Towards On-Line Adaptation of Fault Tolerance Mechanisms

Jean-Charles Fabre[1,2], Marc-Olivier Killijian[1,2], Thomas Pareaud[1,2 *]
[1] CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France
[2] Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France
Jean-Charles.Fabre@laas.fr, Marco.Killijian@laas.fr, Thomas.Pareaud@astrium.eads.net

***Abstract*—In this paper, we address the crucial issue of online software adaptation: how to determine if the system is in an adaptable state? To solve this issue, we advocate the use of both Component-Based Software Engineering (CBSE) and reflective technologies. Such technologies enable a metamodel of the software architecture to be established to represent both structural and behavioral aspects. Based on some requirements expressed by the software designer, and using online animation of the software model (CBSE metamodels and Petri Nets), we propose an approach to 1/decide when the system (or a sub-system) is adaptable, and 2/ guide the system towards an adaptable state. Finally, this approach is applied to the online adaptation of replication mechanisms on a small case study.**

Keywords-component: fault-tolerance, online adaptation, reflection, component-based software engineering.

I. INTRODUCTION

Up to now, adaptation of dependable systems has mostly been investigated off-line, using software engineering techniques. However, online adaptation has received less attention and is a very challenging problem. Indeed, applications and dependability mechanisms are tightly coupled. Henceforth, a change of the application software may lead to a necessary change of its attached fault tolerance mechanisms; a change in the operational conditions (environmental conditions and resources, fault nature and error rates in operation) may necessitate updating the fault tolerance mechanisms. This means that the connection between the application and the dependability mechanisms should be easy to handle and that the fault tolerance software should be easily updatable on-line, without disturbance of the application and its dependability.

The on-line evolution of fault tolerant systems has essentially been addressed using simple approaches, i.e. for adapting the number of replicas [1, 2, 3, 4] or for changing strategies [5, 6, 7, 8]. In such works, the approach is often based on highly parameterized software and/or on several predefined degraded modes of operation. Among other examples, the rate, the compression, the ciphering of checkpoints and/or the number of replicas are usual parameters for replication mechanisms. The use of pre-defined modes of operation means that all variants of the fault tolerance software are available on-line and thus that they must be defined in advance and loaded into the system. Therefore, with such simple approaches, the overall flexibility of the system is impaired.

Our approach relies on reflective software architectures and today's component-based software engineering [9]. Together, these technologies enable design for adaptation to be achieved by providing separation of concerns and fine-grain decomposition of the software, respectively. The work proposed in this paper focuses on the synchronization of the software adaptation with respect to its execution, i.e. the problem of choosing when to perform the adaptation.

This paper is organized as follows. In Section II we describe the problem statement and give an insight of the proposed approach. In Section III we describe for proposed framework for adaptation. The focus of Section IV is the modeling of the software execution in order to perform adaptation. Section V is devoted to the identification of suitable states of execution regarding adaptation. The implementation of the adaptation process is described in Section VI. Section VII illustrates on a simple case study the feasibility of our approach to adapt fault tolerance software.

II. PROBLEM STATEMENT AND OVERALL APPROACH

Long living systems usually have strong dependability requirements, in particular regarding availability and evolvability. The European Network of Excellence ReSIST[1] defines evolvability during operation as "*the ability to dynamically adjust system behaviour and potentially its functional architecture and deployment to cater for new operational contexts, including operational faults and attacks, new threats, and dynamic changes to the system environment*". In this context, mastering the online adaptation of the software in general, and of the fault-tolerance mechanisms in particular, is necessary. The updating of the fault tolerance software should thus be performed during the lifetime of the system according to: changes in the application software or in the operational conditions, availability of new or updated software.

This problem can be addressed today using new software engineering technologies that bring together both reflective and aspect oriented programming, with component-based software engineering (CBSE). Examples of these technologies are OpenCOM [10, 11] and Fractal [12] to name just a few.

These novel technologies provides new means to enforce:

- **Separation of concerns** between the application, the fault-tolerance mechanisms and the adaptation software;
- **Componentization** of the fault tolerance software that can then be easily manipulated at runtime using reflective mechanisms.

[*] This work was carried out when Thomas Pareaud was PhD student at LAAS. He is presently with EADS-ASTRIUM SAS, Toulouse, France.

[1] The Network of Excellence ReSIST, Resilience for survivability in IST (http://www.resist-noe.org/)

- Availability of a **runtime component model** that reflects the current software implementation in order to reason about the software components configuration.
- Availability of a **behavioural model** reflecting the control flow of the software in order to synchronize adaptation with the current software execution.

These design principles ease the adaptation process: thanks to separation of concerns and componentization, the modification necessary for the adaptation can be restricted to a small subset of the components. The availability of the component model will help determining at runtime the exact subset of the components that need to be modified (i.e. *what*). The behavioral model of the computation will then help us synchronizing the modifications with the control flow (i.e. *when*). Additionally, reflection enables the manipulation of the state of the components, in order to control their initialization during the adaptation process. Finally, fine-grain control of the dependability properties of the adaptation process needs is also desirable.

In this paper we focus on the synchronization of the modifications with the software activity. We address the problem in general first. How to determine when the software can be adapted? At the component level, how to control the computation in order to guide the software towards an adaptable state? Then, we show the interest of the approach for evolving fault tolerant systems and illustrate the feasibility on a small case study.

III. A MIDDLEWARE FRAMEWORK FOR ADAPTATION

A. *Framework overview and principles*

The architecture proposed in Figure 1 gives the big picture of a reflective framework targeting on-line adaptation of the fault tolerance software. We assume that the application software and its attached fault tolerance mechanisms are developed using CBSE.

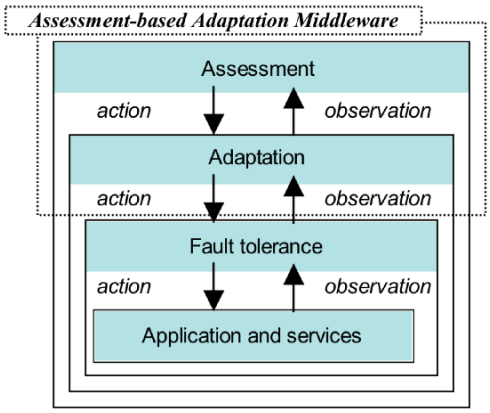

Figure 1. Overall adaptation framework

Figure 1 shows the various abstraction levels that can be defined: application, fault-tolerance, adaptation and assessment. The assessment provides triggers to the adaptation level that performs actions on the fault tolerant application, in particular updating the fault tolerance software. Design of reflective levels is often debatable. Why putting assessment above adaptation and adaptation on top of fault tolerance, etc? The reason is the **control**. In this framework, assessment controls the adaptation, adaptation controls the fault-tolerance mechanisms and fault-tolerance controls the application. In this paper we do not address assessment issues. We assume that such monitoring-based facilities are available to decide when adaptation is required.

As mentioned above, the fault tolerance software must be developed as a reflective component-based architecture and the software configuration can thus be observed and modified at runtime through the exposition of meta-data and control operations. A complete example of a component-based design of several fault tolerance strategies based on replication can be found in [13].

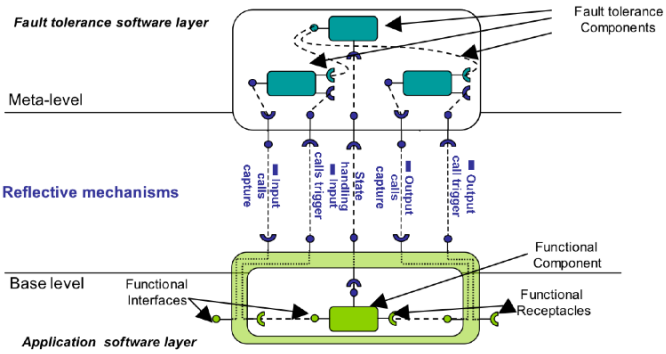

Figure 2. Wrapping component using CBSE

Figure 2 shows the wrapping of a component using a reflective protocol implemented using observation and control features provided by a CBSE like OpenCOM. The reflective mechanisms enable (i) input and output requests to be intercepted and rerouted to the wrapper, (ii) the call to be triggered and (iii) the state to be captured.

A fault tolerance strategy can be connected to the application using these mechanisms. The set of fault tolerance strategies (e.g. replication protocols variants) and their companion services (detection by mutual surveillance, atomic multicast protocol, etc.) form the basis of a fault tolerance middleware. Each actual fault-tolerance strategy corresponds to a set of basic components assembled and connected together in a specific setting, i.e. there's a mapping between a strategy and a component architecture. The set of components corresponding to the software architecture at a given point in time is called a *configuration*. The middleware knows the configurations for various fault tolerance strategies. For instance, two replication strategies (passive and active) can be componentized: the corresponding configurations share some of the components (e.g. group communication, cloning) while some others are specific (e.g. replica synchronization, recovery).

B. *Overview of the adaptation process*

Online adaptation does not only imply consistent updates of the software from a structural viewpoint (i.e. components and bindings) but also from a dynamic viewpoint (i.e. consistent modification regarding the execution in progress).

The major steps of on-line adaptation are the following:

1. For some reason, the assessment layer triggers a signal for requesting an adaptation.
2. The adaptation layer identifies the most appropriate fault-tolerance strategy with respect to the trigger, to the available resources and to the environmental conditions.
3. The current configuration of the replication strategy is compared to the configuration of the target strategy. This leads to a list of component-level modifications in the current configuration: existing components disconnections and removal, new components instantiation and connection.
4. The execution of the system must reach an adaptable state, defined as a state of the computation where the component-level changes identified in step 3 can be done in a consistent manner.

Adaptation can be seen as a process that is concurrent with the execution of the fault tolerant software. Thus, the modification of a given component cannot be done anytime. Looking more carefully to the control flow, it is mandatory to identify states where the adaptation can be performed and other states where it is not possible to adapt. Clearly, the modification of the software configuration during execution process could introduce failures.

As mentioned above, the steps 1 and 2 are beyond the scope of this paper. In the sequel, we address step 3 in Section III.C. Based on assumption on the system model given in Section III.D, we address step 4 in both Sections IV (model) and V (control).

C. *Componentized software manipulation*

The manipulation of the componentized software implies updating the components and bindings in a component architecture. This problem is not new [14] and is not the main focus of this paper. We use a graph modeling of the component architecture for this purpose. Lets take the simple example architecture given in Figure 3. This configuration can also be represented as a graph, as shown on Figure 4.

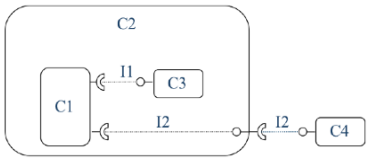

Figure 3. Componentized software simple example

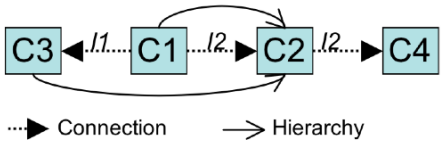

Figure 4. Simple structural model

The solid arrows refer to composition (i.e. nesting) whereas the dashed arrows refer to bindings between receptacle and interfaces. Each node of the graph corresponds to a simple descriptor of an active component. The description of a given software configuration has been done in XML. From the comparison of two configurations, it is possible to derive the operations to be done on-line to transform the current configuration into the target one. The component model implementation provides means to retrieve components interfaces and to update bindings. Thus, the modifications of a configuration are easy: (i) the graph representing the first configuration is modified to reach the graph representing the second one; (ii) the reflective mechanisms provided by the component model implementation are used to perform such modifications on the actual components.

In the example given in Figure 3 for instance, if an adaptation requires replacing the component C4 by a component C5, the following actions have to be done:

- Disconnect C4 from C2;
- Instantiate C5;
- Connect C5 interface to C2 receptacle.

The current component configuration, in fact the graph of components, is available at runtime and is maintained up-to-date to reflect software modifications. In summary, the update of the software configuration is based on the following information:

- interfaces, receptacles and service parameters for each component;
- connection and composition links, parameters of current components at the architectural level.

The graph is a data structure of the adaptation software that is updated according to the modifications performed on the current system configuration.

D. *System and task model assumptions*

We assume that the execution of a request is performed by a *task*. A task execution spans software components belonging to both the application and the fault tolerance layers. Multiple requests can be executed at the same time leading to the *concurrent execution* of tasks within components. The tasks start from an *initial state* and we assume that they *eventually terminates*. The runtime behavior can be represented by a cyclic graph representing the task control flow. When a task terminates, its control flow returns to the initial state, and the task is ready to process a new request. Components are state-full, i.e. tasks can manipulate data over several request processing.

The control structures belonging to the tasks' execution path are quite conventional: sequences, alternatives, loop, internal function call, interactions with other components (i.e. service calls through receptacles and interfaces).

Finally, we assume that concurrency on shared objects between several tasks running within a given component is handled by simple synchronization mechanisms like mutexes or semaphores.

IV. MODELLING EXECUTION AND ADAPTATION

A. *Basic principles and properties*

The runtime modification of the structural model must be done in a consistent manner with respect to the execution flow through the components. The modification can only be carried out when the changes do not introduce inconsistent behavior of the computations in progress. This leads to

define the notion of *Suitable Adaptation States (SAS)* that represent a state in the computation where the modification of the components can be performed safely. To identify such states, we need a model of the computation that reflects the control flow within the current configuration. To maintain consistency of the fault tolerant application and reach the adaptation objectives, some properties must be observed:

- ***Isolation*** *of the components to be modified*: The components to be modified must be suspended and detached from the current configuration and the new ones need to be connected and activated.
- ***Convergence*** *to a Suitable Adaptable State*: The fault tolerant application will eventually reach a state where the modification can be done without violating the specifications (e.g. due to a component modification during execution).
- ***Liveness*** *of the fault tolerant software execution*: The adaptation process that is concurrent with the execution of the fault tolerant software does not introduces a deadlock or a livelock in the computation.
- ***Conformance*** *with configurations expected behavior*: The conformance is defined from a specified behavior, being the initial one or a new one required after adaptation. A correct adaptation process does not introduce unspecified behaviors.

To perform adaptation following the above-mentioned properties, the behavioral model must contain a description of the component in which tasks (e.g. threads) are executed (required for *isolation*). It must also exhibit the synchronization mechanisms since they are a potential source of deadlocks and livelocks (required for *liveness*).

We assume that the initial state of a task is a *Suitable Adaptable State*. Because of the cyclic task behavior of a task and the assumption that it eventually terminates (cf. section III.C), a *Suitable Adaptable State* is eventually reached by any task (required to ensure *convergence*). Last but not least, the model must include the observations of events at the interfaces of the components to trace the interactions (required for *conformance*).

The behavioral model that we propose represents the control flow within the componentized software architecture and fulfills these requirements.

B. Individual computational model

Petri Nets offer good abstractions to represent the global behavior of the component-based system, i.e. the control flow of parallel tasks within a component that can interact through shared objects or events/messages.

The first objective of the modeling is to ***capture the control flow*** in progress within the components. Roughly speaking, we do not need a detailed modeling of the internal computation of the components. We only need to know that a request has been received and that its processing has not been completed yet. This can happen either because the request is currently being processed by the component or because the processing has involved a request to another component (a nested call). This objective can be solved easily by using a request counter that is incremented when the component receives a request and decremented when it issues a reply. The value of the counter indicates the number of request processing in progress in the component.

A second objective of the modeling is to ***represent causality*** among parallel tasks within a component. This causality can be due to two reasons: the tasks share a common resource, or the tasks exchange events or messages. To avoid inconsistencies, two interacting tasks or components should produce and consume events (messages) in the same configuration. Such causal dependency must be obeyed to prevent inconsistencies such as, a message is sent and never processed, an event is raised and captured in a new state where it has no meaning, etc. Solving this second objective is not as easy and depends pretty much on the design of the requests processing and the internals of the components. This is why we rely for this on the component designer. The designer provides us with a model of the concurrency within each component. See, for example, the model of component C3, in Figure 5, where a semaphore is used to synchronize two tasks.

The construction of a model for a given task is based on elementary patterns such as a sequence of actions and control structures like alternatives and loops. It is worth noting that such elementary patterns only have to appear at the behavioral model level when they have an observable effect at the component interface. For instance, a loop in which no interaction is performed does not need to be represented. For this reason, the model is much simpler than a detailed behavioral model in which all execution details are represented. However, when a loop embeds interactions, it can be modeled independently as a cyclic task.

In summary, the above model is necessary and sufficient to address adaptation of independent tasks but, as soon as **concurrent tasks** are considered, it is necessary to represent synchronization among tasks. To this aim, we introduce a place and two transitions for each synchronization object (mutex, semaphore): when the place is marked, the object is free; when it is empty, the object is locked. The ingoing transition corresponds to the release and the outgoing to the acquisition of the object.

C. Execution model

Each individual component in a configuration has its own Petri Net model representing both its interactions with other components and causality among concurrent internal tasks. The execution model of the computation within a complete configuration is the result of the aggregation of the execution models of the individual components. As shown in the example given in Figure 5, the tasks span several components belonging to a configuration.

At runtime, the capture of component interactions animates this global model. The individual component models are locally animated by the capture of incoming request and of synchronization actions. More details about the implementation can be found in Section VI. This animation of the model results in the marking of the Petri Net according to the current state of the system.

A marking of the Petri net represents an execution state at a given instant in time. However, as we will see in Section

V, establishing an adaptable state requires the knowledge of past and possible futures of the current tasks. The history of the fired transitions since the initial state conveys the notion of past and future execution states:

- the **past** is composed of the set of transitions triggered from the initial state
- the **future** is a sequence of transitions that **can be** fired from the current state. Of course, from a state, several futures are possible.

Keeping the complete history (initial marking and complete sequence of fired transitions) of the system could be very costly. However, since we consider only cyclic tasks, when a task is back to the initial state, its history can be forgotten to reduce the required memory space.

D. *Constructing a meta-model: an example*

In order to illustrate the design and the use of the execution model, let's take the example given in Figure 5. This architecture is composed of 4 components, `C1`, `C2`, `C3` and `C4`, and can be interpreted like this:

- `C1` is a *client* of a *service* provided by `C4`.
- `C3` is an *intermediate communication service*.
- `C2` is a *monitoring service* of the system.

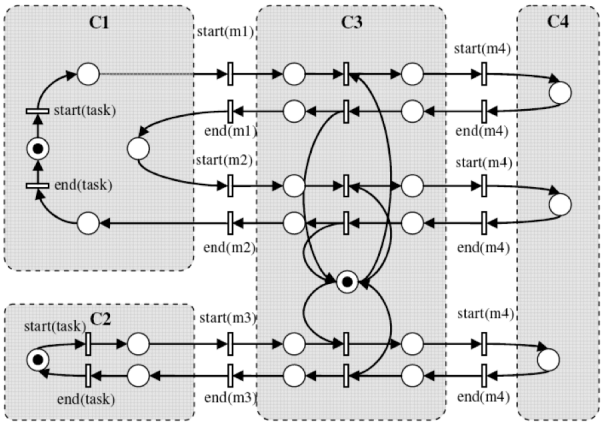

Figure 5. Execution profile with critical sections

In this example, we show two tasks both spanning several components, namely `C3` and `C4`. The complete execution model of the whole system is built when all components are bound to each other. When all components are distributed, i.e. each component is located on a different site, the metamodel (i.e. the structural and behavioral model) is divided into pieces. The metaspace of each site is populated with partial metamodels, those associated to the tasks running locally within the components.

To build this runtime metamodel we need to intercept the creation and destruction of components and their connection and disconnection. When a component is created, we check the number of tasks it includes and create the required number of metatasks. When a component is deleted, the corresponding metatasks data structures are also deleted. When two components are connected, each partial model is completed by the partial models of the tasks belonging to the connected component (e.g., `C1.task` model with `C3.task` model).

V. SUITABLE STATES FOR ADAPTATION

In the previous sections we have detailed how to build a suitable execution model to represent the behavior of an adaptable system. Now, in this section, we address the establishment of suitable states in which the modification of the component architecture does not introduce inconsistency (as defined in Section IV.A). In the following, we assume that we need to adapt the whole service shown in Figure 5, by replacing `C4` by an updated version `C5`.

A. *Case of independent tasks: locally adaptable states*

When the tasks of a component are independent, i.e. they do not need synchronization, suitable adaptation states can be identified easily. The adaptable state of the component architecture is defined as the Cartesian product of the adaptable states of the set of individual tasks.

The adaptable state of a given task is defined according to the isolation property mentioned earlier. Two cases can be identified:

- The task is located within a component that is going to be removed and is not interacting with other components in the software architecture. In this case the task is simply stopped when terminated, i.e. back into its initial state, and, henceforth, is in an adaptable state.
- The task has issued a call to a component that will be replaced (say C4 in our example). All calls in progress within C4 must be terminated to reach a suitable adaptable state.

The second case is obviously the most interesting, in fact the most general. We can observe there is a need for synchronization between the adaptation and the execution in progress in order to identify adaptable states.

We can identify 3 execution regions (later called **major states**) within the task execution (C1.task in our example):

- E1: the execution in progress within C3 has not yet reached the call to C4;
- E2: the execution in progress has terminated its activity within C4 and is back in C3;
- E3: all other cases covering the execution of the task within C4 or the situation where the execution has started within C4, implied nested calls in other components (possibly within C3 again) and, thus, is not terminated within C4.

Intuitively, both E1 and E2 represent suitable adaptation states. The component C4 can be isolated and its removal does not introduce any inconsistency. E3 represents the system states where the adaptation could introduce consistency problems. Looking more carefully to adaptable states E1 and E2 we can argue that:

- In state E1, the task will terminate with the new configuration of the system because the change in the configuration is done before it reaches the call to C4. The call will actually be issued to the new component C5. This implies that the task execution obeyed the

specification after adaptation when terminated. E1 represents the states that are common to both configurations before and after the modification.

- In state E2, the adaptation (switch from C4 to C5) is done after the task has terminated its execution within C4. This implies that the task execution obeyed the *specification before adaptation*. E2 represent the states that are not impacted by the modification for the current task execution.

In summary, the suitable adaptable states correspond to two situations that can be easily identified in the data structures implementing the metamodel. The state is adaptable when:

- The history of the task does not include any call to a component that must be replaced and/or modified;
- For all transitions in the history representing a call to a component to be replaced or modified, there exist a return call transition.

In all other cases the state of the system tasks is considered as non-adaptable, such cases corresponding to E3. It is worth noting that there could exist some adaptable states within E3. The major states E1, E2, E3 can be determined off-line by an analysis of the tasks' control flow within a cycle. In summary, a task cycle goes through E1 first, then E3 (interaction with a component to be updated) and finally E2.

Controlling the execution to maintain the system in a suitable adaptable state consist in placing a lock (*adaptation lock*) before entering E3, i.e. preventing temporarily a call to the replaced component. When all tasks are in a suitable adaptable state, the target component can be replaced.

Finally the Cartesian product of major states (E1, E2, E3) for every task in the system can be computed to give the big picture of the global adaptable states.

B. Case of concurrent tasks: globaly adaptable states

Concurrency among tasks located within a given component may impair the consistency when changes are made in the component architecture. Let's take a simple example to illustrate this fact.

Suppose two concurrent tasks within a component, say t_1 and t_2, having both a critical section protected by a shared mutex. The critical section *CS1* in t_1 does not interact with the rest of the world and thus is non-observable. In the second task t_2, we assume an interaction with a second component within the critical section *CS2*. When t_2 is running in CS2, it holds the mutex whereas t_1 is waiting. According to what we learnt from the previous section, t_2 can be blocked just to prevent entering the E3 state (just before the call to an external component). This means that t_2 is suspended in the critical section whereas it is in a suitable adaptable state (namely E1). If t_1 is not in an adaptable state and is waiting to acquire the mutex, we have a deadlock and the system won't reach any adaptable state. The reason is that t_1 is suspended for functional reasons and t_2 is blocked for adaptation reasons and, henceforth, will never release the mutex. Such a violation of liveness properties cannot be ignored. It shows the possible conflicts between the functional execution flow and the adaptation process.

The presence of synchronization mechanisms between concurrent tasks leads to inaccessible states in the Cartesian product of major states. Let's take an example to better understand this situation: In this example, there are 2 tasks: one launched by `C1` and named `C1.task`, the other launched by `C2` named `C2.task`. These two tasks have 3 critical sections, all located in C3. A mutex (the central place holding a token in Figure 5) is used to ensure mutual exclusion among the 3 critical sections.

To handle adaptation, and to model concurrency among the tasks, we need to augment the graph of major states with information about mutex acquisition and release, see Figure 6. When taken independently from each other, these 2 tasks follow a simple cycle E1, then E3, and finally E2. However, the acquisition of the mutex makes these cycles more complex and may create a deadlock situation. For instance, there is a deadlock when the adaptation engine locks C2.task in C2.t3 in order to make it stay in an adaptable state and, at the same time, C1.task is blocked in C1.t5 before its second critical section, waiting for acquisition of the mutex, just before issuing the call to C4.

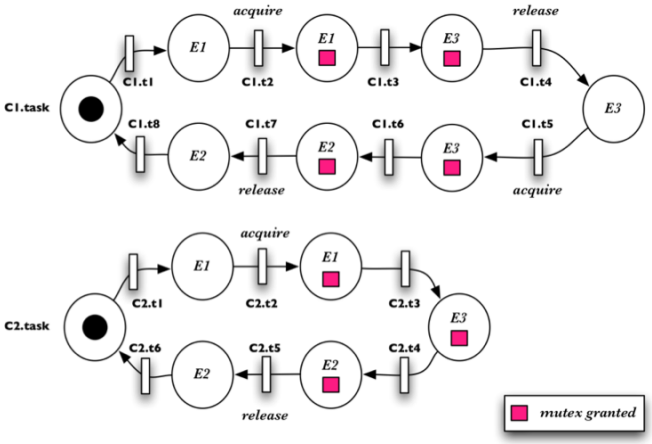

Figure 6. Major states and concurrency model

A basic solution is to put the adaptation lock just before the entrance in the critical section: in C1.t2 and C2.t2 in the example above. The not-adaptable status is extended to E1/E2 with the acquired mutex. This approach oversimplifies the model and is clearly suboptimal: the system is considered as adaptable in a very small portion of its states.

If we want to be optimal, another fine-grain solution needs to be used to prevent deadlocks. We first need to prevent a task to be locked in E3. To this aim, we also need to prevent locking a task possessing a mutex that another task has requested. In the example, we cannot lock C2 in C2.t3 if C1 is in E3 and will try to acquire the mutex in C1.t5. We need to lock C2 before acquisition of the mutex, i.e. in C2.t2.

We end with two types of adaptation locks: *durability locks* that prevent a task to enter E3 (C1.t3, C2.t3) and *convergence locks* before entering a critical section (C1.t2,

C1.t5, C2.t2). The aim of convergence locks is to guide the system towards an adaptable state. As illustrated in this section, synchronization implies causal dependencies that must be taken into account to find suitable adaptation states. Similarly, other sources of causal dependencies, such as message or event-based communication, have an impact on the analysis of suitable adaptation states.

VI. IMPLEMENTATION OF THE ADAPTATION PROCESS

In this section, we describe how to implement the adaptation of a fault-tolerant system. Following the methodology described in the previous sections this requires solving the following six main problems:

1. Describing configurations,
2. Building architectural representation,
3. Building behavioral representation,
4. Event monitoring,
5. Controlling execution to reach a SAS,
6. Initialization of component state.

A. Describing configurations

A configuration is a graph that represents both the hierarchy of the components and the way components are connected together, cf. section IV. Each component in the configuration has parameters initialized with default values. We distinguish two kinds of parameters:

- *Immutable parameters*: their value remains the same for the whole execution, including during adaptation.
- *Initialization parameters*: they aim at setting an initial state. Their value evolves during the execution and they should be set consistently with the past of the execution after adaptation.

The various configurations are written with an ADL producing XML documents. The values of the parameters are stored as serialized values in specific files. Both XML and parameter files are available, at runtime, in a *configuration repository*. These files are built from a graphical tool that checks if the description of each component is consistent with the components' dependencies and that each parameter has a defined default value.

B. Building architectural representation

Our implementation is based on the OpenCOM middleware. OpenCOM provides generic reflective mechanisms. They capture and signal modifications of the software to the meta-level. Any architectural change (components/connections) from OpenCOM main interfaces is automatically and causally reflected in the architectural model. The architectural model consists of a graph as described in the section IV. Its implementation introduces tradeoffs between an efficient browsing or memory usage. To improve adaptation performance in our implementation, we favored an efficient browsing of the component data structure.

C. Building behavioral representation

Our behavioral model is a Petri net built from the description of component *execution paths* (threads, methods). It is stored into a file using a domain specific language. This Petri net is implemented as a quadruplet *(S, T, Pre, Post, M): S* is the set of places, *T* the set of transitions, *Pre* and *Post* are the incidence matrixes and *M* is the marking matrix. This implementation enables behavior monitoring and is very compact.

Nevertheless, modifications on the execution path are quite costly since it requires some matrix transformation and array copies. Each time a component is created, its internal tasks and places representing synchronization mechanism are added to the model. Conversely, its internal tasks and synchronization mechanisms places are removed from the model when it is deleted. When a new component is connected, e.g. through a *method call*, then the execution path must be extended with the path of the method called (when C4 is connected to C3 in figure 5, for instance). When the connection is an asynchronous service connection, e.g. *event/message passing*, a place is added in the model to represent the transient state of event delivery and/or message passing. Transitions representing emission and reception are connected to this place.

D. Event monitoring

An interception mechanism has been added to OpenCOM. This interception mechanism is implemented as a new component inserted between interfaces and receptacles. Two types of interceptors are available. *Connection interceptors* are inserted at the receptacle side (capture of outgoing interaction), and *interface interceptors* are inserted at the interface (capture of incoming interaction). They respectively provide information on origin and destination of the interaction.

Monitoring is performed by a meta-component attached to the interceptors. A meta-component is connected to meta-interfaces providing monitoring features of internal events (emission/reception and lock/unlock events). It modifies the execution model state by activating transitions of the Petri net that correspond to the observed events.

E. Controlling execution to reach a SAS

We have developed a tool that computes the set of SAS from the comparison between two configurations. Additional locks are added to force the software execution to converge toward a SAS and/or to force the system execution to remain in a SAS. Currently, the analysis of SAS is performed off-line. Following the method described in section V, we have identified the E1, E2 and E3 states for all tasks, and combined them to define globally adaptable states. The adaptation (durability and convergence) locks are then inserted at the appropriate identified places.

F. Updating the state

The behavioral model forces the adaptation to be applied when there is no transaction in progress (at the execution level). The result is that the state to be transferred from one component to another one is very limited.

We consider in this work that the state that needs to be initialized when a new component is inserted is accessible using initialization parameters. Actually, we do not have any automatic method to find the part of the state that is relevant

from one configuration to another, and how the rest of the state has to be initialized. In our opinion, this problem is application dependent.

Our solution consists in writing transfer rules identifying the parameters to be saved before modifying the architecture, and parameters to restore from saved parameters using conversion method.

G. *Putting all the pieces together: the middleware*

Our middleware is an extension of the OpenCOM component model implementation. We introduced our architectural model, behavioral model and interception model at the meta-level. Event monitoring is in charge of updating the state of the behavioral model (state of the Petri net) according to observed events.

The adaptation manager looks for the requested configuration in the *configuration repository*, determines the architectural modifications and insert control items in the *behavioral model*. Once an SAS has been reached, the manager saves the required component state, modifies the architecture, initializes the state of newly inserted components, and releases control features. When the adaptation is completed, the software continues its execution with the new configuration. In our current implementation there is no formal assessment, i.e. context monitoring, and the adaptation is trigger by hand.

VII. ON-LINE ADAPTATION OF REPLICATION STRATEGIES

Our case study is just a simple example used to illustrate the feasibility and benefits of the generic approach towards adaptation for fault-tolerant systems, variants of duplex replication strategies.

The proposed approach was applied to a control system that simulates an inverse pendulum located on a cart. The pendulum can rotate around the shaft that connects it with the cart. The cart can move on rails using an electrical engine. Sensors measure the angle of the pendulum and the position of the cart. The acceleration of the engine is controlled by the system through an actuator. The goal is to make the cart moving to a desired position, provided by an operator through a console. The pendulum must remain in equilibrium in the inversed position at anytime.

This system has to be tightly controlled since it maintains an unstable equilibrium. The controller can be subject to failures, a crash for example. A crash failure of the controller leads to the fall of the pendulum. We thus replicate the controller on two processors with two candidate strategies:

- Semi-active replication: two replicas receive requests, but only the leader sends replies.
- Passive replication: the state is periodically sent from the primary to the backup replica.

In [13], we discuss the decomposition of these fault-tolerance strategies; Figure 7 shows the resulting semi-active mechanism configuration.

Depending on some environmental and resource conditions, the best fault tolerance strategy may change. The adaptation consists in switching from the semi-active to the passive replication strategy. This implies i) replacing the Semi-Active Replication Manager by the Passive Replication Manager, and ii) adding to the corresponding structural model several components to handle checkpointing. The leader becomes a primary and the followers become backups when the manager has been changed. From a global point-of-view, the adaptable states of the fault tolerance essentially relate to the synchronization between replicas. For instance, a call from the leader to the follower (i.e. a synchronization message) must terminate (no message in transit) to perform the adaptation.

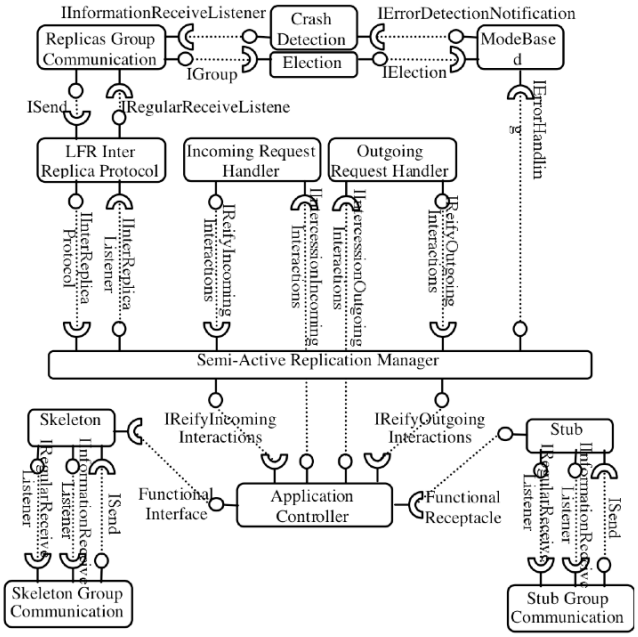

Figure 7. The semi-active strategy configuration

The distributed system running this application is composed of four nodes:

- A console node that signals the expected position of the pendulum to the replicated controllers. It owns a periodic task that sends messages to both replicas.
- Two nodes for replicated controllers with mutual crash detection. They are composed of a periodic task for control-command loop, a periodic task for crash detection and a task for message reception.
- A node with the system sensors. It owns a unique task for message reception.

Figure 8 shows the model of the control-command loop illustrating three software levels: application, fault tolerance and communication. The task goes through these three layers. The task is represented in its initial state waiting for a target position *tp* of the cart delivered by the console, this data item being protected by a mutex. After acquisition of the target position, the task starts by reading the current position *cp* of the cart through sensors. This is done by means of the communication stub (RPC1). When this is done the fault tolerance layer synchronizes the followers with the sensor values. The computation proceeds at the base level to compute the command to be sent to the cart (RPC2). The fault tolerance layer synchronizes then the followers with the command value.

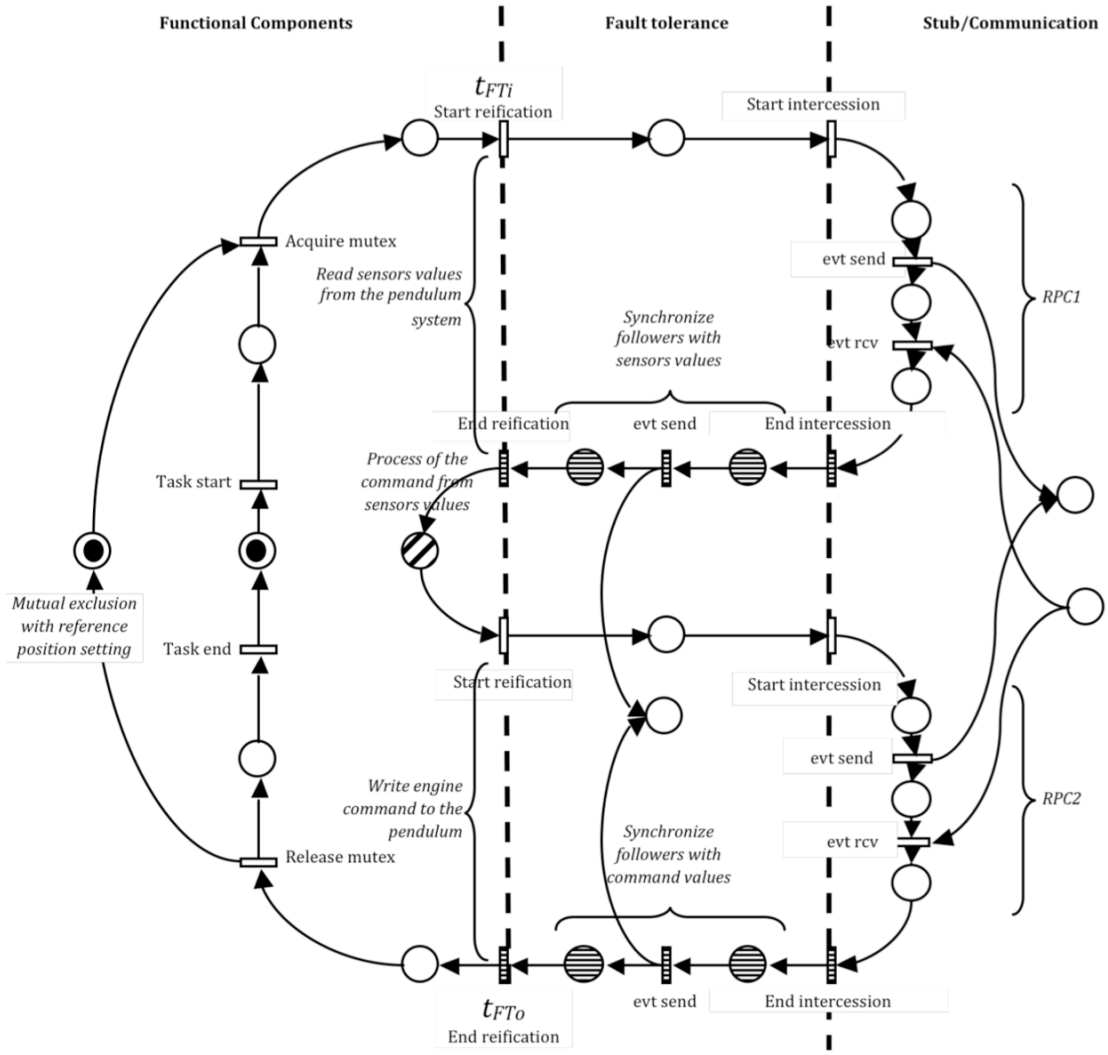

Figure 8. Task *control-command loop* behavioral model

To perform the adaptation, the major states must be determined for each task. These major states are then combined to define the globally adaptable states. In figure 8, the task is in state E3 from transition t_{FTi} to t_{FTo} (the two access to the fault-tolerance layer can be seen as a kind of transaction). It is worth noting that the ***diagonally striped state*** at the application level also belongs to E3.

Ultimately, the global analysis of the case study leads to the following system level SAS:

- **Adaptable states**: (E1) the replicas are synchronized, all tasks have started from their initial state, and the control flow is not within any component to be adapted. (E2): The replicas are resynchronized after the processing, but the tasks have not returned yet to their initial state;
- **Non-adaptable states**: (E3) The replicas are not synchronized; processing has started at one replica only.

In summary, adaptation can only be performed when the state of the replicas is synchronized. State synchronization is a complex mechanism involving several parallel executions. Nevertheless, this study has shown that all transactions are not necessarily finished when the adaptation occurs.

In this case study, the size of the non-adaptable state may appear quite large but it is important to notice that adapting the fault-tolerance mechanisms by switching the replication mechanism requires changing the a central component: the replication manager which articulates the fault-tolerance components. However, this can be optimized with a different design of the fault-tolerance strategy. This stresses the importance of the design for adaptation.

Our systematic analysis of adaptation is very useful to evaluate the adaptation capabilities of a given design. Additionally, finer-grain adaptation can be envisaged with smaller non-adaptable states. For example, introducing compression and/or ciphering of the inter-replicas protocol in the LFR mechanism shown in Figure 7 would be easy. Only the greyed places and transitions are concerned with this adaptation. In this case, the size of the non-adaptable states is very limited. Only two adaptation locks would be necessary to control this system: just before the two End_intercession

transitions. Another easy adaptation is the replacement of the group communication subsystem, i.e. Stub/Skeleton Group Communication in Figure 7.

The main benefit of our work is to provide an approach to investigate the adaptation problem in a generic and systematic way, while trying to optimize control points of the adaptation process. This experiment shows the feasibility of on-line adaptation of replication strategies using CBSE design and on-line behavioral modeling.

VIII. CONCLUSION

For today's systems, online adaptation is a crucial issue. In particular, when one is concerned with evolution of long-living systems, one must take care of evolution of both the application and of the fault tolerance mechanisms. Indeed, adaptation of fault-tolerance is desirable for software maintainability but also to cope with changes in the operational context (environment, fault assumptions, resource leakage, etc.). In this paper, we tried to take advantage of novel software engineering technologies to address the core issues of on-line software adaptation. Indeed, separation of concerns and componentization are two properties that can help us to design and build adaptable fault-tolerant systems. However, an adequate component-based middleware is not enough to provide adaptation. It only provides means to dynamically manipulate the component architecture, to load and unload, to start and stop components.

The work proposed in this paper tries to investigate the issue of adaptation by looking more carefully to the compromise between structural manipulation and on-line computation. We propose a systematic analysis of the behaviour to determine when an adaptation can be made without violating safety and liveness properties. The framework introduces the notion of adaptation engine that stores a description of the component architecture and monitors the tasks running through the components. The reification features enable the behavioural model to reflect the on-going execution. Based on this, adaptation locks can be placed at appropriate places before triggering adaptation. The proposed analysis is able to identify suitable adaptation states, in which the current execution should be maintained, or where the current execution should go, in order to perform a consistent adaptation of the software configuration.

Based on an analysis of the various dimensions of the problem, we have shown the feasibility and the interest of the proposed approach with a small case study. Fault tolerance software developed as a set of components can thus be adapted on-line according to some operational conditions and to the evolution of the system configuration. The limitations relate to the complexity of the execution model, and so scalability implies a high level modeling of the system execution. Through the experiments carried out, we observed that a given design determines the adaptation features, i.e. the tradeoffs between performance and fine-grain adaptation. Such tradeoffs between the granularity of the adaptation/componentization, the complexity of the execution model and state management issues, can only be addressed on a case-by-case basis.

Finally in practice, the proposed reflective architecture and design process lead to software layers developed on top of component model middleware support, forming as a whole, an *adaptive fault tolerance middleware*.

ACKNOWLEDGMENTS

The authors wish to thank for their support the French Defense Research Agency (DGA) and the European Network of Excellence ReSIST.

REFERENCES

[1] Y. J. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri, "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," IEEE Transactions on Computer, vol. 52, pp. 31-50, 2003.

[2] C. Sabnis, M. Cukier, Y. J. Ren, P. Rubel, W. H. Sanders, D. E. Bakken, and D. A. Karr, "Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA," in 7th IFIP Internation Working Conference on Dependable Computing for Critical Applications, 1998, pp. 149-168.

[3] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "The Interception Approach to Reliable Distributed CORBA Objects," in Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS). Portland, Oregon, USA, 1997, pp. 245-248.

[4] P. Narasimhan, T. A. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava, "MEAD: support for Real-Time Fault-Tolerant CORBA," in Concurrency and Computation: Practice and Experience, vol. 17, Wiley&Sons ed, 2005, pp. 1527-45.

[5] K. H. Kim, "ROAFTS: A Middleware Architecture for Real-Time Object-Oriented Adaptive Fault Tolerance Support," IEEE High Assurance Systems Engineering, pp. 50-57, 1998.

[6] M. Hecht, H. Hecht, and E. Shokri, "Adaptive fault tolerance for spacecraft," in IEEE Aerospace 2000 Conference, vol. 5. Big Sky, MT, 2000, pp. 521-533.

[7] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," IEEE Transactions on Parallel and Distributed Systems, vol. 10, pp. 560-579, 1999.

[8] K. H. Kim, J. Goldberg, T. F. Lawrence, and C. Subbaraman, "The Adaptable Distributed Recovery Block Scheme And A Modular Implementation Model," presented at Pacific Rim International Symposium on Fault-Tolerant Systems, 1997.

[9] P. Maes, "Concepts and experiments in computational reflection," presented at Conference on Object-Oriented Programming Systems, Languages, and Applications, Orlando, Florida, 1987.

[10] G. Coulson, G. Blair, P. Grace, F. Taïani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," ACM Transactions on Computer Systems, vol. 26, 2008.

[11] P. Costa, G. Coulson, C. Mascolo, L. Motolla, G. P. Picco, and S. Zachariadis, "A Reconfigurable Component-Based Middleware for Networked Embedded Systems," International Journal of Wireless Information Networks, vol. 14, pp. 149-162, 2007.

[12] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The Fractal Component Model and Its Support in Java," Software Practice and Experience, vol. 36 (11-12), pp. 29, 2006.

[13] T. Pareaud, J.-C. Fabre, and M.-O. Killijian, "Componentization of Fault Tolerance Software for Fine-Grain Adaptation," in Proc of the 14th Pacific Rim Inter. Symp. on Dependable Computing (PRDC'08), Taipei, Taiwan, 2008

[14] S. Sicard, F. Boyer, N. De Palma, "Using components for architecture-based management: the self-repair case", in Proc. of the 30th international conference on Software engineering table of contents, Leipzig, Germany, 2008, pp 101-110 .