
Najla Hadj Kacem¹, Ahmed Hadj Kacem¹ and Khalil Drira²,³

¹ ReDCAD Laboratory - University of Sfax
B.P.1088, 3018 Sfax, Tunisia
najla.hadjkacem@isimsf.rnu.tn, ahmed@fsegs.rnu.tn
² CNRS - LAAS
7 Avenue du Colonel Roche, F-31077 Toulouse, France
³ University of Toulouse - UPS, INSA, INP, ISAE - LAAS
F-31077 Toulouse, France
khalil@laas.fr

Abstract. Technology advances continue to make computing environments ever changing and more complex. In the presence of such environments software systems are increasingly expected to continue operating at run-time. As human intervention becomes costly, time-consuming and error-prone, these systems should be equipped with self-adaptation capabilities in order to adapt themselves in response to environmental changes. While most of the research in this area focuses on individual parts of an adaptive system, our work leverages on this research but tackles the problem where interdependent and distributed adaptations are concurrently performed. In this paper, we approach behavioural changes of component-based systems in two stages. First, we propose a process to individually adapt one component at a time. Second, we elaborate a coordination protocol to maintain globally consistent state when implementing distributed adaptations. To achieve correct coordination, rather than only considering dependency relations between multiple adaptations, our approach further focuses on dependency relations between components at run-time. Motivated by the potential benefits of using formalisms, we construct a formal model of our protocol using Coloured Petri Nets in order for an adaptive system to be trusted after adaptation. In the model, we make sufficient abstraction of details, but still deal with the core of the protocol. This makes the model simpler and the analysis easier due to restricted state space size. We verify key behavioural properties and conduct CTL model checking to assess the correctness of the model and thereby the correctness of the protocol.

Keywords: Self-adaptation, Consistency Preservation, Adaptation Process, Coordination Protocol, Modelling, Analysis.

1 Introduction

Today’s distributed software systems require flexibility and robustness in the presence of ever changing environments. In fact, the environmental conditions in which a distributed system runs are likely to periodically change during the system lifetime. Such changes can significantly impact the system functionality and/or quality characteristics. In consequence, software systems are increasingly expected to adapt during run time in response to these changes while operational, without compromising their consistency [1]. However, manually adapting complex systems is a costly and error-prone process. Automating the adaptation process becomes an imperative undertaking, so as to enable systems to adapt themselves with minimal human interaction.

Self-adaptation is a relatively novel approach [2] that addresses requirements for self-management capabilities in software. As recognized by the DARPA Broad Agency, a self-adaptive software evaluates its own behaviour and changes behaviour when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible [3].

In line with Hofmeister [4], Aksit [5] classifies adaptation into two broad categories, namely structural adaptation and behavioural adaptation. The
first results in architectural changes of the system (e.g., addition or removal of entities), whereas the second causes functionality modification of the computational entities involved. Dynamic change in the functionality of a computing system is of particular interest here, as it is more critical than a change in the structure. To successfully address complex system functionality issues and flexibility requirements, we use the component-oriented technology. More specifically, we are mainly concerned with replacement of a component implementation rather than simple component tuning such as adjusting a parameter.

There are many challenges in developing self-adaptive capabilities for a software system. The major challenge deals with consistency preservation to avoid undesirable transient behaviour [1]. Whatever functionality change to be enforced, it should result in a correctly operating system. When some situations require coordinated adaptations of multiple system components it is important to have coordination mechanisms to maintain a globally consistent state.

As the complexity of adaptive systems increases, so does the need for mechanisms that trust systems to operate correctly after adaptation. Formal methods are perceived as an appropriate way of increasing confidence in adaptive systems [6][7].

This paper describes an approach to behavioural changes of component-based systems. The main features differentiating our approach from existing work are: (1) Unlike most approaches in which the adaptation logic is hard-wired into the business logic, our approach follows the principle of separation of concerns; it separates the adaptation logic of a component from its business logic, thereby increasing system flexibility and reusability. Modularity is also improved by provisioning the business logic specific to each component a number of implementations. To make this more manageable, we split up the implementations into categories. Implementations that belong to the same category are functionally equivalent but each one is tailored to accommodate changes in quality of service requirements. (2) While most of the research focuses on individual parts of a computing system, our approach leverages on this research but tackles the problem where interdependent and distributed adaptations are concurrently performed. To do so, we propose a process to individually adapt one component at a time and elaborate a coordination protocol implementing distributed adaptations. (3) To achieve correct coordination, rather than only considering dependency relations between multiple adaptations, our approach further focuses on dependency relations between components at run-time. (4) Contrary to other ad hoc solutions, we investigate the application of Coloured Petri Nets (CPN) [8][9] as implemented in CPN Tools [10] in order to trust adaptive systems after adaptation. We construct a model of the coordination protocol. In the model, we make sufficient abstraction of details, but still deal with the core of the protocol. Also, we use the simulation and analysis facilities of CPN Tools to assess the correctness of the model and thereby the correctness of the protocol. The CPN formalism is chosen because it supports structured data types, supports construction of compact parameterizable models, and allows models to be hierarchically structured into a set of modules.

The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the model behind our proposal. Section 4 identifies a number of consistency requirements that need to be fulfilled. Section 5 introduces our approach. Section 6 presents the formal model of the protocol. Section 7 shows our state space analysis results. Finally, Section 8 presents conclusions and future work.

2 Related work

Even though software adaptation receives increasing attention from the research community, a common terminology that may be used uniformly is not provided. Some of the terms used in related work may have slightly different meanings depending on the targeted system (e.g., dynamic change [11], dynamic update [12], hot swapping [13], reconfiguration [14]). The associated research focuses on many areas. We limit our discussion to solutions that address the challenge of consistency preservation. Goudarzi [15] identifies two categories of the approaches to system consistency: preservation through recovery and preservation through prevention. In the former the application developer has to implement functionality in the application or in the underlying framework to deal with the inconsistencies introduced at system adaptation time (e.g., Chorus [16], LUCOS [17], POLUS [18]). This approach is closely language and operating system dependent. For this reason, it is complicated and thus too painful for the developers. The potential of the latter case is to prevent the introduction of inconsistencies by driving the system to a safe state before actually applying adaptation. The rest of this section reviews some of the main research towards preventative solutions in the area of behavioural adaptation.

Existing research can be categorized based on the adaptation scope. Adaptive systems can be adapted at the granularity of a procedure, an object or a component. Operating at the procedure granularity-level, both [19] and [20] recognize consistency re-
quirements by waiting until a procedure is inactive. For object-oriented systems, [12] enforces safety of a class by waiting until no methods of the class are active. In [13], object quiescence is defined as a state in which the object is not executing any of its methods. In a component-based context, the pioneering work done by Kramer and Magee [11] provides a fundamental definition of quiescence as any state in which a component is not within a communication and will neither receive nor initiate any new communication. The work of [14] introduces a hot-swapping technique which does not allow the old and the new components to execute simultaneously. In [21], the SOFA (SOFtware Appliances) and its extension DCUP (Dynamic Component UPdating) associate to each component one manager to ensure consistent state while replacing its replaceable part by a new version.

A common characteristic to the presented works is the process they rely on to safely apply changes. Adaptive process consists of at least three stages: bring the affected entities into a safe state, perform the adaptive action, and reactivate the affected entities to resume normal computation. Even though these solutions have some interest features to be helpful for our adaptation process, they are limited by their lack of support for multiple interdependent and potentially distributed adaptations.

While the previously presented trend focuses on individual adaptive entity at a time, several approaches are recently proposed to extend the local scope by considering distribution issues, e.g., [22], [23], [24]. In [22], Kon et al. present a generic architecture for managing dependencies in distributed component systems and discuss how it can be used to support automatic reconfiguration. The main focus is on maintaining both the local and network-wide consistency of a distributed system thanks to the explicit management of inter-component dependencies. The main problem with this approach is that the coordination code is mingled with the application business code, mitigating the advantages of reasoning about the adaptation logic and reusing it. Cactus [23] is a framework for constructing configurable services in distributed systems. Cactus, a host is organized hierarchically into layers, where each layer includes many adaptive components. Each adaptive component encapsulates alternative implementations of a specific service. Adaptations by multiple related components are coordinated using a graceful adaptation protocol. CARISMA project in [24] is a middleware system made up from adaptable services. It is based on the provision of multiple implementations of the same service but with different behaviours. When used, the middleware checks the application profile document and compares with the current execution context to evaluate which behaviour the service component should use when providing its service. These behaviours may conflict. An auction protocol is then proposed for conflict resolution. The system, while having the advantages of cleaner separation of coordination and business logics, is limited to runtime support of adaptation coordination to resolve potential conflicts.

What differs in all the surveyed solutions is the design and the implementation of adaptation mechanisms. Most of the mechanisms are ad hoc; they are highly platform specific and lack adequate formalism. Unless adaptation mechanisms are addressed in a more comprehensive and formal setting, adaptive systems will be error-prone. In this perspective, significant effort is spent for formally specifying structural changes of adaptive systems. Bradbury’s survey of contributions in this area [25] examines a number of specification approaches to dynamic software architecture and further classifies them based on graphs, process algebras logic, and other formalisms. The importance of formal approaches to behavioural adaptation is relatively not emphasized regarding the few effort that is spent on it in existing research. In [26], Magee et al. require the use of a formal configuration model to describe a system. The system model is described in the configuration language Darwin, and is produced during the development process by the application designer. The idea behind this model is to identify which computation of the system should be deferred in order to reach safe state. This is contrary to our approach, in which this is done dynamically at run time. More recently, Biyani and Kulkarni [7] propose an approach to formally verify adaptation in a distributed system. This approach differs from our work in the sense that the system does not need to be in a safe state before an adaptation can be applied. Consistency preservation of an adaptive system depends on the satisfaction of transitional-invariants during and after an adaptation. Another approach for formally addressing the problem of component adaptation is to ensure the correct composition of components. In [27], Xiong and Weishi provide a synthesis of the current state-of-the-art in this research area.

3 System model

The model presented in this paper aims at supporting the self-adaptation of distributed systems, composed by multiple components spanning across multiple nodes. Following the separation of concerns principle we provide a clean decoupling of the adaptation logic from the business logic of a component. This can potentially decrease development costs, by increasing reuse and flexibility. Modularity is also
improved by provisioning the business logic specific to each component a number of implementations grouped into categories. Implementations that belong to the same category are functionally equivalent but each one is intended to accommodate changes in quality of service requirements. Another important aspect of the model is the assignment of an adaptor agent to each component. An adaptor is responsible for the lifecycle management of a component while locally invoking adaptive operations on it.

Furthermore, we assume that the overall system adaptation is controlled by a remote node, the adaptation manager. The manager uses feedback on the state of the managed components and the state of their execution environment to make any necessary adaptation decision. Therefore, at each node there are context sensors. These sensors are able to capture context information locally. Using event-based exchanges sensed information is communicated to the context monitor that stores and interprets it to report relevant changes. Processed information is made available to the manager by the following methods [28]:

- Subscription/notification protocol: The manager can subscribe only to the events of interest. As soon as a relevant change occurs, the monitor is required to provide a notification event about it.
- Explicit query: The manager can ask the monitor for context information by a query and expect an answer.
- Polling: The manager looks for context information periodically from the monitor.

All context informations received by the manager start evaluation process to establish whether the system should be adapted.

Consider, for example, a set of nodes that participate in a distributed multimedia application. All participants are receiving a shared multicast media stream, and different kinds of media are supported, namely text, audio, video. Suppose the manager receives from the monitor context information about a significant change in the current network bandwidth. The manager subsequently may decide to (i) change how the exchanged data is encoded, or to (ii) adjust the media quality (e.g., high, medium, low). Accordingly, updating the receiving affected components depends on whether the new and original implementations belong to different or same categories.

4 Consistency requirements

In this section, we identify consistency requirements that need to be fulfilled in order to ensure that the system is left in a correct state after adaptation.

Consistency requirements we consider are of two types: specific and generic. While specific requirements are typically system-dependent, generic requirements should hold with respect to three aspects including component integrity, communication channels integrity and global system correctness. The first aspect checks whether the new component implementation starts its computation, after being initiated, with appropriate state information. This means that there is a need for state transfer which provides the new implementation enough state information to resume the execution correctly. The second aspect checks whether the communications in progress are compromised by an adaptive action. That is, they should eventually be completed (e.g., before or after adaptation). The latter aspect guarantees that the overall system consistency is preserved in case multiple components are concurrently adapted. Thus, coordination mechanisms are needed for reasoning and determining undesired behaviours, such as deadlocks, cycles, or conflicts. In the following section we show how the proposed adaptation process and coordination protocol meet these requirements.

5 Proposed approach

This section details our approach to behavioural changes in software. The approach takes advantage of using roles as basic building blocks for modelling protocols. In the following, we briefly present the main roles. Next, we introduce an adaptation process to individually adapt one component at a time, and for global adaptation we elaborate a coordination protocol to maintain globally consistent state.

The starting point is to identify the key roles. Here a role can be viewed as an abstraction of functionality whereas an agent is the physical entity that carries out the functionality. In this paper, the approach embodied in the above-described model assumes that the overall system adaptation is controlled by the adaptation manager. The manager’s main functionalities are detecting significant environmental changes, identifying possible solutions and enforcing the optimal one in the running system. The process by which solutions are produced is referred to as planning process. A plan is selected and executed based on its optimality. It can be argued that the problems of conflicting and cyclic decisions between concurrent adaptations are resolved during planning. Nonetheless, other types of problems can occur due to dependency relationships between each affected component and its cooperative components (its clients and the components on which it depends). Namely, clients have to refrain from initiating communications to components under adaptation to not
cause state instability. Recall that an adaptive component has no support for changing its own state. Only its adaptor can impose such state changes. To do so, the adaptor monitors the state of the component and intercepts the messages going into or out of the component during adaptation. Furthermore, it stores adaptation methods that can be fired to invoke operations on the component. From a functional perspective, three main roles are identified as needed: Manager, Adaptor and Initiator.

Manager: This role is instantiated with the agent (A_Manager) that is the global coordinator responsible for detecting relevant environmental changes, for the planning process and for the correct enforcement of the optimal system adaptation plan.

Adaptor: The functionalities associated with this role may be carried out by an agent (A_Adaptor) which is responsible for managing the adaptation logic code of a component affected (AC) by an adaptive action. The adaptation process to be performed by an A_Adaptor is described in more detail later in this section.

Initiator: This role may be carried out by an agent (A_Initiator) which is responsible for managing the adaptation logic code of a component not affected by an adaptive action but capable of initiating communications to an AC. The main functionalities an A_Initiator has to perform are: (i) to react to each passivation request sent by an A_Adaptor so that it refrains its associated component from initiating communications to a specific AC, and (ii) when an activation request arrives, to allow resuming the active behaviour.

5.1 Adaptation process

The prime concern of the adaptation process is how to maintain the consistency at the local component level. During this process, each A_Adaptor is responsible for managing the lifecycle of its associated AC. This is achieved by forcing AC, initially in the running state, in appropriate states (Figure 1) while carrying out the steps of the adaptation process.

A step can either end up with success or failure. In case of failure, a rollback mechanism is needed to return to the original state. A history of chronologically ordered operations, that are performed at adaptation time, has to be recorded by each A_Adaptor.

The adaptation process consists of a sequence of five steps which are briefly explained below.

Step 0: A notification about a new adaptation from the A_Manager triggers the first step. Upon interception of such a message, an A_Adaptor drives AC to the passive state. It calculates the potential benefits of replacing the current AC implementation, in the presence of changes in the available resources or application demands. After this, it decides whether to accept or refuse.

Step 1: When the adaptation plan is intercepted, an A_Adaptor computes the set of all affected components and extracts as parameter of its own task the identifier of the new implementation (New_Imp). It verifies whether the New_Imp and the old implementation (Old_Imp) share category membership. Accordingly, two possible strategies are available to drive AC to the safe state, as follows.

– Case i: Same categories Before suspending the execution of AC, an A_Adaptor must ensure a consistent continuation of the execution flow after adaptation. To do so, it is required that the Old_Imp leaves off where the New_Imp can start correctly. Suspending the execution of the Old_Imp implies freezing processing message communications. There will not be any outgoing message sent by AC, but all incoming messages will be intercepted, serialized and stored into a buffer in FIFO order.

– Case ii: Different categories In this case, driving AC to the safe state must be delayed until all its communication channels are empty. Thus, clients of AC should not start new communications to AC waiting for the adaptation to complete. To ensure this, an A_Adaptor sends a passivation request to the concerned A_Initiator(s). In parallel, intercepted incoming messages are forwarded to AC. When this is finished the execution of AC is suspended.

Step 2: After loading the New_Imp, two strategies are available to drive AC to the ready state, as follows.

– Case i: Same categories State transfer operation has to retrieve the state from the Old_Imp and set it back to the New_Imp. To achieve this, it is required that every implementation provides the typical state access methods get_state and set_state.
– Case ii: Different categories As the Old_Imp
finishes processing all incoming client messages
before being removed, no state transfer needs to
be performed.

Step 3: The existing link an A_Adaptor has
with the Old_Imp is substituted by a link with the
New_Imp. AC is then driven to the adapted state.

Step 4: After all preceding steps succeed, AC
has to resume its execution. The following strategies
are available to drive AC to the resuming state.

– Case i: Same categories An A_Adaptor
fetches out queued incoming messages, furthers
new ones and redirects them to the New_Imp.

– Case ii: Different categories Activation re-
quest can be sent by an A_Adaptor to the con-
cerned A_Initiator(s).

Finally, the Old_Imp can be removed, followed
by driving AC to the initial running state.

5.2 Coordination protocol

The local adaptation scope is extended here by con-
sidering distribution issues. We introduce a message-
based protocol that describes how interacting roles
coordinate to achieve adaptations. We summarize
below the rules governing the interactions at the
A_Manager, the A_Adaptors and the A_Initiators
sides.

A_Manager side The A_Manager initiates the pro-
tocol by broadcasting an adaptation notification to
ACs, asking each A_Adaptor whether it is willing to
exhibit an adaptive behaviour. The notification
may describe the global distributed context informa-
tion and the type of required adaptation. If all
A_Adaptors positively reply then the A_Manager
broadcasts to them the established adaptation plan.
Otherwise, in case at least one A_Adaptor refuses,
the A_Manager cancels the adaptation and prop-
agates this information to the A_Adaptors having
accepted.

Suppose that all A_Adaptors receive the plan.
At this point each A_Adaptor is conducting step 1.
It can either drive AC to the safe state, or fail.
On the receipt of replies from all A_Adaptors, the
A_Manager either (i) sends a message specifying the
next state, if all replies are positive; or (ii) receives
at least one negative reply, cancels adaptation and
sends cancel messages to all A_Adaptors.

Once all A_Adaptors have brought the ACs into
the safe state, the A_Manager allows them to si-
multaneously conduct step 2 and afterwards step 3,
while still taking the same principle into account: if
an A_Adaptor fails locally so does the adaptation
globally. Finally, when all A_Adaptors proceed to
step 4 this implies a successful adaptation. In this
case, the A_Manager waits for all ACs to be in the
resuming state. If this happened, it can pick up its
cyclic behaviour.

A_Adaptors side Before actually performing adap-
tation, each A_Adaptor is in the idle state. Upon
interception of an adaptation notification targeted
to its associated AC, an A_Adaptor executes step 0.
This will result in either an acception or a refusal.
In the first case, it waits for a response from the
A_Manager. In the second case, it exits the adapta-
tion process. This means it will roll back AC to
the original state. An A_Adaptor waiting for a response
remains blocked until it receives either (i) a cancel
message after which it exits the adaptation process,
or (ii) an adaptation plan that causes it to proceed
to step 1. After performing step 1, AC may be driven
to the safe state or not. If the safe state is reach-
able, an A_Adaptor informs the A_Manager and has
to be waiting for a response telling it to proceed or
to cancel. As for step 1, an A_Adaptor performs step
2 and so on sequentially step 3, until it receives a
message asking it to resume the normal execution
according to step 4. Finally, when step 4 is finished
it returns to the initial idle state.

A_Initiators side Initially, each A_Initiator is in
the idle state. Once it intercepts a passivation re-
quest from an A_Adaptor, it immediately performs
some operations to block outgoing channels directed
to the specific AC. When this is achieved it sends the
demanding A_Adaptor a message indicating that
the passivation is done. Then, it remains waiting for an
activation request to resume active behaviour, after
which it sends the A_Adaptor a message indicating
that the activation is done. Finally, it returns to the
idle state.

6 The protocol CPN model

Central in this section is our aim to trust the coordi-
nation protocol to behave as expected. We start by
giving an overview on CPN. Next, we present the as-
sumptions taken into account when constructing the
protocol model. In the remaining subsections we de-
tail the description of some individual modules con-
stituting the model.

6.1 Coloured Petri Nets

Petri Nets (PN) provide a well known formalism for
modelling concurrent and distributed systems, pro-
tocols included. A traditional PN is a directed graph in which each node is either a place (drawn as ellipse) or a transition (straight bar or rectangle). Places can be considered as conditions on the system control states and transitions as actions. Each edge (oriented arc) connects a place to a transition or vice versa. Tokens are the marker of a place. A transition is said to be enabled if a sufficient number of tokens, according to the arc inscriptions, fills every input place. An enabled transition can fire (or occur) and create a specified number of tokens in each output place.

Coloured Petri Nets (CPN) are a high-level PN where tokens are of some specified type (colour set). The arc inscriptions are functions used to determine both the quantity and the value of tokens to be removed or created. The guards are boolean expressions associated with the transitions. A guard is used to restrict possible action occurrences.

6.2 Modelling assumptions

Before proceeding, it is worth noting the following assumptions adopted when constructing our CPN model.

Reliable communication: The starting assumption is that the communication channel between interacting roles is reliable so as to avoid loss, duplication or permutation of messages during adaptation.

Well established plan: It is also assumed that the manager has identified the optimal plan, then it recognizes components involved in the adaptation.

No timeout constraint: The model assumes that all adaptors reply to the manager within a reasonable bounded time.

Abstract modelling of messages: All fields in the messages are omitted as these do not impact the protocol logic.

Termination: The manager and all adaptors do not resume or roll back after adaptation that is successfully performed or not, but rather do either exit with a FAIL or a SUCCESS state. This assumption helps us to prove proper termination.

6.3 Overview of the CPN model

Figure 2 shows the hierarchical structure of the protocol CPN model. Each node in Figure 2 represents a page (module). The complete model is then hierarchically structured into 24 pages. An arc between two nodes means that the superpage (source node) contains a substitution transition whose behaviour is described in the subpage (destination node). We adopt the convention that a substitution transition and its associated page have the same name. As representative pages of the CPN model we consider dashed nodes. In the following sections we explain in more detail how they are modelled.

6.4 The Top page

Page Top, depicted in Figure 3, is the topmost page of the CPN model and provides the highest abstraction view of the model. This page has four substitution transitions: Adaptation Manager, Adaptor Agents, Initiator Agents and Communication Channel.

Also in Figure 3, there are eight places. Each place represents an input or an output message buffer to model interactions between roles. For example, when the manager sends a message, it will appear as a token on the place OutgoingMsg MNG to AA; and similarly a message received by the manager will appear as a token on the place IncomingMsg MNG from AA.

The colour sets AAxMsgMNG and AAxMsgAA are used to model the messages in the protocol that are
present on the two places respectively. They are defined as follows:

1. val NbrAA = 3;
2. colset AA = index A with 1..NbrAA;
3. colset MsgAA with Refuse|Accept|SFStDone|SFStCancelled|
   RdStDone|RdStCancelled|ImpChgDone|ImpChgCancelled|PsvReq|ActReq|RsmDone;
4. colset MsgMNG with AdpNotif|Cancel|AdpPlan|ReadyStReq|
   Rollback|ChgImpReq|RsmReq;
5. colset AAxMsgAA = product AA * MsgAA;
6. colset AAxMsgMNG = product AA * MsgMNG;

Lines 1-2 declare the colour set AA. First, we declare NbrAA to be a constant and give it the value 3. That is, NbrAA is a parameter of the model by which we can change the number of adaptors. Next, we declare AA to be the identifier of the adaptors.

The colour sets MsgAA and MsgMNG (Lines 3-4) are used to define the messages transmitted over the communication medium from the adaptors to the manager and from the manager to the adaptors respectively.

The remaining colour set AAxMsgAA (Line 5) is defined to be the product of the types AA and MsgAA. Tokens of the colour set AAxMsgAA are two-tuples where the first element denotes the identifier of the adaptor source of the message, and the second element containing the message.

6.5 The Adaptation Manager Page

Figure 4 depicts the page Adaptation Manager modelling the manager side of the protocol. The page captures the high-level overview of the module and structures it into five substitution transitions. Each substitution transition is named by a process step and is linked with a subpage which models the manager behaviour in this step. This gives us a better compact and readable model.

There are six places in Figure 4. They represent the core set of states the manager goes through during adaptation. The place Manager, typed by the colour set StMNG, models the initial state ACTIVE and the two terminal (accepting or halt) states FAIL and SUCCESS for the manager. The remaining five places store state changes of the manager related to performing process steps. Typed by the colour set AAxStMNG, these places identify the state of the manager with respect to the adaptors. Note that the states are named to reflect the current step; prefixes are used to identify whether step 0 (N_), step 1 (Sf_), step 2 (Rd_), step 3 (Ch_) or step 4 (Rsm_) is performing.

The first place Manager has an initial marking of one ACTIVE token. This indicates that the manager is initially in the ready state. From this marking, only the Start_NwAd transition is enabled and when it occurs it will initialize the adaptation process by putting on the place Mng_Ntf as many tokens as the manager has involved adaptors. Each token models that the manager is in the N_START state for each of the adaptors.

Fig. 4. The Adaptation Manager page.

After step 0 holds via the substitution transition MNG_Step_0, two possibilities exist for further processing: either the step is successfully terminated by all adaptors or not. Depending on whether the manager is in the state N_END or N_Cancel for all adaptors, the transition Step0_To_1 or Cnl_N is enabled respectively. The occurrence of Step0_To_1 results in changing the state of the manager from N_END to Sf_START with respect to all adaptors, while the occurrence of Cnl_N moves the manager state from N_CANCEL to the terminal state FAIL resulting in a cancelled adaptation.

Once the manager is in the state Sf_START, the adaptation process is considered to be in progress. This will start step 1. The adaptation gets successfully terminated as soon as the transition Cmpl_Ad will be fired when the manager is in the state Rsm_END with respect to all adaptors and will set the state of the manager to SUCCESS.

6.6 The Adaptor Agents Page

The page Adaptor Agents is shown in Figure 5. Similarly to the Adaptation Manager page, this page
is structured in such a way that each substitution transition is used to refer to a process step module for modelling the behaviour of the adaptors in this step. The states of the adaptors during adaptation are modelled by six places. All adaptors are stored in one individual place; their states need to be extended with their identifiers. That is, the six places are typed by the colour set $AAxStAA$.

Briefly, all adaptors are in the initial state IDLE. When an adaptation notification $AdpNotif$ arrives via the $IncomingMsg$ place, the transition $Rcv_AdNtf$ is enabled and when it occurs it sets the state of adaptors, involved in the adaptation, to $N\_START$.

It should be noted that we adopt the convention that certain states of the manager and the adaptors have the same name.

6.8 Modelling Interactions

We now describe the pages capturing the interactions between the Adaptation Manager and the Adaptor Agents modules and between the Adaptor Agents and the Initiator Agents modules, particularly during step 1. We show how we make sufficient abstraction of processing details in the model, but still deal with the core of the protocol.

Modelling interactions between the manager and the adaptors

The $AA\_Step\_1$ page Figure 7 depicts the page $AA\_Step\_1$ which is the subpage of the substitution transition $AA\_Step\_1$ shown in Figure 5. This page models how the adaptors operate during step 1. The place $AA\_Drv\_Sf\_St$ (top-left) is used to associate with each adaptor identifier the initial state $Sf\_START$, the intermediary state $WAIT$ and the two terminal states $Sf\_END$ and $Sf\_CANCEL$.

After having intercepted the adaptation plan, an adaptor is in the $Sf\_START$ state indicating that it is ready to drive the component to the safe state. As previously described, this is conducted in three stages. At first, an adaptor computes the set of all affected components. Secondly, it extracts as parameter of its own task the identifier of the new implementation and verifies whether the new and the old implementations share category membership. The third stage involves bringing the component into the safe state. Since we do not care about processing that does not affect the operation of the protocol, the first and second stages are abstracted away.

Hence, we choose to make this module parameterized through the place $AA\_Categ$ in order to identify for each adaptor whether the two implementations are of the same category or not. The inscription...
AA_CATEG at the bottom-left side is a constant which specifies the initial marking of this place. Typed by the colour set AAImplCATEG, it associates each adaptor identifier with the value SimCtg or DiffCtg.

With respect to Figure 7, the module is decomposed into three main parts: i, ii and iii.

Let us consider the part i which models the behaviour of the adaptors when the two implementations are of different categories. In this case, the transition Start_DCtg is enabled. Depending on the existence of dependency relationships between an affected component and its clients, two cases may arise to fire Start_DCtg. If an adaptor has not reified dependency relationships (modelled via the if-then-else expression), then it can directly proceed to the Frz_READY state. Otherwise, it moves to the Psv_READY state with respect to concerned initiators, indicating that passivation requests should be sent to these initiators in order to refrain the initiation of messages targeted to the component under adaptation. This is done in the subpage of the DCateg_Psv_IC substitution transition. Details of this subpage will be described later. We expect that when this is achieved an adaptor will be in the Psv_COMPL state with respect to the concerned initiators. At this point in time, the Psv2Frz transition becomes enabled. By firing this transition a token is placed on the AA_Frz place, representing that the adaptor is now ready to start to freeze the component (in state Frz_READY). The place RcdAA ensures that the same adaptor will not appear in the place AA_Frz more than once.

When any adaptor is in the Frz READY state, it needs to be ensured that all intercepted incoming messages have to finish processing by its associated component before the actual suspension can take place. Note that this may be ended either successfully or unsuccessfully (e.g., if all intercepted incoming messages are not guaranteed to finish within bounded time). This processing is hidden by the substitution transition DCateg_Frz_AC, and is simply modelled as a non-deterministic choice after which an adaptor is expected to be in the Frz_COMPL or Frz_CNL state. Accordingly, the transition Snd_SfStDone or Snd_SfStCnl is enabled. After firing the Snd_SfStDone (resp. Snd_SfStCnl) transition, a response containing the SfStDone (resp. SfStCancelled) message is sent to the manager and the adaptor changes its state from Frz_COMPL (resp. Frz_CNL) to WAIT.

We now consider part iii to show how the adaptors operate when the two implementations are of the same category. After the transition Start_SCtg fires, an adaptor proceeds to the state Frz READY, in which state it is ready to freeze the component. But before actually suspending the execution of the component, it is important that the old implementation leaves off where the new one can resume correctly. After component suspension, there will not be any outgoing message sent by the component but
all incoming messages will be intercepted, serialized and stored into a buffer in FIFO order. Since these details do not impact the protocol logic, they are abstracted away. The adaptor actions for the reachability of the safe state are hidden by the substitution transition SCateg_Frz_AC, and will simply set an adaptor in the state Frz_COMPL. At this point, the transition Snd_SfStDone becomes enabled. When fired, it passes the message SfStDone to the manager and causes the adaptor to change its state to WAIT.

As shown in part ii, after the messages from the manager arrive this lets any adaptor (in state WAIT) know whether all adaptors involved have ended successfully step 1 or not, depending on the incoming message ReadyStReq and Rollback. The actual interception of such messages is respectively modelled by the transition Rcv_RdyStReq and Rcv_RollReq. Firing the transition Rcv_RdyStReq (resp. Rcv_RollReq) will cause an adaptor to change its state from WAIT to Sf_END (resp. Sf_CANCEL).

The MNG_Step_1 page

Figure 8 depicts the page MNG_Step_1 which is the subpage of the substitution transition MNG_Step_1 shown in Figure 4. In Figure 8, the manager, initially in the state Sf_START for all adaptors, proceeds to the WAIT state after firing the transition Wait. It remains in this state waiting for the arrival of responses from all adaptors. As previously mentioned, an adaptor can either answer with a SfStDone or SfStCancelled message. Transitions Rcv_SfStDone and Rcv_SfStCnl model the receipt of these expected messages from the adaptors. Each time Rcv_SfStDone (resp. Rcv_SfStCnl) fires, the manager changes state from WAIT to Sf_ST_DONE (resp. Sf_ST_CANCELLLED) and the reception counter is incremented by one. This counter is maintained in the place Count typed by the colour set COUNT. The manager controls the number of received messages based on the current state of the counter found in the single token value cnt (initially 0) of the Count place. Therefore, a guard is added to SfStCnl_Rcvd and Cncl_Ad which only allows these transitions to fire when the value of cnt is equal to the value of NbrAA. Recall that NbrAA is the maximum number of adaptors.

Hence, when receiving all responses two possibilities are taken into account by the manager: (1) All adaptors drive their associated components to the safe state, then the transition Snd_RdyReq is enabled and when fired passes a message ReadyStReq to all adaptors, and sets the manager in the Sf_END state; (2) There exists at least one adaptor which fails to drive its associated component to the safe state such that the token placed on the place Boolean, used to control whether or not this happens, changes from the initial state false to true. In such a case, the sending of a Rollback message to all adaptors is modelled by the transitions SfStCnl_Rcvd and Cncl_Ad, and causes the manager to change its state to Sf_CANCEL.

Modelling interactions between the adaptors and the initiators

The DCateg_Psv_IC page

Figure 9 depicts the page DCateg_Psv_IC, subpage of the AA_Step_1 page (Figure 7). Initially, an adaptor is in the state Psv_READY with respect to all concerned initiators. This means it is ready to broadcast passivation requests (PsvReq) to these initiators. The broadcasting is modelled by the transition Snd_PsvReq, which causes the adaptor to move to the WAIT state for each of the initiators. Upon receiving an initiator response (PsvDone) via the transition Rcv_PsvDone, the adaptor in the state WAIT for that initiator, will be in the Psv_Done state after this transition fires. Once all Psv_Done messages are collected, the transition All_Rsp_Rcvd becomes enabled. Firing it sets the adaptor in the Psv_COMPL state for each of the initiators.

Fig. 8. The MNG_Step_1 page.

Fig. 9. The DCateg_Psv_IC page.
The IA_Psv page  Page IA_Psv depicted in Figure 10 is the subpage of the Initiator Agents page (Figure 6). As shown in Figure 6, all initiators are initially in the IDLE state. Firing the transition Wait_PsvReq brings each initiator to the WAIT state with respect to the concerned adaptors if any. According to Figure 9, each time a message PsvReq from an adaptor reaches an initiator which is in the state WAIT for that adaptor, the initiator must invoke a passivation method to block outgoing channels directed to the specific component under adaptation. As this processing does not affect the protocol logic, it is abstracted away in the model. An initiator implements the operation of passivation via the occurrence of the transition Passivate. After Passivate fires, the initiator changes its state from WAIT to PASSIVATED. The sending of the response is modelled by the transition Snd_PsvDone, which causes the initiator in the PASSIVATED state to move to the WAIT state, and passes the message PsvDone to the specific adaptor.

![Fig. 10. The IA_Psv page.](image)

7 Protocol verification

The purpose of this section is to explain how the modelled protocol is validated using the analysis facilities of CPN Tools. This is conducted in two steps. We first use the simulation to investigate different scenarios of the model. Next, after conducting the state space analysis we introduce the standard properties of CPN and how they can be used to prove behavioural properties of the model. Finally, we verify additional properties by considering CTL formulas.

7.1 Simulation

During the construction of the model, single-step simulation is used to investigate different scenarios in detail and check whether the model works as expected. With its visual feedback, simulation helps us to understand the behaviour of the protocol, locate errors and modify the model. Moreover, simulation provides us flexibility to adjust parameters, which are detailed later, for evaluating the system.

Simulation is in fact a powerful facility for increasing our confidence in the correctness of the model. But, conducting several simulations does not ensure that all possible scenarios are covered. To give further confidence, we apply state space analysis.

7.2 State space analysis

An important property of our model is that it is parameterizable with respect to:

- The maximum number of adaptors (NbrAA).
- The maximum number of initiators (NbrIA).
- The relations existing between adaptors and initiators (AA_IA). Implicitly, this refers to the dependency relationships between components and their clients. These relationships must be reified at run-time by the adaptors.
- The information used to identify to each adaptor whether the new and the old implementations share category membership (AA_CATEG).

Accordingly, this allows us to perform analysis by setting the initial state of the model. State space analysis relies on computing all reachable states of the model, and representing these as a directed graph where nodes represent states and arcs represent occurring events. Table 1 shows the chosen values, as declared in CPN ML, to set the parameters of the model in order to carry out three representative tests. To limit the calculation time we conduct state space analysis based on small values.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NbrAA</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>AA_IA</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AA_CATEG</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. Values in tests 1-3 to set parameters.

After state space generation, we investigate some properties using the standard report of CPN Tools. This report provides information about the size of the state space and contains answers to a number of standard properties. Some analysis results are presented in Table 2 based on a partial state space report. In the following, consider as an example the test 2 depicted in Table 1.

The first part referred to as statistics shows that the CPN Tools calculates a full state space, containing 47,100 nodes and 193,323 arcs in 513 seconds.

The second part contains information about the home properties. A home marking is a marking it is always possible to return to. In our model adaptation can either be successfully achieved, or cancelled. The model should consequently have no home marking, as shown in the report.

In the third part, examining the liveness properties shows that there are 5 dead markings. A dead marking is a leaf node of the directed graph; a
Properties | 1 | 2 | 3
--- | --- | --- | ---
State Space Nodes | 4702 | 47100 | 67076
State Space Arcs | 14843 | 193323 | 274907
State Space Secs | 5 | 513 | 932
State Space Status | Full | Full | Full

Home Properties

| | None | None | None
--- | --- | --- | ---
Home Markings | None |None | None

Liveness Properties

| Dead Markings | [729, 2744, 3037, 3416, 4702] | [6932, 28523, 30932, 34857, 47100] | [6905, 37419, 39828, 43731, 67076]
--- | --- | --- | ---
Dead Transition Instances | None | None | None
Live Transition Instances | None | None | None
Fairness Properties | No infinite occurrence sequences

Table 2. State space analysis results for tests 1-3.

state in which no transition is enabled. To investigate whether these states represent desired terminal states of the protocol, we use the *query function NodeDescriptor* to get a representation of the information associated with each dead marking. The last marking $M_{47100}$ differs from the remaining markings in that it represents the case when the protocol terminates with the manager and all adaptors in the SUCCESS state. That is, the adaptation is successfully achieved. However, the remaining markings $M_{6932}$, $M_{28523}$, $M_{30932}$ and $M_{34857}$ denote the cases when the protocol terminates with the manager and all adaptors in the FAIL state. More precisely, they refer to the adaptation which is cancelled either at step 0, 1, 2 or 3 respectively. Hence, the 5 markings correspond to the desired terminal states of the protocol. This expectation is confirmed by the absence of live transition instances.

The final part contains information about the *fairness properties*. We see that there is no cyclic behaviour in the model.

Another issue which should be noted involves the boundedness properties of the message buffer places. These properties specify that the minimal number of tokens that can reside on each place is always zero. This implies that the messages are exchanged and processed as expected.

If analysis shows that the model is correct with respect to the CPN standard properties for each of the three representative tests, then this will increase our confidence in the fact that the model is also correct when varying the parameters.

### 7.3 Model checking

Even though the standard report proves to be useful to investigate the behaviour of the model, some properties which are more particular for the model have to be verified. Thus, we conduct CTL model checking [29] that represents the act of checking the truth value of a given CTL formula for a given state space.

For brevity, an interesting property we check here states that *if the manager receives at least one negative reply from an adaptor then the adaptation is cancelled*. We consider only a representative property that shows that the adaptation is eventually cancelled in case an adaptor fails to bring its associated component into the safe state. To do this, we use the function *eval_node* which takes two arguments: the CTL formula to be checked and the state from where the model checking starts. The ML code [30] implemented for checking the property is explained below.

```ml
1. fun CtlResp((aa, msgaa): AAxMsgAA)=msgaa;
2. val CtlResp’n=ext_col CtlResp;
3. fun StateAA((aa, staa): AAxStAA)=staa;
4. val StateAA’n=ext_col StateAA;
5. fun AdpIsCancelled(m)=CtlResp;
6. (Mark.TopOutgoingMsg_AA_to_MNG 1 m=’SfStCancelled);
7. val CnlAdState=List.nth(SearchNodes(8. EntireGraph,
9. fn m => (AdpIsCancelled m),
10. NoLimit,
11. fn m => m,
12. []),0);
13. op ::),0);
14. fun FailStateDone(m)=
15. (Mark.Adaptation_Manager ’Manager 1 m=’FAIL)
16. andalso (StateAA’
17. (Mark.Adaptor_Agents’Adaptor_Agents 1 m=’FAIL));
18. val FailState = NF(‘noFailState’, FailStateDone);
19. val CnlAdp = NF(‘noAdpIsCancelled’, AdpIsCancelled);
20. val myASKCTLFormula = FORALL_NEXT(EV(FailState));
21. eval_node myASKCTLFormula CnlAdState;
```

Lines 7-13 implement the value CnlAdState, second argument of eval_node (Line 21). This value uses the standard query function SearchNodes to find all markings m in the state space that evaluate to true with respect to the predicate function AdpIsCancelled. As implemented in Lines 5-6, this function checks whether a SfStCancelled message is sent by an adaptor to the manager, via each mark-
ing on the place OutgoingMsg AA to MNG. Starting from the marking which is found, we check for all successor states that the adaptation will eventually be cancelled (Line 20). This is done by checking that the state in which the manager and all adaptors will be in the FAIL state. The result is given in Figure 11, therefore the property is satisfied.

Fig. 11. The model checking result.

8 Conclusions and future work

Driven by the ever increasing need for mastering systems complexity in dynamic environments, self-adaptation becomes crucially important for building today's software systems. Throughout this paper we describe an approach to behavioural adaptation of component-based distributed systems. The main aim of our approach is to comprehensively meet the consistency needs, ranging from single component to distributed system. Even in the presence of failures during adaptation, we guarantee the consistency. This feature is lacked by most of the existing approaches. Furthermore, in order to trust an adaptive system to operate correctly after adaptation, we investigate the application of formal methods by adopting the CPN formalism. After constructing a CPN model of the protocol, we use the simulation and analysis facilities of CPN Tools to assess the correctness of the protocol.

For future work, several issues require further investigations. (1) Our approach is pessimistic since a local failure causes adaptation to be cancelled. Therefore, causes triggering adaptation cancellation have to be relaxed. (2) We seek to investigate timeout in order to ensure that every adaptation is performed in a reasonable time. (3) Planning process to identify possible adaptation plans is a work in progress. We will also focus on the mechanisms that can be implemented to evaluate and choose the most efficient plan. (4) Even though the adoption of centralized solution guarantees globally optimal adaptation decisions while respecting the coordination constraints, it may not scale well when applied to managing a great number of components. A better scalability will be featured by a decentralized approach which we address in our ongoing investigations.

References


Najla Hadj Kacem received her Master degree in Computer Science in 2005 from the Faculty of Economic Sciences and Management (FSEG), University of Sfax, Tunisia. She is now a Ph.D. student at FSEG and member of the ReDCAD Laboratory. Her current research interests include self-adaptation, coordination protocols in distributed software systems and formal methods.

Ahmed Hadj Kacem obtained his diploma of Ph.D. from the University of Paul Sabatier, Toulouse III (France) in 1995. He joined the University of Sfax as an associated professor in 1996. He participated to the initiation of many graduate courses. His current research areas include multi-agent systems and design of adaptive software architectures.

Khalil Drira received the Engineering degree and the M.S. degree (DEA) in Computer Science from INPT, the National Polytechnic Institute of Toulouse, in 1988 and the Ph.D. degree in Computer Science from UPS, University Paul Sabatier Toulouse, in 1992. He is, since 1992, Chargé de Recherche CNRS, a full-time research position at the National Center for Scientific Research of France. His current research interests include design of adaptive software architectures and QoS management. He is or has been involved in several national and international projects in the field of distributed communicating systems. He is author of more than 150 regular and invited papers in international conferences and journals.