An Adaptive Distributed System Based on Conditional Dependencies

Luci Pirmez
Luiz F. Rust C. Carmo
{rust, luci}@nce.ufrj.br

Reinaldo de B. Correia
Renata F. Corrêa, Roberta L. Gomes
{refalcao,reinaldo}@posgrad.nce.ufrj.br
beta@laas.fr

Núcleo de Computação Eletrônica
Universidade Federal do Rio de Janeiro
P.O.Box 2324, 20001-970
Rio de Janeiro – RJ
Brazil

Luiz F. Huet de Bacellar
Bacelllf@utrc.utc.com

United Technologies
Research Center
411 Silver Lane
East Hartford, CT, 06108
USA

Abstract

Distributed programming is applicable in a wide range of domains such as control systems. These systems are subject to mutable environments and must also provide a time varying adaptive set of functionalities. An adaptive system is defined by means of two main features: (i) an approach to specify the correspondence between objects and resources and (ii) the related mechanisms to assure dynamically the right set of objects running. The conditional dependency concept has proven to be an efficient way to specify this kind of correspondence between resources and objects for adaptive systems in the context of multimedia distributed processing (ServiMidia Project [1]). This paper presents a framework for designing general purpose adaptive systems anchored on the concept of conditional dependencies. As the use of such approach may lead to inconsistent system behaviour when specifying very large systems, we show that it is easy to design verification tools based on the same framework mechanisms.

1: Introduction

A modern computing environment is characterized by a high degree of mobility, heterogeneity, and interactions among computing devices. The key requirement for systems in such an environment is the flexibility to adapt to drastic changes that may occur. These changes profoundly impact the performance of user applications.

A distributed system is a good choice to address these issues. Distributed applications are sets of objects working in a cooperative fashion. Any one of the machines in the environment can host objects. In this way, these applications are able to utilize the available resources more efficiently. In addition, performance is higher since parallel processing is the main goal. Objects are also considered components that have specialized functions. The execution coordination of objects is achieved by exchanging messages so that applications are capable of guaranteeing the correctness of its execution.

Although previous research exists, distributed systems do not offer enough support for managing, adapting, and reacting to these changes. For a safe distributed application, timely and dependable services it should be maintained despite any component failures or environmental changes. Our goal is to present an adaptive approach that will make it possible for applications to not only effectively deal with the changing environment in which they are run but also be able to fulfill their varied functionalities.

This adaptive approach may interrupt or start an object depending on the functionality that the application is supposed to accomplish. Further, objects may be interrupted due to a lack of resources in one machine and resume on another in which resources are available.

Failure recoveries in dependable systems are essentially a process of adaptation [2] and many adaptive mechanisms have already been proposed. Some of these mechanisms implement the adaptation process inside the operating systems [3, 4], others in the *middleware* [5, 6]. There are also others that build general models to facilitate the deployment of the adaptation process [7]. Examples of these approaches are the concurrency control of database transactions [8], real-time parallel systems [9, 10], and high-speed communication protocols [11]. Our approach differs in that it proposes an original and simplified way of specifying adaptability

requirements with conditional dependencies, which makes it possible to verify the consistency of the specification by employing the same mechanisms necessary for its processing.

This paper introduces *in Section* 2 the concept of conditional dependencies and how to use it to specify distributed adaptive applications. *Section* 3 discusses architectural issues to create adaptive systems according to the proposed strategy. *Section* 4 points out several inconsistencies that may occur during the specification process. *Section* 5 provides some directives to verify the consistency of a specification. Section 6 finally reports some of the conclusions of this paper.

2: Specification of an Adaptive Application

The process of adaptation is defined as the substitution of a group of running objects (G_{Oa}) for a different group of objects (G_{Ob}). In a more general sense, the process of adaptation can also be defined as the replacement of an object by another one or by a group of objects. Moreover, an object may substitute a group of objects.

Basically, an adaptation strategy allows distributed applications to specify different states. Application states are related to specific sets of actions that must be taken during a period of time. An application state is established by stopping or running objects as shown in Figure 1.

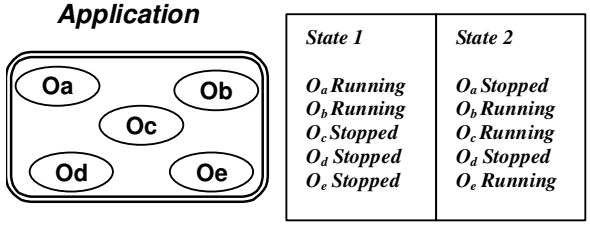

Figure 1 - Application states

To coordinate the execution of a distributed application, it is necessary to determine the spatial positioning of the objects and the establishment of temporal relationships among the objects. Spatial positioning allows application developers to organize the physical positioning of the objects in the environment according to resources availability or/and optimisation requirements such as network bandwidth and delay. Temporal relationships are specified by defining initial instants and durations of the involved objects. The temporal relationship among the objects must be associated to a synchronization model, which governs how these objects are linked to each other. Schemes based solely on timelines present a series of limitations. In other words, it is very difficult to structure the application's execution and establish relationships among objects of variable or unknown duration. To overcome these limitations, we propose a specification approach that combines: (1) a model of temporal synchronization based on timelines; and (2) a model of causal synchronization based on conditional relationships among the involved objects associated to the temporal model above.

2.1: Temporal requirements

An application consists of different types of objects that are activated/deactivated at different instants of time and durations. The beginning instants and durations of these objects' activities are specified by either an inflexible (hard) or flexible time specification. In the case of an inflexible specification, these instants and durations are fixed, whereas, in a flexible specification, these instants vary within a range of values.

A flexible temporal specification is obtained through the establishment of margins of tolerance for the beginning of an object's execution, i.e., when temporal intervals are defined, objects may be started at any instant within this interval. The timing aspects are expressed via a margin of tolerance (range), the initial instant, and the duration.

2.2: Causal requirements

The key to deploying distributed applications that are able to adapt themselves when they are confronted with changes in the environment is to provide for these applications the means of changing their states. These adapted states allow preserving the set of functionalities, which the application was designed to perform. To deal with this problem, the application developer must specify the correct temporal and logical relationships among the application's objects so that a coherent adaptation can be accomplished.

In using the concept of causal synchronization, the application developer can specifies conditional dependencies among the objects to construct a net of causality that determines the application be correctly executed.

The specification strategy for adaptive applications is based on the link concept. The use of links is quite common and is fundamental in the hypermedia and hypertext areas. In these areas, the links are defined as pointers inside a hypermedia or hypertext document. The pointers point to locations inside the same document or inside other documents. These locations are called nodes.

In this way, the links create a mesh of connected nodes so that it is possible for the users to navigate throughout the interlinked documents.

The MHEG-5 standard [12] defines a Link object, which consists of a LinkCondition and a LinkEffect. The LinkEffect, which is a list of elementary actions, is executed when the LinkCondition becomes *true*. In MHEG-5, a LinkCondition is always fired when an event occurs. An event always emanates from only one object. In this paper, the definition of link concept, while based on the MHEG-5 standard, is broadened to include the ability to express adaptability requirements. A Link object is made up of a LinkSource and a LinkTarget. A LinkSource consists of a list of SourceConditions associated to different source objects. A LinkTarget related to only one target object is formed by a TargetAction and a TargetDexpr (dependency expression). TargetDexpr interrelates SourceConditions to describe the semantic requirements that must be followed during the application processing. When a link is fired, it executes its TargetDexpr. It is then verified, and when it becomes true, the TargetAction is executed. Figure 2 depicts a Link object whose LinkSource is composed of N1, N2 and N3 conditions. This Link object has also a LinkTarget that, in turn, is formed by a TargetDexpr and the TargetAction N4.

To specify these conditional dependencies among objects, two link object types are defined: *startLink* and stopLink. They are related to the actions of starting and stopping a unique object execution.

Two states associated to SourceConditions are defined as *running* either *stopped*. Specification of a SourceCondition is defined according to ***source:state*** syntax. For example, when object ***x*** is running, its SourceCondition is specified by the ***x:running*** expression. In the same way, a link's TargetAction, is specified by the ***target:action*** expression. In the case of a link's *startlink for object y*, its TargetAction must be specified by the ***y:start***.

The specification of TargetDexpr is done via a Boolean expression that combines the SourceConditions within this Link object. For example, the ***x:running*** and ***y:stopped*** dependency expressions can be specified to test if object x is running and if object y is stopped. Finally, the specification of a link is defined according to the following syntax: ***link-id = TargetAction ← TargetDexpr***, in which ***link-id*** is the Link identification. This nomenclature symbolizes that *Link-id* possesses a dependency expression that must be satisfied so that the TargetAction can be executed.

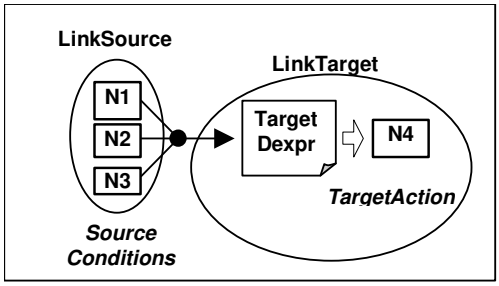

Figure 2 - A Link object representation

2.3: Application Example

A video stream transmitted over the Internet perfectly illustrates how the adaptive mechanism works. The original video is coded with ***Cx*** coder and stored in a video server. This compressed video stream must be decoded by running ***Dx*** object on the client machine, which continues until the end of the presentation if resource availability remains constant. Unfortunately, the amount of resources in the network and in the client machine varies over time due to concurrent processes that share these same resources. The situation described above in which the ***Dx*** object decodes the compressed stream minimizes the usage of network bandwidth. However ***Dx*** object consumes a huge amount of resources (CPU cycles and memory) of the client machine. If a resource shortage occurs due to other concurrent processes that are running on the client machine during this video presentation, an adaptive mechanism must be triggered to cope with this resource shortage. The main idea is to substitute object ***Dx*** by two alternative ***Dp*** and ***Df*** objects. The ***Dp*** object, which partially decodes the video stream from the server, is run on an adjacent machine (in the same subnet) while, as the final step, the ***Df*** object, which decodes the stream from the ***Dp*** object, is run on the client machine. In this way, resources are freed in the client machine since object ***Df*** is a decoder that consumes fewer resources. In order to comply with this adaptive functionality, the objects listed in table 1 must be specified.

Object	Behaviour
Mc	Client Monitoring monitors resources in the machine.
Cx	Complex Coder codes original video.
Dx	Complex Decoder decodes Cx's stream.
Dp	Partial Decoder partially decodes Cx's stream into Cp's stream.
Df	Final Decoder completely decodes Cp's stream.
Uh	Uncompressed Handler handles uncompressed stream.
Sr	Stream Retransmitter redirects stream from video server to adjacent machine.

Table 1: Adaptive objects

As can be seen above, ***Dx***, a complex but efficient decoder, must decode the video that was coded by the ***Cx*** coder object. While the video stream is being decoded, ***Dx*** consumes most of the resources available on client machine ***A***. Since concurrent processes may end up consuming all of its remaining resources, it is necessary to trigger the partial **Dp** and the final ***Df*** decoders in order to block such an occurrence. In this regard, the ***Dp*** decoder must be run on adjacent machine ***B,*** which would then send the partially decoded stream to the client machine ***A.*** The ***Df*** decoder on the client machine ***A*** finally decodes the video stream from ***Dp*** object. The resulting scenario makes it possible to immediately detect any and all conditional dependencies among the objects. The expressions below describe these dependencies.

Stop (Dx in A) ← Mc(50% occupied)
Start (Dp in B) ← Mc(50% occupied) and (Dx:stopped)
Start (Df in A) ← (Dx:stopped) and (Dp:running)

If further resources are needed, the video stream must be totally decoded on the adjacent machine ***B*** by ***Dx*** object. Therefore, it is necessary to stop decoder ***Df*** on client machine ***A***, start both decoder object ***Dx*** on adjacent machine ***B*** and object ***Uh*** on client machine ***A***.

Stop (Df in A) ← Mc(95% occupied)
Stop (Dp in B) ← (Df:stopped)
Start (Dx in B) ← Mc(95% occupied) and (Df:stopped)
Start (Uh in A) ← (Df:stopped) and (Dx:running)

3: The Architecture

The system's architecture is based on the client-server paradigm. Two kinds of machines are involved: the controller and servers (Figure 3). Every server machine hosts a monitoring server object, which responds to requests from the controller machine to: (i) send object state information back to it and (ii) alter the state of the objects.

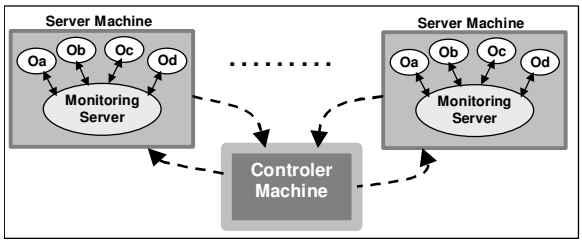

Figure 3 –Architecture

The system's architecture calls for a single **controller machine**, which is in charge of controlling the execution/adaptation of the application in accordance with the net of causality requirements derived from the conditional dependencies specification. This adaptation is also based on the object state information retrieved from the **monitoring-sever-object** in the server machines. The software components in the controller machine are structured in a layered manner (Figure 4). The **monitoring client layer**, the first component, is responsible for retrieving state information from **monitoring-sever-objects.** This information is stored in a specific table within this layer. This table also associates all the application objects with the respective machines that compose the application's environment. Each change in this table results in a notification to the **adaptation layer**. The **monitoring client layer** also sends messages to the **monitoring-server-objects** to control the states of the objects. These messages are related to the state of the objects as shown in Figure 1. The **adaptation layer**, the second component, is concerned with interpreting the **application specification**, which is a script file that defines the regular and adapted behavior. The adaptation is triggered in compliance with the state information recovered by the **monitoring client layer** and the a**pplication specification**. Each adaptation consists of: (i) finding out, via the dependency expressions, which actions should be taken (ii) signalling the **execution layer** what actions are to be performed (with the assigned objects). The **execution layer**, the third component, deals with the coordination of the application execution in accordance with the **adaptation layer** and based on the table information within the **client layer**.

This architecture can be better explained by the example mentioned in section 2. Ideally, the "Mc" (Client Monitor) object should be run on the **controller machine** to decrease the response time of the system since both the adaptation and the execution layers are hosted in this same machine. The other objects could be run in any of the machines in the system's environment. To illustrate these interactions, let's suppose that the client machine is 50% occupied. As soon as the **monitoring server object** hosted in the same machine receives this event, it sends an asynchronous message (trap) to the **monitoring client layer**, which updates its table and informs the **adaptation layer** of this event. The **adaptation layer** then signals the **execution layer** to stop "Dx" (Stop(Dx) ← Mc(50% occupied)). The **execution layer** acts on these objects by way of the **monitoring client layer**.

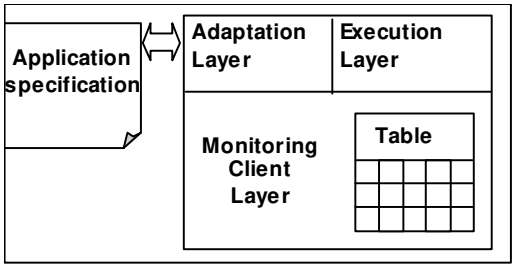

Figure 4 – The Adaptation Layered Architecture

4: Identifying inconsistencies

The specification of an adaptive application is determined in a two-step process: all the objects needed to perform the required functionalities are established, and the dependencies and the conditional links are set. To the extent that the specification of the conditional dependencies are critical (that defines the adapted behavior), a judicious verification process is carried out so that the correctness of the adaptation is guaranteed. The following three types of inconsistencies are the most common errors found in complex specifications. In section 5, we propose a verification mechanism that automatically points out those kinds of inconsistencies:

1) A chain of conditional links between original objects and their alternative ones may create loops, as illustrated in Figure 5. These loops may appear in three different ways: (i) an alternative object is simultaneously started and stoped, (ii) an alternative object is started, when it is running, and (iii) an alternative object is stoped, when it is not running anymore. Based on the example presented in section 2, the following inconsistencies may be erroneously introduced into the execution file by the programmer:

//First system fault
1 Stop (Dx in A) ← Mc(50% occupied)
2 Start (Dp in B) ← Mc(50% occupied) and (Dx:stopped)
3 Start (Sr in A) ← (Dx:stopped)
4 Start (Df in A) ← (Dp:started) and (Dx:stopped)
//Second system fault
Stop (Df in A) ← Mc(95% occupied) and (Df:running)
5 Start (Uh in A) ← (Df:stopped) //Figure 3b
6 Start (Dx in B) ← (Uh:started) and (Df:stopped)
7

//Resources available at client machine
8 Stop (Df in A) ← Mc(5% occupied) and (Df:running)
9 Stop (Dp in B) ← Mc(5% occupied) and (Df:stopped)
10 Start (Dx in A) ← Mc(5% occupied) and (Df:stopped)
11 Stop (Sr in A) ← (Dx:started)
//50% available resources
12 Stop (Uh in A) ← Mc(50% occupied) and (Uh: running)
13 Stop (Dx in B) ← (Dx:running) and (Uh:stopped)
14
15 Start (Dp in B) ← Mc(50% occupied) and (Uh: stopped)
Start (Df in A) ← Mc(50 % occupied) and (Dp:started)

Lines 1, 2 and 14 produce an inconsistency because the Dp object will be started twice in machine B. Lines 6 and 12 also produce an inconsistency because Uh is started and stopped at the same time (Figure 3).

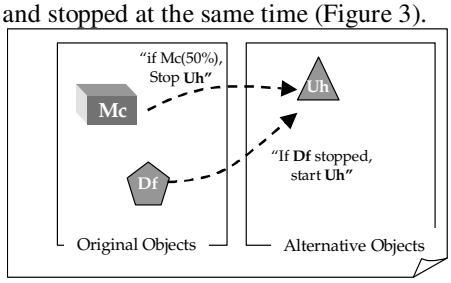

Figure 5 - Loop between Conditional links

2) A dependency expression of a conditional link can never be true. Consequently, the conditional link will never be triggered. As illustrated in Figure 6, the handler Uh will be started if Dp and Dx were stopped, but by the example below whenever Dx is stopped Dp is started, so they will never be at the stopped state at the same time.

1 Stop (Dx in A) ← Mc(50% occupied)
2 Start (Dp in B) ← (Dx:stopped)
3 Start (Df in A) ← (Dp:started) and (Dx:stopped)
4 Start (Uh in A) ← (Df:stopped) and (Dx:stopped)

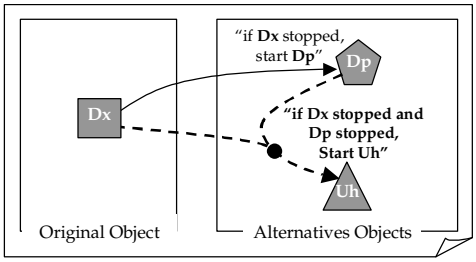

Figure 6 – Conditional Link never triggered.

3) The conditional links defined among objects that are started at different times, or objects that are started together but have distinct durations, may cause the interruption of an object that have already been executed. Figure 7.a shows the timeline of the original objects. Figure 7.b shows the conditional links specified within the specification. In the lines below, let ***tf*** be the instant when Df finishes its task, ***tp*** be the instant when ***Dp*** finishes its own, and ***ts*** be the instant when Mc detects that the system is 95% occupied. ***Uh*** and ***Dx*** must replace ***Dp*** and ***Df*** respectively at ***ts*** instant. If ***ts*** < ***t***p the objects will be replaced correctly. However, if ***tp*** < ***ts*** ≤ ***tf***, there is an inconsistency because ***Dx*** will never be started since ***Dp*** has already been stopped.

1 Stop (Df in A) ← Mc(95% occupied)
2 Stop (Dp in B) ← Mc(95% occupied)
3 Start (Uh in A) ← Mc(95% occupied) and (Df:interrupted)
4 Start (Dx in B) ← Mc(95% occupied) and (Dp:interrupted)

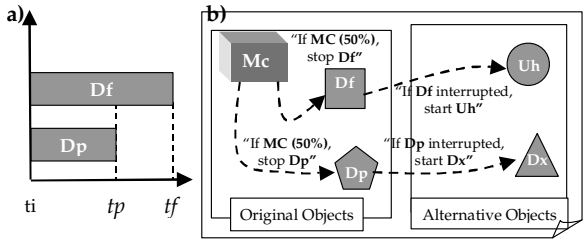

Figure 7 - Conditional Link deactivating an object that has already been executed ($t_p<t_s\leq t_f$)

5: Verification procedures

Specification inconsistencies can be detected by creating a list of all possible events related to all application's objects. List analysis enables to identify the first type of inconsistency described in section 4 (Figure 5), which consists of identifying for each object the existence of more than one event (**StartLink** or **StopLink**) referring to it. To represent all possible adaptations, it is necessary to keep track of all events within the **application specification**, which can make changes in the system's state. The simulation of those events is the first step in the verification process. Our solution is to use the controller machine to simulate those events, which is done by exchanging messages with the **adaptation layer.** Messages are sent to the **adaptation layer** as well as reports to the **execution layer** for every event generated. These messages are intercepted and used in our verification procedure.

The event simulation is also a solution that allows for the identification of the second type of inconsistency (Figure 6). After the execution of all simulations, it is possible to identify conditional links that have never been triggered.

The detection of the third type of inconsistency (Figure 7), which is time dependent, is a more complex task. In this case, the simulation of these events related to the same object in distinct moments allows the adaptation process to be verified. Once more, the simulation of events can be done as often as needed to validate the **application specification** since the **controller machine** is used. In this manner the **application specification** will only be validated if no changes occur in any object that no longer exists.

In order to determine the moment when these simulations must occur, the solution proposed in this paper makes use of a state diagram. In this diagram, transitions between states occur when at least one object is activated or deactivated. These transitions are represented by lists of events. The application state is defined by two sets of objects: the set of running objects (AM) and the set of objects that are not running (PM).

Figure **8** illustrates a state diagram related to the timeline showed in the same figure.

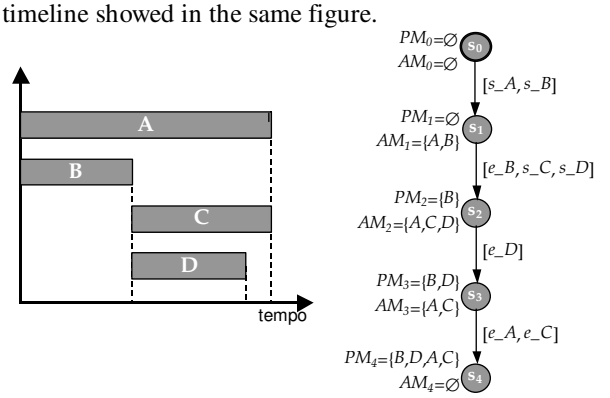

Figure 8 – State diagram of an adaptive document

The state diagram allows the adaptation behaviour to be analysed in accordance with the states in which the event is simulated. As previously described, inconsistencies are detected in any particular states. For example, the replacement of ***A*** by its alternative object is only classified as an inconsistency if the adaptation is activated in state s_i, where ***A*** is no longer activated (i.e. $A \notin AM_i$ and $A \in PM_i$).

In the state diagram, each states s_i is covered (with the exception of the first and last states) and for each object ***A*** that is running in the state ($A \in AM_i$), an event (ex. stopped) is simulated. For each event, an adaptation process is triggered and the **adaptation layer** is called. The **adaptation layer** executes the adaptation process and signals the changes that must be made to the **execution layer**. By intercepting those signals, it is possible to create a list of all actions and events that has been triggered.

For each list created, a new state diagram is built to represent it. This new state diagram is then added to the original one. To illustrate this process, lets consider the example shown in Figure 9. A timeline of the original objects (Figure 9.a) is shown and its state diagram (Figure 9.b). An interruption event is simulated in object ***C*** inside the state s_{A2}. According to the conditional links defined in Figure 9.c, the adaptation consists of replacing object ***C*** by object ***F***. Figure 10 shows a timeline of the adapted objects (Figure 10.a) and its corresponding state diagram (Figure 10.b).

Suppose that the ***C*** object crashes. The ***F*** object must be activated from the beginning. Since, the ***F*** object is started only in s_{B2}, the first two states can be ignored. Therefore, the adaptive process is really started

in s_{B2} (Figure 10.b). Both diagrams (the regular and adapted one) are joined together with a new transition. This new transition connects the state where the interruption event occur (s_{A2}) and the state in which the adaptation is started (s_{B2}). The event list that represents this transition is defined in terms of a set of activated objects from both states. In the example, $M_{A2}=\{A, C\}$ and $M_{B2}=\{A, F\}$, the event list, about the transition between s_{A2} and s_{B2}, must be ***[e_C, s_F]*** (Figure 11.a).

The adapted set of objects generated by the aggregation process branch off from the state where the interruption was simulated. Depending on the result of the adaptation process, this branch may return to the main path (defined by the original objects), which occurs when the adaptation process only handles (replaces or cancel) objects from a specific fragment of the state diagram. In this manner, the final states from the adapted set of objects are equivalent to the original set of objects. In the last example, described in Figure 11.a, the final s_{B3} and s_{B4} states are equivalent to the s_{A3} and s_{B4} states. Therefore, this equivalency allows s_{B3} and s_{B4} to be ignored. Thus, the s_{A2} state becomes the subsequent state of s_{A3}. The final result from the aggregation process is shown in Figure 11.b. In this specific example shown in Figure 11, it is important to notice that the last state in the adapted diagram is always equivalent to the last state in the original one.

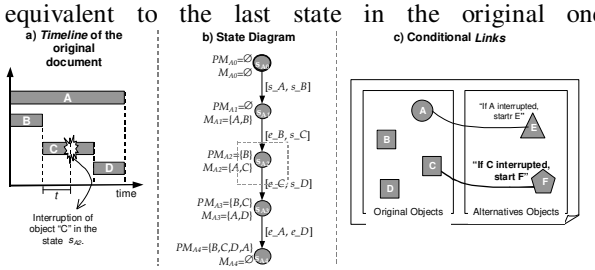

Figure 9 - The adaptive document

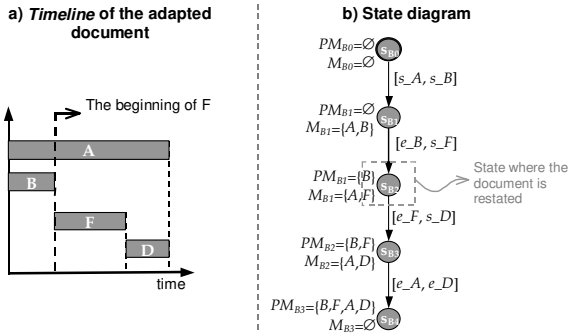

Figure 10 - The adapted document

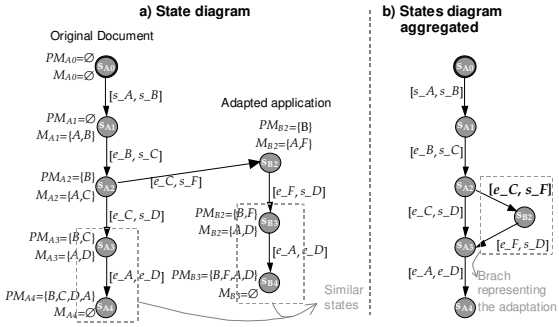

Figure 11 - State diagram aggregation

Finally, after the simulation of all possible adaptations, a state diagram is built. This final aggregated diagram describes the whole gamut of adaptive behavior. For each adaptive branch, a list of triggered **stopLinks** and **startLinks** can be seen.

6: Conclusions

In this paper, we have presented a new way of adapting distributed systems that combines the traditional temporal model with an approach based on the specification of causal relationships among objects. The integration of the link concept with dependency expressions provides a powerful tool for specifying adaptive applications that are easily manipulated. The novelty of our approach is that the implementation of verification tools to find out inconsistent behavior and the framework to coordinate distributed applications rely on the same mechanisms.

This approach has already been used in distributed multimedia systems and all mechanisms discussed in this paper have been intensely tested inside the **ServiMidia** Project. The key difference from multimedia specification is that the regular and adapted behavior (conditional dependencies) is defined in a single file. The scheme adopted in **ServiMidia** specifies two different files. One file stores the original multimedia document (SMIL language) while the other file, the adapted multimedia document. This scheme is employed to preserve backward compatibility with legacy multimedia systems and also to simplify the development process of a player for adaptive documents.

Depending on the area of interest (control systems, building automation...), we can foresee the use of the same strategy.

References

[1] E.C. Cunha, L.F.R.C. Carmo, L. Pirmez, "Design of an Integrated Environment for Adaptive Multimedia Document Presentation Through Real Time Monitoring", Lecture Notes in Computer Sciences v. 1718, Springer Verlag, 1999.

[2] J. Goldberg, I. Greenberg, and T. Lawrence. "Adaptive fault tolerance". In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 127–132, Oct 1993.

[3] Lea R., Yokote Y. and Itoh J., "Adaptive Operating System design using reflection" In Proceedings of the 5 th Workshop on Hot Topics on Operating Systems, pages 95--100, 1995.

[4] Fabio Kon, Roy Campbell, Marshall Mickunas, Klara Nahrstedt, and Francisco Ballesteros. "2k: *A distributed operating system for dynamic heterogeneous environments"*. In IEEE International High Performance Distributed Computing (HPDC), 2000.

[5] E. Nett, M. Gergeleit, and M. Mock "An Adaptive Approach to Object-Oriented Real-Time Computing" - IEEE. Published in the Proceedings of ISORC'98, 20-22 April 1998 in Kyoto, Japan.

[6] Fabian E. Bustamante, Greg Eisenhauer, Patrick Widener, Karsten Schwan, and Calton Pu "Active Streams: An approach to adaptive distributed systems" - In Proc. 8th Workshop on Hot Topics in Operating Systems (HotOs-VIII), 2001.

[7] Hiltunen, Matti A. and Schlichting, Richard D., "Adaptive Distributed and Fault-Tolerant Systems" Department of Computer Science - University of Arizona June, 1995

[8] B. Bhargava, K. Friesen, A. Helal, and J. Riedl. "Adaptability experiments in the RAID distributed database system." In *Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems*, pages 76–85, 1990.

[9] T. Bihari and K. Schwan. "Dynamic adaptation of real-time software." *ACM Transaction on ComputerSystems*, 9(2):143–174, May 1991.

[10] K. Schwan, T. Bihari, and B. Blake. "Adaptive, reliable software for distributed and parallel real-time systems". In *Proceedings of the 6th IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 32–42, Mar 1987.

[11] D. Schmidt, D. Box, and T. Suda. "ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment". *Concurrency: Practice and Experience*, 5(4):269–286, Jun 1993.

[12] ISO/IEC DIS 13522-5, "Information Technology Coding of Multimedia and Hypermedia Information, Part 5: Support for Base-Level Interactive Applications, MHEG-5 IS Document Pre-release 5", 1996.

[13] J.P. Courtiat, L.F.R.C. Carmo, R.C. de Oliveira, "A General-purpose Multimedia Synchronization Mechanism Based on Causal Relations", IEEE Journal on Selected Areas in Communications - Synchronization Issues in Multimedia Communications, Vol. 14, N. 1, January, 1996.

[14] G. Blakowski, R. Steinmetz, "A Media synchronization Survey: Reference Model, Specification, and Case Studies", IEEE Journal on Selected Areas in Communications - Synchronization Issues in Multimedia Communications, Vol. 14, N. 1, January, 1996.

[15] Aurrecoechea C, Campbell AT, Hauw L, "A survey of QoS architectures", ACM Multimedia Systems 6:138-151, 1998.

[16] Hiroaki Higaki, et al, "Protolcols for Groups for of Pseudo-Active Replicated Objects" IEEE Computer Society Fifth.