Design Patterns for Fault Containment *

Titos Saridakis

NOKIA Research Center
PO Box 407, FIN-00045, Finland
titos.saridakis@nokia.com

Abstract

Fault containment is an important constituent of fault tolerance. Means for fault containment allow a system to limit the impact of manifested faults to some predefined system boundaries. This document presents some of the best known techniques for fault containment formatted as design patterns. These patterns are elicited from the areas of self-stabilization, specification closure and fault tolerant OS kernels. The presented fault containment patterns are: the `Input Guard` pattern which confines an error outside the guarded system boundaries; the `Output Guard` which confines an error inside the guarded system boundaries; and the `Fault Container` pattern which is the fault tolerant counterpart of the well-known Adapter pattern and which combines the properties of the `Input Guard` and `Output Guard` patterns.

1 Introduction

This paper describes a set of patterns that provide solutions to the fault containment problem, i.e. how to prevent an error from contaminating other parts of a system besides the place where it occurred. For the reader's convenience, we provide a brief summary of the basic fault tolerance terms and definitions taken from [7] that are necessary for following the presentation of these patterns.

A *system* is an entity with a well-defined behavior in terms of output it produces and which is a function of the input it receives, the passage of time and its internal logic. By "well-defined behavior" mean that the output produced by the system is previously agreed upon and unambiguously distinguishable from output that does not qualify as well-defined behavior. The well-defined behavior of a system is called the system *specification*. A system interacts with its environment by receiving input from it and delivering output to it. It may be possible to decompose a system into constituent (sub)systems, often called system *components*. A *failure* is said to occur in a system when the system's environment observes an output from the system that does not conform to its specification. An error is the part of the system, e.g. one of the system components, which is liable to lead to a failure. A fault is the adjudged cause of an error and may itself be the result of a failure. Hence, a fault causes an error that produces a failure, which subsequently may result to a fault, and so on. Let us consider the following example:

*Copyright 2003 © by NOKIA. All rights reserved. Permission is granted to copy for EuroPLoP 2003.

> A software bug in an application is a **fault** that leads to an **error** when the application execution reaches the point affected by the bug, which in turn makes the application crash which is a **failure**. By crashing, the application leaves blocked the socket ports it used which is a **fault** and the computer on which the application crashed has socket ports which are not used by any process nevertheless not accessible to running applications which is an **error**, and which in turn leads to a **failure** when another application requests these ports.

Based on the above, an error caused by a fault in a system may propagate to the system's environment unless the fault is contained. A system is called fault tolerant when it can deal with faults and their consequent errors in such a way that it does not violate its specification, i.e. the environment of a fault tolerant system does not perceive a failure of the system. Hence, a fault tolerant system does not propagate faults to its environment. The distilled experience of three decades of developing fault-tolerant software systems, as summarized in [10], indicates that fault tolerance has the following six constituents:

- *error masking,* which deals with the dynamic correction of the occurred errors;
- *error detection,* which spots and reports the occurrence of errors;
- *fault containment,* which prevents the error propagation across defined boundaries;
- *fault diagnosis,* which identifies the component where resides the fault that produced a reported error;
- *fault repair,* which provides the means for the elimination, the replacement or the bypassing of a component;
- *system recovery,* which ensures the re-establishment of a legitimate system state (i.e. a state that is part of the system specification) after an error has been reported.

These constituents of fault tolerance are not independent from each other. For example, a fault tolerance mechanism that implements roll back recovery (e.g. see [6]) where upon the occurrence of an error the last checkpointed state of the system is restored, uses means for error detection, fault containment, fault diagnosis and system recovery. Another fault tolerance mechanism that implements the State Machine Approach [16] with dynamic system reconfiguration in order to eliminate faulty replicas, uses means for error masking, error detection, fault containment, fault diagnosis and fault repair. A number of design patterns for error masking, error detection and fault repair, and system recovery which capture well-known fault tolerance techniques have been presented and classified in our previous work [15].

This paper presents three design patterns for fault containment: the `Input Guard` pattern which ensures that no error is propagated from the outside to the inside of the guarded system boundaries; the `Output Guard` pattern which ensures that no error is propagated from the inside to the outside of the guarded system boundaries; and the `Fault Container` pattern which is the fault tolerant counterpart of the well-known Adapter pattern and which will be shown to combine the properties of the `Input` and `Output Guard` patterns.

2 Fault Containment

Means for fault containment are suggested by different directions in fault tolerance literature. Work on self-stabilization [17] as well as work on closure and convergence [1] introduce the notion of *guards*. The implementation of those guards (e.g. see [3, 4]) provides the basic fencing material that stops error propagation beyond defined system boundaries often called fault compartments. Work in fault containment is also done at the OS kernel level, either for Mach [13] or for Chorus [14] or even for share memory multiprocessors [12]. These cases introduce guards to ensure that the output values of a system component belong to a valid set defined in the system specification. If an error is detected by these guards, they stop the output of the guarded component from being passed over to the rest of the system and contaminate it with the propagated error.

On the other hand, the concept of guards is not specifically introduced for fault tolerance purposes. Design by contract [9] introduces the notion of pre-conditions which serve as guards for the execution of statements. Unless the pre-condition for a given statement holds, the statement does not guarantee correct results or the behavior described in the system specification. The design by contract principles compel a particular way of describing the system specification. Namely, pre-conditions are used to describe the conditions that the system input must satisfy (i.e. they describe the correct system input), post-conditions are used to express the guarantees provided by the system upon successful processing of correct input, and invariants are used to express the conditions that will hold true while the system is processing correct input. This specific way of describing the system specification bears many similarities with the concepts of guards that are introduced in this paper. However, this does not imply that the system specification must be expressed in terms of pre- and post-conditions and invariants in order for these patterns to be applicable.

The fault containment techniques captured in the patterns presented in this paper stop the propagation of errors to (and from) the system on which they are applied. Such errors have the form of erroneous input (and output respectively) with respect to the system specification. However, if certain input or output is erroneous according to the specification of the environment of a given system but not according to the specification of the system itself, the fault containment patterns described in the subsequent sections cannot stop the propagation of the corresponding error. Let's consider the following example, which clarifies both the cases where the presented patterns can effectively stop the propagation of errors and where they cannot.

Consider a component called $T2N$ that operates on strings of English words and transforms numbers from their textual form (e.g. one, two, three) to their numeral form (e.g. 1, 2, 3). The specification of this component defines that the input can be any string of alphanumeric ASCII characters and spaces (e.g. *"I have 4 eggs and one potato"*) and the output is the same string only with the numbers transformed from their textual to their numeral form (e.g. *"I have 4 eggs and 1 potato"*). Another component, called $GETIN$, reads either the user input from the keyboard until they key $\langle ENTER \rangle$ is pressed or from an indicated file until EOF is met and delivers in its output the stream of characters it read. Finally, the system S is composed by connecting the output of component $GETIN$ to the input of component $T2N$, as shown in three instances in Figure 1. The specification of S defines that the system takes its input either from a file until EOF is met or from the keyboard until $\langle ENTER \rangle$ is pressed. The output of the system is identical to its input modulo the transformation of numbers from their textual to their numeral form.

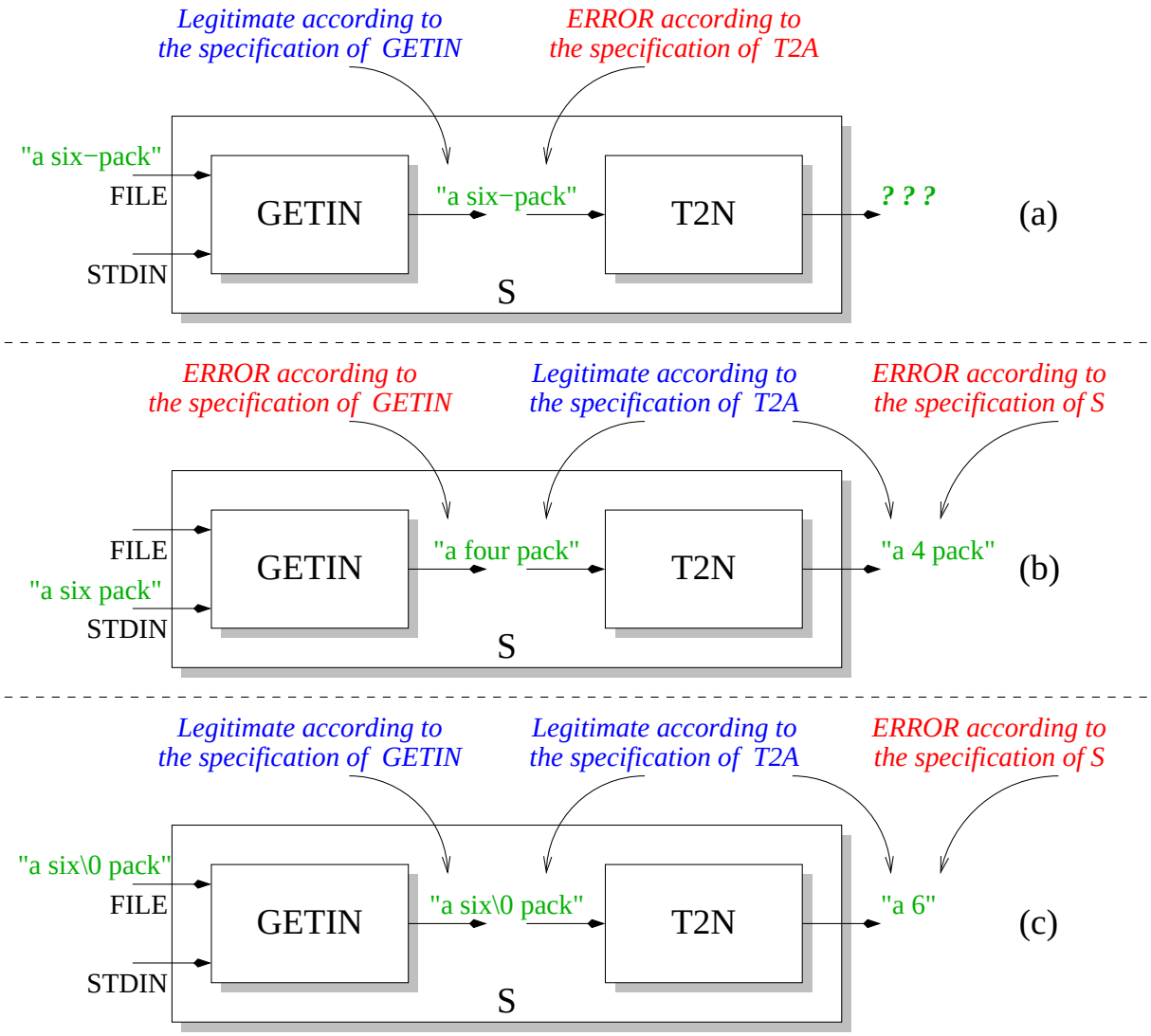

Figure 1: Example of errors that may occur to system S and its constituent components.

Figure 1 shows three cases of errors that may occur in S. Case (a) graphically illustrates an error that is due to a design fault; the specification of $GETIN$ allow output that is not correct input for $T2N$ and in the composition of S does not provide any kind of filtering between $GETIN$'s output and $T2N$'s input. In case (a) the component $GETIN$ reads from a file the string *"a six-pack"* and passes it as input to component $T2N$. This string however contains the dash character (*"-"*), which is not part of a correct input to $T2N$. A mechanism that checks the specification of $GETIN$ in order to stop the propagation of errors that occurred inside $GETIN$, would not provide fault containment in this case, since both input and output of $GETIN$ are legitimate according to its specifications. On the contrary, a mechanism that checks the specification of $T2N$ before allowing any input to be processed by $T2N$ would prevent the error propagation inside $T2N$, since it would be able to identify the erroneous input (the dash character in the string *"a six-pack"*). The application of the `Input Guard` pattern (Section 3) on $T2N$ will provide the desired fault containment for this case.

Case (b) shows an error that can be either due to a software bug in $GETIN$ or due to temporary anomalies in the operation of $GETIN$ (e.g. electromagnetic disturbances, high temperature, etc). Rather than delivering the user typed string *"a six pack"* as input to $T2N$, component $GETIN$ outputs the string *"a four pack"*. That is an error according to the specification of $GETIN$ but it is perfectly legitimate input for $T2N$. Hence, a mechanism that checks the specification of $T2N$ in order to stop error propagation inside $T2N$ would not provide fault containment in this case, since the string *"a four pack"* is legitimate input for $T2N$. What can stop the propagation of the error from $GETIN$ to its environment is a mechanism that checks the specification of $GETIN$ in order to detect errors in its output. The application of the `Output Guard` pattern (Section 4) on $GETIN$ will provide the desired fault containment for this case.

Finally, case (c) depicts another error that is due to a design fault similar to case (a). This time, the input that $GETIN$ reads from a file is a stream of binary data, the first five bytes are *"a six"* followed by the *NULL* character (ASCII code zero), followed by *" pack"*. This is a legitimate input according to the specification of $GETIN$ and the produced output in case (c) is also correct. The first six bytes of this output (*"a six\0"*) is a proper string with alphanumeric characters, which makes it a legitimate input for $T2N$ (e.g. for the definition of string in the C programming language, where whatever follows the first *NULL* character in a string buffer is just ignored). As a result, the propagation of the error cannot be stopped by checking the binary stream *"a six\0 pack"* against the specification of $GETIN$ or $T2N$. The boundaries inside which the error propagation can be stopped are the boundaries of system S. A mechanism that checks the output of S (i.e. *"a 6"*) for the given input (i.e. *"a six\0 pack"*) against the specification of S will be able to provide the desired fault containment for this case, and prevent the error occurred in S from being propagated to its environment.

3 Input Guard

Often, large scale systems are an assembly of independently developed components, like in the case of systems built out of Commercial off the Shelf (COTS) components. In such cases the developer of an individual component wants to prevent errors that may occur elsewhere in the system from infecting the component he developed. One way to achieve this is to verify that every input fed to his component by the system conforms to the input for his component as described in the system specification.

3.1 Context

The `Input Guard` pattern applies to a system which has the following characteristics:

- The system is composed from distinguishable components, which can play the role of fault compartments and which interact with each other by feeding one's output into another's input.
- It is possible, either directly or implicitly, to validate the input of a component against the legitimate input described in the component's specification.
- The errors that can be propagated into a system component have the form of erroneous input, i.e. input whose content or timing does not conform to the system specification.

The second characteristic implies that internal errors (e.g. changes to the internal state due to electromagnetic disturbances in the environment where the system operates) are not considered by this pattern since they are not expressed as erroneous input according to the system specification. Moreover, this pattern does not deal with cases where the input to the system conforms with the system specification but it still contains errors according to the specification of the system's environment (e.g. see case (b) in Figure 1).

3.2 Problem

In the above context, the `Input Guard` pattern solves the problem of stopping the propagation of an error from the outside to the inside of the guarded component by balancing the following forces:

- A system may be composed of components that are developed independently from each other, without keeping strict consistency with each others specifications.
- A system may be infected with errors from its environment even when the environment is not experiencing any errors according to its specification.
- Different systems have different requirements regarding size impact of the fault containment mechanism.
- Different systems have different requirements regarding the time penalty of the fault containment mechanism.
- Fault containment is usually integrated with other solutions provided for other fault tolerance constituents (e.g. error masking, error detection, fault diagnosis and the others mentioned in Section 1) in order to provide wider fault tolerance guarantees.

3.3 Solution

To stop erroneous input from propagating the error inside a component a guard is placed at every access point of the component to check the validity of the input. Every input to the guarded component is checked by the guard against the component specification. If and only if the input conforms with that specification then it is forwarded to the guarded component.

Notice that the above solution does not define the behavior of the guard in the presence of erroneous input, besides the fact that it does not forward it to the guarded component. This is intentionally left undefined in order to allow implementations of the `Input Guard` to be combined with error detection mechanisms (e.g. when a check fails, an error notification is sent to the part of the system responsible for fault diagnosis) or with the implementations of error masking mechanisms (e.g. the *comparator* entity of the `Active Replication` pattern [15]). Hence, the behavior of the guard when the checks performed on the input fail depends on the other fault tolerance constituents with which the input guard is combined.

3.4 Structure

The `Input Guard` pattern introduces two entities:

- The *guarded component* which is the part of the system that is protected against the fault contamination from external errors propagated to it through its input.
- The *guard* which is responsible to check for errors in the input to the guarded component against its specification.

There may be many instances of the *guard* entity for the same *guarded component*, depending on the system design and on the number of different access points the *guarded component* may have. For example, a software component with a number of interfaces and a number of operations declared in each interface may have one guard per interface or one guard per operation declared in its interfaces or any possible combination of those. Figure 2a illustrates graphically the structure of the *Input Guard* pattern for a *guarded component* with a single access point. Figure 2b contains the activity diagram that describes the functionality of the *guard*.

One possibility is to implement the *guards* as separate components in the system. This approach allows to have a number of *guards* proportional only to the number of the access points of the *guarded component*. The time overhead introduced by this approach is quite high since it includes the invocation of an additional component (i.e. the *guard*). Also, the space overhead of this approach is rather elevated since it increases the number of the components in a system by the number of *guards* that are implemented. Furthermore, in the case where components are mapped to individual units of failure (i.e. each component can fail as a whole and independently of other components) this approach introduces a well-known dilemma in fault tolerance: *"QUIS CUSTODIET IPOS CUSTODES?"* ("who shall guard the guards?"). This dilemma has also well-known consequences, which are difficult and very costly to resolve (e.g. redundant instances of the guards, distributed among the constituent components of a system, which have their own synchronization protocol).

Despite the above inconveniences, this implementation approach is valuable in the case of COTS-based systems composed from black-box components where the system

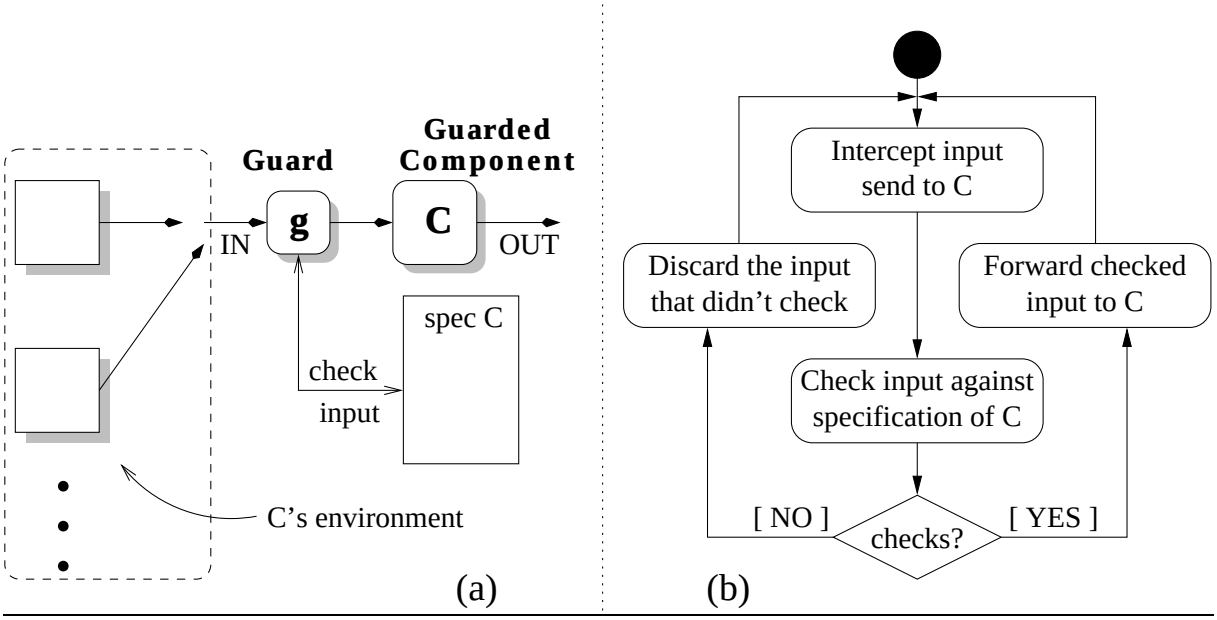

Figure 2: The structure (a) and the activity diagram (b) of the `Input Guard` pattern.

composer does not have access to the internals of the components. Also, this approach can be applied when fault containment comes as a late- or after-thought in the system development and a quick fix is needed in form of a patch. This implementation approach does not require any modification on existing components of a system; rather, *guards* are introduced as separate add-on components to the existing system.

Another implementation approach is to make the *guard* part of the implementation of the *guarded component*. This practice is often employed in programming where a method checks its arguments before using them to perform its designated task. This allows the coupling of the *guard(s)* and the *guarded component*. By integrating the *guard* with the*guarded component* the space overhead of the `Input Guard` implementation is kept low since it does not introduce another component in the system. Coupling the *guard* and *guarded component* implementation is usually applied in the development of COTS software where the developer has no knowledge about the rest of the system in which the component will be integrated. Hence, in order to assure robust functioning of a component, the developer checks the input of the component on every call. The drawback of this implementation approach is the fact that the time overhead is high and fixed. This is because the *guard* is engaged on every call to the *guarded component*, even when the supplied input has already been checked by other fault tolerance means.

A third implementation possibility is to place the *guard* inside each of the components which may provide input to the *guarded component*. This approach allows the integration of the *guard* with other fault tolerance mechanisms (e.g. the *guard* of the `Output Guard` pattern for each component that provides input to the *guarded component*; see Section 4 for more details). Furthermore, this approach allows the elimination of redundant checks for errors which can increase the time and space overhead of fault tolerance solutions in a system. On the other hand, this approach is not applicable to COTS software. Third party developers may not have information about the specification of the other components to which they will feed their output, hence they do not know what conditions

to check in the *guard*. A drawback of this implementation approach is the elevated space overhead; the number of *guards* is not only proportional to the access points of the *guarded component* but also to the number of components that provide input to the *guarded component*. Another drawback is that this *guard* cannot protect the *guarded component* from communication errors that occurred during the forward of the checked input from the *guard* to the *guarded component*. On the positive side however, this approach allows the *guard* to be selectively integrated only with those components that are considered not robust enough and subject to produce erroneous input for the *guarded component*. This can be used to reduce the elevated space overhead of the approach.

3.5 Consequences

The `Input Guard` pattern has the following benefits:

+ It stops the contamination of the *guarded component* from erroneous input that does not conform to the specification of the *guarded component*.

+ The undefined behavior of the *guard* in the presence of errors allows its combination with error detection and error masking patterns, and fault diagnosis mechanisms. Whenever this is applicable, the system benefits in terms of reduced run-time overhead introduced by the implementation of the fault tolerant mechanism (e.g. the combination of fault containment and error detection in the context of system recovery from errors).

+ The similarities between the *guard* entities of the `Input Guard` pattern and `Output Guard` pattern (see Section 4) allow the combination of the two in a single entity. This entity will operate on the same data and will perform two checks: one against the specification of the component that produced the data as output and the other against the specification of the component that will consume the data as input. When applicable, this combination can provide significant benefits in terms of time and space overhead since two separate checks will be performed by the same piece of code.

+ There are various ways that the `Input Guard` pattern can be implemented, each providing different benefits with respect to the time or space overhead introduced by the *guard*. It is also possible to integrate the *guard* with an existing system without having to modify the internals of the system components (first implementation alternative). That reduces significantly the amount of system re-engineering required for applying the `Input Guard` pattern to COTS-based systems made of black-box components.

The `Input Guard` pattern imposes also some liabilities:

- It is not possible to minimize both the time and the space overhead of this pattern. To keep low the time overhead introduced by the `Input Guard` pattern, the functionality of the *guard* must not be very time consuming. This results in a tendency to introduce a separate *guard* for each different access point (e.g. one *guard* per interface or even per operation declared in an interface) of the *guarded component*. Each such guard checks only a small part of the specification of the *guarded component*, minimizing thus the execution time of an individual *guard*. However, this

results in a large number of *guards*, hence in an elevated space overhead. On the other hand, to keep low the space overhead introduced by the `Input Guard` pattern, the number of *guards* needs to remain as small as possible. This implies that each *guard* will have to check a lager number of input for the *guarded component*, becoming a potential bottleneck and thus penalizing the performance of the system with elevated time overhead.

- For certain systems that require *guards* to be implemented as components (e.g. systems composed from black-box COTS software), the `Input Guard` pattern results unavoidably to an elevated time and space overhead. The space overhead is due to the introduction of the new components implementing the *guards*. The time overhead is due to the fact that passing input to the *guarded component* requires one additional indirection through the component implementing the *guard* that check the given input.

- The `Input Guard` pattern cannot prevent the propagation of errors that do conform with the specification of the *guarded component* (e.g. see case (b) in Figure 1). Such errors may contaminate the state of the *guarded component* if it has one. Although these errors cannot cause a failure on the *guarded component* since it operates according to its specification, they can cause a failure on the rest of the system. Such a failure of the entire system will be traced back to an error detected in the contaminated *guarded component*. Unless the error detection and fault diagnosis capabilities of the system allow to continue tracing the error until the initial fault that caused it, it is possible that inappropriate recovery actions will be taken targeted only at the *guarded component*, which, nonetheless, has been operating correctly according to its specification.

- The `Input Guard` pattern can effectively protect a component from being contaminated by erroneous input according to its specification. However, unless it is combined with some error detection and system recovery mechanisms, this pattern will result in a receive-omission failure (i.e. failure to receive input) of the *guarded component*. For certain systems, such a failure of one of their components may cause a failure on the entire system. Hence, the `Input Guard` pattern has limited applicability to such systems if it is not combined with other fault tolerance patterns.

3.6 Known Uses

The *guard* entity can be seen as one possible realization of the pre-conditions validation prior to the execution of a piece of code, as this is described in the design by contract principles [9]. The concept of conditions guarding the execution of tasks has been introduced as *monitors* already in the '70s [5]. Monitors, however, would not prevent the flow of erroneous input to the guarded component; rather, they would prevent the guarded task from being executed if the starting/input conditions were not met. Nowadays, the majority of the books that introduce a programming or scripting language instruct also the validation of input arguments upon the call of a function, procedure, method, routine or whatever the name of the a functionality block in the corresponding language.

3.7 Related Patterns

The `Input Guard` pattern has similarities with the `Output Guard` pattern presented in the following section. Also, the *guard* entity of this pattern complements the *acknowledger* entity of the `Acknowledgment` pattern [15] in combining fault containment and error detection. The *acknowledger* is responsible to inform the sender of some input about the reception of it. The combination of the *acknowledger* and the *guard* will provide a confirmation of the reception of correct input and a notification in the case of reception of erroneous input.

4 Output Guard

Whereas the `Input Guard` pattern prevents an error in the input of the guarded component from contaminating that component, it is often highly desirable stop the error when it occurs rather when it is about to be propagated to another component. The `Output Guard` pattern describes how to confine an error in the component that contains the fault which led to that error. The technique described by the `Output Guard` pattern is very similar to the one described by the `Input Guard` pattern: the output of a component is checked against the specification of the component to ensure conformance. Despite the similarity of the technique, these two patterns serve different purposes. The `Input Guard` pattern prevents the contamination of a component from an error occurred elsewhere while the `Output Guard` pattern confines an error inside the component where that error occurred.

4.1 Context

The `Output Guard` pattern applies to a system that has the following characteristics:

- The system is composed from distinguishable components, which can play the role of fault compartments and which interact with each other by feeding one's output into another's input.
- It is possible, either directly or implicitly, to validate the output of a component against the legitimate output described in the component's specification.
- The errors that occur in a system are expressed as erroneous system output according to its specification, i.e. output whose content or timing does not conform to the system specification.

Similar observations as for the `Input Guard` pattern apply here. The second characteristic implies that internal errors (e.g. changes to the internal state due to electromagnetic disturbances in the environment where the system operates) are not considered by this pattern unless they result in erroneous output according to the system specification. Moreover, this pattern does not deal with cases where the output of the system conforms with the system specification but it still contains errors according to the specification of the system's environment (e.g. see case (a) in Figure 1).

4.2 Problem

In the above context, the `Output Guard` pattern solves the problem of confining an error inside the component where it occurs by balancing the following forces:

- A system may be composed of components that are developed independently from each other, without keeping strict consistency with each others specifications.
- A system may infect with errors its environment when the environment is not able to distinguish the erroneous output from a correct one.
- Different systems have different requirements regarding size impact of the fault containment mechanism.

- Different systems have different requirements regarding the time penalty of the fault containment mechanism.
- Fault containment is usually integrated with other solutions provided for other fault tolerance constituents (e.g. error masking, error detection, fault diagnosis and the others mentioned in Section 1) in order to provide wider fault tolerance guarantees.

4.3 Solution

To stop an error from being propagated outside the component where it occurred, a guard is placed at every exit point of the component (be it message emission or invocation return point). Each such guard checks the produced output against the specification of the component. If and only if the output conforms with the component specification then it is allowed to reach the component's environment.

Notice the above solution does not define the behavior of the guard in the presence of erroneous output, besides the fact that it does not allow it to reach the component's environment. Similarly to the `Input Guard` pattern, this behavior is intentionally left undefined in order to allow implementations of the `Output Guard` pattern to be combined with error detection mechanisms (e.g. when a check fails, a notification is sent to the error detection mechanism [8]). Hence, the behavior of the guard when the checks performed on the output fail depends on the other fault tolerance constituents with which the input guard is combined.

4.4 Structure

Since the technique captured by the `Output Guard` pattern is very similar to the one captured by the `Input Guard` pattern, it does not come as a surprise that the entities the former introduces are similar to those introduced by the latter:

- The *guarded component* which is the part of the system which is guarded against the occurrence of errors and in which occurred errors will be confined.
- The *guard* which is responsible to check for errors the output to the guarded component against its specification.

Similarly to the `Input Guard` pattern, there may be many *guards* for the same guarded component, depending on the system design and on the number of different exit points the *guarded component* may have. For example, a software component with a number of interfaces and a number of operations declared in its of them can have one guard per interface, or one guard per *message-send* and *return* operation, or any possible combination of those. Figure 3a illustrates graphically the structure of the *Output Guard* pattern for a *guarded component* with a single exit point. Figure 3b contains the activity diagram that describes the functionality of the *guard*.

One way to implement the *guards* is as separate components in the system. This approach has the same advantages and inconveniences as its counterpart in the `Input Guard` pattern. It allows to have a number of *guards* proportional only to the number of the access points of the *guarded component*. However, the time overhead is quite high since it includes the invocation of an additional component (i.e. the *guard*). Also, the space overhead of this approach is rather elevated since it increases the number of the

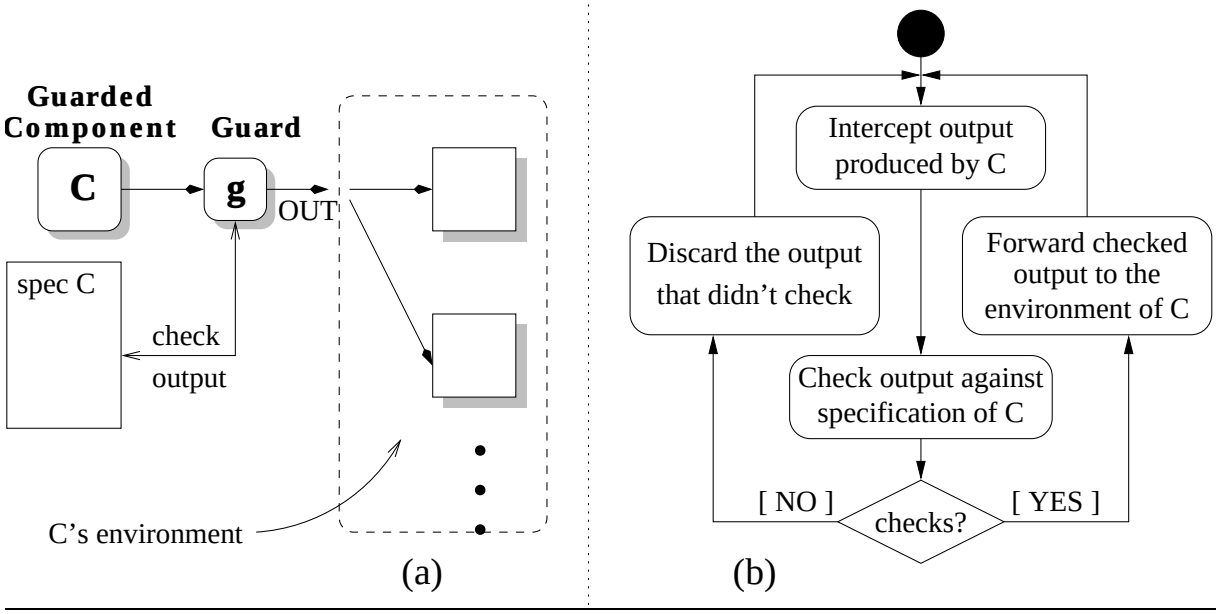

Figure 3: The structure (a) and the activity diagram (b) of the `Output Guard` pattern.

components in a system by the number of *guards* that are implemented. Furthermore, in the case where components are mapped to individual units of failure (i.e. each component can fail as a whole and independently of other components) the problem of "guarding the guards" raises again as for the `Input Guard` pattern. This dilemma has also well-known consequences, which are difficult and very costly to resolve (e.g. redundant instances of the guards, distributed among the constituent components of a system, which have their own synchronization protocol).

Despite the above inconveniences, this implementation approach is valuable in the case of COTS-based systems composed out of black-box components where the system composer does not have access to the internals of the components. Also, this approach can be applied when fault containment comes as a late- or after-thought in the system development and a quick fix is needed in form of a patch. This implementation approach does not require any modification on existing components of a system; rather, *guards* are introduced as separate add-on components to the existing system.

Another implementation approach is to make the *guard* part of the implementation of the *guarded component*. This practice is often employed in programming where a method checks the validity of the output it produced before returning it to its environment (e.g. to its caller). This allows the coupling of the *guard(s)* and the *guarded component*. By integrating the *guard* with the*guarded component* the space overhead of the `Output Guard` implementation is kept low since it does not introduce another component in the system. Coupling the *guard* and *guarded component* implementation is usually applied when developing software components that are meant to upgrade an existing system. In these cases, the developer of the new software component wants to make sure that the produced output will not introduce any errors to the existing system. To archive this, the developer of the component introduces self-checking code for the output the component produces (e.g. see [18] and [11] for more information on self-checking code). The drawback of this implementation approach is the fact that the time overhead is high and fixed. This is because the *guard* is engaged every time the *guarded component* produces output, even

when that output will be anyway checked as whether it is valid input for the component which will receive it (e.g. see the `Input Guard` pattern in Section 3).

A third possibility is to place the *guard* inside each of the components which may receive the output produced by the *guarded component*. This approach allows the integration of the *guard* with other fault tolerance mechanisms (e.g. the *guard* of the `Input Guard` pattern for each component that receives the output produced by the *guarded component*; see Section 3 for more details). Furthermore, this approach allows the elimination of redundant checks for errors which can increase the time and space overhead of fault tolerance solutions in a system. This is the case when the *guard* entities of the `Input Guard` and `Output Guard` patterns are integrated. Then, each *guard* will perform on the *same* data its own checks with respect to different component specification: the former *guard* checks the data against the specification of the component that produces the output and the latter against the specification of the component that receives the input.

On the other hand, this approach is not applicable to COTS software. Third party developers may not have information about the specification of the other components from which they will receive input, hence they do not know what conditions to check in the *guard*. A drawback of this implementation approach is the elevated space overhead; the number of *guards* is not only proportional to the exit points of the *guarded component* but also to the number of components that receive as input the output produced by the *guarded component*. On the positive side however, when this implementation approach is applicable it also protects the environment of the *guarded component* from communication errors that occurred during the forward of the produced output to the *guard*. Attention is required when this approach is combined with en error detection and fault diagnosis mechanism, because it is difficult to deduce the source of the error, i.e. whether it is a communication error or it is due to a fault in the *guarded component*. Another advantage of this approach is the fact that the *guard* can be selectively integrated only with those components which are not robust enough to resist the propagation of errors occurred in the *guarded component*. This can be used to reduce the elevated space overhead of the approach.

4.5 Consequences

The `Output Guard` pattern has the following benefits:

+ It confines an error to the component where it occurred by forwarding to the component's environment only output that conforms to the specification of the component.

+ The undefined behavior of the *guard* in the presence of errors allows its combination with error detection and error masking patterns, and fault diagnosis mechanisms (e.g. see [8]). Whenever this is applicable, the system benefits in terms of reduced run-time overhead introduced by the implementation of the fault tolerant mechanism (e.g. the combination of fault containment and error detection in the context of system recovery from errors).

+ The similarities between the *guard* entities of the `Input Guard` and `Output Guard` patterns allow the combination of the two in a single entity. This entity will operate on the same data and will perform two checks: one against the specification of the component that produced the data as output and the other against the specification of the component that will consume the data as input. When applicable, this

combination can provide significant benefits in terms of time and space overhead since two separate checks will be performed by the same piece of code.

+ There are various ways that the `Output Guard` pattern can be implemented, each providing different benefits with respect to the time or space overhead introduced by the *guard*. It is also possible to integrate the *guard* with an existing system without having to modify the internals of the system components (first implementation alternative). That reduced significantly the amount of system re-engineering required for applying the `Output Guard` pattern to COTS-based systems made of black-box components.

The `Output Guard` pattern imposes also some liabilities, similar to those of the `Input Guard` pattern:

- It is not possible to minimize both the time and the space overhead of this pattern. To keep low the time overhead introduced by the `Output Guard` pattern, the functionality of the *guard* must not be very time consuming. This results in a tendency to introduce a separate *guard* for each different exit point (e.g. one *guard* per invocation-return or per message-send) of the *guarded component*. Each such guard checks only a small part of the specification of the *guarded component*, minimizing thus the execution time of an individual *guard*. However, this results in a large number of *guards*, hence in an elevated space overhead. On the other hand, to keep low the space overhead introduced by the `Output Guard` pattern, the number of *guards* needs to remain as small as possible. This implies that each *guard* will have to check a lager number of output for the *guarded component*, becoming a potential bottleneck and thus penalizing the performance of the system with elevated time overhead.

- For certain systems that require *guards* to be implemented as components (e.g. systems composed from black-box COTS software), the `Output Guard` pattern results unavoidably to an elevated time and space overhead. The space overhead is due to the introduction of the new components implementing the *guards*. The time overhead is due to the fact that passing output from the *guarded component* to its environment requires one additional indirection through the component implementing the *guard* that check the given output.

- The `Output Guard` pattern cannot prevent the propagation of errors that do conform with the specification of the *guarded component* (e.g. see case (a) in Figure 1). Such errors are not due to a malfunction of the *guarded component* and do not affect its internal state. Although these errors do not have their source in the *guarded component* which is checked to produce output according to its specification, they can cause a failure on the rest of the system. Such a failure of the entire system will be traced back to an error detected in the *guarded component*. Unless the error detection and fault diagnosis capabilities of the system allow the detection of faults in the system design, it is highly probable that inappropriate recovery actions will be taken targeted at the *guarded component*, which, nonetheless, has been operating correctly according to its specification.

- The `Output Guard` pattern can effectively protect the component's environment from being contaminated by erroneous output produced by the component according

to its specification. However, unless it is combined with some error detection and system recovery mechanisms, this pattern will result in a send-omission failure (i.e. failure to deliver output) of the *guarded component*. For certain systems, such a failure of one of their components may cause a failure on the entire system. Hence, the `Output Guard` pattern has limited applicability to such systems if it is not combined with other fault tolerance patterns.

4.6 Known Uses

Contrary to the relation of the `Input Guard` with the design by contract principles [9] where the *guard* maps to the pre-conditions, the `Output Guard` pattern has little relation with them. In the design by contract the post-conditions are properties *guaranteed* at the end of a successful process execution, rather than being conditions checked to verify whether the execution was successful or not. Applications of the `Output Guard` pattern are found in *Double Modular Redundant* and *Triple Modular Redundant* systems where the *guard* checks whether the output of two or three replicas of the guarded component is identical and, if not, it does not deliver it to the environment of the guarded system. A special case of application for the `Output Guard` pattern is the use of exceptions in various programming languages (e.g. C++, Java) and distributed computing frameworks (e.g. various implementations of the CORBA specifications). In those cases, the *guard* explicitly knows about output of the guarded component that is exceptional and must be treated by the exception handlers rather than been let through to the environment of the *guarded component*.

4.7 Related Patterns

Besides the obvious similarities with the `Input Guard` pattern described in Section 3, the `Output Guard` pattern relates to the `Fail Stop Processor` (FSP) pattern [15]. In the FSP pattern a *comparator* entity receives the output of two or more identical *processor* entities and compares it. If all output received is identical, then the output is forward to the environment; otherwise the comparator stops delivering any further output to the environment. A combination of the *guard* and *comparator* entities would enable the self-checking code to identify which of the *processors* has failed and notify the error detection and system recovery mechanisms in order for them to take the appropriate actions.

5 Fault Container

The `Fault Container` pattern aggregates the fault containment techniques described by the `Input Guard` and the `Output Guard` patterns. This pattern proposes the use of a wrapper that transforms a software component into its fault containing counterpart.

5.1 Context

The `Fault Container` pattern applies to a system which has the following characteristics:

- The system is composed from distinguishable components, which can play the role of fault compartments and which interact with each other by feeding one's output into another's input.
- It is possible, either directly or implicitly, to validate the input and output of a component against the legitimate input and output described in the component's specification.
- The errors that occur in the system are expressed as erroneous input (i.e. input whose content or timing does not conform to the system specification) or as erroneous output (i.e. output whose content or timing does not conform to the system specification).

The second characteristic implies that the `Fault Container` pattern cannot deal with the types of errors that are not addressed by the `Input Guard` and the `Output Guard` patterns. In other words, this pattern does not deal with internal errors of a component (e.g. changes to the internal state due to electromagnetic disturbances in the environment where the system operates). Moreover, this pattern does not address cases where the input or the output of a component conforms to the component specification but still contains errors according to the specification of the system in which the component operates (e.g. errors due to design faults). For example, the `Fault Container` pattern cannot provide fault containment when applied either to $GETIN$ or to $T2N$ components in Figure 1(c).

5.2 Problem

In the above context, the `Fault Container` pattern solves the problem of prohibiting the propagation of an error both from inside a component to the rest of the system and from the component's environment into the component itself, by balancing the following forces:

- A system may be composed of components that are developed independently from each other, without keeping strict consistency with each others specifications.
- A system may infect with errors its environment or be infected with errors from its environment despite the fact that the exchanged data that propagate the error not be perceived as erroneous by the system's environment.
- Different systems have different requirements regarding size impact of the fault containment mechanism.

- Different systems have different requirements regarding the time penalty of the fault containment mechanism.
- Fault containment is usually integrated with other solutions provided for other fault tolerance constituents (e.g. error masking, error detection, fault diagnosis and the others mentioned in Section 1) in order to provide wider fault tolerance guarantees.
- The system to which the `Fault Container` pattern is applied must be indistinguishable from the mechanism that implements the `Fault Container` pattern, i.e. it must not be possible to address or access system without addressing or accessing that mechanism at the same time.

5.3 Solution

As stated by its name, the `Fault Container` pattern provides a container that embraces the system on which the pattern is applied. This container forms the fence that stops the propagation of errors in both ways, from inside the container to the outside and vice versa. Whenever the system receives input from its environment or produces output to be delivered to its environment, the container checks it against the specification of the system. If and only if the input (or the output) confirms with the system specification then it is allowed to reach the system (or reach the system's environment respectively).

A similarity of this pattern with the other two patterns presented previously in this paper is the fact that the behavior of the container is not defined in the presence of an error, besides the fact that it does not allow it to enter or leave the system. This behavior is intentionally left undefined in order to allow implementations of the `Fault Container` pattern to be combined with error detection, fault diagnosis and system repair mechanisms. It is worth mentioning that in practice there has not been any pure implementation of the `Fault Container` pattern as described here in the fault tolerance literature. Rather, very often the functionality of this pattern is combined with other functionalities. For example, the container functionality is combined with error detection and fault repair through reflection functionalities to result in a mechanism that provides fault containment and repair for the Chorus OS [14]. In another case, the container functionality has been combined with the functionality described by the `Adapter` pattern [2] in order to result in a fault containing Hive cell made out of four (or more) modified Unix kernels for shared memory multiprocessor architectures [12].

Strictly speaking, from the fault containment perspective the `Fault Container` pattern provides the same benefits as the combination of the `Input Guard` and the `Output Guard` patterns, i.e. it prevents an error from being propagated inside and outside a given component. From a conceptual perspective however, this pattern contains more than the mere aggregation of the guarding of the access and exit points of a component. It also contains the potential of coordinating these two functionalities, i.e. the possibility to correlate an error in the output of the component to the input that, when processed by the component, led to the erroneous output. This additional functionality becomes visible when combining the `Fault Container` pattern with some fault diagnosis, error detection, fault repair, or system recovery mechanism. In practical terms, an implementation of the `Fault Container` pattern will provide to the environment of the contained component an interface similar to the aggregation of the interfaces provided by implementations of the `Input Guard` and `Output Guard` patterns. However, behind this interface and in ad-

dition to the functionality of the guards corresponding to the aforementioned patterns, the functionality of their coordination may be provided.

5.4 Structure

The `Fault Container` pattern introduces two entities, similar to those of the `Input Guard` and `Output Guard` patterns:

- The *contained component*, which is the part of the system that is shielded from getting contaminated by errors propagated from its environment through its input and from contaminating its environment with errors propagated through its output.
- The *container*, which is responsible to check against the *contained component* specification for errors in the input the component receives and in the output the component produces. In addition, the *container* may correlate the output checks with the checks of the input that led to that output in order to derive causal relations between input and erroneous output.

In practice, the *container* is a new component in the system which embeds the *contained component* and implements also the error fencing functionality. That makes the *container* and the *contained component* a single addressable constituent in the system composition. Figure 4a illustrates graphically the structure of the *Fault Container* pattern for a *contained component* with a single access and a single exit point. Figure 4b contains the activity diagram that describes the functionality of the *container*.

The implementation approach of the `Fault Container` pattern that is most widely used is the tight coupling of the *container* with the *contained component*. This approach is a straightforward application of the solution described above and it has appeared in two variants. In the first variant, usually applied to custom-made software, the *container* functionality is integrated with the *contained component*. Every time the latter receives some input or is about to deliver some output, the *container* functionality is engaged to check that input or output against the *contained component* specification. In the second variant, usually applied to systems composed from black-box COTS software, the *container* is a new component that embeds the *contained component*. Every input send to the *contained component* and every output produced by it are intercepted by the *container*, which checks them against the specification of the *contained component*. Both variants of this implementation approach are applicable in many cases of software system development, ranging the composition of the system from custom-made software component to the use of black-box COTS software. As long as the specification of the *contained component* is available, this implementation approach can be applied to transform a software component into its fault containing counterpart. The main characteristic of this approach is that the *container* introduces a fixed time and space overhead. In cases where the input and output of a component is already checked by other means, the error fencing functionality provided by the *container* is redundant. Consequently the system performance is unnecessarily penalized in terms on time and space overhead.

Another implementation alternative is to integrate the *container* with the environment of the *contained component*. This results in redundant instances of the *container* inside the components interacting with the *contained component*, which introduce a higher space overhead to the system compared to the first implementation alternative. Moreover, with this approach the *contained component* is entirely shielded from its environment but only

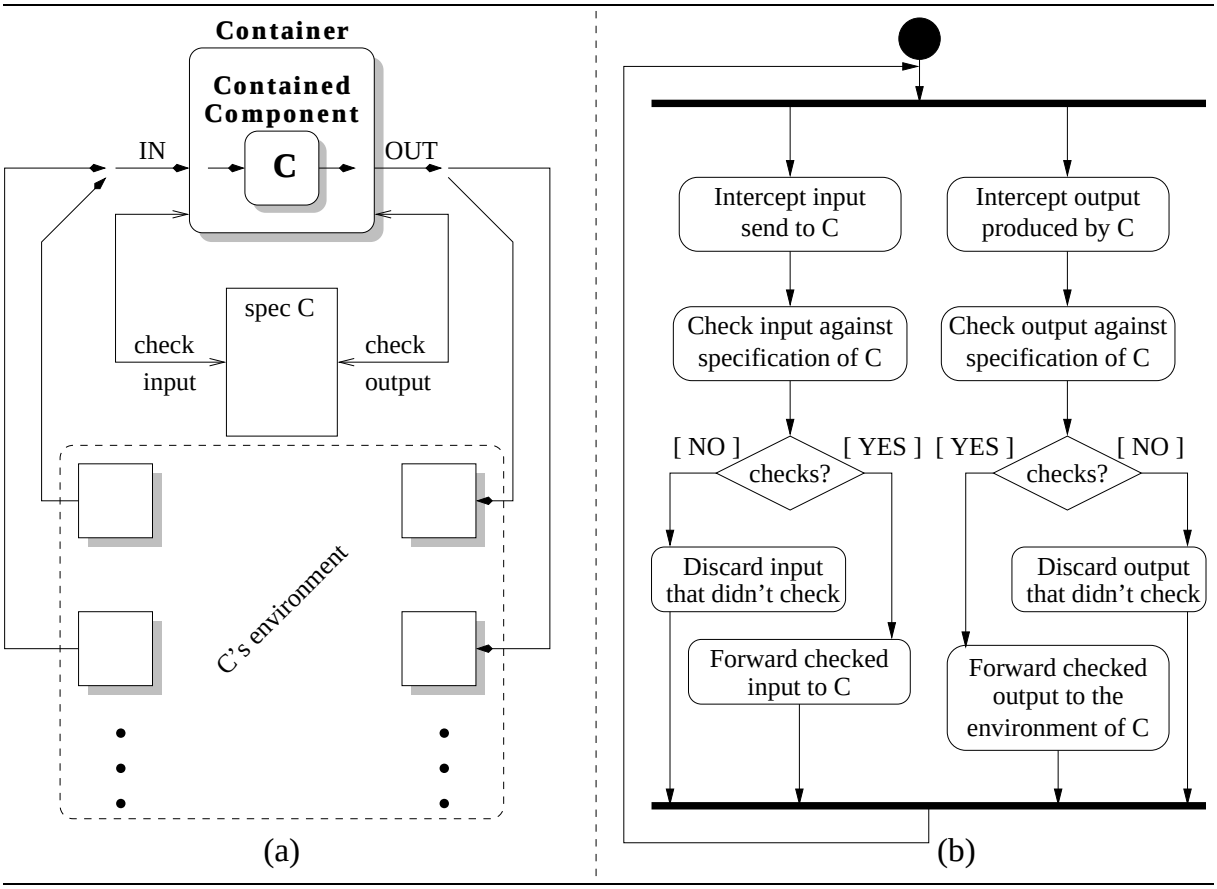

Figure 4: The structure (a) and the activity diagram (b) of the `Fault Container` pattern.

shielded from those parts of its environment that contain the *container*. At the same time however, this approach allows to integrate the *container* only with selected component from the environment of the *contained component*. Consequently, the system developer has better possibilities for tuning the time overhead introduced by the `Fault Container` pattern since the *container* functionality will not be engaged in every single interaction of the *contained component* with its environment.

This approach has an important drawback: it allows the *contained component* to be addressed separately from the *container*. This contradicts the second force of this pattern which requires the *contained component* to be indistinguishable from its *container* and results in a partially wrapped component. Nevertheless, this approach can be favored in practice when performance issues are of primary importance. The choice and the responsibility of partially wrapping a component in order to gain performance benefits lies entirely with the system designer. Another drawback of this approach is the additional communication overhead that is introduced by the coordination messages that need to be exchanged among the distributed instances of the *container* in order to deduce the causal relations between input and erroneous output of the *contained component*. Some of this overhead can be amortized by integrating these messages with other coordination messages for fault tolerance specific purposes (e.g. with acknowledge messages sent for error detection purposes).

In theory, a third implementation alternative is to place the *container* in a separate component which acts as proxy for the *contained component*. However, this approach is

very similar to the second variant of the first implementation alternative, and there are no application cases where the first approach does not apply and this one does. On the other hand, the implementation of the *container* as a separate component bears two drawbacks: it has an elevated space overhead due to the additional component implementing the *container*, and it also has an elevated time overhead due to the indirection through the component implementing the *container* of every interaction of the *contained component* with its environment. For these reasons, this implementation approach has very low practical interest.

5.5 Consequences

The `Fault Container` pattern has the following benefits:

+ It stops of errors expressed as input and output content or timing that does not conform to a component specification from entering or exiting that component.

+ The undefined behavior of the *container* in the presence of errors allows its combination with error detection and error masking patterns (e.g. see [14] and [12]). Whenever this is applicable, the system benefits in terms of reduced run-time overhead introduced by the implementation of the fault tolerant mechanism (e.g. the combination of fault containment and error detection in the context of system recovery from errors).

+ The eminent similarities of this pattern with the *Adapter* pattern [2] allow the straightforward combination of the two patterns in the same implementation. Employing the `Adapter` pattern to adjust the interface of a component that is otherwise incompatible with the interface its environment expects from it may introduce faults in the system. The `Fault Container` pattern compliments the `Adapter` by ensuring that the adaptation of the component interface to the interface expected by its environment does not cause erroneous input and output to propagate from the component to its environment and vise versa.

+ The different implementation alternatives allow the system developer to chose the aptest way to apply the `Fault Container` pattern in a given system. By embedding the *contained component* inside the *container* the former is completely shielded from its environment for a fixed space and time overhead. When appropriate however, the system developer may chose to integrate the *container* with selected parts of the *contained component*'s environment. This decreases the time overhead introduced by the `Fault Container` pattern for the price of a higher space overhead (redundant instances of the *container*) and for lower shielding guarantees since some selected interactions of the *contained component* will not be checked by the *container*.

The `Fault Container` pattern imposes also some liabilities:

- It is not possible to minimize both the time and the space overhead of this pattern. To keep low the time overhead introduced by the `Fault Container` pattern, the functionality of the *container* can be integrated with selected components in the environment of the *contained component*. In addition to reducing the fault containment guarantees only to those selected parts of the system, this approach increases also the space overhead because of the redundant instances of the *container*. On the

other hand, to keep low the space overhead introduced by the `Fault Container` pattern one *container* can be used, which will embed the *contained component*. Then, every interaction between the *contained component* and its environment will have to go through the *container*, causing an elevated and fixed time penalty for the system execution.

- For certain systems that require the *container* to embed the *contained component* (e.g. systems composed from black-box COTS software), the `Fault Container` pattern results to an elevated implementation overhead. This is because the system developer has to implement the code necessary for embedding one component into another, in addition to the implementation of the *container* functionality.

- The `Fault Container` pattern cannot prevent the propagation of errors that do conform with the specification of the *contained component* as is illustrated in Figure 1. Such errors are not due to a malfunction of the *contained component*. Despite the fact that these errors do not have their source in the *contained component* which is checked to receive input and to produce output according to its specification, they can cause a failure on the rest of the system. Such a failure of the entire system will be traced back to an error detected in the *contained component*. As a result recovery actions targeted at the *guarded component* will probably be taken. However, such recovery actions do not deal with the source of the problem, which is the fault that caused the initial error (e.g. a design fault or an error in the input of the *guarded component*). To allow effective system recovery, sophisticated (and thus space and time consuming) error detection and fault diagnosis techniques must be employed which will allow the error tracing to be continued through the *guarded component* until the real source of the propagated error is revealed.

- The `Fault Container` pattern can effectively protect a component from being contaminated by erroneous input according to its specification. It also prevents the component from delivering to its environment erroneous output according to the component specification. However, unless it is combined with some error detection and system recovery mechanisms, this pattern will result in send- or receive-omission failures (i.e. failure to send output or receive input) of the *contained component*. For certain systems, such a failure of one of their components may cause a failure on the entire system. Hence, the `Fault Container` pattern has limited applicability to such systems if it is not combined with other fault tolerance patterns.

5.6 Known Uses

The `Fault Container` pattern has been applied in practice in the implementation of fault containing Hive cell made out of four (or more) modified Unix kernels for shared memory multiprocessor architectures [12]. Other cases where the `Fault Container` pattern has been used to shield the propagation of errors to and from an operating system kernel are the cases of fault-tolerant Mach [13] and the Chorus kernel [14].

5.7 Related Patterns

The `Fault Container` pattern has obvious similarities with both the `Input Guard` and `Output Guard` patterns presented in Sections 3 and 4 respectively. Also, the `Fault`

`Container` pattern complements the `Adapter` pattern [2] with fault containing properties. Finally, this pattern can be seen as the customization of the `Proxy` pattern [2] to meet the fault containment requirements of a system.

6 Summary

This paper has presented three design patterns for fault containment which are inspired from a variety of applications domains: system specification (e.g. design by contract [9]), self-stabilization [17], OS kernels (e.g. fault tolerant Mach [13] and Chorus [14]) and shared memory multiprocessor architectures (e.g. the Hive system for the FLASH multiprocessor [12]). This work is a complement of our previous work on design pattern for fault tolerance [15] and it can be used to extend the classification of the pattern system presented there.

The variety of implementation approaches that can be employed when the presented patterns are applied make them suitable for different cases ranging from systems built from custom-made software components to systems composed from black-box COTS software. For example, the `Input Guard` pattern is commonly used when integrating a new component with an existing system whose specification is not available or not very clear. The `Output Guard` pattern is usually applied on components liable to produce errors that may contaminate the rest of the system (e.g. third party software). Finally, the `Fault Container` pattern is especially useful for wrapping critical components for the overall functionality of the system and to prevent errors from propagating into and outside from those components.

When employing these patterns, the system developer must pay attention to certain issues that play a major role in the overall performance of the system. The first issue is the tradeoff between the time and space overhead introduced by these patterns: optimizing one overhead results to a negative impact on the other. The system developer must make a design decision on this issue keeping in mind the consequences this decision bears for the system performance.

Another important issue is the risk of introducing redundant software that checks the same conditions. For example, this may happen if the system developer integrates the `Input Guard` with the environment of a third party component and that component is eventually delivered having the `Input Guard` integrated in itself. That results in a useless duplication of the *guard* functionality which penalized both the time and the space performance of the system.

Last, but certainly not least, the system developer must face and resolve the well-known dilemma of fault tolerance regarding the responsibility of "guarding the guards". In common practice this issue is dealt with in the following way. The failure model of the system maps the units of failure to the system's constituent components. Then, the *guard* and *container* entities of the presented patterns are integrated with some of these components. It follows then that the fault containment specific entities may fail only when some component of the system fails. Thus their failures can be monitored by the error detection mechanism that monitors the component into which they are integrated and their recovery becomes part of the recovery of the failed component. However, some implementation alternatives map the *guard* and *container* entities to separate components. If the aforementioned failure model is applied again then the error detection mechanism of the system must be extended to monitor also these new components. Moreover, the error masking, fault repair and system recovery mechanisms must be also applied to these new components. This is both a tedious implementation task as well as time and space consuming since it requires additional acknowledge messages, replica groups, checkpoints actions, etc.

Acknowledgments

The author would like to thank his shepherd, Michael Kircher, for his insightful comments, his time and his patience through the revisions of this paper. Also, the paper owes its final form to the constructive criticism and the suggestions of the participants of Workshop C in EuroPLoP 2003: Eduardo Fernandez, Aalder Hofman, Duane Hybertson, Ewald Kaluscha, Andy Longshaw, Markus Schumacher, and Peter Sommerlad.

References

[1] A. Arora and M. Gouda. Closure and Convergence: A foundation for Fault-Tolerant Computing. *IEEE Transations on Software Engineering*, 19(11):1015–1027, November 1993.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, December 1994.

[3] S. Ghosh, A. Gupta, T. Herman, and S.V. Pemmaraju. Fault-Containing Self-Stabilizing Algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, May 1996.

[4] S. Ghosh, A. Gupta, and S.V. Pemmaraju. Fault-Containing Network Protocols. In *Proceedings of the ACM Symposium on Applied Computing*, pages 431–437, April 1997.

[5] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[6] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems . In *Proceedings of Fall Joint Computer Conference*, pages 1150–1158, November 1986.

[7] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1992.

[8] N.G. Leveson, S.S. Cha, J.C Knight, and T.J. Shimeall. The User of Self Checks and Voting in Software Error Detection: an Empirical Study. *IEEE Transactions on Software Engineering*, 16(4):432–443, April 1990.

[9] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International, 1988.

[10] V.P. Nelson. Fault-Tolerant Computing: Fundamental Concepts. *IEEE Computer*, 23(7):19–25, July 1990.

[11] D.S. Rosenblum. Towards a Method of Programming with Assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.

[12] M. Rosenblum, J. Chapin, D. Teodosiu, S. Devine, T. Lahiri, and A. Gupta. Implementing Efficient Fault Containment for Multiprocessors. *Communications of the ACM*, 39(9):52–61, September 1996.

[13] M. Russinovich, Z. Segall, and D. Siewiorek. Application Transparent Fault Management in Fault Tolerant Mach. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 10–19, June 1993.

[14] F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat. MetaKernels and Fault Containment Wrappers. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 22–29, June 1999.

[15] T. Saridakis. A System of Patterns for Fault Tolerance. In *Proceedings of the 7th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 535–582, June 2002.

[16] F.B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[17] M. Schneider. Self-Stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[18] S.S. Yau and R.C. Cheung. Design of Self-Checking Software. In *Proceedings of the International Conference on Reliable Software*, pages 450–455, April 1975.