press1 :      Blank(void);
ASSIGN
 init(S7) := 0;
 init(S8) := 1;
 init(S9) := 0;
 init(S10):= 0;
 init(A4) := stop;
 next(loadpress1.rolePress1) :=
       case
        loadpress1.rolePress1 = nonactive
        & at_load :   activated;
        loadpress1.rolePress1 = returning :
                    nonactive;
        OTHERWISE :  loadpress1.rolePress1;
       esac;
 next(forgepress1.rolePress1) := ...
 next(unloadpress1.rolePress1):= ...
```

This controller process has access to CA actions `LoadPress1`, `ForgeBlank1`, and `UnloadPress1`. It iterates calling its roles in these CA actions. For example, it activates its role in CA action `LoadPress1` if `at_load` is true and the role is not yet activated. When the role is in the returning state, the call is finished by setting its state to nonactive. The roles of press 1 in the other CA actions are treated similarly. The actuators and sensors are modeled as objects of (the controllers of) the devices and must be initialized appropriately.

Blanks are the main objects of interest in the Production Cell case study as far as the application's functionality is concerned; they are passed on from one device to the next. The blank currently being held by a device is represented in the corresponding controlling process by a variable that models blanks as records of two entries: a name for identifying the blank and a component expressing whether a blank has already been forged:

```
MODULE Blank(name)
 DEFINE
  present := !(id = void);
 VAR
  id :       { void, anon, id1 };
  state :    { plain, forged };
 ASSIGN
  init(id)    := name;
  init(state) := plain;
```

Initially, a blank is not forged (i.e., is plain). The name component `id` is assigned value `void` if there is currently no blank at the position represented by the variable, `id1` if a blank with name `id1` is present at the position represented by the variable, and `anon` if a blank with a name different from `id1` is present at the position represented by the variable. The instances defined for the controllers are all initialized with `void`, indicating that, initially, no device holds a blank.

An unlimited supply of identifiers would be necessary for observing each blank individually. The distinction between `id1` and `anon` is made so as to observe a specific blank named `id1` on its way through the cell. This is done in order to abstract from a nonfinite system to a finite state representation; it is sufficient to formalize the requirements for the system in this way since each requirement needs at most one blank to be identified. Our formalization is such that, at any time, there is at most one blank named `id1` in the system.

The other CA actions and controllers of the production cell were encoded in SMV analogously; the combination of all the actions and the controllers provides the state transition system that was used for checking the properties of the Production Cell.

6 Formalizing Properties of the Fault-Tolerant Production Cell

We have formalized and 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 behavior 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 behavior 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 only if 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 nontolerable 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 (modeled) failure should be tolerable. In this case, the conditional safety and liveness properties reduce to the nonconditional 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 characterizes the fault-free cases.

We will now 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 (here, press 1) fails. Failures of press 1 are signaled by the CA action LoadPress1 with the exceptions press1_failure or b_sensor_failure. The property encoding the continuous service requirement should express that if, during execution of the cell, a press1_failure or a b_sensor_failure occurs, then any blank in other devices of the Production Cell or blanks inserted afterward 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)))
```

7 Design for Validation

The analysis of properties of the Fault-Tolerant Production Cell 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. This shows the usefulness of model-checking for developing and improving the CA action-based design of the Fault-Tolerant Production Cell. 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 an 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 maneuver itself into a situation from which no further activities were possible. Such "critical" sequences of actions can be derived from counterexample paths generated by the model-checker. The counterexample 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 (see also the related discussion in Section 4.1).

Another problem that we discovered was connected to the traffic light and the way in which the controller of the deposit belt interacts with the environment (i.e., the consumer). The interaction was originally organized by means of messages sent between the consumer and the feed belt controller and by checking the traffic light. The solution suggested from the formal analysis led to an improved and simpler design in which the feed belt controller interacts with the consumer only by means of the traffic light.

Two SMV specifications for our CA action-based design were produced: a full model and an abstraction of the full model. The full model contains 1,251 lines of SMV code; the abstracted model has 1,079 lines of the code. The potential state space of the full model is about 10^{34}; the size of the abstracted one is reduced to about 10^{26}. Two models were instantiated with the number of blanks to be put into the production cell and the actual model checking was performed on a SUN Sparc Ultra-II platform. The full model produced no result for three or more blanks after one week of computation. However, the abstracted model responded in all possible situations, including the possible maximum number of blanks, within about 15 hours. Further efficiency and timing results can be found in [2].

8 An Implementation

We have implemented the design of the control program that was discussed in the previous sections using a Java implementation of a distributed CA action support scheme [21]. (This scheme makes use of the nested multithreaded transaction facilities provided by the Arjuna transaction support system [14].)

Fig. 7 shows a screen dump of the FZI Fault-Tolerant Production Cell simulator controlled by our implementation. Outlines of 12 top-level CA actions are displayed on the simulator diagram. During system execution, these outlines are colored in gradually to show the progress of a CA action execution dynamically. In the figure, there are three CA actions that are active and being executed: LoadTable, LoadPress1, and ForgeBlank2. If an exception, or two concurrent exceptions, are raised in an action, the color within the outline will change to indicate the dynamic process of exception handling.

Fig. 8 shows a slightly modified version of the device and sensor failure injection panel provided by FZI. By using this panel, failures can be easily injected into the Production Cell simulator. For example, a rotary motor failure or a rotary sensor failure of the robot can be injected by pressing the corresponding buttons in the panel. We have extended the original FZI panel to permit the injection of concurrent failures: A pair of failures can be injected into the simulator if the failure mode selection is set to "double." In this mode, two different failure buttons may be pressed sequentially, but only the second press will stimulate the actual injection of the two concurrent failures. After one or more failures are

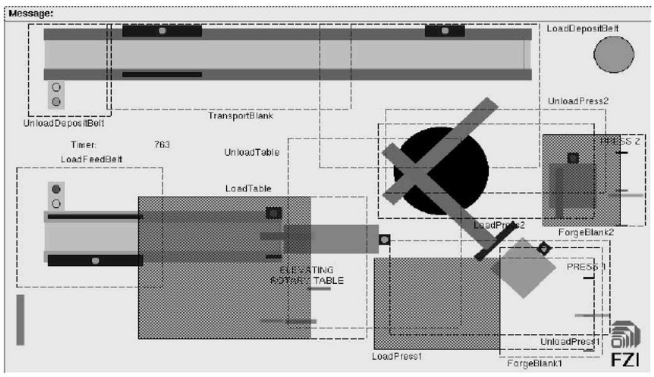


Fig. 7. The Fault-Tolerant Production Cell simulator.

injected into the simulator, the error detection measures embedded in our control program should be able to detect them promptly. One or more corresponding exceptions will thus be raised. The simulator will then portray how such exceptions are handled within the CA action framework, in particular how the system is, if necessary, brought to a stop in a safe state.

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.

9 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 involving multiple moving mechanical components. 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 specification of Production Cell II.

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 very large, extension of the approach we had used in Production Cell I [24]. 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 means of 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. Matos and White [12] describe 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; safety-related requirements are addressed by both proper synchronization inside CA actions and necessary mutual exclusion of the action execution whenever the concurrent execution of two CA actions is unsafe. Liggesmeyer and Rothfelder [9] developed a Formal Risk Analysis approach for analyzing the runtime behavior of Production Cell II and studied 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 (for the complete treatment, see [21], [22]).

Production Cells I and II do not involve any consideration of timing deadlines. We have extended our ideas on exception handling and resolution to deal with such complications as possibly concurrent timing and value faults [16]—these being situations that can arise in Production Cell III, the "Real-Time Production Cell" model [11], for which several control programs have been developed.

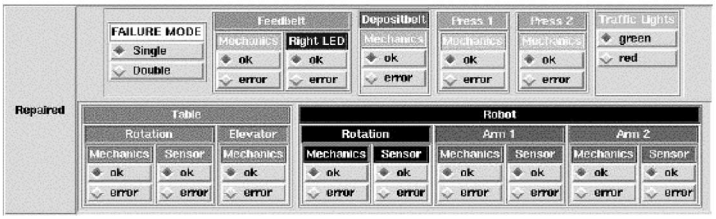


Fig. 8. Revised failure injection panel.

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 [4]. Ideally, of course, such an approach would be allied to a formal means of validating the system design as it is developed—something that could, we believe, take advantage of the formal treatment of CA actions, as shown in this paper and [2], [17], and the formal language for specifying CA actions, COALA [18].

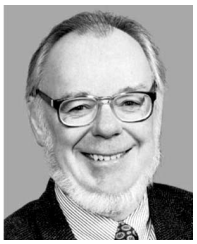
ACKNOWLEDGMENTS

This work was supported by the ESPRIT Long Term Research Project 20072 on Design for Validation (DeVa), the IST Programme RTD Project 1999-11585 on Dependable Systems of Systems (DSoS), the EPSRC Flexx Project on Dependable and Flexible Software, and the EPSRC IBHIS Project on Diverse Information Integration for Large-Scale Distributed Applications. A.F. Zorzo was also supported by CNPq/Brazil under grant number 520503/00.7.

REFERENCES

[1] R.H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Trans. Software Eng.,* vol. 12, no. 8, pp. 811-826, Aug. 1986.
[2] E. Canver, D. Schwier, A. Romanovsky, and J. Xu, "Formal Verification of CAA-Based Designs: The Fault-Tolerant Production Cell," third year report, ESPRIT Long Term Research Project 20072 on Design for Validation, pp. 229-258, Nov. 1998.
[3] S.J. Caughey, M.C. Little, and S.K. Shrivastava, "Checked Transactions in an Asynchronous Message Passing Environment," *Proc. First IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing,* pp. 222-229, Apr. 1998.
[4] R. DeLemos and A. Romanovsky, "Co-Ordinated Atomic Actions in Modelling Object Co-Operation," *Proc. First IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing,* pp. 152-161, Apr. 1998.
[5] E.A. Emerson, "Temporal and Modal Logic," *Handbook of Theoretical Computer Science,* J. van Leeuwen, ed., vol. B, chapter 16, pp. 995-1072, Elsevier Science Publishers, 1990.
[6] J.N. Gray, "A Census of Tandem System Availability between 1985 and 1990," *IEEE Trans. Reliability,* vol. 39, no. 4, pp. 409-418, 1990.
[7] K.H. Kim, "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Trans. Software Eng.,* vol. 8, no. 3, pp. 189-197, 1982.
[8] C. Lewerentz and T. Lindner, *Formal Development of Reactive Systems: Case Study "Production Cell"*. Springer, Jan. 1995.
[9] P. Liggesmeyer and M. Rothfelder, "Improving System Reliability with Automatic Fault Tree Generation," *Proc. 28th Int'l Symp. Fault-Tolerant Computing,* pp. 90-99, June 1998.
[10] 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.
[11] A. Lötzbeyer and R. Mühlfeld, "Task Description of a Flexible Production Cell with Real Time Properties," FZI Technical Report, (ftp://ftp.fzi.de/pub/PROST/projects/korsys/task_descr_flex_2_2.ps), 1996.
[12] G. Matos and E. White, "Application of Dynamic Reconfiguration in the Design of Fault-Tolerant Production Cell," *Proc. Fourth Int'l Conf. Configurable Distributed Systems,* pp. 2-9, 1998.
[13] K.L. McMillan, "Symbolic Model Checking," revised version of PhD thesis, Carnegie Mellon Univ., Kluwer Academic Publishers, 1993.
[14] G.D. Parrington, S.K. Shrivastava, S.M. Wheater, and M.C. Little, "The Design and Implementation of Arjuna," *USENIX Computing Systems J.,* vol. 8, no. 3, 1995.
[15] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Eng.,* vol. 1, no. 2, pp. 220-232, 1975.
[16] A. Romanovsky, J. Xu, and B. Randell, "Coordinated Exception Handling in Real-Time Distributed Object Systems," *Int'l J. Computer Systems Science & Eng.,* vol. 14, no. 4, pp. 197-207, July 1999.
[17] D. Schwier, F. von Henke, J. Xu, R.J. Stroud, A. Romanovsky, and B. Randell, "Formalization of the CA Action Concept Based on Temporal Logic," second year report, ESPRIT Long Term Research Project 20072 on Design for Validation, pp. 3-15, Dec. 1997.
[18] 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 Co-Ordinated Atomic Actions," third year report, ESPRIT Long Term Research Project 20072 on Design for Validation, pp. 43-86, Nov. 1998.
[19] 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," *Proc. 25th Int'l Symp. Fault-Tolerant Computing,* pp. 499-508, June 1995.
[20] J. Xu, A. Romanovsky, and B. Randell, "Co-Ordinated Exception Handling in Distributed Object Systems: From Model to System Implementation," *Proc. 18th IEEE Int'l Conf. Distributed Computing Systems,* pp. 12-21, May 1998.
[21] 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," third year report, ESPRIT Long Term Research Project 20072 on Design for Validation, pp. 167-188, Nov. 1998.
[22] J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A.F. Zorzo, E. Canver, and F. von Henke, "Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions," *Proc. 29th Int'l Symp. Fault-Tolerant Computing,* pp. 68-75, June 1999.
[23] J. Xu, A. Romanovsky, and B. Randell, "Concurrent Exception Handling and Resolution in Distributed Object Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 11, no. 10, pp. 1019-1032, Oct. 2000.
[24] 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," *Software—Practice & Experience,* vol. 29, no. 2, pp. 667-697, 1999.

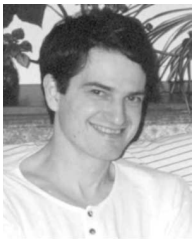
Jie Xu received the PhD degree from the University of Newcastle upon Tyne on advanced fault-tolerant software. He is a lecturer in computer science at the University of Durham, United Kingdom. From 1990 to 1998, Dr. Xu was with the Computing Laboratory at Newcastle, where he was promoted to senior researcher in 1995. He moved to a lectureship at Durham in 1998 and cofounded the Distributed Systems Engineering Group and the DPART Laboratory supporting highly dependable distributed computing. Dr. Xu has published more than 100 book chapters and research papers in areas of system-level fault diagnosis, exception handling, software fault tolerance, and large-scale distributed applications. His major work has been published in leading academic journals, such as the *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, and *IEEE Transactions on Reliability*. He has been involved in several research projects on dependable distributed computing systems, including two EC-sponsored ESPRIT BRA projects and one ESPRIT Long Term Research project. He is principal investigator of the FTNMS project on fault-tolerant mechanisms for multiprocessors and coleader of the EPSRC Flexx project on highly dependable and flexible software and the EPSRC IBHIS project on diverse information integration for large-scale distributed applications. Dr. Xu is editor of *IEEE Distributed Systems*, PC cochair of the Seventh IEEE WORDS on Object-Oriented Real-Time Dependable Systems, workshop chair of the IEEE SRDS Workshop on Reliable Distributed Object Systems, and tutorial speaker of the IEEE/IFIP International Conference on Dependable Systems and Networks. He has given invited lectures at international colloquiums and served as session chair and PC member of various international conferences and workshops, including the IEEE SRDS, IEEE ISOR, EDCC, and ACM SAC-AIMS.



Brian Randell graduated in mathematics from Imperial College, London, in 1957 and joined the English Electric Company, where he led a team that implemented a number of compilers, including the Whetstone KDF9 Algol compiler. From 1964 to 1969, he was with IBM in the United States, mainly at the IBM T.J. Watson Research Center, working on operating systems, the design of ultra-high speed computers, and computing system design methodology. He then became a professor of computing science at the University of Newcastle upon Tyne, where, in 1971, he set up the project that initiated research into the possibility of software fault tolerance, and introduced the "recovery block" concept. Subsequent major developments included the Newcastle Connection and the prototype Distributed Secure System. He has been principal investigator on a succession of research projects in reliability and security funded by the Science Research Council (now Engineering and Physical Sciences Research Council), the Ministry of Defence, and the European Strategic Programme of Research in Information Technology (ESPRIT), and now the European Information Society Technologies (IST) Programme. Most recently, he has had the role of project director of CaberNet (the IST Network of Excellence on Distributed Computing Systems Architectures), and of two IST Research Projects, MAFTIA (Malicious- and Accidental-Fault Tolerance for Internet Applications) and DSoS (Dependable Systems of Systems). He has published nearly 200 technical papers and reports and is coauthor or editor of seven books.

Alexander Romanovsky received the MSc degree in applied mathematics from Moscow State University in 1976 and the PhD degree in computer science from St. Petersburg State Technical University in 1988. He was with this university from 1984 to 1996, doing research and teaching. In 1991, he worked as a visiting researcher at ABB Ltd. Computer Architecture Lab Research Center, Switzerland. In 1993, he was a visiting researcher at the Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. From 1993 to 1994, he was a postdoctoral fellow at the Department of Computing Science, the University of Newcastle upon Tyne, United Kingdom. In 1992-1998, he was involved in the *Predictably Dependable Computing Systems* (PDCS) ESPRIT Basic Research Action and the *Design for Validation* (DeVa) ESPRIT Basic Project. In 1998-2000, he worked on the *Diversity in Safety Critical Software* (DISCS) EPSRC/UK Project. Now, he is a senior research associate with the Department of Computing Science, University of Newcastle upon Tyne, working on the *Dependable Systems of Systems* (DSoS) EC IST RTD Project. His research interests include software fault tolerance, software diversity, concurrent programming, concurrent object-oriented and object-based languages, real-time systems, exception handling, operating systems, software engineering, and distributed systems. He has coauthored more than 120 scientific papers, book chapters, reports, and a patent in these and related areas.

Robert J. Stroud received a first degree in mathematics from Cambridge University, followed by the MSc degree and then, in 1988, the PhD degree in computing science from the University of Newcastle upon Tyne, for a thesis in the area of distributed systems. He is a reader in computer science at the University of Newcastle upon Tyne and a member of the Dependability Research Group. He leads Newcastle's work on the IST project MAFTIA and is also involved in two other dependability projects, DSoS and DIRC. His research interests include security of mobile code, reflection, coordinated exception handling in concurrent/distributed systems, object-oriented systems, fault tolerance, and Internet technologies. He has published extensively on distributed systems and fault tolerance.



Avelino F. Zorzo received the BSc and MSc degrees at UFRGS/Brazil and the PhD degree from the University of Newcastle upon Tyne, United Kingdom. Currently, he is a senior lecturer at the Pontifical Catholic University of Rio Grande do Sul (PUCRS/Brazil). Since 2000, he has been a researcher financed by the Brazilian agencies CNPq (350277/2000-1) and FAPERGS (99/2049.3). His interests include fault-tolerant, distributed, and parallel systems. Since 1990, he has (co)authored more than 50 technical papers on these subjects.

Ercument Canver received the MSc degree in computer science from San Diego State University in 1991 and the Diplom in mathematics from the University of Ulm, Germany, in 1992. From 1991 to 2000, he worked as a researcher at the University of Ulm on various national and international (ESPRIT) research projects on formal methods for dependable systems, such as VSE (Verification Support Environment), GUARDS (Generic Upgradeable Architecture for Real-Time Dependable Systems), DeVa (Design for Validation), and SafeRail (Software Specification Techniques for Safety Critical Applications in Railway Engineering). Since 2000, he has been working at Siemens AG, Germany, in the mobile telecommunication sector.

Friedrich von Henke received the diploma and Dr.rer.nat. degrees from the University of Bonn, Germany, in 1971 and 1973, respectively. He is currently a professor of informatics (computer science) at the University of Ulm, Germany, where he is the head of the Artificial Intelligence Department. His main research interests are in the development and application of formal methods, theorem proving, and other areas of artificial intelligence. Before joining the university, he held research and management positions in the Computer Science Laboratory of SRI International, the Artificial Intelligence and Computer Systems Laboratories of Stanford University, and the German National Research Center for Information Technology (GMD). He is a member of the ACM, IEEE Computer Society, and the German Informatics Society (GI).