A Case Study in Statistical Testing of Reusable Concurrent Objects

Hélène Waeselynck and Pascale Thévenod-Fosse

LAAS - CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse Cedex 4 - FRANCE
{waeselyn, thevenod}@laas.fr

Abstract. A test strategy is presented which makes use of the information got from OO analysis and design documents to determine the testing levels (unit, integration) and the associated test objectives. It defines solutions for some of the OO testing issues: here, emphasis is put on applications which consist of concurrent objects linked by client-server relationships. Two major concerns have guided the choice of the proposed techniques: component reusability, and nondeterminism induced by asynchronous communication between objects. The strategy is illustrated on a control program for an existing production cell taken from a metal-processing plant in Karlsruhe. The program was developed using the Fusion method and implemented in Ada 95. We used a probabilistic method for generating test inputs, called statistical testing. Test experiments were conducted from the unit to the system levels, and a few errors were detected.

1 Introduction

A large number of testing techniques have already been defined for programs developed according to hierarchical approaches and written in procedural languages (see e.g., [3]). But object-oriented (OO) development process corresponds to a different approach to software construction. The design of a program is organized around the data it manipulates and their relationships, which leads to highly decentralized architecture. OO languages provide powerful mechanisms like entity instantiation, genericity, inheritance, that have no equivalent in procedural languages. Hence, testing approaches must be revisited to take into account the characteristics of OO technology (see e.g., [5, 12]). Yet, few complete experiments that follow design and testing in a systematic way have been reported. The paper focuses on such an experiment.

We present a test strategy based on the information got from analysis and design documents produced during an OO development process. This strategy defines proper solutions for some of the OO testing issues: here, emphasis is put on applications which consist of *concurrent objects* linked by *client-server relationships*. The case study used as an experimental support to our investigation is adapted from an existing industrial production cell [13]. Its aim is to forge metal blanks got from a feed belt. An OO version of the control program [1] provides us with an example of a complete OO development: Fusion analysis and design [6], Ada 95 implementation.

Some OO specific difficulties are raised by the case study: How to determine the unit and integration testing levels (decentralized architecture of objects)? At each testing level, how to define the test objectives from the analysis and design documents? How to determine conformance of the test results to the expected ones (oracle checks)? How to solve the controllability and observability problems that are drastically increased by object encapsulation? The choice of the proposed solutions is guided by the consideration of two major concerns:

- *Component reusability*. OO design is intended to favor reusability: components are defined by interface models, and their internal behavior is hidden (encapsulation). In order to define a cost-effective strategy, testing of reusable components has to be performed without making any assumption on the operational context.
- *Nondeterminism* involved by concurrency between objects communicating by asynchronous events. It means that there are many possible interleavings of event reception and event treatment, depending on the reaction time of the environment. This issue has already been identified in work on protocol testing [14]; unlike in protocol testing where the hypothesis of a slow environment may sometimes be acceptable, no timing assumption may be done for concurrent objects.

Main related work is briefly reviewed in Section 2. Section 3 introduces the probabilistic method for generating test inputs we use to automatically generate test input sequences in accordance with given test objectives [15]. Section 4 describes the case study. An overview of the global strategy is presented in Section 5. Then, emphasis is put on unit testing (Section 6). Section 7 summarizes the results of the experiments conducted from the unit to the system levels; and finally, we highlight the main lessons learnt from this theoretical and experimental investigation.

2 Related Work

There is now a general agreement among the testing community that OO concepts raise a number of problems from the perspective of testing (see e.g., [5, 12]). The case study presented in this paper does not exhibit all of these problems. For example, its design is more object-based than object-oriented: there are few inheritance relations, and only between very basic classes. Hence this paper does not address the problem of how to test new and inherited features (see e.g., [8]). The focus is on decentralized architecture of communicating concurrent objects, which raises the following issues:

- *Determination of testing levels*. The traditional unit and integration levels of testing do not fit well in the case of OO programs. Their decentralized architecture makes it difficult to determine where to start testing, and there is no obvious order for an integration strategy. Related work defines the testing levels by considering the number of stubs to be produced [10], or the most significant paths and interactions in the application [9].
- *Selection and control of test sequences.* State-based testing techniques are usual for small OO subsystems [11, 16]. But concurrency introduces nondeterminism, which

has to be taken into account. The problem is well-known in the field of protocol testing. For example, it is shown in [14] that a simple system of two communicating deterministic automata yields nondeterministic behavior that cannot be modeled by a finite number of states. Transferring classical state-based criteria – such as the Wp method chosen by these authors – to communicating systems requires assumptions to be made (bounded queues and communication channels, system run in a "slow" environment).

- *Observability and oracle checks.* Observability is impeded by the principle of encapsulation and information hiding: as shown in [17], this is detrimental to state error propagation, and hence to the revealing of faults during testing. The systematic introduction of built-in test capabilities in the design of classes (set/reset capabilities, executable assertions, operations reporting the internal state) is studied in [4]. In our case, the issue is further compounded by concurrency which raises the problems of (i) choosing an appropriate conformance relation as the basis of the oracle checks and, (ii) synchronizing the control and observation with the activities of the objects.

Reusability is also one of our concerns: we are interested in assessing the behavior of software components that may be reused in the framework of different applications. This problem shares similarities with other work concerned with the interface robustness of commercial off-the-shelf software components [7, 18]; however, unlike them, we do not consider fault injection and fault containment techniques. Our aim is not to assess the ability of the component to handle faults in its environment, but to increase confidence in its ability to react according to its interface model for a large population of possible users. Hence we have to consider a larger range of solicitation profiles (e.g., frequency and sequencing of calls to the operations of the objects) than the ones that are possible in the original application.

Central to the case study is the problem of the interactions of objects with their environment. The interaction schemes may be very complex depending on the many possible interleavings of concurrent events. Since it is not tractable to test for all possible interleavings in a given operational context, and since reusable objects should be tested for several operational contexts, sampling approaches have to be defined. The approach that we propose is based on statistical testing designed according to functional criteria.

3 Statistical Testing Designed According to a Test Criterion

Test criteria are related to the coverage of either a white box or a black box model of the software (see e.g., [3]). For example, the program control flow graph is a well-known white box model; automata are black box models that may be used to describe some software functions. Given a criterion (e.g., branch or transition coverage), the conventional method for generating test inputs proceeds according to the *deterministic principle*: it consists in selecting a priori a set of test inputs such that each element defined by the criterion is exercised (at least) once. But a major limitation is due to the imperfect connection of the criteria with the real faults: exercising once, or very few

times, each element defined by such criteria cannot ensure a high fault exposure power. Yet, the criteria provide us with relevant information about the target piece of software.

A practical way to compensate for criteria weakness is to require that each element be exercised several times. This involves larger sets of test inputs that have to be automatically generated: it is the motivation of *statistical testing* designed according to a test criterion [15]. In this approach, the information provided by criteria is combined with an automatic way of producing input patterns, that is, a random generation.

The statistical test sets are defined by two parameters, which have to be determined according to the test criterion: (i) the input distribution, from which the inputs are randomly drawn and, (ii) the test size, or equivalently the number of inputs that are generated. As in the case of deterministic testing, test criteria may be related to a model of either the program structure, which defines *statistical structural testing*, or of its functionality, which defines *statistical functional testing*.

The *determination of the input distribution* is the corner stone of the method. The aim is to search for a probability distribution that is proper to rapidly exercise each element defined by the criterion. Given a criterion C – say, transition coverage – let S be the corresponding set of elements – the set of transitions. Let P be the occurrence probability per execution of the least likely element of S. Then, the distribution must accommodate the highest possible P value. Depending on the complexity of this optimization problem, the determination of the distribution may proceed either in an analytical or in an empirical way (see [15] for detailed examples). The latter way is used in the case study presented in this paper (Section 6.3).

The *test size* N must be large enough to ensure that each element is exercised several times under the input distribution defined. The assessment of a minimum test size N is based on the notion of test quality with respect to the retained criterion. This quality, denoted q_N, is the probability of exercising at least once the least likely element of S. Relation (1) gives the value of N under the assumption of statistical independence (see [15] for examples where the assumption does not hold). It can be explained as follows: $(1-P)^N$ is an upper bound of the probability of never exercising some element during N executions with random inputs. Then, for a required upper bound of $1-q_N$, where the target test quality q_N will be typically taken close to 1.0, a minimum test size is derived.

$$N = \ln(1-q_N) / \ln(1-P) \qquad (\mathbf{1})$$

Returning to the motivation of statistical testing – exercising several times each element defined by the criterion, it is worth noting that Relation (1) establishes a link between q_N and the expected number of times, denoted n, the least likely element is exercised: $n \cong - \ln(1-q_N)$. For example, $n \cong 7$ for $q_N = 0.999$. This relation is used to tune the test size N when the input distribution is empirically determined.

The main conclusions arising from previous work conducted on procedural programs were that: (i) statistical testing is a suitable means to compensate for the tricky link between test criteria and software design faults, (ii) the most efficient approach should be to retain weak criteria (e.g., transition coverage) facilitating the search for an input distribution, and to require a high test quality with respect to them (e.g., $q_N = 0.999$).

We are now studying the use of statistical testing for OO software. In the work presented in Sections 5 and 6, we adopt a functional approach consisting in basing the probabilistic generation of test patterns on the information got from the OO analysis and design documents. The production cell case study allows us to investigate this issue taking the example of the Fusion method [6].

4 Presentation of the Case Study

The case study is adapted from an industrial production cell. It was launched by the FZI (Forschungszentrum Informatik) inside the framework of the German Korso Project [13]. In addition to the informal specification, a Tcl/Tk simulator was made available by the FZI: it mimics the movements of the physical devices of the cell (Fig. 1).

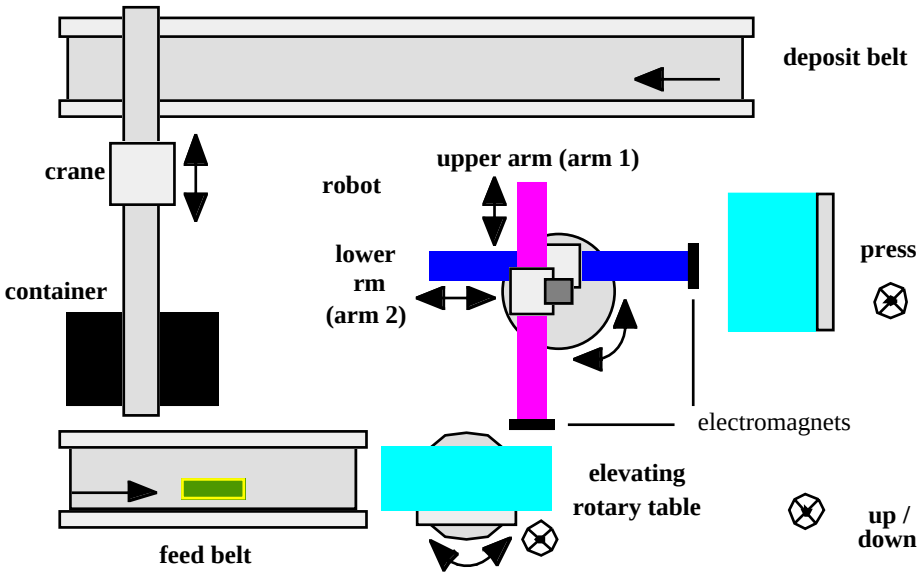

Fig. 1. Top view of the production cell

The aim of the production cell is the transformation of metal blanks into forged plates and their transportation from the feed belt into a container. The production cycle of each blank is the following: the blank is put on the feed belt by an operator; the feed belt conveys the blank to the table which rotates and lifts to put the blank in the position where the first robot arm is able to magnetize it; this arm places the blank into the press where it is forged; the second robot arm places the resulting plate on the deposit belt; then the crane magnetizes the plate and brings it from the deposit belt into a container. At a given time, several blanks may be in transit within the cell.

The cell is an industry-oriented problem where safety requirements play a significant role: 21 safety requirements and 1 liveness requirement (« every blank introduced in the cell will be forged and eventually dropped into the container ») are listed in the informal specification. Two examples of safety requirements are given below.

Requirement 18. A new plate may only be put on the deposit belt if the photoelectric cell, installed at the end of the belt, has detected the previous plate.

Requirement 21. If there is a blank on the table, the robot arm 1 may not be moved above the table if it also conveys a blank (otherwise the two blanks collide).

A control program for the production cell has been developed using the Fusion method [1]. This provides us with an example of a complete OO development process: Fusion analysis, Fusion design, and Ada 95 implementation. Fusion [6] is presented by its authors as a synthesis of prominent OO methods (OMT/Rumbaugh, Booch, Objectory, CRC). It includes some aspects coming from formal specification methods (pre- and postconditions). In this section, only few elements of the Fusion analysis phase are described: the ones that are used for the definition of the test experiments.

The Fusion analysis starts with the creation of the *system context diagram* of the controller (Fig. 2): it consists of six concurrent agents communicating by asynchronous events. Note that, since the controller is an active concurrent system, it has been separated – as proposed in Section 3.5 of [6] – to view it as a set of cooperating agents, each of which being developed using Fusion. The meaning of arrows is the following: an event may be sent at any time; the receiving agent queues the event until it is able to treat it; the sending agent is blocked until the end of the event treatment. The principle underlying event emission is that each agent is autonomous: it will do as many actions as it can independently. And for this, an agent may reorder the events it has queued: the order of event treatments does not always correspond to the order of event receptions into the waiting queues.

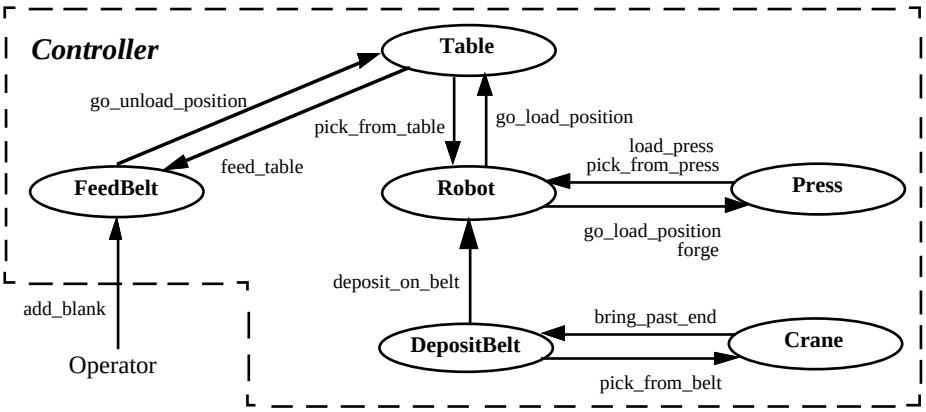

Fig. 2. System context diagram (inside view)

The expected behavior of an agent is described by its *interface model* which consists of a lifecycle and an operation model. The *lifecycle* specifies the order of event treatments and emissions by the agent. It is defined in terms of regular expressions. For example, the Table lifecycle is shown below:

initialize.#feed_table.(go_unload_position.#pick_from_table.go_load_position.#feed_table)*

It means that the first event to be treated is initialize which leads to the emission of feed_table (prefix # denotes emission); then any number of cycles of go_unload_position and go_load_position (with event emissions) may be treated by the Table agent.

The treatment of each event by the receiving agent is specified in the operation model which includes pre- and postconditions. For example, the treatment of go_load_position by Table corresponds to the following conditions:

Precondition:	No blank on the table.
Postcondition:	The table is in load position.
	An event feed_table has been sent to FeedBelt.

In this example, the precondition of go_load_position is implied by the postcondition of pick_from_table treated by the Robot agent. But the order specified in the lifecycle does not always guarantee that the preconditions hold. In such cases, the receiving agent has to queue the event until the precondition becomes true: other events can be treated in the meantime. Thus, the interface model implies the reordering of events by the receiving agent: the order of event treatments must comply with both the agent lifecycle and the operation preconditions.

5 Overview of the Test Strategy

The test strategy is based on two kinds of information: the list of 21 safety requirements, and the models got from the Fusion analysis phase. A first problem is the determination of testing levels. To tackle this problem, we consider the possibility of associating a functional description with a (set of) class(es):

- The *unit level* corresponds to small subsystems that are meaningful enough to have interface models in Fusion. From Section 4, there are 6 such subsystems: FeedBelt, Table, Robot, Press, DepositBelt and Crane. Each of them is already an aggregation of classes. It means that basic classes (e.g., the Electro_Magnet class) are not tested in isolation: they are tested through their embodying subsystems (e.g., Robot).
- The *integration process* is guided by the consideration of the safety and liveness requirements. For example, requirement 21 leads to the definition of an integration test for the subsystem Robot+Table. Thus, four integration tests (FeedBelt+Table, Robot+Table, Robot+Press, Robot+DepositBelt) and one system test are defined.

The respective focus of each testing level (unit, integration) is determined in order to define a cost-effective test strategy for *reusable* components. The concern is to identify what can be tested once for all at the unit level and what has to be tested during the integration process specific to each application.

Unit testing is focused on verifying conformance to interface models. For reusability purposes, unit testing is designed without making any assumption on the operational context: it is well-recognized that a component that has been adequately tested in a given environment is not always adequately tested for another environment. Hence the test sequences are not restricted to the ones that are possible in the production cell context. Roughly speaking, the units are placed in an "hostile" environment: there is no timing or sequence assumption related to the events sent to them. This allows us to verify the correct reordering of events, and the correct treatment of the reordered events, in response to arbitrary solicitations. If the unit is not robust enough to pass the test, its design can be improved. Or, at least, the test results allow the identification of usage assumptions that must be added to its interface model.

Whether or not these assumptions hold in a given context (e.g., the production cell) has to be verified during the *integration process*. Also, integration testing allows the verification of safety requirements involving several units of the production cell. The liveness requirement is verified by the final system test. Contrary to the design of unit testing, the design of integration testing takes into account some characteristics of the cell. This leads to a more constrained version of the environment. For example, when testing Robot+Table, it is not possible to sent a load_press event to Robot while the previous one had not yet been treated. This is so because the robot is connected to a single press, and this press is blocked until the previous load_press is treated.

An overview of all experimental results is shown at the end of this paper. Section 6 focuses on the design of unit testing only (see [2] for the complete case study).

6 Unit Testing of Reusable Components

The unit testing process can be decomposed into phases that accompany the Fusion phases. The main objective – verify the conformance of the units to their interface models – has first to be refined in terms of workable test criteria and conformance relations. Fusion analysis models are examined with a view to testing: this leads to their reformulation in a compact and non ambiguous form (Section 6.1) from which test criteria are derived (Section 6.2). Statistical test sets are designed in accordance with the criteria (Section 6.3) and an oracle procedure is specified (Section 6.4). Then the development of a test environment supporting the refined objective requires a number of controllability and observability problems to be handled (Section 6.5): solutions to these problems must be determined in relation to the choices taken in the late phases of the production cell development (Fusion design and Ada implementation). Experiments are performed using the resulting test environments (Section 6.6).

6.1 Reformulation of the Fusion Interface Model

The lifecycle model specifies the order in which each unit should process input events and send output events. The processing of input events is made more precise in the operation model, where in particular pre- and postconditions are stated. The textual

lifecycle expression can be put into an equivalent form: a finite state automaton recognizing the regular expression. However, this automaton is not sufficient to describe the allowed sequences of processed events because no operation should be triggered outside its precondition: the set of allowed sequences should be further constrained by considering whether the postcondition of one operation implies the precondition of the next one. This leads us to reformulate the Fusion interface model.

First, a *completed version of the operation model* is required for testing purposes. According to the Fusion method, a condition that has to be true both before and after the operation is not listed, neither as a precondition nor as a postcondition; and the granularity of the operation model does not distinguish between the case where the condition remains true throughout the operation execution, and the case where it turns to false and then returns to true before the end of the operation. In both cases, it is important to check the validity of the condition after the operation. Hence, pre- and postconditions are expanded in the completed operation model. For example, in the model of the Table operation go_load_position (Section 4), pre- and postconditions are added, including the following ones: the rotation and elevation motors are off.

Then, combining the lifecycle expression and the completed operation model, the allowed sequences of event treatment for each unit are reformulated as *finite state automata*. Figure 3 shows the reformulation of the lifecycle of Robot which is the most complex unit (for the other units, we get 2-states automata). The textual lifecycle expression would have given us a 4-states automaton, depending on which arm is carrying a plate. The examination of the operation model shows that presence or absence of a plate in the press should also be taken into account: Robot is not allowed to process pick_from_press if it did not previously load the press.

The lifecycle automata describe the allowed sequences of event treatments, but they say nothing about the event reordering functionality to be supplied by the units. Input events are supposed to be stored in a waiting queue if the unit is not ready to process them. When no timing and sequence assumption related to the events sent to the unit is made, the size of the waiting queue may be infinite: the reordering mechanisms of event processing cannot be modeled by an automaton with a finite number of states.

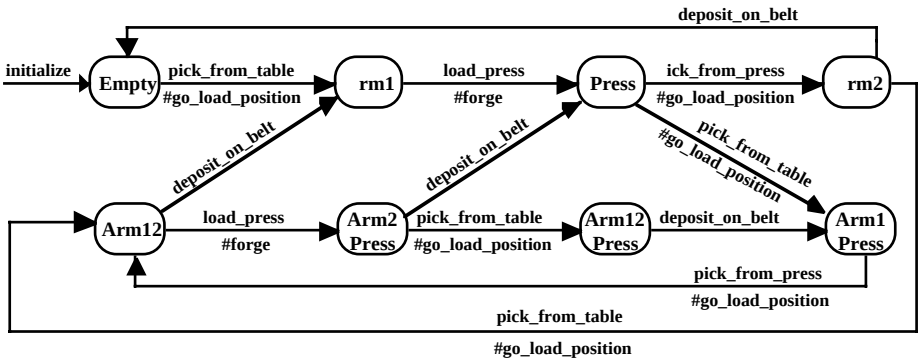

Fig. 3. Robot lifecycle automaton

In order to represent both the reordering mechanisms and the allowed sequences of event treatments, we need a formalism that is more powerful than finite state automata. The solution retained is to attach an *input event queue* to each state of the lifecycle automaton: this queue is characterized by the number of events of each category (for Robot, pick_from_table, pick_from_press, load_press, deposit_on_belt) that are waiting to be processed. It is worth noting that the reformulated model (automaton + queues) may involve nondeterminism, for example (Fig. 3) when state Arm12 is reached while both deposit_on_belt and load_press events are queued.

6.2 Test Criteria

Two different test criteria are needed, based on the reformulated models. As regards the *correct treatment of reordered events*, the criterion retained is the coverage of the *transitions of the lifecycle automata* with a high test quality of $q_N = 0.999$. This test quality implies that the least likely transition should be exercised 7 times on average by a given test set (Section 3).

As regards the *correct reordering of events*, the retained criterion is the coverage of *four classes of queue configurations* for each category of event: the number of queued events of that category is 0, 1, 2 or ≥3. We require the same test quality as previously. Note that queue configurations with more than one event are not possible in the production cell context. Yet, they are forced during testing for reusability purposes.

6.3 Design of Statistical Test sets

Controlling coverage of transitions and queue configurations is not trivial. Due to concurrency, the internal behavior of one unit depends not only on the order in which events are received, but also on the interleavings of event reception and event treatment.

To illustrate the problem, let us assume that a test sequence first takes the robot to the Arm12 state (Fig. 3) with no queued event, and then is continued by subsequence load_press.deposit_on_belt.pick_from_table. The robot behavior may be different depending on the triggered interleaving (Fig. 4). If the time intervals between the three events are such that none of them is received before the previous one has been processed (Fig. 4a), then deposit_on_belt is processed before pick_from_table and the queue remains empty. If both deposit_on_belt and pick_from_table are received before completion of the load_press operation (Fig. 4b), then one of the events is queued and the other processed depending on some implementation choice. The choice in Figure 4b is the one of the Ada program: deposit_on_belt has lower priority than load_press and pick_from_table. Examples 4a and 4b show that the triggered coverage of transitions and queue configurations may be quite different for two interleavings of the same sequence of events. This leads us to introduce some notion of time in the definition of test sequences.

It is not possible to know in advance the exact duration of one operation: the best that can be done is to estimate a conservative upper bound of it. Then, if the delays between input events are long enough compared to the reaction time of the units (slow environment), specific interleavings such as the one of Figure 4a are forced, making it

possible to assess the coverage supplied by a given sequence of events. Such an a priori assessment is impossible when the units are not run in a slow environment, because the occurrence of a specific interleaving depends on uncontrollable factors: the behavior may be non reproducible from one run to the other. Yet, interleavings such as the one of Figure 4b should not be excluded from the test input domain if they may occur in operational contexts. For example, if the treatment of events is not atomic, it must be possible to trigger the reception of events during an on-going treatment. And irrespective of the atomicity hypothesis, it must be possible to trigger schemes where several concurrent events are received in the meantime between two treatments. To account for all these schemes, the proposed sampling approach is to consider several load profiles, i.e. profiles of time intervals between two successive events. Then a test sequence is defined as a sequence of input events with time intervals between events.

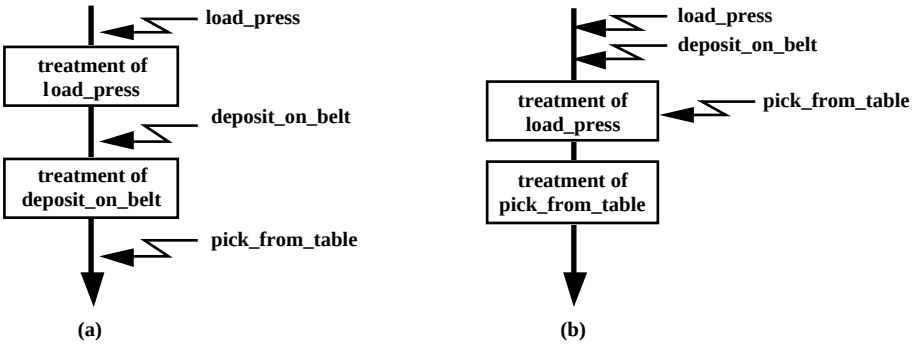

Fig. 4. Different possible interleavings for a same sequence of events

Event sequences are first designed. Assuming a slow environment, it is possible to search for a probability distribution of input events that is proper to ensure rapid coverage of the criteria. The analysis proceeds empirically: coverage measures are collected from programs simulating the reformulated model of each unit (automaton + queues). Then, input distributions are determined by successive trials with random sets of events. Robot is the unit for which the empirical process required the largest number of trials: in the retained distribution, the probability of events is tuned according to the current state and queue configuration. For the other units, making all events equally likely (uniform stimulation) turns out to provide a balanced coverage of transitions and queue configurations. Sets of event sequences are randomly generated according to the retained distributions. Each set provides the required test quality of $q_N = 0.999$ with respect to the criteria: every transition and queue configuration is exercised more than 7 times. Note that since the initialization is one of the transitions to be exercised, each set contains at least 7 test sequences beginning with initialize. For example, the Robot test set contains 12 test sequences involving a total number of N = 306 events. An example of sequence included in this set is provided below. It is one of the shortest sequences (the size varying over the range [5..40]):

initialize . pick_from_table . load_press . deposit_on_belt . pick_from_table . pick_from_press . load_press . pick_from_press . load_press . pick_from_table . deposit_on_belt

The *time intervals* between successive events are generated according to 3 load profiles:

- A low load profile: the time intervals are large compared to the reaction time of the unit (long delays).
- A high load profile: the time intervals are shorter than, or the same order of magnitude as, the reaction time of the unit (short delays).
- An intermediate load profile (mix of short, long and middle delays).

Hence, each set of events will be executed three times, once under each load profile. In this way, different interleavings are triggered for a same sequence of events. One of these interleavings, the one forced by the low load profile, ensures coverage of the criteria with the test quality previously assessed. The other two profiles induce interleavings like the one of Figure 4b, possibly in a nondeterministic way.

The values associated to the three profiles take into account the average response time of the FZI simulator, but could be calibrated for other environments in a similar way. With the simulator, the reaction time of each unit to process one event is of the order of magnitude of a few seconds. Accordingly, the time intervals are generated as follows: 1) uniform generation over [15s..20s] for the low load profile; 2) uniform generation over [1s..15s] for the intermediate load profile; 3) uniform generation over [0s..5s] for the high load profile. For the previous Robot test sequence, the values generated under the low load profile are shown below:

initialize . (16s) pick_from_table . (15s) load_press . (18s) deposit_on_belt . (15s) pick_from_table . (15s) pick_from_press . (17s) load_press . (15s) pick_from_press . (18s) load_press . (20s) pick_from_table . (18s) deposit_on_belt

6.4 Oracle checks

The role of the oracle is to determine conformance of the test results to the expected ones. Most of the corresponding checks are based on our reformulation of the interface models. They take advantage of the fact that the lifecycle automata turn out to possess a remarkable property: for any test sequence, the final state and the number of processed events at the end of the sequence do not depend on the nondeterministic choices made during the execution. For example, the interleavings of Figure 4 both take the robot to the Arm1/Press state with an empty queue, irrespective of the intermediate behavior. Accordingly, the oracle conformance checks are specified as follows:

- The observed sequences of input events processed and output events sent must be recognized by the lifecycle automaton.
- The number of events that are processed by the unit is the same as the number of events that would be processed by the lifecycle automaton exercised with the same test sequence. Due to the property mentioned above, adding this check to the previous one ensures that the implementation has the same deadlocks as the specification.
- After each treatment of event, the associated postconditions got from the completed operation model must hold.

In addition to the previous checks, our oracle procedures include the verification of 15 safety requirements that can be related to one unit taken in isolation. For example,

requirement 18 (see Section 4) is included in the DepositBelt oracle. Although verifying safety requirements is not the main objective of unit testing, it is interesting to consider them at the unit level: if they are shown to hold when testing in a "hostile" environment, then *a fortiori* they should hold in the production cell context. Indeed, the experimental results will provide examples of requirements that are violated at the unit level when no assumption is made on the operational context; yet, integration tests show that they hold when the unit is placed in the production cell.

6.5 Unit Test Environments

Having defined the test sets and test oracles, we have now to develop the test environments allowing us to perform the experiments. As expected, this raises a number of controllability and observability issues.

Let us recall that the test environment must be able to *control any arbitrary input sequence*: there is no ordering assumption related to the events sent to the units. Since the sending of events is blocking, we must be careful not to introduce deadlocks when the event order departs from the one specified in the lifecycle. Let us take the example of a unit having a lifecycle defined as (E1.#e1.E2#e2)*, and exercised with an input sequence having order E2.E1. The expected behavior of the unit is to treat both events in the lifecycle order: the treatment of E2 is delayed until E1 has been processed and e1 has been sent. If the test driver is implemented by a single Ada task sequentially sending E2 and E1, then the driver will be blocked on E2: E1 will never be sent. This is not acceptable since the test environment should always be able to send the next event, as defined in the sequence. To handle this controllability problem, the adopted solution is to have the input events sent by several concurrent drivers.

Observability is prerequisite to the feasibility of oracle checks: appropriate information is to be monitored by the test environment. Checking conformance to the lifecycle automata relates to the observation of the treatment of input events and emission of output events. However, the reordering of input events according to the lifecycle is encapsulated in the units: once the input events have been received by a unit, the test environment may be unable to know the order of their internal treatment. This problem is solved by inserting a print statement at the start point of the treatment in the Ada code, thus making observable the beginning of each operation. Postconditions and safety requirements relate to the state of the physical devices controlled by the software units (e.g., position of the robot arms). In our test environment, their verification is implemented by instrumenting the FZI simulator used to mimic the reaction of the physical devices. All the observations must be synchronized with the behavior of the units, and some ordering relations must be preserved by the results of (possibly concurrent) observers: for example, for a given unit, observation of the beginning of one operation is always reported before the end of this operation; postconditions checks are always performed and reported before the beginning of the next operation of this unit.

A generic view of the resulting test environments is provided in Figure 5. The corresponding design choices are justified in [2], and we do not further detail them in

this paper. Let us just mention two important remarks. First, it can be seen that a number of test components had to be developed in order to control and observe the unit under test. Second, it is worth noting that a few postconditions were not implemented in the oracle procedures: due to synchronization constraints, making them observable would have required an important instrumentation of the corresponding units and this was considered as too intrusive.

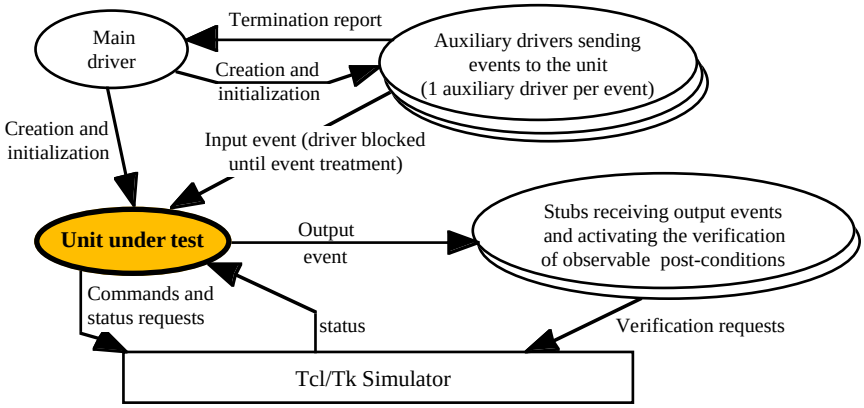

Fig. 5. Generic view of unit test environments

With these test environments, each test experiment generates a trace file recording:

- The sequence of every event treated or sent by the unit under test.
- The number and category of input events not treated at the end of the test experiment.
- The results of the checks for the status of the devices at the end of the operations.
- The error messages of the FZI simulator. The FZI simulator has built-in mechanisms to issue an error message in case of abnormal situations like collision of devices, or falling of metal plates.

Then the trace file is analyzed off-line by an oracle program in order to issue an acceptance or rejection report.

6.6 Experimental Results

The test experiments first revealed a *synchronization problem* that was observed whatever the unit under test (violation of postconditions and safety requirements). It concerns communication with the FZI simulator: the synchronization mechanisms do not ensure that requests to stop the physical devices are received in time by the simulator. For example, when the Robot operation load_press is executed, the extension of the upper arm (arm 1) over the press may be actually stopped too late so that the blank falls beside the press. Failures are all the more frequent as the load

profile is high, but their occurrences are not repetitive from one run to the other: executing several times a same test set systematically produces failures, but possibly at different times during the experiment. These intermittent faulty behaviors are closely related to the real-time behavior of the program which is not adequately addressed by the Fusion method. Nevertheless, the fault is repeatedly exposed by the statistical test sets. It is fixed by modifying the Ada code of two basic classes (Sensor_Server, Actuator_Server) in charge of the communication with the physical devices. With this fix, four agents (FeedBelt, Table, Press, Crane) pass their test. For the other two agents (Robot, DepositBelt) errors are detected.

As regards *Robot*, the number of deposit_on_belt treated by the Ada program is lower than the expected one: when several deposit_on_belt are queued, only one of them is treated and the others are ignored. Failures are observed whatever the load profile, but their rate is raised under the high load profile since the waiting queues are larger. These test results show that the Robot agent is not robust enough to be used in any context of applications: its correct behavior assumes that no new deposit_on_belt event is sent to Robot before the previous one has been processed. The fault is fixed by a simple modification of the deposit_on_belt operation (replacement of a Boolean flag by a counter), and the fixed Robot passes its unit test.

As regards *DepositBelt*, it is first observed that two events sent during the treatment of the initialize operation occur in the reverse order compared to the one defined in the Fusion lifecycle. After analysis, our diagnosis is that both orders are consistent with the initial (informal) specification of the application, and none of them may lead to the violation of safety requirements. Our oracle procedure is modified in accordance with this diagnosis and it is requested that this modification is integrated in the Fusion analysis document. A second and more significant problem is related to the dropping of plates on the belt. It is observed that plates may be dropped too close to each other, so that they collide or fall at the end of the belt; two safety requirements are violated (e.g., requirement 18 in Section 4). The problem is due to the fact that the dropping of plates is not encapsulated into an operation of DepositBelt. Then the DepositBelt agent is not robust enough to be used in any context of applications: its correct behavior assumes that plates are dropped only when expected by the belt, that is, when the previous plate has already reached the end of the belt.

Fixing this fault requires the modification of the DepositBelt interface, and thus of the interface of the Robot agent which interacts with DepositBelt. The modifications would have repercussions at every level from the Fusion analysis document to the Ada code. In the context of the production cell, such substantial modifications are not needed if it can be shown that the correct usage assumption holds. Indeed, one of the preconditions of the Robot operation deposit_on_belt should ensure that the robot cannot drop a plate while there is another one at the beginning of the belt. Hence, the decision to perform the modifications is delayed: the correct behavior of the subsystem Robot+DepositBelt has to be verified by integration testing (Section 7).

Besides the exposure of faults, the test experiments give us feedback on the internal behavior of the Ada program under the various load profiles. We have instrumented the oracle procedures checking conformance to the lifecycle automata, in order to collect statistics on the number of times each transition is actually executed. The statistics

confirm that transition coverage heavily depends on the load profile. For instance, some transitions of the Robot automaton are never or seldom executed under the high load profile, and others are much more exercised. This is due to the fact that when numerous events are available in the waiting queues, their order of treatment is enforced by the priority choices taken in the Ada implementation. For example (Fig. 3), the robot transition from Arm2 to Arm12 state (triggered by pick_from_table) is never exercised under the high load profile: lower load profiles are required to reveal faults related to this transition. Conversely, our test results show that the high load profile may be more effective in some cases. This corroborates our expectation that the use of different load levels should strengthen the fault revealing power of test sets.

7 Summary and Conclusion

Table 1 displays the results of the whole set of test experiments. A complete description of the integration testing process is available in [2]. Let us note that the integration test experiments do not reveal any fault related to the joint behavior of the units under test: in particular, the two safety requirements violated during unit testing of DepositBelt are satisfied by the subsystem Robot+DepositBelt: as expected from the Robot interface model (Section 6.6), the assumption governing the correct behavior of DepositBelt does hold. At the system level, only one input can be controlled: the introduction of blanks in the production cell; it is worth noting that the experiments confirm that the interleaving of the internal events is quite different depending on the time intervals between the successive blanks.

This theoretical and experimental investigation allows us to draw more general conclusions on testing of reusable concurrent objects. The main lessons learnt are summarized below. They are classified according to three issues: object-oriented technology, concurrency and reusability.

Table 1. Overview of the test results

Unit Tests	FeedBelt, Table, Press, Crane	Robusts, 8 safety requirements satisfied
	Robot	Made robust ➥ 5 safety requirements satisfied
	DepositBelt	Assumption on the usage context, 2 safety requirements violated
Integration Tests	Robot+DepositBelt	Assumption holds in the production cell context ➥ the 2 safety requirements are satisfied
	FeedBelt+Table Robot+Table Robot+Press	2 safety requirements satisfied 1 safety requirement satisfied 2 safety requirements satisfied
System Test	1 liveness requirement satisfied	

As regards *object-oriented* technology, the information got from OO development methods should allow us to design efficient statistical functional test sets: this has been exemplified using Fusion analysis document; but, the information used is quite similar to the one provided by other OO methods. However, whatever the chosen testing approach, the requirements of the test environments must be identified early in the development process, in accordance with the pursued test objectives. Especially, it must be identified what is to be controlled and what is to be observed. Then, how to handle the resulting controllability and observability issues is necessarily constrained by the design and the implementation choices taken for the application. This leads us to recommend that the development of the test environments accompany the corresponding development phases of the application. This is not new, since recommended for a long while in the well-known "V development process". But, it is still more crucial in cases of OO software because of the testability problems increased by encapsulation mechanisms. Returning to the Fusion example, the determination of the testing levels, test criteria and oracle checks must accompany the analysis phase; as for the design and implementation of the test environments, they must be conducted during the Fusion design and implementation phases.

As regards *concurrency*, the concern is not to exclude any event interleaving that may occur in operational contexts, that is, to account for nondeterminism induced by asynchronous communication. To tackle this problem, it is necessary to introduce some notion of time in the definition of the test sequences: for each sequence of events to be sent during testing, time intervals between successive events must be generated according to several load profiles that mimic different types of operational contexts. We propose to use a sampling technique based on three typical load profiles: long delays compared to the reaction time of the subsystem under test; short delays; and a mix of short, long and middle delays. Another notable impact of concurrency is related to the design of test environments which may become complex compared to the subsystem under test: the environments must consist of concurrent drivers and stubs in order to handle the controllability and observability problems added by concurrency.

Finally, the *reusability* concern influences the choice of the test criteria that guide the selection of test inputs: the test sequences must not be restricted to the ones that are possible in the context of the application under test. The test input domain is enlarged. Yet, the benefit of testing for reusability – when it remains feasible – is significant: either the unit under test is robust enough to pass the test, or it is not. If it is, it may be used in any context without requiring further testing. If it is not, its design can be made robust, or at least the test results allow the identification of usage assumptions that are required for the unit to conform to its interface model.

Acknowledgments. This work was partially supported by the European Community (ESPRIT Project n° 20072: DeVa). It has benefited from many fruitful discussions with our DeVa partners, in particular: Stéphane Barbey, Didier Buchs, Marie-Claude Gaudel, Bruno Marre and Cécile Péraire. We also wish to thank Damien Guibouret and Olfa Kaddour very much for their useful contribution to the case study, within the framework of student projects.

References

1. Barbey, S., Buchs, D., Péraire, C.: Modelling the Production Cell Case Study Using the Fusion Method. Technical Report 98/298, EPFL-DI, 1998
2. Barbey, S., Buchs, D., Gaudel, M-C., Marre, B., Péraire, C., Thévenod-Fosse, P., Waeselynck, H.: From Requirements to Tests via Object-Oriented Design. DeVa Year 3 Deliverables (1998) 331-383. Also available as LAAS Report no 98476
3. Beizer, B.: Software Testing Techniques. 2nd edn. Van Nostrand Reinhold, New York (1990)
4. Binder, R.: Design for Testability in Object-Oriented Systems. Communications of the ACM 37 (1994) 87-101. Also available in [12]
5. Binder, R.: Testing Object-Oriented Software – A Survey. Software Testing, Verification & Reliability 6 (1996) 125-252
6. Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P.: Object-Oriented Development – The Fusion Method. Object-Oriented Series, Prentice Hall (1994)
7. Fabre, J-C., Salles, F., Rodriguez Moreno, M., Arlat, J.: Assessment of COTS Microkernels by Fault Injection. In Proc. 7th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7), San Jose (1999) 19-38
8. Harrold, M-J., McGregor, J., Fitzpatrick, K.: Incremental Testing of Object-Oriented Class Structures. In Proc. 14th IEEE Int. Conf. on Software Engineering (ICSE-14), Melbourne (1992) 68-80. Also available in [12]
9. Jorgensen, P., Erickson, C.: Object-Oriented Integration Testing. Communications of the ACM 37 (1994) 30-38. Also available in [12]
10. Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C., Kim, Y-S., Song, Y-K.: Developping an Object-Oriented Software Testing and Maintenance Environment. Communications of the ACM 38 (1995) 75-87. Also available in [12]
11. Kung, D., Lu, Y., Venugopala, N., Hsia, P., Toyoshima, Y., Chen, C., Gao, J.: Object State Testing and Fault Analysis for Reliable Software Systems. In Proc. 7th IEEE International Symposium on Software Reliability Engineering (ISSRE'96), White Plains (1996) 76-85. Also available in [12]
12. Kung, D., Hsia, P., Gao, J. (eds): Testing Object-Oriented Software. IEEE Computer Society (1998)
13. Lewerens, C., Linder, T. (eds): Formal Development of Reactive Systems – Case Study Production Cell. Lecture Notes in Computer Science, Vol. 891, Springer-Verlag (1995)
14. Luo, G., Bochman, G. v., Petrenko, A.: Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method. IEEE Trans. on Software Engineering 20 (1994) 149-162
15. Thévenod-Fosse, P., Waeselynck, H., Crouzet, Y.: Software Statistical Testing. In: Randell, B., Laprie, J-C., Kopetz, H., Littlewood, B. (eds): Predictably Dependable Computing Systems. Springer-Verlag (1995) 253-272
16. Turner, C., Robson, D.: The State-Based Testing of Object-Oriented Programs. In Proc. IEEE Conference on Software Maintenance (1993) 302-310. Also available in [12]
17. Voas, J.: Object-Oriented Software Testability. In Proc. 3rd International Conference on Achieving Quality in Software , Chapman & Hall (1996) 279-290
18. Voas, J.: Certifying Off-the-Shelf Software Components. IEEE Computer 31 (June 1998) 53-59.