Brito PHS, de Lemos R, Rubira CMF *et al.* Architecting fault tolerance with exception handling: Verification and validation. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 24(2): 212–237 Mar. 2009

Architecting Fault Tolerance with Exception Handling: Verification and Validation

Patrick H. S. Brito[1], Rogério de Lemos[2], Cecília M. F. Rubira[1], and Eliane Martins[1]

[1] *Institute of Computing, State University of Campinas, Brazil*

[2] *Computing Laboratory, University of Kent, U.K.*

E-mail: {pbrito, cmrubira, eliane}@ic.unicamp.br; r.delemos@kent.ac.uk

Received March 11, 2008; revised January 28, 2009.

Abstract When building dependable systems by integrating untrusted software components that were not originally designed to interact with each other, it is likely the occurrence of architectural mismatches related to assumptions in their failure behaviour. These mismatches, if not prevented during system design, have to be tolerated during runtime. This paper presents an architectural abstraction based on exception handling for structuring fault-tolerant software systems. This abstraction comprises several components and connectors that promote an existing untrusted software element into an idealised fault-tolerant architectural element. Moreover, it is considered in the context of a rigorous software development approach based on formal methods for representing the structure and behaviour of the software architecture. The proposed approach relies on a formal specification and verification for analysing exception propagation, and verifying important dependability properties, such as deadlock freedom, and scenarios of architectural reconfiguration. The formal models are automatically generated using model transformation from UML diagrams: component diagram representing the system structure, and sequence diagrams representing the system behaviour. Finally, the formal models are also used for generating unit and integration test cases that are used for assessing the correctness of the source code. The feasibility of the proposed architectural approach was evaluated on an embedded critical case study.

Keywords exception handling, fault-tolerant software architecture, model-based test, model checking, software verification and validation

1 Introduction

The adoption of software components, which used to be restricted to the construction of enterprise systems, has expanded to other application areas where the cost of failure might be unacceptable. Software systems that can cause risks for human lives or great financial losses can be made fault-tolerant, so that they are capable of providing their intended operations, even if only partially, despite the presence of faults. Amongst the several existing techniques for building fault-tolerant systems, exception handling is a well-known mechanism for structuring error recovery in software systems[1]. Exception handling complements other techniques for error recovery, such as atomic transactions[2], and aims to support the construction of programs that are more reliable, concise, and easy to evolve[3]. The use of exception handling to develop large-scale software systems[4,5], together with the fact that it is implemented by several modern object-oriented languages, such as, Java, Ada, C#, and C++, and component models, such as, CCM, EJB, Ice, and .NET, confirms its importance to the current practice of software development. Furthermore, in applications where a rollback is not possible, such as those that interact with physical environments, exception handling may be the only choice available. On the other hand, it is also accepted that the exception handling mechanism might have its disadvantages, if we consider the fact that a large part of a system's code is devoted to error detection and handling[1,5].

To cope with the inherent complexity of the exception handling mechanism, it has been claimed that the abnormal behaviour should be systematically incorporated as early as possible in the software development process, especially during the requirements engineering and the architectural design[6]. Software architectures explicitly represent the structure of systems, and it is one of the earliest artefacts that permits the analysis of system quality attributes, such as, dependability, including reliability, availability, security, and

Regular Paper

Patrick Brito is supported by Fapesp/Brazil under Grant No. 06/02116-2 and CAPES/Brazil under Grant No. 0722-07-3. Cecília Rubira is partially supported by CNPq/Brazil under Grant Nos. 301446/2006-7 and 484138/2006-5.

safety[7]. Fault tolerance at the architectural level has received considerable attention, mostly in the context of fault handling. In particular, issues related to architectural reconfiguration that includes replacing, adding, removing architectural elements, or changing the topology of the configuration[8]. The same cannot be said about error propagation and error handling at the architectural level. However, recent work has looked into architectural abstractions as a means for structuring fault-tolerant software systems based on exception handling[9,10]. If the abnormal behaviour is considered since the beginning of the software development process, other issues have to be considered during the architectural design phase, such as the existence of architectural mismatches involving different types of exceptions, the existence of useless exceptions, the association of proper handlers for the exceptions and the activation of the exception handlers when such exceptions are raised.

In a complementary way, in order to identify and remove faults related to the system's abnormal behaviour, verification and testing techniques should be used during the architectural design and implementation. Few contributions have exploited the verification of exception handling abnormal behaviour at the architectural level[11,12]. For instance, the work by Castor *et al.* proposes a solution for specifying and verifying exception control flow at the software architecture using the Alloy specification language. However, their approach does not consider the behaviour of exception handlers as part of the verification process. Also, this solution does not scale very well when the verification process has to deal with many different types of exceptions[11]. Regarding the testing activities, the verified software architecture can be used for generating model-based test cases in order to assess the system's implementation against the desired properties of the software architecture. Test data such as the generation of test oracles are not discussed in this paper.

In this paper, as a means to overcome some of the limitations mentioned above, we propose the use of architectural abstractions and scenarios. A major advantage of this approach is that it scopes the model to be verified, which, consequently, reduces the state-space, thus improving the overall scalability. In the proposed rigorous architectural approach, architectural elements are modelled in a high-level way by instantiating an architectural abstraction based on exception handling: the idealised fault-tolerant architectural element (iFTE). The role of the iFTE in this context is to abstract away from mechanisms for detecting and handling errors in order to minimise their impact on the overall system complexity[10]. Based on the architectural abstraction, fault-tolerant software architectures can be described using stereotyped UML2.0, which can then be used as a basis for automatically generating the formal specification of the software architecture. This formal model allows the formal verification of error handling properties, and the automatic generation of test cases for assessing the correctness of iFTE-based software architectures, thus improving the system dependability. Regarding the formal representation, although Architecture Description Languages (ADLs) are have the specific purpose of formally representing software architectures, these languages usually lack on support for representing specific aspects of the system. Regarding exception handling, which is essential in the context of our approach, the existing ADLs neither provide support for verifying the exception control flow involving architectural elements, nor general properties related to scenarios of exception handling. Moreover, for generating test cases from the formal specification it is necessary to represent the interfaces of the architectural elements with information about their respective operations. This kind of detailed information is not usually represented by ADLs. To overcome these limitations, it is necessary to use a formal notation that allows the representation of types in an explicit way, in order to distinguish different exceptions. Moreover, for representing the chaining of exception control flow, conversion and masking, the formal notation should also support the specification of scenarios involving architectural elements. Using B-Method[13], which permits the specification of different types through mathematical sets, architectural elements, interfaces and exception types are explicitly represented. Architectural scenarios are modelled using CSP[14], which is a process algebra indicated to represent sequence of events. The model checker used in the proposed solution is ProB[15], which permits the combined use of B-Method machines and CSP specifications in a complementary way.

Although this paper deals with verification and testing based on architectural properties, it does not detail how the exceptional behaviour can be systematically specified. This complementary activity has been addressed in previous work[6,16,17]. In the context of high confidence systems, the essential contributions of the paper are in the specification, verification and testing of fault-tolerant software architectures. The aim of the proposed approach is the provision of assurances that the exception handling based approach being proposed can be used as a basis for the design and implementation of fault-tolerant system. For that, we have, first, formally specified and verified an

architectural abstraction based on exception handling for the provision of fault tolerance, second, defined an integrated approach for the specification and verification of software architectures that are based on this architectural abstraction, including the analysis of exception propagation and handling at the architectural level, and finally, defined an approach for testing the final system against its architectural specification. For the formal specification, we have defined properties related to the signalling, propagation and handling of exceptions. Finally, unit and integration test cases are generated for assessing the implementation of the software architecture.

The rest of this paper is organised as follows. Section 2 presents some background regarding the concepts considered in this paper. Section 3 contextualises the proposed approach with some related work. Section 4 presents the iFTE abstraction. Section 5 describes the rigorous development approach proposed for developing dependable component-based systems. Section 6 describes the case study that will be used throughout the paper for exemplifying and evaluating our contribution. Details about the formal verification and testing of iFTE-based architectural elements, as well as the respective formal specification, are presented in Section 7. Section 8 describes how we systematise the verification and testing of the exception flow in a software architecture. Section 9 evaluates the feasibility of the overall approach through a critical real-time application. Finally, Section 10 provides some concluding remarks and future directions of research.

2 Background

2.1 Software Fault Tolerance

Fault tolerance is the ability of a system to continue its normal operation despite the presence of faults. Software systems that can cause risks for human lives or great financial losses should be made fault-tolerant since they are considered to be critical. Because fault tolerance has a global system scope, it should be related to both architectural elements (components and connectors) and architectural configurations which implement the rules by which they interact.

The provision of software fault tolerance relies on the existence of redundancy, which can be incorporated either implicitly or explicitly at the architectural level. Implicit redundancy refers to extra design elements associated to architectural elements, which are activated for handling internal errors. An example of implicit redundancy is the usage of exception handling for supporting error recovery. If special care is not taken when structuring the system, the normal and abnormal specifications can be entangled thus increasing system complexity. Explicit redundancy is an inherent aspect of strongly-structured systems[18] and refers to extra architectural elements, usually activated to replace failed elements. Examples of explicit redundancy are N-version programming and N-self-checking programming, which are two software fault tolerance techniques[19].

Although the techniques of explicit redundancy deal with software fault tolerance, the implementation of these techniques can increase the system cost in a considerable way. As an alternative, techniques of implicit redundancy can be used. In techniques of implicit redundancy, the redundant code is responsible to implement error recovery, which is activated after the detection of an erroneous state. One of the most used ways for implementing implicit redundancy strategy is through the use of the exception handling mechanism of programming languages.

2.2 Idealised Fault-Tolerant Component

The idealised fault-tolerant component (iFTC)[20] is a structuring concept based on exception handling that makes it possible that normal and abnormal behaviour can be kept separate. Fig.1 presents the structure of an iFTC. The *normal activity* corresponds to those situations where service is provided as specified, while the *abnormal* (*or exceptional*) *activity* corresponds to those situations where errors are detected, and the component cannot provide the requested service. Exceptions can be classified into two different categories: *internal*, when they are raised by a component in order to invoke its own error recovery measures, and *external*, when they are signalled to inform that, for some reason, the component cannot provide the requested service. External exceptions can be partitioned into *interface*

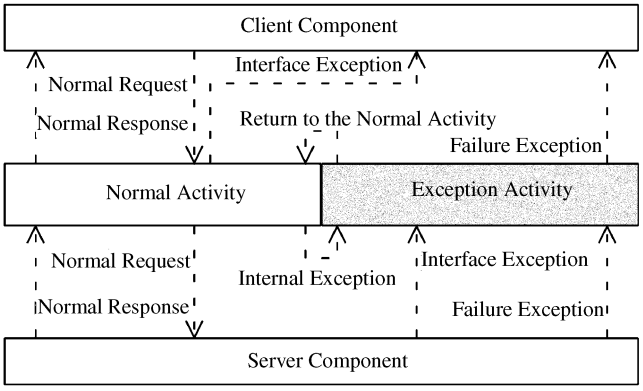

Fig.1. Idealised fault-tolerant component (iFTC).

exceptions, which are due to an invalid service request, and *failure exceptions*, which are due to a failure in the processing of a valid request. In this sense, exceptions and exception handling provide a suitable framework for structuring the fault tolerance activities into a system. iFTCs may be organized into layers, so that components may handle exceptions raised by components located in other layers.

2.3 Formal Notation and Verification

For overcoming the ADLs' limitations related to exception handling and discussed in Section 1, it is necessary to use a formal language that makes it possible to represent types in an explicit way, in order to distinguish different exceptions. Moreover, for representing the chaining of exception propagation, conversion and masking, the formal notation should also support the specification of scenarios involving the architectural elements.

B-Method is a general-purpose formal language based on set theory for specifying and verifying system models with explicit representation of the state, and a modular representation through the concept of refinement[13]. *Refinement* allows us to build a model gradually by making it more precise. The modularisation of the development facilitates the design and improves the scalability of the verification because it is conducted incrementally. Once the refinement is formally guaranteed, the properties verified at the abstract level are reused at the refined level, hence do not need to be re-verified.

A limitation of the B-Method is its inability to easily restrict the correct order for executing operations. Communicating Sequential Process (CSP)[14] is a process algebra that allows an easy representation of execution sequences, and if combined with B-Method, it compensates the aforementioned limitation[15]. As a combined solution, ProB[15] is a model checker that uses B-Method and CSP in a complementary way. In ProB, a CSP specification can be used to restrict the sequence of B-Method operations that are executed.

3 Related Work

In this section we review selected publications related to the three main topics covered in this paper: structuring of system abnormal behaviour, verification of system abnormal behaviour, and architecture-based testing.

The idealised C2 component (iC2C)[9] is another structuring technique based on the idealised fault-tolerant component (Subsection 2.2), which focus on software systems compliant with the C2 architectural style[21]. The internal protocol followed by the internal elements of an iC2C enforces error confinement and makes it possible to define multiple exception handling contexts at the architectural level. Later work by Castor *et al.*[22] defined and implemented an architectural level exception handling mechanism based on the concept of iC2C. The abstraction presented in this paper can be seen as an extension of the iC2C for a broader class of software architectures that adhere to the peer-to-peer architectural style[23].

Simons and Stafford[24] presented a framework, called CMEH, for specifying dependable software architectures based on Commercial Of The Shelf (COTS) components. CMEH allows the contextualisation of exception types, as well as the association of exception handlers with the architectural connectors. This is complementary to ours, since it focuses on implementation, instead of verification and testing. A semantic difference between this work and ours is that CMEH does not consider the exception control flow at the software architecture, but only architectural exception handlers.

Regarding the system verification, several contributions have proposed static evaluation of source code for analysing exception flow[25,26]. Exception flow analysis consists of identifying propagation paths in a program, for example, to identify uncaught exceptions in languages with polymorphic types, such as, Java. Recent work by Castor *et al.*[11], within the Aereal framework, leverages existing languages and tools to support the description and analysis of exception flow in software architectures. Concerning the verification of the architectural exceptions that work is similar to ours; however, there are some important differences. Our work makes it possible the specification of properties based on scenarios related to the software architecture. Moreover, in the common verification also conducted by Aereal, our approach is shown to be more scalable for the purpose of model checking. Finally, besides the verification activities, our work also considers the generation of test cases for assessing the implementation of the software architecture.

Approaches that generate model-based test cases from architectural formal specifications have been previously proposed[28–30]. An example of such an approach specifies integration test cases using a graph that represents system behaviour[30]. Starting from a formal description of the software architecture, the solution is based on the derivation of a graph that represents the system behaviour in terms of the interactions between the components. Next, the solution guarantees that all the graph edges are covered in order to identify test cases. The main limitation of this approach is the

high number of test cases that are generated. Moreover, beyond the generation of test cases, it is also necessary to specify the order in which they should be executed for reducing the costs of system integration. There are a couple of contributions that improve the above mentioned integration testing approach[28,29], and in one of them, the authors adopt a strategy for identifying a suitable set of reduced graphs, focusing on specific architectural properties. This approach is similar to ours since we have used CSP specifications of the scenarios related to the software architecture for constructing graphs that contain a reduced number of vertices. In addition to that, our approach determines the best order for executing test cases by analysing the dependencies amongst the architectural elements. There are other differences between these approaches and ours, mainly related to the fact that we focus on architectural fault tolerance, i.e., we are looking into exception mismatches in the context of a software architecture that is based on a fault-tolerant architectural abstraction.

4 iFTE: Idealised Fault-Tolerant Architectural Element

The idealised fault-tolerant architectural element (iFTE) is an architectural abstraction for structuring fault-tolerant systems. This abstraction enforces the principles associated with the concept of the idealised fault-tolerant component presented in Subsection 2.2, and incorporates mechanisms for detecting errors, as well as handling and propagating exceptions in a structured way. The iFTE abstraction can be used for both components and connectors. The only difference is the role that each element develops in the software architecture. While components are considered the places of computation, connectors are the places of communication, coordinating the interaction between components.

In order to provide a clear separation of concerns between the normal and abnormal behaviour, the iFTE defines four types of interfaces, which are presented in Fig.2: (i) the Provided Normal interface (I_iFTE_PN) defines an access point for the (fault-tolerant) operations provided by the iFTE, without requesting external operations; (ii) the Provided Abnormal interface (I_iFTE_PA) defines an access point where iFTE signals its external exceptions; (iii) the Required Normal interface (I_iFTE_RN) specifies operations required by the iFTE for implementing its normal behaviour or handling exceptions; and (iv) the Required Abnormal interface (I_iFTE_RA) specifies the external exceptions that the iFTE is able to handle. In other words, while I_iFTE_PN and I_iFTE_RN are responsible for the normal behaviour, I_iFTE_PA and I_iFTE_RA are responsible for the abnormal behaviour.

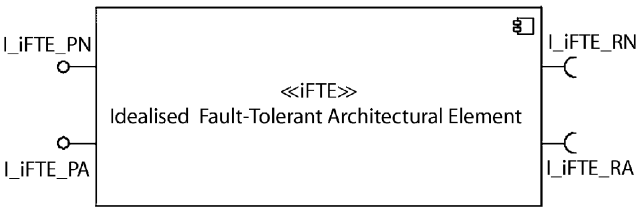

Fig.2. Idealised fault-tolerant architectural element (iFTE).

The operations associated with the provided interface have a set of exceptions, which are known as provided exceptions. The operations of the required interfaces have a set of exceptions that can be caught by the iFTE, which are known as required exceptions. The exceptions signalled by an iFTE can either be internally raised, or propagated as a consequence of an exception caught from a required operation. A raised exception can either be an interface exception when an operation is wrongly requested, or a failure exception as a consequence of an internal iFTE error. Finally, for each provided operation, we may associate a set of required operations that can be invoked as part of its execution.

4.1 Relations Between Interfaces

As presented in Fig.3, the internal behaviour of the iFTE can be described in terms of ten relations among the four types of interfaces. These internal relations explicitly represent the existing relations between the interfaces of the iFTE. Although those relations are not behavioural scenarios, they can be combined in order to compose them. The first two letters of the relation's name means its domain (origin) and the following two letters means its image (destination): pn for a provided normal interface, pa for a provided abnormal interface, rn for a required normal interface, and ra for a required abnormal interface. The ten internal relations of the iFTE are: (i) pnpn, which represents the normal response of a provided operation without requesting external operations; (ii) pnpai, which represents the signalling of interface exceptions; (iii) pnpaf, which represents the signalling of failure exceptions without requesting external operations; (iv) pnrn, which represents the request of external operations; (v) rnpn, which represents the normal response of a provided operation, after receiving a normal response of a required operation; (vi) rnpa, which represents the signalling of an exception after receiving a normal response of a required operation; (vii) rapn, which represents a normal response of a provided operation after receiving an exception from a required operation; (viii) rapa, which

represents the signalling of an exception after receiving an exception from a required operation; (ix) rnrn, which represents a requesting of a second required operation after receiving a normal response of a previous request; and (x) rarn, which represents a requesting of a required operation after receiving an exception from a previous request.

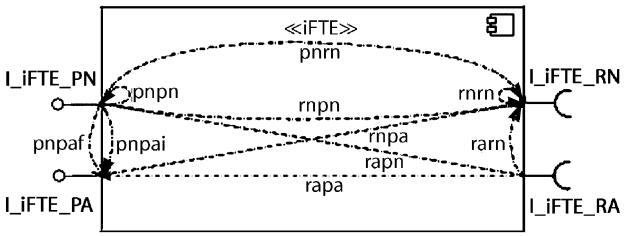

Fig.3. Internal relations of iFTE.

4.2 iFTE Behavioural Scenarios

A scenario is a sequence of events expected during the system operation, which includes environment conditions, expected stimuli and responses[27]. Basically, scenarios focus on how the system behaves to implement its functionalities. Since the internal behaviour of the iFTE is defined through the aforementioned ten relations, the identification of scenarios consists of combining these relations with requests and responses of operations, and with the signalling and capturing of exceptions. Based on these combinations, nine different scenarios are identified, which describe the external behaviour of an iFTE: (i) *internal request/response scenario*, which involves only the pnpn relation, and consists of a successful execution of a provided operation; (ii) *interface exception scenario*, which involves only the pnpai relation, and consists of a wrong request of a provided operation, which causes the signalling of an interface exception; (iii) *internal failure exception scenario*, which involves only the pnpaf relation, and consists of an unsuccessful execution of a provided operation, as a consequence of an internal error; (iv) *external request/response scenario*, which involves the pnrn and rnpn relations, and consists of a successful execution of a provide operation that uses external operations; (v) *failure exception scenario*, which involves the pnrn and rnpa relations, and consists of an unsuccessful execution of provided operations, which raises an exception after requesting external operations; (vi) *masking scenario*, which involves the pnrn and rapn relations, and consists of a successful execution of a provided operation after masking an exception caught from a required operation. In the context of fault tolerance, this scenario is extremely important, since it explicitly represent that a handler has been executed and successfully recovered the state of the architectural element, thus tolerating the fault of its server; (vii) *exception propagation scenario*, which involves the pnrn and rapa relations, and consists of an unsuccessful execution of a provided operation, after catching an exception from a required operation, which could not be successfully masked; (viii) *iterative request response scenario*, which involves the pnrn, rnrn, and rnopn relations, and consists of a successful execution of a provided operation after requesting more than one required operations; and (ix) *iterative exception propagation scenario*, which involves the pnrn, rarn, and rnpa relations, and consists of an unsuccessful execution of a provided operation, after catching an exception and requesting a further required operation, the exception is not successfully masked. These nine scenarios are considered basic since they can be combined for generating other more complex normal and abnormal scenarios.

4.3 Detailed iFTE

Fig.4 presents the detailed design of an iFTE, which contains five architectural elements, each one with a specific and well-defined role: (i) the Normal component implements the normal behaviour of the iFTE; (ii) the Abnormal component implements the exception handlers for the exceptions raised by the Normal component, and those caught through the I_iFTE_PA interface; (iii) the Provided component acts like a bridge between the operations provided by the iFTE and its environment, including the signalling of exceptions; (iv) the Required component also acts like a bridge, but between the required operations of the iFTE and its environment; and (v) the Coordinator connector coordinates the interaction between the four internal components. For realising the boundary between the iFTE and the environment, the Provided and Required components are also responsible for adapting the data interchanged between architectural elements, in order to prevent architectural mismatches.

For providing an explicit separation of concerns, each internal element of the iFTE has a specific role regarding either the normal behaviour (Normal component), the abnormal behaviour (Abnormal component), or the resolution of architectural mismatches (Required and Provided components). Moreover, the definition of separate interfaces for the normal and abnormal behaviour facilitates the reuse of the normal part of the iFTE, even when the exceptions and exception handlers have to be adapted to a different architectural context. The Abnormal component is the only one that handles exceptions; the other elements are only capable of identifying erroneous conditions of its own state, raise the

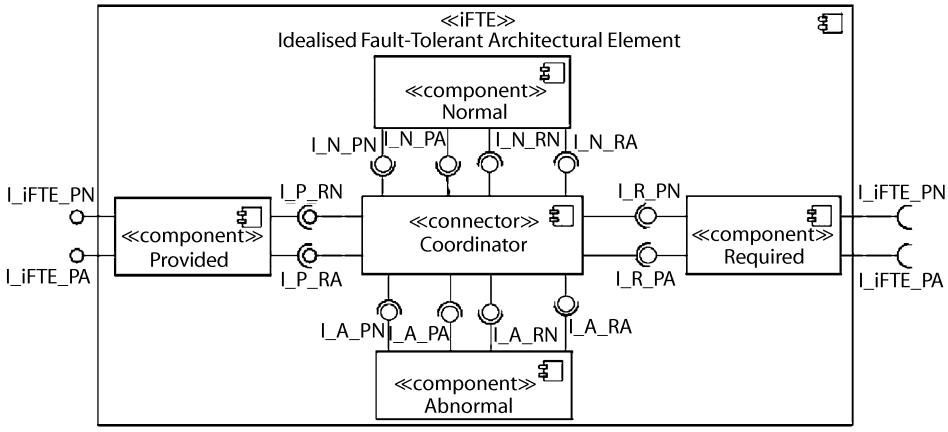

Fig.4. Internal structure of the idealised fault-tolerant architectural element (iFTE).

corresponding exceptions, and propagating it to the Abnormal component through the Coordinator. The internal architectural elements of the iFTE interact through internal interfaces, and these interfaces also enforce the separation between normal and abnormal behaviour.

Analysing the interaction amongst the internal elements of the iFTE, we have identified 12 basic scenarios. After receiving a request through I_iFTE_PN, the Provided component may respond in two different ways: (i) signals an interface exception through I_iFTE_PA (1st basic scenario); or (ii) requests the respective operation of the Normal component from I_P_RN to I_N_PN, mediated by the Coordinator connector. When the Normal component receives an operation request, it can behave in three different ways: (i) returns normally to the Provided from I_N_PN to I_P_RN (2nd basic scenario); (ii) signals an internal exception through I_N_PA; or (iii) requests an external operation. When the Normal component signals an exception through I_N_PA, the Coordinator connector propagates it to the Abnormal component through I_A_RA. After executing the handler, whose behaviour is not considered here, the Abnormal either signals a failure exception through I_A_PA (3rd basic scenario), or return normally through I_A_PN, masking the error (4th basic scenario). When the Normal component requests external operations through I_N_RN, the Coordinator propagates the request to the Required through I_R_PN. After this, the Required requests the operation for an external element through I_iFTE_RN and can receive two different responses: (i) a normal response through I_iFTE_RN (5th basic scenario); or (ii) an exception through I_iFTE_RA. In the last case, the external exception is propagated to the Abnormal (from I_R_PA to I_A_RA through the Coordinator), which tries to handle it. The Abnormal can provide either a failure exception through I_A_PA (6th basic scenario), or a normal response through I_A_PN, masking the external exception (7th basic scenario).

The other five scenarios where derived from the composition of the seven basic scenarios that were presented. Before the iFTE raises an internal exception, it could have successfully executed an external operation (external request followed by an internal exception). In this case, the iFTE can either mask the exception (8th basic scenario), or fail (9th basic scenario). When the Normal requests an external operation after it had masked an internal exception, it can receive an abnormal response (masked internal exception followed by an external exception). In this case, the Abnormal component tries to handle it. If the external exception is masked, it constitutes the 10th basic scenario, which masks both internal and external exceptions. If the external exception could not be successfully handled, the Abnormal component returns abnormally and the iFTE crashes (11th basic scenario). Finally, the 12th basic scenario occurs when although the iFTE had masked an external exception (Scenario 7), it could not mask a following internal one.

Since the Normal component is responsible for providing the functionalities of the iFTE, it might be an existing component that needs to be incorporated into the architecture. For using an existing component it might be necessary to adapt the reused component in order to make it compatible with the four internal interfaces of the Normal. As presented in Fig.5, the structure of the reused Normal component is composed of three elements: the ReusedComponent, which is being reused and can even be a COTS component; a NormalProvided adapter, which is responsible for converting all the provided interfaces of the reused component into the I_N_PN and I_N_PA interfaces; and a NormalRequired

adapter, which is responsible to convert all the required interfaces of the reused component into the I_N_RN and I_N_RA interfaces.

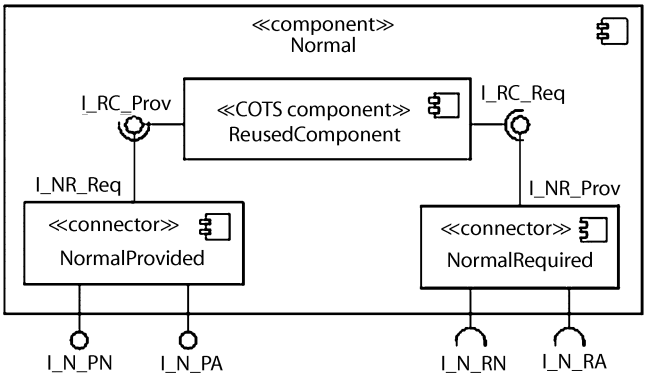

Fig.5. Adaptation of an existing Normal component.

5 Rigorous Development Approach Using iFTE

In our approach, the software architecture is considered a first-level unit, which guides the development from the specification to the implementation of the application. Fig.6 presents an overview of the proposed approach for developing fault-tolerant software architectures. Activity 1 specifies the software architecture, which can be done graphically using a CASE tool. From the use case abnormal scenarios, two artefacts should be specified: a UML component diagram with normal and abnormal interfaces representing the structure of the software architecture, and a set of UML sequence diagrams with abnormal scenarios representing the abnormal architectural scenarios of exception control flows and handlers. For generating the architectural scenarios, the use case scenarios are refined according to the

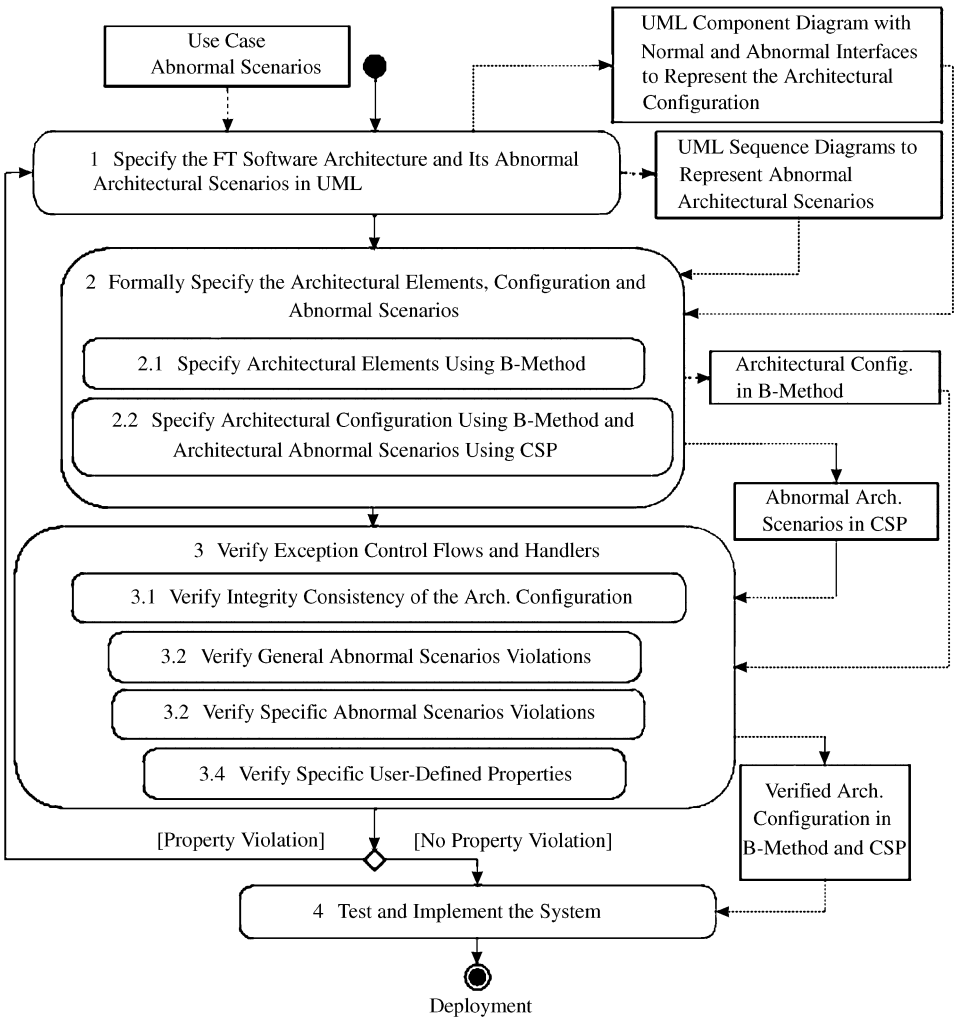

Fig.6. Process for developing abnormal behaviour.

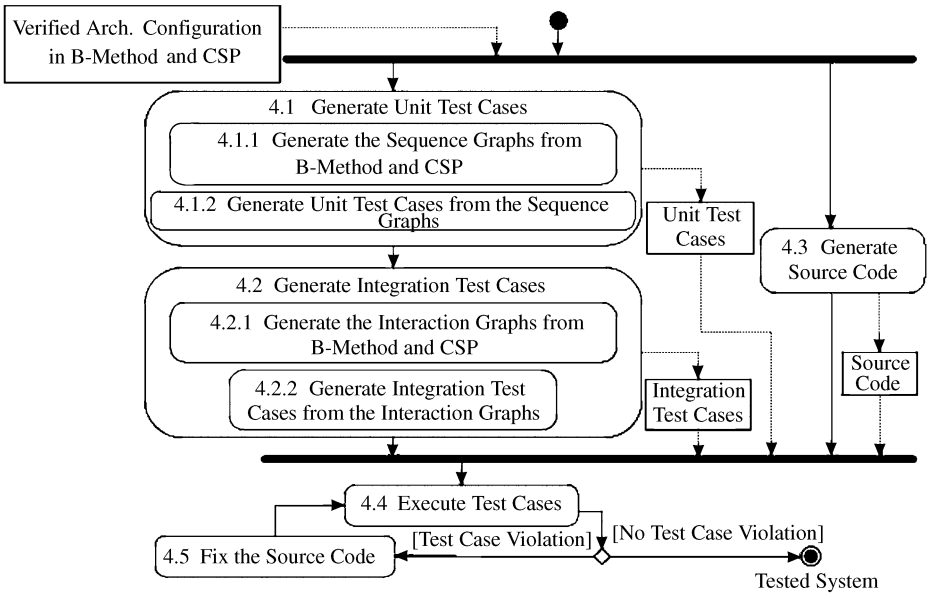

Fig.7. Test case generation and system implementation.

architectural configuration of the system. Activity 2 formally specifies the software architecture (architectural configuration and scenarios). This activity consists of an automatic model transformation from UML (XMI files) to B-Method and CSP. This transformation consists of instantiating the formal templates presented in Subsections 7.1 and 8.1 with the structural and behavioural specifications extracted from the UML models. Activity 3 is the formal verification of the software architecture, in order to identify design faults related to the exception control flows and handlers. Activity 4, which consists of the generation of test cases and the system implementation, is detailed in Fig.7. This process is considered recursive, since it can be executed either for the entire system, or for the internal structure of an architectural element.

As presented in Fig.7, the generation of test cases and the system implementation should be executed in parallel. For generating unit test cases, the proposed approach uses the formal specification of the architectural elements in B-Method and CSP. First, in Activity 4.1.1 the sequences of requests and responses involving the provided and required interfaces of each iFTE are graphically represented using sequence graphs, which uses the notation of UML activity diagrams. Afterwards, in Activity 4.1.2 the sequence graphs are used for generating unit test cases. Sequence graphs can be seen as UML activity diagrams whose activities represent events which can be either the execution of operations or the signalling of exceptions. The sequence of events is indicated by the edges of the graph, which in our approach are derived from the behavioural formal specification in CSP. The generation of integration test cases, which is conducted in Activity 4.2, follows a similar rationale. However, since integration focuses on the interaction between architectural elements, it is necessary to represent such interactions through interaction graphs. Interaction graphs are UML activity diagrams that graphically represent the interactive behaviour involving the provided and required interfaces of different architectural elements. The generation of such diagrams can be automatised, since they capture the interactions already represented by the architectural configuration (B-Method) and the respective architectural scenarios (CSP). Finally, in Activity 4.2.2 the interaction graphs are used for generating integration test cases, which assesses the existence of architectural mismatches involving exception propagation. It is important to stress that both unit and integration test cases can be automatically generated by using case tools for supporting model-based test case generation.

With the system properly verified and the test cases already generated, Activity 4.4 consists of the execution of test cases in order to assess the correctness of the source code against the scenarios of the software architecture. Finally, if any fault is identified during the test execution, it has to be fixed in Activity 4.5.

5.1 Verification Process

We have defined a single process to be followed when verifying properties associated with the system abnormal behaviour. This process is a refinement of Activity 3 presented in Fig.6, and is composed of four

sub-activities to be executed sequentially. These sub-activities should be employed in the verification of both architectural elements and architectural configuration. Sub-activity 3.1 (*verify integrity consistency*) comprises the syntactical analysis of the model, as well as the integrity of its state. Sub-activity 3.2 (*verify general abnormal scenarios violations*) verifies whether there exists violations of the specified scenarios, by checking the existence of impossible scenarios. Sub-activity 3.3 (*verify specific abnormal scenario violation*) tries to find desired patterns of exception control flows and handlers. Finally, Sub-activity 3.4 (*verify user-defined properties*) analyses application specific properties.

The properties of interest associated with Sub-activities 3.1 and 3.2 are specified as *assertions*, i.e., formal rules that the model should be compliant with. If one of these rules is violated, an error message and a counterexample are presented by the model checker. Differently, the properties associated with Sub-activity 3.3 are specified as *definitions*, i.e., formal state patterns that the model checker tries to find (not their violations). If there is a state that satisfies those patterns, a warning message and an example are presented by the model checker. Finally, since the properties of Sub-activity 3.4 are defined by the user, it is possible to use both assertions and definitions.

For executing the verification, the ProB model checker calculates all the possible states of the model at runtime. For each state, it checks if it violates any assertion, and if it satisfies any definition. To reduce the statespace of the model checking, we have adopted a scenario-based approach, where architectural scenarios in CSP are used to restrict the way that the B-Method operations change the state of the machine. To support the verification of behavioural properties, the B-Method machines define the `sequenceHistory` variable, which stores the sequence of events that were executed to achieve the current state. As presented in Subsections 7.1 and 8.1, these events consist on requests and responses of operations.

5.2 Validation Process

For allowing the specification of test cases in different levels of abstractions without knowing the source code, our approach implements a grey-box testing strategy based both on the scenarios and on the structure of the software architecture and iFTEs. The generated test cases can be used for validating the software system against its functional and non-functional requirements. Regarding the functional requirements, application-specific behavioural scenarios can be defined for representing the expected behaviour of the whole application. Moreover, the proposed approach also uses the structural information of the software architecture to validate the system against its non-functional requirements. Validation is supported by three activities. First, unit test cases can also be generated for the application, if we consider the whole system as a single component (Sub-activity 4.1 of Fig.7). Second, integration test cases can validate the software system against the existence of architectural mismatches, which can hinder non-functional requirements related to dependability (Sub-activity 4.2 of Fig.7). Finally, the scenario-based strategy also allows us to generate both unit and integration test cases for specific behaviour related to either functionalities or error handling using, for instance, architectural reconfiguration.

6 Case Study: A Mining Control System

Subsection 6.1 presents the description of the case study used for exemplifying and evaluating the approach proposed in this paper, including the respective goals and steps of execution. Subsection 6.2 presents the specification of the fault-tolerant software.

6.1 Description of the Case Study

In this subsection, it is presented a case study of a mining control system[31], which was been conducted by the authors. The extraction of minerals from a mine produces water and releases methane gas. In addition to extracting minerals, the mining control system is used to drain water from the sump, and to remove air from the mine when the methane level becomes high. The system is composed by three main sub-systems: MineralExtractorController, which controls the extraction of minerals, PumpController, which controls the level of water, and AirExtractorController, which controls the level of methane. When the water reaches a high level, the pump is turned on and the sump is drained until the water reaches a low level. A water flow sensor is able to detect the flow of water in the pipe. However, the pump is situated underground, and for safety reasons it must not start, or continue to run, when the amount of methane in the mine exceeds a safety limit. For controlling the level of methane, there is an air extractor controller that monitors the level of methane inside the mine, and when the level is high an air extractor is switched on to remove air from the mine. The whole system is also controlled from the surface via an operator console that should handle any emergencies raised by the automatic system. For improving the reliability and availability of the transactions associated with the system controllers, we have

defined a fault-tolerant software architecture based on the iFTE abstraction, following the approach proposed in this paper.

The main objective of this case study is threefold. First it aims to assess the feasibility of the proposed solution in what concerns the verification and testing of scenarios of exception control flow and handling at the software architecture. Second, it aims to assess if the proposed approach helps in finding bugs that would go unnoticed otherwise. Finally, this case study also aims to assess if the use of architectural abstractions improves understandability. Regarding scalability assessment, which was not the focus of this case study, we have done other case studies with bigger and more complex applications. In these case studies we have noticed that the proposed solution has shown to be more scalable than other existing solutions such as the Aereal framework. Besides, the case studies have also showed that the main advantage of our solution concerning scalability is the adoption of architectural abstractions together with architectural scenarios, which considerably reduces the number of architectural elements as well as the respective state space.

The system requirements where specified in terms of use cases abnormal scenarios, according to the MDCE methodology[6]. Abnormal scenarios are characterised by the presence of exceptions, e.g., raising, propagation, and handling of exceptions.

6.2 Specification of the Software Architecture

The software architecture of the mining control system is presented in Fig.8 through a UML component diagram. As it can be seen, the architecture is composed of eleven architectural elements, four of them are sensors: (i) MethaneLevel, which detects the level of methane inside the mine; (ii) AirFlow, which detects the flow of air inside the pipes; (iii) WaterLevel, which detects the level of water inside the mine; and (iv) WaterFlow, which detects the flow of water inside the pipes. It is assumed that in this system all the architectural elements are iFTEs, except for the four sensors (AirFlow, MethaneHigh, WaterLow, WaterHigh).

The three identified controllers (MineralExtractorController, AirExtractorController, and PumpController), have the role of architectural connectors (≪iFTEConnectors≫). Each controller is responsible for dealing with the normal behaviour of the system, and handling any exceptions that are propagated by the components. Depending on the state of the sensors, one of the controllers will always be activated: (i) water low & methane low ⇒ MineralExtractorController; (ii) water high & methane low ⇒ PumpController; and (iii) methane high ⇒ AirExtractorController. In case there is a failure that cannot be handled by the system, the AirExtractorController notifies the OperatorInterface element that such a failure has occurred.

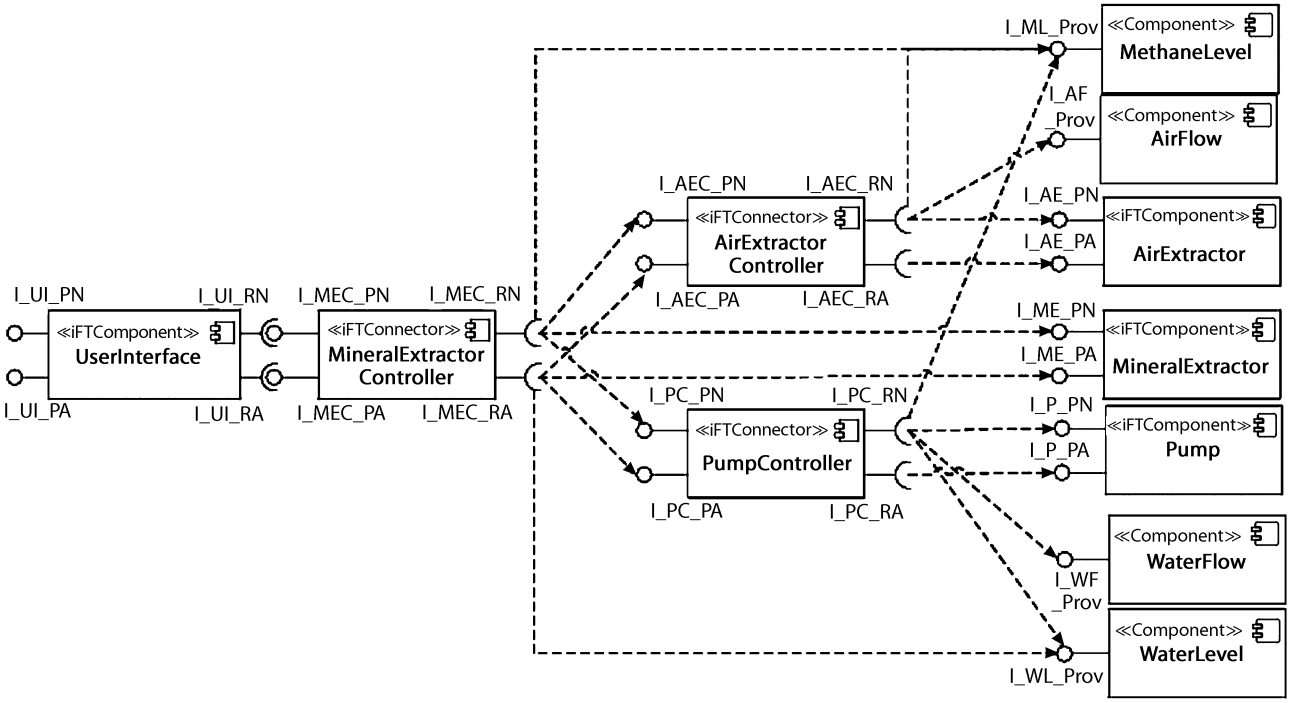

Fig.8. Architectural configuration of the mining control system.

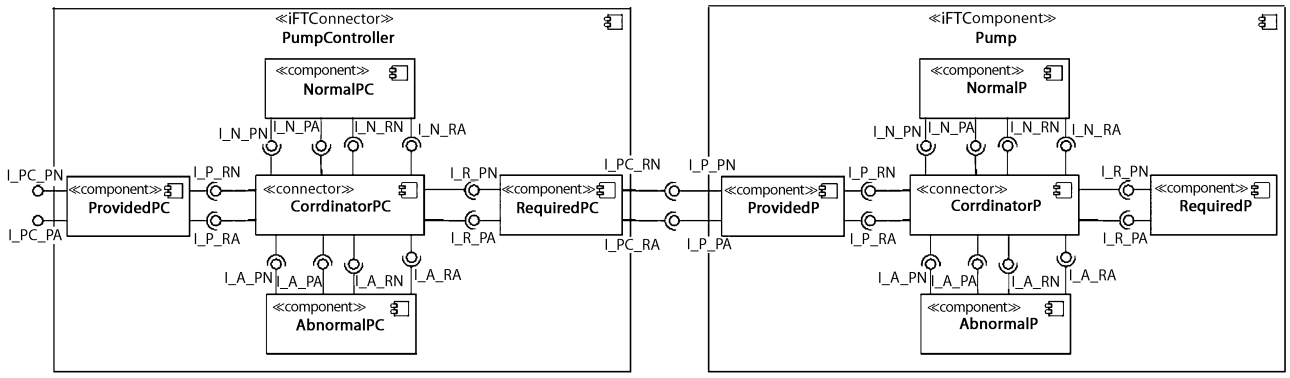

Fig.9. Detailed view of a partial architectural configuration.

For this architectural configuration, a total of 13 architectural exceptions were identified related to errors in the system architecture. For exemplifying the flow of exceptions, in the following, we consider the case when an error is detected inside the AirExtractor, and an internal exception is raised[32]. If AirExtractor fails to handle this exception locally, it propagates an exception to the AirExtractorController. Again this architectural element attempts to handle the exception once it is caught, but if it fails, it propagates the exception to the MileralExtractorController. If the concentration of methane is high and the AirExtractor has failed, there is nothing that MileralExtractorController can do, except to propagate an exception to its collaborating architectural elements. Upon receiving this exception, the MineralExtractor, the PumpController and the AirExtractorController should shut down their activities, and the OperatorInterface should raise an alarm for the operator to take the appropriate measures.

To illustrate the structure of the iFTEs of the software architecture, Fig.9 presents the internal details of the PumpController and Pump architectural elements. As can be seen, in the Required component of the PumpController and the Provided component of the Pump are responsible for enabling the interaction, adapting the received operation requests (Provided), and the respective return values (Required).

7 Verification and Testing of iFTEs

Although the iFTE architectural abstraction was already partially verified using extended timed automata notation[32], this is not sufficient for verifying its instantiation into architectural elements, and the distinction necessary between different types of exceptions. In the following, we present the formal model for representing iFTE-based architectural elements (in B-Method and CSP), as well as the properties for verifying the consistency of the architectural element with the proposed abstraction. An overview of the B-Method and CSP formal notations used in this paper is presented in Appendix A.

7.1 Formal Specification of iFTEs

The formal models presented in this paper are general, meaning that they can be used as templates for representing different software architectures. But to facilitate their understanding, we present them in the context of the mining control system presented in Section 6 (Fig.8).

Fig.10 presents part of the B-Method machine of the iFTE abstract model for the PumpController connector (pc). A B-Method machine explicitly represents an iFTE in terms of three basic features: its *interfaces*, through the `pc_Interfaces` set (line 4); its provided and required *operations*, through the `pc_Operations` set (line 5); and its provided and required *exceptions*, through the `pc_Exceptions` set (line 6). The B-Method machine represents three types of events (request, normal response, and abnormal response) through the `eventType` set (line 8). Essentially, these events are used to define the behavioural scenarios of an iFTE. After representing the basic features of the iFTE abstract model through sets, it is necessary to categorise and relate them by means of B-Method variables. lines 15 to 18 present the variables that categorise the interfaces of the `pc_Interfaces` set. The formal model guarantees, through properties, that each interface can only participate in one of these subsets (details in Section 7). Depending on whether it is an operation or an exception, it has to be associated with either a normal or abnormal interface, respectively. These associations are made through relations, which are defined in lines 23, 24, 27 and 28. These relations also capture whether the operations and exceptions are provided or required. The first relation (line 23) is from `pc_pnInterfaces` to a power set of `pc_Operations`. The provided and

required exceptions are associated to the abnormal interfaces of the iFTE abstract model through the relations presented in lines 27 and 28. Moreover, the exceptions are also associated to the operations of the

```
1  MACHINE pc
2  /*=======================*/
3  SETS
4      pc_Interfaces = {i_pc_pn, i_pc_rn, i_pc_pa, i_pc_ra};
5      pc_Operations = {controlPump, getMethaneLevel, ...};
6      pc_Exceptions = {MethaneHighPumpOnException, ...};
7
8      events = {request, normalResponse, abnormalResponse}
9  /*=======================*/
10 VARIABLES
11 ... /*declaration of variables*/
12 /*=======================*/
13 INVARIANT
14       /*interface category*/
15       pc_pnInterfaces:POW(pc_Interfaces) &
16       pc_paInterfaces:POW(pc_Interfaces) &
17       pc_rnInterfaces:POW(pc_Interfaces) &
18       pc_raInterfaces:POW(pc_Interfaces) &
19
20       /*interfaces — operations*/
21       pc_pnOperations:pc_pnInterfaces +−> POW(pc_
           Operations) &
22       pc_rnOperations:pc_rnInterfaces +−> POW(pc_
           Operations) &
23       /*interfaces — exceptions*/
24       pc_paIntExceptions:pc_paInterfaces +−> POW(pc_
           Exceptions) &
25       pc_raIntExceptions:pc_raInterfaces +−> POW(pc_
           Exceptions) &
26
27       /*exceptions — operations*/
28       pc_pnOpExceptions:pc_Operations +−> POW(pc_
           Exceptions) &
29       pc_rnOpExceptions:pc_Operations +−> POW(pc_
           Exceptions) &
30 ...
31 /*=======================*/
32 OPERATIONS
33 exception <− − pc(interface, operation, eventType)=
34 PRE
35     interface : pc_Interfaces &
36     operation : pc_Operations &
37     eventType : events &
38     ...
39 THEN
40     ...
41 END
```

Fig.10. B-Method machine of PumpController iFTE connector.

iFTE abstract model through the relations presented in lines 30 and 31. Other 16 variables are defined by the machine: 10 for representing the iFTE abstract model internal relations defined in Subsection 4.1 (Fig.3), and 6 to store information used during the verification process (e.g., the traceability of the events, and the exceptions that where caught by the iFTE), which are not shown in Fig.10. Finally, the B-Method machine defines an operation for representing the occurrence of an event (lines 39∼47), which is characterised by an interface and an operation (`interface` and `operation` arguments respectively). The type of the event is represented through the argument `event`. The return value (`exception`) represents the exception that has been returned by the event. When the event does not provide an abnormal return (e.g., a request or a normal response), `exception` is assigned as an empty set (∅).

The formal model of the iFTE can be instantiated according to specific architectural elements, which have specific interfaces, operations and exceptions. After that, as shown in Subsection 8.1, the formal models of various architectural elements are used to compose the software architecture.

7.2 Verification of iFTEs

The verification of software architectures considers properties regarding both the architectural elements, and an architectural configuration. The properties associated with the architectural elements should enable to check whether they are consistent against the behaviour of the iFTE; while the properties associated with the architectural configuration should be able to check whether architectural configuration follows the composition rules dictated by the architectural elements. The verification is conducted using model checking, and when a property is violated, an error message and a counter example are presented by the tool.

The properties to be verified cover four basic goals: (i) *consistency of the formal model*, which comprises the syntactical analysis of the models, as well as the integrity of the instantiated variables; (ii) *scenarios violations*, which amounts to identify possible violations on the specified scenarios; (iii) *extra behavioural information*, which analyses if the model satisfies all the possible valid scenarios (behavioural completeness), and does not satisfy invalid (or undesired) scenarios (behaviour consistency); and (iv) *user-defined properties*, which verifies particular aspects of an application and should be defined by the user. Table 1 details the properties related to each goal in the context of iFTE-based architectural elements.

Table 1. Properties of Interest of iFTE

Consistency of the Formal Model	
No.	Property Name & Description
1	*Internal relations integrity property.* An operation cannot be considered at the same time as internal and external.
2	*Internal exceptions consistency property.* Internal exceptions cannot be propagated to other architectural elements.
3	*Interface classification property.* All the interfaces have to be classified in exactly one of the four types defined by the iFTE abstract model.
4	*Interface and operation consistency.* The associations between an interface and its operations have to be consistent with their types (provided or required).
5	*Normal and abnormal interfaces integrity property.* Interfaces classified as normal can neither signal nor catch exceptions, and the interfaces classified as abnormal can neither receive operation requests nor provide normal returns.
6	*Cardinality of relations property.* Verifies if the cardinalities of the relations remain correct after the iFTE abstract model instantiation.
7	*Provided and required exceptions integrity property.* The sets of provided and required exceptions should be disjoint.
8	*Interface and raised exceptions integrity property.* The sets of interface and failure exceptions should be disjoint.
9	*Prevention of unused provided exceptions property.* Every provided exception should be signalled by at least one provided operation.
10	*Masking correctness property.* Only maskable exceptions can be masked by the iFTE, any other exception that would be masked is considered a design failure.
11	*Abnormal interfaces and operations integrity property.* The set of exceptions associated to an abnormal interface has to be equivalent to the set of exceptions associated to its operations.
Scenarios Violations	
No.	Property Name & Description
1∼9	One property for each basic scenario.
Extra Behavioural Information	
No.	Property Name & Description
1	*Behavioural completeness property.* Verify if the architectural element satisfies all the basic scenarios of the iFTE.

7.3 Testing of iFTEs

Test cases generation follows the model-driven approach[33,34], and most of which can be automated. All the testing artefacts can be reused each time the component is tested: during its development or each time it is reused. As a consequence, the iFTE component testing can be performed in a black-box way, allowing test cases reuse even without component source code.

For generating test cases for a provided operation of an iFTE, it is necessary to generate a *sequence graph*, which represents the execution of the internal and required operations that the provided operation requires, as well as the respective normal and abnormal returns. The sequence graph consists of a graphical representation of the formal models of the iFTE (Subsection 7.1), which is constructed for each one of its provided operations. The nodes of the sequence graph are identified from the B-Method machine, while the edges are identified from the CSP specification. The nodes of the sequence graph are disposed in different partitions according to the respective interface of the iFTE that is referred. Thus, four partitions are defined for the abstract model, and sixteen for the detailed one.

The test cases generated for testing iFTEs can either take into consideration the internal elements individually (its detailed structure), or generate test cases for a higher abstraction, abstracting away its internal structure and considering only the external interfaces of the iFTE, which is I_iFTE_PN, I_iFTE_PA, I_iFTE_RN, and I_iFTE_RA (Fig.2).

For illustrating the artefacts generated, Fig.11 presents the execution sequence graph for the controlPumping() operation of the PumpController connector. This graph represents only the external exceptions (provided and required) of the iFTE for generating test cases for a black-box testing. Analysing this graph, 22 paths are identified by a depth-first search algorithm, producing 22 test cases. Although in this case study the test cases have been manually generated, case tools could use the "interaction graphs" as input for automatically generating them. In our research group, a prototype tool is being developed for this purpose[35]. The `controlPumping()` operation is represented as an initial node of the `I_PC_PN` partition. In the same way, each required operation that can be executed by `controlPumping()` is represented as a node

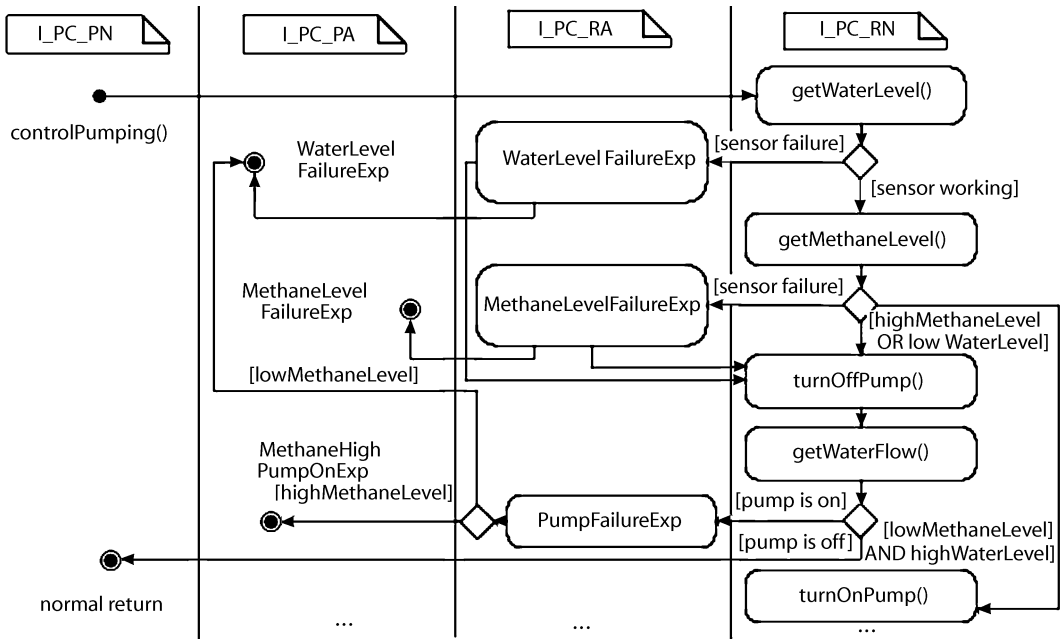

Fig.11. Part of execution sequence graph for the controlPumping operation.

of the I_PC_RN partition. The subset of required operations of controlPumping() is determined by the internal relations of the iFTE, defined in the B-Method machine. The exceptions that can be caught by the PumpController element are represented as nodes of the I_PC_RA partition, while the signalled exceptions are disposed in the I_PC_PA partition. Finally, it is necessary to define a node to represent the normal return of the element. This node is disposed in the I_PC_PA partition. The outputs of the architectural element (normal and abnormal) are considered final nodes, representing the end of an execution scenario.

For exemplifying the stub creation, one of the test cases simulates the MethaneHighPumpOnException throwing. In this case, when the pump is turned on, the stub that simulates the getMethaneLevel() required operation was prepared to return the highMethaneLevel. After that, for safety reasons the controller tries to turn off the pump. The stub of the turnOffPump() operation was prepared to throw the PumpFailureException exception when the operations were called. Because it indicates an emergency exception, the controller informs this warning to the MineralExtractorController connector through the MethaneHighPumpOnException. After executing this scenario, the test oracle has to execute the contract verification, which checks exception class type and context.

The sequence of operation requests and responses is derived from the CSP specification, which represents the architectural scenarios of the application (details in Subsection 8.1). For example, analysing the following CSP (hypothetical) specification: "pc.i_pc_pn.controlPump.request $->$ pc.i_ pc_rn.ROP.request" it means that the process pc.i_pc_pn.controlPump.request comes before process pc.i_pc_rn.ROP.request. In other words, the controlPump provided operation of the PumpController connector is always executed before any of its required operations, represented by the ROP variable.

After constructing the graph, the test cases can be generated in a straightforward way: each path from the start to a final node is considered a test-case. Besides the identification of the test cases (paths of the graph), the graph is also useful for deriving stub synchronisation commands, because it illustrates the sequence on which the required operations are called, as well as the respective expected returns. During the test execution, stubs should replace required elements, simulating their behaviour in a controlled way, and making it possible to observe behaviour of the component under test in normal and abnormal situations.

Besides the generation of the test cases, it is necessary to determine the correct ordering for executing the test cases of the component's provided operations. Analysing the semantic of the provided operations, it is possible to exist a mandatory logical sequence of execution. For example, because the Pump component is initially turned off, its turnOnPump() operation should be tested before the turnOffPump() one. For determining the testing order, we have to generate the *execution*

flow graph, which is a dependency graph that illustrates the sequential dependencies among the provided operations of an architectural element. Beyond its usefulness for determine the best order for executing the test, the execution flow graph also derives test drivers for executing test cases.

8 Verification and Testing of iFTE-Based Software Architectures

Besides the verification of the architectural elements, it is also necessary to verify the integration of these elements into architectural configurations. The focus of the architectural verification is to support the analysis of signalling, handling and propagation of exceptions in terms of their types and conversions during the flow between architectural elements. In the following, we present the formal model for representing iFTE-based software architectures (in B-Method and CSP), as well as the properties for verifying the exception flow. The architectural formal model is presented in the context of the mining control system presented in Fig.8.

8.1 Formal Specification of iFTE-Based Software Architectures

A software architecture based on iFTE architectural elements is also specified through a B-Method machine and a CSP specification. The B-Method machine represents the structural and behavioural information about the software architecture, e.g., architectural elements, architectural configuration, and exception flows among architectural elements. The CSP specification defines the possible execution scenarios, restricting the sequence of events that can occur in the architecture, and synchronising the occurrence of architectural events and internal events of each architectural element.

Fig.12 presents part of the architectural configuration template instantiated according to the mineral extraction controller system's software architecture (Fig.8). Differently from the specification of the architectural elements, the architectural model contains many architectural elements (iFTEs and non-iFTEs), which are represented by the `mineral_ArchElements` set (line 4). Besides, the full specification of each architectural element is incorporated into the software architecture, i.e., the B-Method machines of the iFTEs and non-iFTEs are imported by the architectural specification (line 8).

Using two complementary subsets, the architectural elements are then classified in iFTEs (`mineral_ifteArchElements`, line 14) and non-iFTEs (`mineral_nonIfteArchElements`, line 15). An integrity property was defined for guarantee that *mineral_ifteArchElements* ∪ *mineral_nonIfteArchElements* = *mineral_ArchElements*. The properties verified are presented in Subsection 8.2. Further, the architectural elements are interconnected in terms of an architectural configuration, which is defined through a relation between required and provided interfaces (`mineral_archConfiguration`, line 17). In addition, to contextualise the provided and required exceptions during propagations, another relation is defined (line 18). Finally, there is a B-Method operation for representing events involving a pair of architectural elements (lines 20∼28). Besides the three parameters that are also present in the iFTE abstract model (`interface`, `operation`, and `eventType`), the architectural model also identifies the architectural element

```
1   MACHINE mineral
2   /*==========*/
3   SETS
4       mineral_ArchElements = {ui, mec, aec, pc, p, ...};
5   ...
6   /*==========*/
7   EXTENDS
8       ui, mec, aec, pc, p, ...
9   /*==========*/
10  VARIABLES
11  ... /*declaration of variables*/
12  /*==========*/
13  INVARIANT
14      mineral_ifteArchElements:POW(mineral_Arch-
          Elements) &
15      mineral_nonIfteArchElements:POW(mineral_Arch-
          Elements) &
16
17    mineral_archConfiguration : mineral_reqInterfaces +->
          POW(mineral_provInterfaces) &
18    mineral_provExcep_reqExcep : mineral_provExceptions
          +-> POW(mineral_reqExceptions) &
19  ...
20  /*==========*/
21  OPERATIONS
22  exception <-- mineral (from, to, interface, operation, eve-
      ntType)=
23  PRE
24      from: mineral_ArchitecturalElements & to: mineral_
          ArchitecturalElements &
25  ...
26  THEN
27    ...
28  END
```

Fig.12. B-Method machine of an architectural configuration.

```
1   MAIN = UI ;;
2   ···
3   MEC_PC = mec.i_mec_rn.controlPump.request −> mi-
        neral.mec.pc.i_mec_rn.controlPump.request −> mi-
        neral.mec.pc.i_pc_pn.controlPump.request −> PC
        (i_mec_rn, controlPump);;
4   ···
5   − − pump controller − −
6   PC(RN1, ROP1) = pc.i_pc_pn.controlPump.request −>
        PC_INT (RN1, ROP1, i_pc_pn, controlPump);;
7   − − pc−ifte − −
8   PC_INT (RN1, ROP1, PN1, POP1) = (pc.PA.POP1. ab-
        normalResponse?EXP −> MEC_PC_RETURN (RN1,
        ROP1)) []
9   (pc.i_pc_rn.pumpWater.request −> PC_P(RN1, ROP1,
        PN1, POP1, i_pa_rn, pumpWater));;
10  ···
11  − − pump − −
12  PC_P (RN1, ROP1, PN1, POP1, RN2, ROP2) = pc.
        i_pc_rn.pumpWater.request −> mineral.pc.p.i_ pc_rn.
        pumpWater.request −> mineral.pc.p.i_ p._pn.pump-
        Water.request −> P(···);;
13  ···
```

Fig.13. CSP specification of a software architecture.

that have originated the event (`from`), as well as its respective destination (`to`).

In order to define the architectural scenarios of the application, it is necessary to define a CSP specification to complement the structural definition of the B-Method machines. Fig.13 presents the CSP specification of the architectural model. According to the ProB model checker, each B-Method operation corresponds to an event in CSP, followed by its proper parameters (Fig.13). For example, in line 9, the request `pc.i_pc_rn.pumpWater.request` corresponds to the execution of the `pc` operation in the B-Method machine, with the following values of parameters: "`interface = i_pc_rn`", "`operation = pumpWater`", and "`event = request`". For representing the possibility of occurring many different events, the CSP specification uses the external choice operator (`[]`). In this way, after the PumpController receives an operation request (`pc.i_pc_pn.controlPump.request`), the execution sequence is defined by the `PC_INT` process (line 8), which states that the PumpController should either provide an abnormal response to the MineralExtractionController (`mec`), or request external operations to the Pump (`p`). In the first case, it is important to notice that the response refers to the same operation that has been previously requested, which is represented by the `ROP1` variable. In the second case, when the PumpController requests an external operation, the execution sequence is defined by the `PC_P` process, which should either provide a normal or an abnormal response to the PumpController.

Analysing Fig.13 we can also see the scenarios of CSP are also responsible for synchronising the execution between events at the architectural level and internal events of the iFTEs. For example, line 3 states that the internal events of the PumpController connector (`PC`) can only occur after an operation request from the MineralExtractorController to its provided interface (`mineral.mec.pc.i_pc_pn.controlPump.request`). After receiving a request through its provided interface, the PumpController either returns abnormally to the MimeralExtractorController (line 8), or requests an external operation to the Pump (line 9). These example scenarios have two important characteristics to be highlighted: first, note that PumpController is not able to provide a normal return without a previous request to the Pump; second, the sequence of events involves both internal events of the PumpController connector and architectural ones representing the interaction between architectural elements (started by `mineral...`). The CSP specification is responsible to synchronise whenever one of them occurs.

After this instantiation, the iFTE-based architecture in the context of the mining control system, approximately 1000 scenarios were identified. These scenarios were defined by combining different order of execution for the system functionalities, as well as the respective returns. For example, different scenarios can be defined for the mineral extraction, since it is possible to extract air from the mine before extracting minerals. Moreover, each execution can provide either normal or abnormal returns and the abnormal returns can be either successfully tolerated or not. Each one of the iFTE architectural elements was verified through the 22 properties presented in Table 1. For the software architecture, besides the 14 properties of Table 2, we have specified three more properties related to the identification of divergences in specific scenarios considered critical to the business logic. These scenarios occur when the Pump is turned on, and the MethaneLevel components informs that the level of methane is high. In this case, the only sequence of operations that should be possible is turn off the Pump, and turn on the AirFlow component. If these operations are successfully executed, the system continues working. If either the Pump does not turn off, or the AirFlow does not turn on, an alarm should be raised into the UserInterface.

8.2 Verification of iFTE-Based Architectures

In the same way as the architectural elements, the verification of iFTE-based architectures is conducted using model checking, and when a property is violated,

Table 2. Properties of Interest of the Software Architecture

Consistency of the Formal Model	
No.	Property Name & Description
1	*Legitimacy of the architectural dependencies property.* A required interface cannot depends on a provided interface of the same architectural element.
2	*Normal dependency correctness property.* For every required operation, it is necessary to have a correspondent provided operation to be activated.
3	*Operation usage property.* Every operation declared in the architectural model should be associated to an interface.
4	*Interface usage property.* Every interface declared in the architectural model should be associated to an architectural element.
5	*iFTE decomposition property.* For each iFTE, an iFTE abstract model should be defined.
6	*Operation mapping correctness.* The operations associated to iFTEs in the architectural model should have a correspondent operation into the respective iFTE abstract model.
7	*Exception usage property.* Every exception should be associated to at least one operation.
8	*Abnormal dependency correctness property.* There should exist a mapping between the required and provided exceptions of two connected interfaces.
9	*Exception mapping correctness property.* Every architectural exception associated to iFTEs should have a correspondent exception into the respective iFTE abstract model.
Scenarios Violations	
No.	Property Name & Description
1	*Deadlock freedom property.* The architectural configuration has to be free of deadlocks.
2	*Communication uniformity property.* The architectural elements have to communicate each other in a call/return way (response after request).
3	*Dependence integrity property.* An architectural element a can only request operations of another architectural element b if a depends on b.
4	*Request/response property.* Every response event should refer to the same source and destination elements, as well as the same operation that was immediately requested.
5	*Exception propagation property.* When an exception is propagated from an architectural element to another, the provided exception of the first element should be mapped to an equivalent required exception of the second element.
Extra Behavioural Information	
No.	Property Name & Description
–	*Scenario patterns.* Application-specific properties for identifying specific patterns of exception propagation. Usage examples of these properties are: verify if the software architecture is compliant with specific strategies of exception propagation, and obtain the trace of an exception in a specific scenario.

an error message and a counter example are presented by the tool. The properties to be verified for the software architecture also cover the same four basic goals: (i) *consistency of the formal model*; (ii) *scenarios violations*; (iii) *extra behavioural information*; and (iv) *user-defined properties*. Table 2 details the properties related to each goal in the context of IFTE-based software architectures.

The behavioural properties specified for the system define patterns containing a sequence of requests and responses (normal and abnormal) of operations. These patterns specify sequences of exception propagation, exception mapping, and exception handling that are expected during the system execution. If there is an application's scenario that is inconsistent with any pre-specified behavioural pattern, the model checker presents a counterexample that shows the violation.

During the verification of the software architecture, the model checker has detected deadlocks, and the violation of properties. An example of the former was the deadlock caused by omitting at the architectural design the declaration that the PumpController is able to propagate the exception MethaneLevelSensorFailureException. Since PumpController is not able to fully mask this exception, a deadlock occurs whenever it catches this exception from the MethaneLevel sensor. Concerning the violation of exception propagation properties, one of the identified violations was caused by the absence of two handlers (required exceptions) in the MineralExtractorController.

In addition to identify faults introduced during the process of instantiating the architectural abstractions, other two faults were identified related to the violation of the verification properties: one for the MineralExtractorController connector, and another for the software architecture. In the following, we present the formal specification of these two properties.

Property 10 of the iFTE (Table 1), which states that

only maskable exceptions can be masked by the iFTE, is specified for the MineralExtractorController connector as follows: $mec\_maskedExceptions \subseteq dom(mec\_rapn)$, where `mec_maskedExceptions` stores all the exceptions that the MineralExtractorController has masked during the model checking, and `mec_rapn` is a relation from a required exception to a subset of the respective provided operations that can mask it. The violation of such a property reveals that the MineralExtractorController connector should propagate the RaiseAlarm exception, but this exception has been previously handled and swallowed. Since this error was caused by a human mistake during the system specification (in extended UML), it characterises a design fault that would go unnoticed if the proposed rigorous architectural approach was not used. Although it is a simple error, it could be expensive to correct in further phases of the software development.

Property 8 of Category 1 of the software architecture (Table 2), which states that there should exist a mapping between the required and provided exceptions of two connected interfaces, is specified for the mining control system as follows: $\forall provEx, \exists reqEx | provEx \in mining\_providedExceptions \wedge reqEx \in mining\_requiredExceptions \wedge reqEx \in mining\_provExcep\_reqExcep(provEx)$, where `mining_providedExceptions` is the set of the system provided exceptions, `mining_requiredExceptions` is the set of the system required exceptions, and `mining_provExcep_reqExcep` is a relation from a provided exception to a subset of required exceptions to which it can be converted during the exception propagation. The violation of such a property characterises either an architectural mismatch or a human mistake during the architectural design. Since it can compromise the efficacy of the exceptional behaviour and the system dependability, it should be fixed before generating the source code of the application.

Besides the violation of two properties, we have injected other three real design faults in the UML model in order to assess if the proposed approach could identify them during verification. The design faults injected were: (i) omission that the Pump component raises the PumpFailureException; (ii) deletion of the connection between I_MEC_RN and I_AEC_PN interfaces; (iii) addition of a new exception (Exp1) to be raised by the PumpController connector, but no component is able to catch it (a useless exception). After the automatic generation of the formal model in B-Method and CSP and the verification of the specified properties, the injected faults have been identified. The omission of the PumpFailureException exception raised by components has been automatically detected by the ProB model checker as an inconsistency of the B-Method machine (undefined type). The absence of connection between interfaces I_MEC_RN and I_AEC_PN was detected by the violation of Property 1 of Category 2 of the software architecture (Table 2), which states that the architectural configuration has to be free of deadlocks. Finally, the addition of the Exp1 has violated Property 9 of the PumpController connector (Table 1), which prevents the existence of useless exceptions.

8.3 Testing of iFTE-Based Software Architectures

Regarding the testing of software systems against their architectural specification, this paper is focused on integration testing. In the following, we provide details about how to conduct a dependency analysis for determining the integration testing order. After that, we present how integration test cases are identified from the specification of the software architecture.

8.3.1 Dependency Analysis

The existence of dependencies between architectural elements, in terms of their provided and required operations, should impose an order on how these components are integrated, thus facilitating fault localisation. The basis for establishing this order is obtained from the dependency analysis between components. In our work, the order for executing test cases is defined in such a way to reduce the effort of stub creation and consequently the cost of test. A complementary technique for reducing the effort of stub creation was proposed by Zhang and Ryder, which uses program analysis technology to reduce the number of unnecessary nodes and dependencies in the dependency graph. One big challenge in integration testing of component-based systems is how to obtain enough information for dependency analysis given that we cannot assume source code availability. The approach presented here uses the specifications of the architecture and of the iFTEs to apply dependency analysis. At this level an abstract view of the system is considered, but when architectural elements are instantiated by physical components, this analysis can be updated.

For the dependency analysis, we use the chaining approach[36], which relates the architectural elements by chaining the existing dependencies between them. Links represent dependencies among these elements and connect elements that are directly related. Analysing those dependency chains, it is possible to support different interests. For example, to know the components

which are affected by changes in another component. In the context of this work, we use dependency analysis to identify the best order for integrating the architectural elements in order to facilitate the execution of integration test cases.

The features considered in the dependency analysis rely on the notation used to specify the architecture. In the B-Method and CSP specifications presented in Subsection 8.1, the architecture is described in terms of architectural elements, interfaces, operations, exceptions and events. These features are used to construct a matrix representing the direct dependences among the architectural elements. From this matrix it is possible to obtain, for a given architectural element c, three types of chains: affected-by, affects and related. The *affected-by chain* consists of a set of architectural elements that could potentially affect an architectural element c. The *affects chain* is the set of architectural element that can be affected by c. The *related chain* is a combination of the affected-by and affects chains for c. To determine the integration order, we use two metrics, called *influence factor* (IF) and *late integration factor* (LIF)[37]. These metrics were proposed to help in determining the integration order when testing of object oriented systems and in our work it was adapted for testing component-based systems.

8.3.2 Constructing the Dependency Matrix

The dependency matrix (DM) represents the relationships among architectural elements. The columns of a DM represent the dependency in the relationship, and its rows represent the depended-on element. A cell DM[a, b] = 1 indicates that b depends on a. Both structural and behavioural dependencies are indicated in the matrix, and this information could be obtained from the architectural model. For example, in our case, since we are using B-Method and CSP to describe the architecture, we can obtain structural dependencies from B-Method notation (architectural configuration), and the behavioural ones from CSP (scenarios of use).

The rows and columns of the matrix can represent the architectural elements, interfaces, operations and signalling of exceptions. Events from and to the external environment should also be added, such as, a *start* event provided by the user to initiate the system. Then the dependencies between these elements, as described in the B-Method specification, are inserted into the cells. It is also possible to represent the internal relationships among interfaces of an iFTE, as described by their respective B-Method specifications. After inserting the structural relationships, the matrix is completed with behavioural information obtained from the CSP model. Taking the example of the mining control system (Fig.13), if the MineralExtractorController (MEC) connector requests services to the PumpController (PC), which requests services to the Pump (P). So, MEC depends on PC, and PC depends on P.

Once the DM is constructed, we can determine the affects(c) and the affected-by(c) chains. The affected-by(c) is determined by the column of the matrix that is related to c. The affects(c) is determined by the row of the matrix that is related to c. Although in the context of this work the dependency matrix was manually generated, the CSP specification of the software architecture could be used as input for automatically generating it.

8.3.3 Determining the Integration Order

As already mentioned, the integration order is obtained based on two metrics: the *influence factor* (IF) and the *late integration factor* (LIF)[37]. The first one computes the number of architectural elements that should be integrated after the architectural element of interest c. It is obtained by counting the number of elements that c directly affects. In other words, it is the number of $c' \neq c$ such that $DM[c, c'] = 1$. The second factor is $LIF(c) = \Sigma\ IF(c')$ for all $c' \neq c$ such that $DM[c', c] = 1$. The LIF is proportional to the total number of elements that an element depends on. After calculating the LIF of all the elements, we proceed to determine the integration order by performing the following steps.

Step 1. Select the elements with the least LIF. It means that in our strategy the fewer elements an element depends on, the sooner it should be integrated. This rationale is motivated by the reduction of effort for creating stubs. We designate the selected elements by a set, $selected_i$, where i, in this case, is 1, to indicate that they should be integrated in the first step.

Step 2. Recalculate the LIF of the remaining elements. For each $c' \notin selected_i$ and for each $c \in selected_i | c \in affected\text{-}by(c')$ then $LIF(c') = LIF(c') - IF(c)$.

Step 3. Return to Step 1 until all the elements were integrated.

If there are various elements with the same LIF, this indicates the existence of a cycle in the dependency, that is, if $LIF(c) = LIF(c')$ in one step of the integration order determination, then we can conclude that c depends on c' and c' depends on c. They are also designated as *strongly connected elements*[36]. In this case, whichever the element is integrated first, a stub is necessary to substitute the other. Some heuristics

were proposed by Lima and Travassos to decide which element to select first[37].

For exemplifying our approach, Table 3 presents the two first matrices. To improve the readability, the first column only presents an acronym of the architectural element's name, and the columns that have no dependency were omitted. Following the algorithm already presented, the influence factor (IF), and the late integration factor (LIF) of the first integration step were calculated. After each step of integration, we have defined the better integration order of the architectural elements, which is presented in the last column of Table 3. Notice that in the first step of integration, seven architectural elements have no dependencies (LIF = 0, not presented in Table 3). Those are the first elements to be integrated. Then in Step 2, two other elements had LIF = 0 (AEC and PC), which were the next components to be integrated. Finally, the order for integrating the UserInterface (UI) and MineralExtractorController (MEC) is defined only in the third step of integration, which is not presented in Table 3.

Table 3. Dependency Matrix Among Architectural Elements of the Mining Control System

	Step 1				Step 2				IF	Integration Order
	UI	MEC	AEC	PC	UI	MEC	AEC	PC		
UI	0	0	0	0	0	0	0	0	**0**	11th (Step 3)
MEC	1	0	0	0	1	0	0	0	**1**	10th (Step 3)
AEC	0	1	0	0	0	1	0	0	**1**	8th (Step 2)
PC	0	1	0	0	0	1	0	0	**1**	9th (Step 2)
ML	0	1	1	1	–	–	–	–	**3**	1st (Step 1)
AF	0	0	1	0	–	–	–	–	**1**	2nd (Step 1)
AE	0	0	1	0	–	–	–	–	**1**	3rd (Step 1)
ME	0	1	0	0	–	–	–	–	**1**	4th (Step 1)
P	0	0	0	1	–	–	–	–	**1**	5th (Step 1)
WF	0	0	0	1	–	–	–	–	**1**	6th (Step 1)
WL	0	1	0	1	–	–	–	–	**2**	7th (Step 1)
LIF	**1**	**8**	**5**	**7**	**1**	**2**	**0**	**0**	–	–

8.3.4 Generating Integration Test Cases

In this study our concern is to show how architectural formal models can be used for generating test cases. For the sake of conciseness, we sketch here how to use existing techniques to obtain integration test cases. In the particular point of view of this paper, the objective of these test cases is to exercise the interactions between the implementation of architectural elements, through the operations at their interfaces, to identify mismatches related to the flow of exceptions between architectural elements.

First, we need to identify the invocation sequence of operations associated to different interfaces. This can be obtained from the CSP specification, which gives the synchronization sequence of internal and architectural events. From this information we can construct an *interaction graph* from which test cases can be derived for each integration step, as identified in Subsection 8.3.3.

In this graph, operations and the respective possible returns (normal return and exceptions) are represented as nodes. The internal and architectural events (requests and responses) that refer to these operations are represented as edges. When it is an operation request (`ifte...request` in CSP), the edge points to the operation that is being requested; when it is an operation response (`ifte...normalResponse` or `ifte...abnormalResponse`), the edge points to the returned value. The interaction graph also has a start node, which requests the first operation that starts the interaction, and many final nodes, which represent the possible responses of the operation under test. After the interaction graph has been constructed, the test cases can be identified: each path (from the starting node to a final node) is considered a test case for integration testing. But since we focus on the system abnormal behaviour, only the paths that have at least one exception node are considered.

9 Case Study Evaluation

Overall, this case study has shown how a fault-tolerant architectural abstraction based on exception handling can be used to formally specify software architectures. The proposed approach has provided the means to apply model checking to verifying key properties of the architecture, and to generate architecture-based integration test cases for assessing the correctness of the source code against the architectural specification. The rigorous process has helped to identify and correct design faults of the architecture in earlier stages of the software development. Although most of these problems were simple to correct, if they were left to be corrected in the later phases of the development, it would have been much harder.

Regarding the iFTE architectural abstraction, the modelling of the mining control system was made easier because the building blocks of its architecture were based on the iFTE. The reuse of the formal models for specifying architectural elements and configurations have been facilitated, since only 13% of their formal specification, approximately, is affected by changes. So, 87% of the templates presented in Subsections 7.1 and 8.1 were reused straight forward during specification of the application. For example, the information

updated for the architectural elements were the names of interfaces, operations and exceptions (lines 4∼6 of Fig.10) and initialization of variables (lines 23∼31 of Fig.10). The pre-conditions and behaviour of the B-Method operation (not presented in Fig.10) where preserved. Moreover, most of the formal properties were also reused and the modified properties where instantiated from a template, in order to represent specific interfaces, operations and exceptions.

Besides the reuse of the formal model, it was also clear that the explicit separation between the structural (B-Method) and behavioural (CSP) specifications has facilitated the specification of architectural elements based on the iFTE. Since the architectural elements have their own B-Method machines, this enabled to represent a software architecture containing elements based on different abstractions, e.g., the explicit inclusion of B-Method machines for iFTEs and non-iFTEs as shown in Fig.12. On the other hand, the use of CSP for specifying scenarios has allowed to represent complex abnormal architectural scenarios, e.g., the propagation of the MethaneHighPumpOnException exception from the PumpController to the MineralExtractorController, followed by an exception handler that forces a second try to turn the pump off before raising an alarm.

Regarding the adaptability of the software architecture, if we reuse an architectural element into another architecture, the impact of change varies from a case to another. For example, the system's functionalities (normal behaviour), which can be related to the business domain, would be completely reused; though changes might be necessary in the way exceptions are propagated and handled (abnormal behaviour). The iFTE abstraction used in this paper make this kind of adaptation easier. First, the explicit separation of concerns between normal and abnormal interfaces concentrates the focus of change on abnormal interfaces. Second, the existence of a component with the role of implementing the exception handling (Abnormal component) facilitates the update of exception handlers. Third, the Provided and Required components can be used as internal adapters in order to reuse the Normal component with no modifications.

Regarding the formal verification of the software architecture, first of all, we notice that the properties of interest initially identified for the architectural abstraction could be re-used on different architectural elements, thus facilitating the verification process. Moreover, since each architectural element can have an internal architecture, the software architecture can be incrementally specified and verified, which helps to control the complexity and improves the scalability of the verification approach. Another advantage comes from the combination of B-Method and CSP, which facilitates the verification of both structural and behavioural properties, e.g., properties regarding the exception control flow and handling. Regarding the verification of the abnormal behaviour, the existing approaches cannot handle certain properties which are handled by the proposed approach, especially properties related to architectural scenarios of exception control flow and exception handling involving architectural reconfiguration.

Finally, for evaluating the scalability of the proposed solution, we have compared our results with the results obtained when verifying the same system using the Aereal framework[11]. Although in both cases the verification of the software architecture has been concluded, the approach presented in this paper has used approximately 50% less memory. The scalability improvement was a consequence of two characteristics of the proposed solution. First, the explicit behavioural specification in CSP restricts the architectural scenarios in which the formal model is verified, thus reducing the state space used during verification. Second, the use of architectural abstractions has reduced the number of architectural elements, since it allows the specification of composed architectural elements as single elements.

10 Conclusions and Future Work

This paper has presented a rigorous development approach for developing and testing fault-tolerant software architectures. The focus of the paper is on an integrated solution involving formal specification, verification and test of exception control flows and handlers at the software architecture. This approach is built around the concept of architectural abstractions, which intends to improve the control of the architecture complexity and the scalability of the proposed solution. An architectural abstraction has been presented: the idealised fault-tolerant architectural element (iFTE), an architectural abstraction based on exception handling mechanism, which is used for structuring dependable systems at the architectural level.

Regarding the formal representation, the proposed approach combines the use of B-Method and CSP for representing exception types, exception control flows and handlers in terms of architectural scenarios. The architectural scenarios and properties are also used for generating unit, integration and robustness test cases, which aims to assess the consistency between the verified software architecture and the final source code of the application. This assessment is necessary, since even when the software architecture is formally verified,

the consistency of the final source code with the software architecture is not guaranteed.

The case study used for evaluating the overall approach has showed that the proposed approach provides an appropriate solution for modelling, analysing and testing fault-tolerant software systems at the architectural level.

A limitation of the proposed solution is the assumption that the communication between architectural elements follows a call-return protocol. Although this assumption simplifies the specification of software architectures, it lacks concurrency, which is essential in the context of a wide range of applications. To overcome this limitation, current work is looking on how to adapt our solution to support the specification of event-based software architectures. For this, the new formal template should resolve concurrent exceptions and coordinate scenarios involving the execution of concurrent components. Moreover, the concepts of architectural abstractions and scenarios presented in this paper should provide considerable benefits in the context of Model-Driven Architecture (MDA). First, since abstractions and scenarios can be specified in different levels of details, their refinement can guide the transformation from a platform-independent model to a platform-specific model. To support MDA, it is necessary to develop tools for automatic model transformation, including the generation of source code. Regarding model-based testing, the proposed approach lacks on the generation of testing data. In order to supply such limitations, it would be necessary to use complementary approaches such as mutation testing[38].

Acknowledgment The authors would like to thank the anonymous referees for the valuable feedback to improve the paper's quality and contribution.

References

[1] Cristian F. Exception Handling. Dependability of Resilient Computers, Anderson T (ed.), Blackwell Scientific Publications, 1989, pp.68–97.

[2] Gray J, Reuter A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[3] Parnas D L, Würges H. Response to undesired events in software systems. In *Proc. the 2nd Int. Conf. Software Engineering*, San Francisco, USA, October 1976, pp.437–446.

[4] Castor Filho F, Cacho N, Figueiredo E, Ferreira R, Garcia A, Rubira C M F. Exceptions and aspects: The devil is in the details. In *Proc. the 14th ACM SIGSOFT FSE*, Portland, Oregon, USA, November 5–11, 2006, pp.152–162.

[5] Reimer D, Srinivasan H. Analyzing exception usage in large Java applications. In *Proc. Workshop on Exception Handling in Object-Oriented Systems* (*ECOOP'2003*), Darmstadt, Germany, July 21–25, 2003, pp.10–19.

[6] Rubira C M F, de Lemos R, Ferreira G, Castor Filho F. Exception handling in the development of dependable component-based systems. *Software – Practice and Experience*, March 2005, 35(5): 195–236.

[7] Bass L, Clements P, Kazman R. Software Architecture in Practice. 2nd Edition, Addison Wesley, 1999.

[8] Bradbury J S. Organizing definitions and formalisms for dynamic software architectures. Technical Report 2004-477, School of Computing, Queen's University, March 2004.

[9] de Castro Guerra P A, Rubira C M F, de Lemos R. A Fault-Tolerant Software Architecture for Component-Based Systems. *Architecting Dependable Systems*, *LNCS* 2677, Berlin, Germany: Springer, 2003, pp.129–149.

[10] de Lemos R, de Castro Guerra P A, Rubira C M F. A fault-tolerant architectural approach for dependable systems. *IEEE Software*, 2006, 23(2): 80–87.

[11] Castor Filho F, Brito P H S, Rubira C M F. Specification of exception flow in software architectures. *Journal of Systems and Software*, 2006, 79(10): 1397–1418.

[12] Castor Filho F, Brito P H S, Rubira C M F. A Framework for analyzing exception flow in software architectures. *SIGSOFT Software Engineering Notes*, 2005, 30(4): 1–7.

[13] Abrial J R, Lee M K O, Neilson D, Scharbach P N, Sorensen I. The b-method. In *Proc. the 4th Int. Symp. VDM Europe on Formal Software Development* (*VDM'91*), Noordwijkerhout, the Netherlands, Oct. 21–25, 1991, Vol.2, pp.398–405.

[14] Brookes S D, Hoare C A R, Roscoe A W. A theory of communicating sequential processes. *J. ACM*, 1984, 31(3): 560–599.

[15] Leuschel M, Butler M J. Prob: A model checker for b. In *Proc. Int. Conf. Formal Methods* (*FME'2003*), *LNCS* 2805, Pisa, Italy, Sept. 8–13, 2004, pp.855–874.

[16] Patrick H S Brito, Camila Ribeiro Rocha, Fernando Castor Filho, Eliane Martins, C M F Rubira. A method for modeling and testing exceptions in component-based software development. In *Proc. the 2nd Latin American Symposium on Dependable Computing* (*LADC 2005*), *LNCS* 3747, Salvador, Bahia, Brazil, Oct. 25–28, 2005, pp.61–79.

[17] F Castor Filho, P A de C Guerra, V A Pagano, C M F Rubira. A systematic approach for structuring exception handling in robust component-based software. *Journal of the Brazilian Computer Society*, April 2005, 10(3): 5–19.

[18] Randell B. Turing memorial lecture facing up to faults. *Comput. J.*, 2000, 43(2): 95–106.

[19] Laprie J C, Arlat J, Béounes C, Kanoun K. Definition and analysis of hardware- and software-fault-tolerant architectures. *IEEE Computer*, 1990, 23(7): 39–51.

[20] Anderson T, Lee P A. Fault Tolerance: Principles and Practice. Prentice-Hall, 1981.

[21] Taylor R N, Medvidovic N, Anderson K, Whitehead J E J, Robbins J. A component-and message-based architectural style for GUI software. In *Proc. the 17th Int. Conf. Software Engineering*, Seattle, Washington, USA, April 1995, pp.295–304.

[22] F Castor Filho, Guerra P A de C, C M F Rubira. An architectural-level exception-handling system for component-based applications. In *Proc. the 1st Latin-American Symposium on Dependable Computing*, *LNCS* 2847, São Paulo, Brazil, Oct. 21–24, 2003, pp.321–340.

[23] Clements P *et al.* Documenting Software Architectures: Views and Beyond. Addison-Wesley, 2003.

[24] Kevin Simons, Judith A Stafford. Cmeh: Container managed exception handling for increased assembly robustness. In *Proc. the 7th Int. Symp. Component-Based Software Engineering* (*CBSE'04*), *LNCS* 3054, Edinburgh, Scotland, May 24–25, 2004, pp.122–129.

[25] Chang B M, Jo J W, Yi K, Choe K M. Interprocedural exception analysis for Java. In *Proc. the 2001 ACM Symp. Applied Computing* (*SAC'01*), Las Vegas, USA, March 11–14, 2001, pp.620–625.

[26] Schaefer C F, Bundy G N. Static analysis of exception handling in ada. *Softw. Pract. Exper.*, 1993, 23(10): 1157–1174.

[27] Siau K, Halpin T A (eds.). Unified Modeling Language: Systems Analysis, Design and Development Issues. Idea Group, 2001.

[28] Bertolino A, Inverardi P, Muccini H, Rosetti A. An approach to integration testing based on architectural descriptions. In *Proc. the Third IEEE Int. Conf. Engineering of Complex Computer Systems* (*ICECCS'97*), Washington DC, USA, IEEE Computer Society, 1997, pp.77–85.

[29] Muccini H, Bertolino A, Inverardi P. Using software architecture for code testing. *IEEE Trans. Softw. Eng.*, 2004, 30(3): 160–171.

[30] Richardson D J, Wolf A L. Software testing at the architectural level. In *Proc. Int. Workshop on Multiple Perspectives in Software Development* (*Viewpoints'96*) *on SIGSOFT'96 Workshops*, New York, NY, USA, ACM, 1996, pp.68–71.

[31] Sloman M, Kramer J. Distributed Systems and Computer Networks. Hertfordshire: Prentice Hall International (UK) Ltd., UK, 1987.

[32] de Lemos R. Architectural Fault Tolerance using Exception Handling. Architecting Dependable Systems IV, *LNCS* 4615, Springer, 2007, pp.142–162.

[33] Binder R V. Testing Object-Oriented Systems: Models, Patterns, and Tools. Boston: Addison-Wesley Longman Publishing Co., Inc., MA, USA, 1999.

[34] Bertolino A, Marchetti E, Muccini H. Introducing a reasonably complete and coherent approach for model-based testing. *Electr. Notes Theor. Comput. Sci.*, 2005, 116: 85–97.

[35] Perez I, Martins E, Viégas J. Using UML models for component test. In *Proc. the 8th Brazilian Workshop on Test and Fault Tolerance* (*WTF 2007*), Belém, Pará, Brazil, 2007, pp.99–102. (in Portuguese)

[36] Stafford J A, Wolf A L. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 2001, 11(4): 431–451.

[37] Lima G M P S, Travassos G H. Integration testing applied to object-oriented software: Heuristics for class ordering. Tech. Rep. ES-632/04, COPPE/UFRJ, 2004. (in Portuguese)

[38] Abreu B, Martins E, Sousa F. Generalized extremal optimization: A competitive algorithm for test data generation. In *Proc. the 21st Brazilian Symposium on Software Engineering* (*SBES 2007*), Joâo Pessoa, Paraíba, Brazil, Oct. 15–19, 2007, pp.342–358.

Patrick H. S. Brito received the B.S. degree from Federal University of Alagoas - Brazil in 2003, and the M.S. degree from University of Campinas - Brazil in 2005. Currently, he is working toward the Ph.D. degree in the Institute of Computing, State University of Campinas (UNICAMP), where he is a member of the Software Engineering and Dependability Research Group under the supervision of Prof. Cecília Rubira. His research interests include dependable software development methodologies, exception handling at the software architecture, and component-based software development.

Rogério de Lemos is a lecturer in computing science in the University of Kent, UK. Before joining the Computing Laboratory at the University of Kent he was a senior research associate at the Centre for Software Reliability (CSR) in the University of Newcastle upon Tyne, UK. He has over 50 scientific publications in international journals, book chapters and conferences, and recently co-edited five books on Architecting Dependable Systems, and the sixth volume is under preparation. His main research areas are in software architectures for dependable systems, self-adaptive software systems, and software development for safety-critical systems.

Cecília M. F. Rubira is an associate professor of the Institute of Computing at State University of Campinas (UNICAMP), Brazil. She received her Ph.D. degree in computing science in 1994, from the Department of Computing Science in the University of Newcastle upon Tyne, UK. Her current research interests are fault tolerance, exception handling, software architectures, and component-based software engineering applied to the development of dependable object-oriented software. She has co-authored more than 50 scientific papers, book chapters, and books in these areas, and also supervises a number of M.Sc. and Ph.D. students at UNICAMP.

Eliane Martins received her Ph.D. degree in computer science by École Nationale Supérieure de l'Aéronautique et de l'Espace (ENSAE) and Laboratoire d'Automatique et d'Analyse de Systèmes (LAAS), Toulouse, France, 1992. She is at the Institute of Computing of the State University of Campinas (UNICAMP) since 1993, and in 2001 she became associate professor. Her research interests include conformance and robustness testing of software, especially, communication protocols and embedded systems. For robustness testing purposes, the use of fault injection is the main concern. She is also interested in methods and tools for model-based test case generation. She spent a sabbatical year at the Institut National des Télécommunications (INT) at the Department of Software and Networks (LOR), in Evry, France.

Appendix A Overview of the Formal Notations

This section presents an overview of the formal notations used in this paper.

A.1 B-Method Machines

B-Method is a state-based method developed by Abrial for specifying, designing and coding software systems. It is based on set theory with the axiom of choice. Sets are used for data modelling, thus allowing the definition of personalised types according to what is being modelled. B-Method has been used in industry with some success, in a number of applications ranging from the development of control systems to smart cards. As all formal methods, the B-Method provides a formal language to describe systems, allowing for analysis and verification of certain system properties prior to implementation. An important characteristic of B is that it covers the whole development process, from specification to implementation. In the context of this paper, we have used only the specification and verification of B-Method, not the implementation.

```
1   MACHINE counter
2
3   VARIABLES
4       value
5
6   INVARIANT
7       value : INT
8   INITIALIZATION
9       value : =0
10
11  OPERATIONS
12      inc() =
13      BEGIN
14          value := value +1
15      END;
16
17      add(number)=
18      PRE
19          number : INT
20      THEN
21          value := value + number
22      END;
23
24      output <-- getValue()=
25      BEGIN
26          output := value
27      END
28  END
```

Fig.A. Example of a B-Method machine.

B-Method specifications are centred on the notion of abstract machine. Abstract machines are the units of modularisation of specifications in B-Method, and resemble modules of imperative languages. An abstract machine encapsulates data and behaviour. Data are stored in terms of *variables*, and behaviour in terms of *operations*. Fig.A presents an example of an abstract machine in B-Method. This machine has a single variable (`value`), and three operations: (i) `inc()`, which increments one in the value of the `value` variable; (ii) `add(number)`, which increments "number" in the value of the variable, and (iii) `output` ← `getValue()`, which returns the value of the variable. The type of the variable is defined in terms of *invariants*. Besides defining the variable types, invariants might contain other statements in order to indicate which properties must be maintained throughout the lifecycle of the abstract machine.

Operations are specified by means of pre-conditions and multiple assignments. Pre-conditions can be used to define general rules that should be valid before executing the operation. When there is no precondition (operations `inc()` and `getValue()`), it is assumed a "true" value (logical tautology). Assignments are used to change the value of variables, thus updating the state of the machine. For example, lines 14 and 21 of Fig.A changes the value of the machine's variable. When executing an operation, it is required that it preserves the invariants specified for the machine. Operations might have parameters and provide a return. While the types of parameters are specified as pre-conditions (line 19 of Fig.A), the type of return is defined as an assignment into the operation (line 26 of Fig.A). In the example of the `add(number)` operation in Fig.A (line 17) a parameter is passed containing an integer value (INT). Examples of other symbols for defining more complex types are presented in Table A. These definitions can be used into invariants and preconditions, in order to define logical expressions, mathematical relations, power sets, etc.

Table A. Some Symbols for Defining Types in B-Method

Symbol	Meaning	Symbol	Meaning
POW(A)	Powerset of A	A −−> B	Total Function from A to B
A +−> B	Partial Function from A to B	A B	A union B
A * B	Cartesian Product of A and B	seq(A)	Sequence of Elements of A

A.2 Communicating Sequential Processes (CSP)

Communicating sequential processes (CSP) is a

process algebra that describes patterns of communication by algebraic expressions. The basic element of behaviour and communication in CSP is an *event*. Events may be atomic, or they can have associated data. For example `evt1` and `evt2` are atomic events, while `evt3!5` and `evt4?x` represent events that output the value 5 and input a value represented by x respectively. A *process* is the unit which associate related events. The simplest process, `STOP`, is the one that engages in no events. Another useful process is `SKIP`, representing successful termination.

When associating events together, processes can define either sequential execution or alternative executions. A process can define sequential events by using the arrow operator (−>), in such a way that if `PROCESS1 = (evt1−> evt2−> evt3!7)`, `PROCESS1` defines an exact sequence of events: first `evt1`, followed by `evt2` and then `evt3` outputting the number 7. A process can also define sequences of other processes; for this, it is necessary to define a list of processes separated by the semicolon operator (`;`). For example, process `PROCESS2 = (P1;P2)` behaves like `P1` until `P1` executes a final successful event (`SKIP`), at which point it behaves like `P2`.

Besides specifying sequential execution, CSP also allows the specification of alternative choices. For this, the alternative sequences should be separated by the choice operator (`[]`). For example, if `PROCESS3 = (PROCESS1 [] (evt5 −> evt6))`, the first event to be executed could be either `evt1` or `evt5`; but after choosing one of them, the respective sequence would be followed.