Journal of Universal Computer Science, vol. 14, no. 13 (2008), 2212-2238
submitted: 31/10/07, accepted: 27/6/08, appeared: 1/07/08 © J.UCS

A Safe Dynamic Adaptation Framework for Aspect-Oriented Software Development

Miguel A. Pérez-Toledano, Amparo Navasa, Juan M. Murillo
(Quercus Software Engineering Group
University of Extremadura, Cáceres, Spain
{toledano, amparonm, juanmamu}@unex.es)

Carlos Canal
(GISUM Group
University of Málaga, Spain
canal@lcc.uma.es)

Abstract: One focus of current software development is the re-use of components in the construction of systems. Software Adaptation facilitates the consequent need to adapt these components to the new environment by employing adaptors which are obtained automatically and hence with a certain guarantee of suitability, from formal descriptions of the interface behaviour. One appropriate technique for Software Adaptation is Aspect-Oriented Programming (AOP) which makes use of aspects to facilitate the dynamic adaptation of components transparently and non-intrusively. However, owing to the way that aspects are integrated, these can unexpectedly modify the functionality of the system, and consequently completely alter its semantics. It is hence necessary to study the final behaviour of the system to ensure its correctness after adding aspects for its adaptation. This study must go beyond just detecting problems at the protocol level, to analyze the potential semantic problems. This is the main focus of the present communication. We start from the Unified Modeling Language (UML 2.0) specification of both the initial system and the aspects. This specification is validated by generating an algebraic Calculus of Communicating Systems (CCS) description of the system. Next, extended (finite) state machines are automatically generated to verify, simulate, and test the modeled system's behaviour. The result of that process can also be compared with the behaviour of the new running system. To facilitate this task, we propose grouping components so as to centre the study on the points actually affected by the behaviour of the aspects.

Keywords: Aspect-Oriented Programming, Software Adaptation, UML, CCS, Extended State Machines, Interaction Pattern Specification.
Categories: D.2.7, I.6.5, I.6.4

1 Introduction

Companies' information systems change rapidly, and their existing software has to evolve without negatively affecting the comprehension, modularity, and quality of those systems. The development of component-based systems allows one to re-use software, thus reducing delivery times and costs without affecting quality.

In this context, Software Adaptation provides the tools needed to integrate new components into a system with the use of adaptors. These are software entities designed to obviate problems of interactions between the software system and the

new element to integrate. It is advisable for these adaptors to be obtained automatically in order to guarantee their correctness. Several formalisms are in use to describe the interface behaviour of the components. Examples are Process Algebra and Labeled Transition Systems. These formalisms allow one to model the interface behaviour and to check that the system is deadlock free ([Inverardi, 03], [Canal, 06], [Inverardi, 04]).

Nevertheless, the description of the specifications from which the adaptors are obtained is a complex task. The adaptors should also be designed without having first to prepare the elements that are being adapted. It is in this context that Aspect-Oriented Programming (AOP) is such an appropriate tool, since it allows code to be adapted transparently and non-intrusively. This is possible because AOP applies Quantification and Obliviousness Principles -[Filman, 05]-: "Quantification refers to the ability to write unitary and separate statements that have effect in many non-local places in the system. Obliviousness means that the places these quantifications apply do not have to be specifically prepared to receive them. In particular, an aspect must be able to affect several modules, while modules receiving aspects should not have to be specially prepared for this purpose". In this way, aspect technology can be used to adapt new behaviour dynamically at run-time. In addition, using aspect oriented technology provides support not only for behavioural adaptation but for semantic adaptation as well: since aspects can alter the functionality of affected components, the semantics can be completely changed.

The use of aspects has unquestionable advantages over standard Software Adaptation [Canal, 06], but it has some disadvantages too. The most significant disadvantage is that with traditional adaptation techniques, adaptors are automatically generated using methods that guarantee their correctness. The difficult analogous task of automatic generation of aspects in adapting a system is a current line of research. The published work that does exist addressing the automatic generation of adaptor aspects [Autili, 07] has focused on adaptation at the protocol level, so that aspects that are designed to adapt the system semantics are still generated manually. Therefore, mechanisms must be provided to check their correctness. For that reason, it is necessary to analyze what has occurred with the integration of the aspect and its effect on the system. This analysis can not be limited to detecting problems of protocols among components, but must also analyze the possible semantic problems caused, because the adaptation of new components can change the semantics of the system. This requires specifying the aspect's behaviour and checking the correctness of its integration. Furthermore, the specification should be intuitive and scalable.

In order to study the overall result of adapting a software system by adding new aspects, we propose the use of Unified Modeling Language (UML 2.0) specifications of the system under consideration. These specifications are obtained during the system's analysis and design phase, and must be updated with the descriptions of the aspects to integrate. The aspects' behaviour is described using an Interaction Pattern Specification (IPS) protocol [France, 04]. These patterns are coded by means of sequence diagrams, and are integrated into the system's UML specification at places where the aspects are to be applied. The resulting documentation, completed with the remaining UML diagrams, allows one to obtain finite state machines from each of the system components. The set of machines thus obtained models the expected behaviour of the system. This model will be used to validate, test, and simulate the

behaviour of the integrated aspect. Comparing the model's behaviour and properties with those of the software code obtained on including the aspects will allow one to check that the code behaves as expected. To facilitate these operations we propose compositional verification, grouping the state machines in order to focus on the study of the components involved, and hence reduce the magnitude of the simulation and model checking operations.

This paper is an improved and extended version of previous paper ([Pérez-Toledano, 07]) in which aspects are described by means of tables and examples to detail the steps of this proposal. In addition, it includes the definition of extended state machines and composition operations.

The article is organized as follows: Section 2 describes the use of aspects in software adaptation and the problems with checking for correctness, Section 3 presents our proposal, Section 4 discusses related work, and Section 5 presents the conclusions. Finally, the acknowledgements and references are presented.

2 Problem description

Several problems arise when one needs to adapt software by integrating a new component into the system. The different problems of interoperability may be categorized into four levels [Becker, 07]:

- Signature Level. These are problems of syntax between the signatures of the interfaces of the components that have to be adapted. This kind of problem will not be dealt with in the present study.
- Behavioural Level. These are problems caused by the protocols for the use of the methods defined in the interfaces of the adapted components.
- Semantic Level. These are questions of whether the new adaptations really provide the required behaviour.
- Service Level. These are questions of whether the new adaptations conserve the non-functional properties of the system (security, reply,…).

As was noted above, the use of AOP to adapt software has certain advantages deriving from the application of the Quantification and Obliviousness Principles, but it does not allow one to obtain adaptors automatically. This makes it difficult to know whether the adaptation is correct. Moreover, the focus of AOP is not just the adaptation of software. Its goal is rather to provide new modularization techniques to solve the problems caused by crosscutting concerns which it does by isolating the crosscutting concerns in modules called aspects[1]. The adaptation of software using aspect-oriented technologies creates new interoperability problems in the systems thus constructed. They can be summarized in the following points [McEachen, 05]:

1) Unintended aspect effects. Pointcuts of new aspects may be applied to undesired join points, which could provoke unintended side effects.

[1] An aspect executes a method (advice) when a condition (a regular expression called a pointcut) is satisfied during the execution of an application. The points where the advice is executed, interrupting code execution, are called join points.

2) Arbitrary aspect precedence. Pointcuts of new aspects may be applied to the same join point as other (unknown) aspects already are. This may cause problems with the sequence of application of the aspects.
3) Unknown aspect assumptions. When pointcuts of new aspects are applied, they may not find join points matching existing requirements.
4) Partial weaving. When the code of a system is modified, the aspects within it may not be applied to future modifications.
5) Incorrect changes to control dependencies. The advice type 'around' can alter the behavioural semantics of a system. This advice type is termed 'around', as against 'before' or 'after', in the sense of a bypass in that it executes a method in place of the join point it operates over.
6) Failure to preserve state invariants. When an aspect is applied, it could break the system's state invariants.

These situations complicate the study of the interoperability problems mentioned above. Points 1–4 mainly affect the Behavioural Level, because they can modify the correct sequence in which the methods defined within component interfaces must be instantiated. Points 5 and 6 mainly affect the Semantic and Service Levels, because they can modify a component's expected behaviour and the system's non-functional properties.

When software is adapted using automatically obtained adaptors, syntactic problems (Signature Level) and problems of deadlocks between the protocols of components which are being adapted are solved (Behavioural Level) ([Inverardi, 03] [Inverardi, 04]). Instead, when the adaptation is by means of aspects, there is no starting formalism with which to study how the adaptation has been performed. This drawback is aggravated by the fact that the application of aspects adds new problems at this level. It is therefore necessary to consider solutions that allow one to study the integration or deletion of aspects in a system, and to test the constructed system's correctness. This study can not be limited to detecting problems of protocols among components, but must also analyze the possible semantic problems caused by the adaptation.

3 The framework

To adapt a software system using AOP, whether adding new services (functional adaptation) or updating some existing service (technical adaptation), one needs to specify the changes in order to study how they affect the system. It is also possible, depending on the type of study, that specifying the interface behaviour will not be enough. The behaviour and properties of the adapted system are best studied on a model constructed with the purpose of performing simulation, testing, and model-checking operations. One can then investigate any possible problems found at the Behavioural, Semantic, or Service Levels, and thereby implement dynamic adaptations[2] with safety.

Nevertheless, the model constructed must take certain considerations into account in order to be efficient. First, it must be scalable to facilitate the adaptation of new

[2] Dynamic adaptation refers to a component's being adapted at runtime.

aspects. It must be adaptable to each aspect to facilitate the individual study of each aspect's behaviour. It must be as complete and precise as possible to facilitate model-checking operations. Finally, the description of the system should be by means of some graphical tool to facilitate the intuitive comprehension of the model.

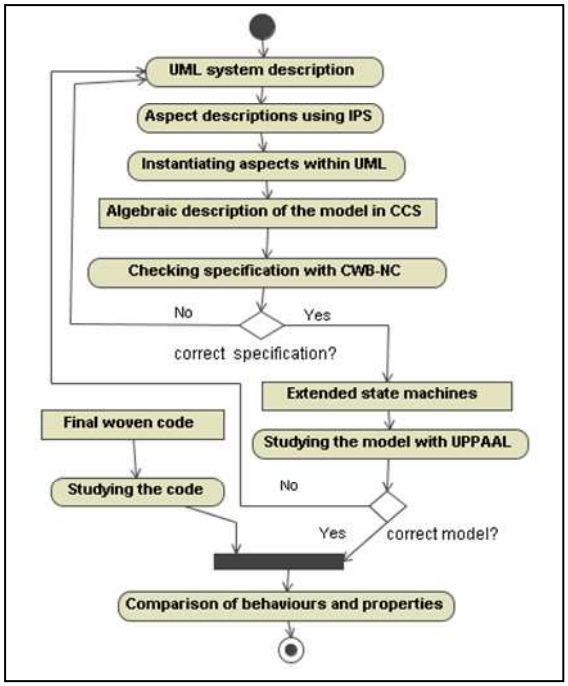


Figure 1: The TITAN activity diagram.

Given this context, in this section we present our proposal TITAN, a framework that allows one to model a system and to study the integration of aspects within it. The fundamental activity diagram of TITAN is shown in Figure 1, in which round-cornered rectangles represent actions performed within the framework, and rectangles represent testable and verifiable representations of the models. In brief, the use of TITAN begins with using UML to model the required behaviour before the adaptation. Then, for aspect modeling, TITAN uses the Interaction Pattern Specification (IPS) approach to describe the aspects' behaviour. Aspects are instantiated in the original UML specification. The next step is the validation of the UML specification. To that end, algebraic descriptions of the system are generated and used to perform checks of the model. The validated model is then used to generate extended state machines in order to verify, simulate, and test that the modeled behaviour is as the designer expects. Finally, that model is compared with the one produced by the extended code obtained from adding aspects, the aim being to detect mismatches. In this way, a check has been made as to whether the evolved system produces the expected behaviour. The following subsections describe these steps in more detail.

3.1 UML system description

The goal in TITAN is to insert the description of the aspects into the specifications obtained during the analysis and design phase of the system. Naturally then, the first step is to obtain a reference specification of the original system. System description is performed by using all the necessary UML diagrams, but TITAN focuses on the interaction flows described in sequence diagrams.

Sequence diagrams are a good starting point to specify the interactive behaviour of a system. They allow us to describe the main working scenarios initially known, and they can be completed and refined during system design. However, sequence diagrams have some limitations to be used as a modeling tool. Sequence diagrams provide a partial view of the system that must be contextualised with the rest of the scenarios in order to study the particular evolution of each participant. Different sequence diagrams are related by using UML invariant labels. These labels describe the state of participants inside the scenario. States represented with labels can be defined by means of OCL equations, variable vectors, or simply with state names. The labels are later used to check whether a pattern is applicable inside the scenario, as well as to build algebraic specifications and state machines. In order to achieve this, the lifeline (projection of the interactions of a participant inside a diagram) of each participant must start and end with a special label showing the state in which it starts and ends inside the described scenario. Therefore, labels have different aims: to describe the evolution of the different states that a participant can reach, to establish the current state of a participant, and to serve as a link between specifications from different diagrams.

3.1.1 UML system description: an example

The case study described in this section, inspired by the one used by Jacobson in [Jacobson, 05], will be used throughout the paper to illustrate the features and usage of the framework. Let us consider a system which provides functionality to a Hotel Management. Specifically, consider the requirements for a Room Reservation facility which will be used by the hotel customers. Figure 2 shows the corresponding use case diagram and the sequence diagram for the main scenario. When a customer wants to reserve a room, its availability is checked; then, if it is available, the room state is changed and a reservation is made. On the other hand, if it is not available, the customer is informed.

In the sequence diagram of Figure 2, both the initial and final state labels have been represented except for the “Customer” participant that only has the initial state label. The final state label of this participant is omitted deliberately in order to detect this situation later in the framework. In this example, in view of the fact that only one state of each participant is described, the initial and final labels of the rest of the participants coincide.

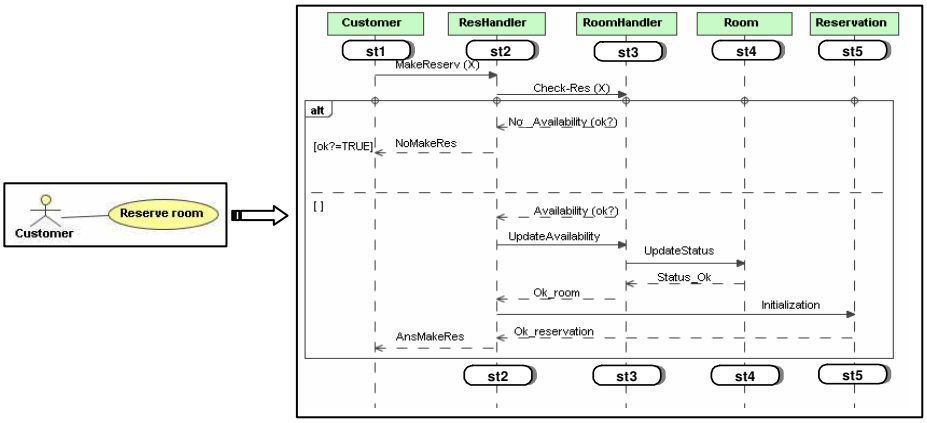


Figure 2: The Reserve Room use case and the associated sequence diagram.

3.2 Aspects description

Currently, UML is the most extensively used modeling language. But the modeling process is more complex when aspects have to be considered, and various solutions have been proposed to model aspect behaviour ([Elrad, 05], [Aldawud, 03], [Stein, 02], [Jacobson, 05]). TITAN starts by describing aspects as extensions of system use cases, in the same way as [Jacobson, 05]. Use cases affected by the aspects to be integrated are identified in a first description. Later, TITAN describes each extension by using the IPS approach to model aspects ([France, 04], [Araujo, 04]). IPS's are used to describe the interrelationships between the aspects and the system, representing interaction patterns by means of sequence diagrams. In an IPS, each participant represents a different role. A role is a UML metaclass that must be later instantiated by a UML element. That element must satisfy the partial order of the messages defined in the role's lifeline as well as other possible specified requirements.

Element	***Description***
Aspect Name	Specifies the name for the aspect
Join Point	Each cut point identified in the sequence diagram
Aspect Operation	Name of the aspect operation to be applied
Components	Names of the affected components
Event	Event type triggering application of the aspect
Pointcut	Conditions that must be satisfied allowing the aspect to be executed
When Clause	When the aspect can/must be applied

Table 1: Aspect description table.

After designing the aspect with UML, it is necessary to describe how to adapt it to the system. To this end, a table was designed –Table 1– [Navasa, 05] which lists the

points in the system at which the aspect must be applied, the type of event, the application conditions, the components involved in the adaptation, and when it is to be applied.

3.2.1 Aspects description: an example

With the description of the behaviour of the system (in the present example, the system is reduced to just one single scenario), an additional requirement has been modeled as an aspect. This aspect –Counter– is responsible for counting the number of room requests that could not be satisfied because the hotel was fully booked. The adaptation requires both the aspect's behaviour and how it is to be integrated into the system to be described. Following Jacobson, for the first of these tasks, the use case diagram in which the aspect is to be used –Figure 3– must be extended to show the addition of the new behaviour. To model the aspect's behaviour, IPS's will be used instead of sequence diagrams.

The IPS designed to describe the sequence of events when the reservation is not available (Figure 3) has references to role names and abstract methods (marked with a leading vertical bar in the diagram). There are two role names ("requester", and "replier"), and two events ("available?(x)", and "no_available(x)") which must be instantiated, linking the aspect ("Counter") to the rest of the system.

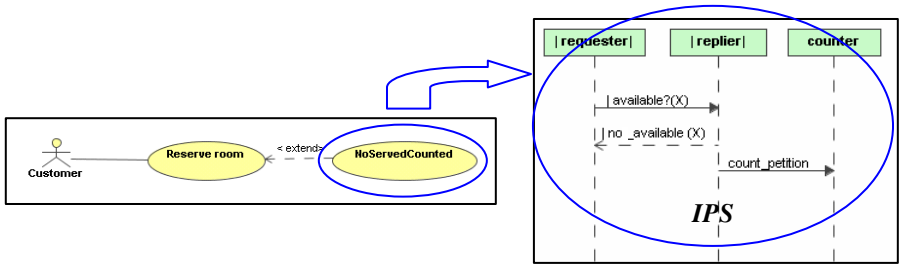


Figure 3: The Reserve Room use case extended and the Counter aspect IPS.

The use case information is completed by indicating how the adaptation to the system is to be performed. Table 2 shows how the Counter aspect is associated to the ending of the event "Check-Res", defined within the "RoomHandler" participant. Thus, it is specified that the aspect is to be launched when the condition "no rooms available" is satisfied. With this information, it is possible to proceed to the following phase, in which the appearance of the system will be adapted.

Element	*Value for the example*
Aspect Name	Counter
Join Point	Check-Res
Aspect Operation	count_petition
Components	RoomHandler
Event	Received Message Synchronous
Pointcut	NoRoomAvailable=True
When Clause	After

Table 2:"NoServedCounted" common items information.

3.3 Instantiating aspects within UML

Once the aspects have been modeled one needs to find points in the sequence diagrams of the model that match the requirements stated by the aspects. These are the join points where aspects will be applied. Notice that, depending on the description of the aspects (Table 1), different operations must be performed in order to instantiate the patterns. For each aspect to be integrated every message and role defined in an IPS must be linked to messages and participants defined in the system specifications. Further information about IPS instantiation can be found in [Whittle, 04].

3.3.1 Instantiating aspects within UML: an example

Taking Table 2 as described in Section 3.2.1 as referent, the IPS has now to be linked to the join points – in the present example, the role "|requester|" to the "ResHandler" element of Figure 3, and the role "|replier|" to the "RoomHandler" element. In addition, it is necessary to link the "|available?(x)" event to "Check-Res(x)", and "|no_available(x)" to "No_Availability(ok?)". Figure 4 represents the final scenario once the aspect has been integrated into the sequence diagram. When the process is complete, the resulting specification describes the expected behaviour of the system including the adapted component. Note that the adapted aspect –"Counter"– has been provided with initial and final state labels, as have the rest of the components of the diagram, in order to facilitate subsequent validation tasks. These validation tasks will be explained in the following sections.

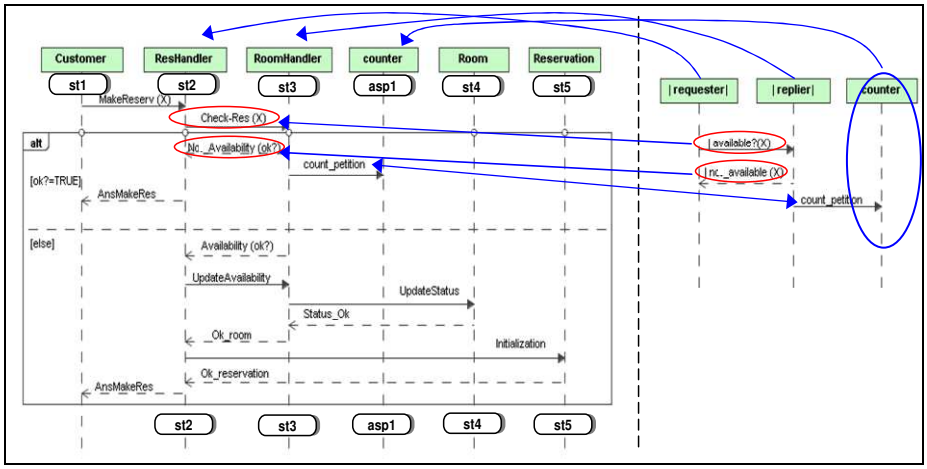


Figure 4: Counter aspect integrated into Figure 2.

3.4 Validating specifications

Validation is the process of gaining confidence that a system behaves as intended. The behaviour of the system is compared to the expectations of the designer who is validating the system. Validation is thus always inherently informal.

The first step in this phase is to compare the sequence diagrams with other UML diagrams such as state diagrams and class diagrams. This kind of study can usually be facilitated by UML modeling tools ([Simmonds, 05], [Straeten, 03]). Nevertheless, interoperability problems such as deadlocks or inconsistencies will not be detected using this analysis.

To cover this deficiency, TITAN automatically obtains Calculus of Communicating Systems (CCS) algebraic descriptions from the projections of the lifelines of the sequence diagrams[3]. The links in this case are the state labels. In these algebraic descriptions, each possible reference values of variables of the software system is abstracted by representing internal transitions -"t"-. The abstraction from internal transitions inside algebraic descriptions simplifies the analysis and allows errors to be detected. Since the aim is to check the model, these algebraic descriptions are exported to the tool 'Concurrency Workbench of the New Century' – CWB-NC- ([CWB-NC, 00]). Such model checking with algebraic descriptions allows one to detect gaps, deadlocks, and forbidden event sequences in the specifications [Uchitel, 04] – the usual problems at the 'Behavioural Level'. Any gaps detected must be filled by extending the specification with new scenarios which can be positive as well as negative (a negative scenario is a forbidden sequence of events). The objective is to construct a sufficiently meaningful, error-free, model of the system in a simple way.

[3] Algebraic descriptions are obtained from the internal graphical representation of the UML tool, since the description of UML fragments are not included in the OMG/XMI format.

A similar approach would be to use state machines for that purpose. However, performing validation with model checking using process algebras is easier because they facilitate the operations of comparing or detecting forbidden sequences of events, described by negative scenarios. This simplicity is due to the capacity of process algebra to abstract some operations by marking them as internal. The abstraction also simplifies the analysis because the operations will not depend on the values of the data received in each message. The operations of equivalence used in the validation are also facilitated by this feature, although the abstraction makes it impossible for the algebraic descriptions obtained to delimit the values used in decisions. TITAN does nonetheless use extended state machines to simulate the system's behaviour because they provide more precise information than CCS algebraic descriptions. However, the accuracy of these machines complicates the validation operations, which are simpler using algebraic descriptions because of the latter's abstraction capability.

The validation consists of two phases. In the first, temporal model checking operations are applied in order to detect deadlocks in the specifications. And in the second, if there exist forbidden sequences of events –described in UML 2.0 with the fragment "negative"– these are integrated into the CCS axioms and we need to apply process algebraic equivalence checking. These operations, however, provoke an increase in the number of states to study – the so-called state explosion problem.

There are different methods of alleviating the problem of state explosion. They may reduce the size of a state either by hiding some information, or by representing the information in a dense form. It is thus reasonable to expect that state explosion cannot be significantly alleviated without losing some capacity of analysis. To avoid incorrect answers, the designer needs to know which properties are preserved in the reduced state space, and use this information to guide the construction of the reduced space.

TITAN deals with state explosion using the compositional Labeled Transition System (LTS) construction ([Valmari, 98]). This method is commonly used in verifying the process-algebraic equivalence between systems with synchronous interprocess communication. Due to the properties of hiding and parallel composition applied to algebraic processes, one can hide those events in whose execution one is not interested, thus reducing the size of the processes that use those events, and then compose the system using these reductions. The goal of compositional LTS construction is to build a reduced state space equivalent to the full state space in the sense of some process equivalence. In this way, specification may be checked and questions of analysis may be answered with the same algorithms and tools as with full state spaces.

The validation process focuses on studying the coherence of specifications in such a way that the task only requires a major effort in the initial construction of the system, but not in subsequent adaptations. Each adaptation will most often only require partial validation, enabling actions not involved in the join point where the adaptation is applied to be hidden.

3.4.1 Validating specification: an example

Once the specifications have been obtained, the system is represented as the composition of all the components involved. These specifications are then exported

to CWB-NC for analysis. TITAN uses CWB-NC because it is a simple tool that allows straightforward execution of operations of equivalence and model checking.

The model checking operations are applied by means of Computation Tree Logic –CTL– ([Clarke, 86]) because the CTL model-checking algorithm is linear in both the state space and the property sizes. TITAN uses "check" and "search" commands to detect deadlocks in the specifications.

Figure 5 shows the CCS specifications of the scenario described in Figure 4, where *m* is used for the delivery of a message *m* and *'m* represents the corresponding reception, the operator "." indicates sequence and "+" indicates alternative behaviour, and "t" represents an internal action which provides an internal choice when combined with "+" as in the example in Figure 5.

In the example, model checking with the specifications from Figure 5 allows a deadlock in the system to be detected. The cause is that a state label was omitted. This situation arises when the "Customer" element requests a reservation. When the request is attended to, "Customer" evolves from its initial state ("st1") to an unspecified state. To solve this problem, a state label has to be integrated into "Customer" in order to describe the final state and to be able to return to this state when execution is finished.

```
proc st1 = MakeReserv. (t.'NoMakeRes.nil + t.'AnsMakeRes.nil)
proc st2 = 'MakeReserv. Check-Res. (t. 'No_Availability. NoMakeRes.
        st2 + t.'Availability. UpdateAvailability. 'Ok_room. Initialization.
        'Ok_reservation. AnsMakeRes. st2 )
proc st3 = 'Check-Res. (t. No_Availability. st3 + t. Availability.
        'UpdateAvailability. UpdateStatus. 'Status_Ok. Ok_room. st3)
proc asp1 = t. 'count_petition. asp1
proc st4 = t. 'UpdateStatus. Status_Ok. st4
proc st5 = t. 'Initialization. Ok_reservation. st5
…..
proc system = st1 | st2 | st3 | asp1 | st4 | st5
```

Figure 5: CCS algebraic descriptions obtained from Figure 4.

If there are forbidden sequences of events these are integrated into the CCS axioms and execution is tested within the system using the command "eq". Nevertheless, this kind of operation provokes an explosion in the number of states to consider. An example is the process "system" described in Figure 5, with a size of 6337 states and 42 679 transitions. Therefore, executing equivalence operations demands that the number of states of the processes involved be reduced. To this end, it is convenient to mark as internal transitions ("*t*") the events not involved in the sequence of forbidden events, to perform partial compositions, and to reduce the size of the process obtained by means of reduction algorithms. For example, by marking as internal transitions the events between the "RoomHandler" and "Room" participants –"UpdateStatus", "Status_OK"– and between the "ResHandler" and "Reservation" participants –"Initialization" and "Ok_reservation"– and by

minimizing the resulting process (with the "min" command), the size of the equivalent process has 181 states and 796 transitions, much more reasonable for this kind of operation. Nevertheless, if the size of the system is still too large, it is possible to execute partial validations among processes instead of using the final system obtained by composing all the processes.

3.5 Obtaining extended states machines

Once the specifications have been validated, extended state machines can be obtained automatically for each element of the system. There exist several algorithms to perform this task [Scenarios-Machines, 05]. TITAN uses the algorithm proposed in [Whittle, 00] which is based on selecting the lifelines of every scenario in which each participant is involved using labels as links between them.

An extended state machine is a labelled transition system with two types of labels: atomic labels and time labels. Let *Act* be the set of atomic labels of the system and let Δ be the set of timed labels, whose elements are denoted by $\varepsilon(d) \in \Delta$, $\forall d \in R^+$. Let L be the set of system labels such $L = Act \cup \Delta$. Let E the set of system variables, C be the set of clocks and D the set of data variables, such that Var= (E $\cup$ C $\cup$ D). Let the set of guards over Var be represented by G(Var).

We define an extended state machine as a quadruple $S = \langle s, s_0, \rightarrow, I \rangle$, where s is a set of states, s_0 is the initial state, $\rightarrow$ is a transition relation and $I : S \rightarrow$ G(Var) is an invariant assignment function for each state (this must be satisfied by all variable whilst operating in that state). Each state in this machine consists of the pair: $s=(l,u)$, where l is a node of the labelled transition system, and u is a variable assignment function s.t. for clocks X $\subseteq$ C, u: X $\rightarrow R^+$, for data variables Y $\subseteq$ D, u: Y $\rightarrow Z$ and for systems variables F $\subseteq$ E, u: F $\rightarrow Z$. The initial state s_0 is defined to be $s_0 = (l_0, u_0)$ *where* l_0 is the initial node of the transition system, and u_0 initialises all system variables, clocks and data variables.

The transitions for the extended states machine above:

$$\rightarrow = \langle l, g, a, r, l' \rangle$$

where l, l' are nodes of the machine and $a \in L$, but also:

- g is a guard, s.t . g ::=x~c where x $\in$ Var, ~::= < | $\leq$ | = | <> | > | $\geq$, and c$\in$ Var or c is a constant
- r is a assignment function s.t. for $r \subseteq$ Var, $[r \mapsto r']u$

The transition must satisfy the following rules:

$$(l,u) \xrightarrow{g,a,r} (l',u') \text{ if } g \text{ is satisfied by } u \text{ and } u' = [r \mapsto r']u$$

$$(l,u) \xrightarrow{\varepsilon(d)} (l',u') \text{ if } (l = l'),\ u' = u + d, \text{ and } u' \text{ satisfies } I(l')$$

With respect to the transitions, their main purpose is to represent the protocol of events produced during the execution of one participant. To give sense to the sequence, it is necessary to include some information in the transitions. This information represents each action described in the UML sequence diagrams. Therefore, each transition consists of three parts – condition, event, and action:

- "condition" is a Boolean expression, returning true or false. If condition evaluates to false, then neither the event nor the possible action is performed.
- "event" consists of a structure <*t, n*> where "*t*" describes the type of event (delivery "*!*" or reception "*?*") and "*n*" is the name of the event ($n \in Act$).
- "action" is needed when using loops or clocks in sequence diagrams. Actions can contain initialisation operations or simple arithmetic operations. The values of counter variables or clock variables are positive integers.

The resulting extended state machines provide more precise descriptions than classical statecharts. In particular, they provide the possibility of representing time requirements and complex operations over groups of events (such as critical regions) described with UML fragments. This provides support to perform more precise model checking than statecharts or CCS algebraic descriptions. There are many powerful model checkers available. Although in the current state of our proposal we do not consider real-time issues, since UML v2.0 specifications can include timing parameters, a real-time model checker will be needed for this task. We have chosen UPPAAL ([UPPAAL, 96]) for the simulation and model-checking operations of the model constructed. UPPAAL is an appropriate tool for distributed systems that can be modeled as a collection of processes with finite control structure and real-valued clocks, communicating through channels or shared variables. Its simulator enables one to examine possible dynamic executions of a system during the modeling stage, and its model-checker covers exhaustively the system's dynamic behaviour, and is also able to check invariant and reachability properties by exploring the state-space.

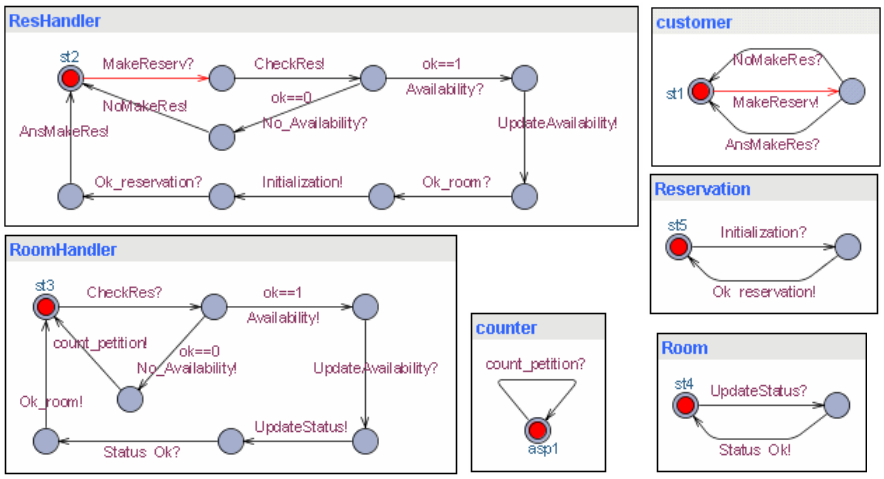


Figure 6: Extended state machines of the components described in Figure 5.

3.5.1 Obtaining extended state machines: an example

In the example of Section 3.4.1, once the deadlock has been eliminated, the next step is to obtain state machines for the system. In the extended state machines proposed by TITAN, the states contain a data structure representing information about the UML

fragments and the values of system variables. The transitions can represent event sequences as well as conditions on the variables, and their initialization and updating. Figure 6 shows the machines obtained from the example, and which will constitute the model to use as a reference in the rest of the steps to be described in this section. Figure 6 does not show the information associated with each node of the machines in order to allow the transitions that are produced to be seen clearly.

3.6 Studying the model

Once the model has been constructed, it will be tested to analyze how the planned adaptations behave. Simulation operations with the machines obtained allow problems at the Behavioural and Semantic Levels to be detected, i.e., those relative to the sequencing and the places at which the aspects are introduced, and to the expected behaviour of the system. A model-checking process completes the study, verifying operations at the Service Level. The model thus allows one to study how new adaptations affect the system before their actual implementation, and thus perform dynamic adaptations in a safe manner.

However, using extended state machines can provoke the state-explosion problem. This problem is managed using CTL, because CTL has an efficient model checking algorithm. Moreover, in order to reduce the number of states we can use the parallel composition to compose machines. The composition must observe the order of the traces defined. Thus, to study new adaptations inside the system, it is not necessary to study the evolution of each component, but only of those components involved. The selection of involved components can be managed using the tables obtained in the aspect description.

During the model checking operations of the model constructed, it is possible to detect errors in the original specifications that were not detected when the validation process was executed. This situation arises because algebraic descriptions allow one to increase the abstraction level at the cost of reducing the precision of the model. When a pathology of this type is detected, it is necessary to return to UML specifications in order to eliminate the problem.

3.6.1 Studying the model: an example

The end result of the model constructed –Figure 6– is a set of extended state machines. These machines model the expected behaviour of the system, and allow one to proceed with the simulation and testing stage. For the case of the present example, the operations must focus on discovering whether the modeled system behaviour is correct and whether the Counter aspect is applied at the correct places in the system. To this end, the simulation shown in Figure 7 describes the “Counter” aspect invocation. Moreover, different random and directed traces are generated modifying the value of the system variables to study the aspect invocation. These simulations are focused on changing the conditions in order to detect unintended join points or unrecognized aspect assumptions.

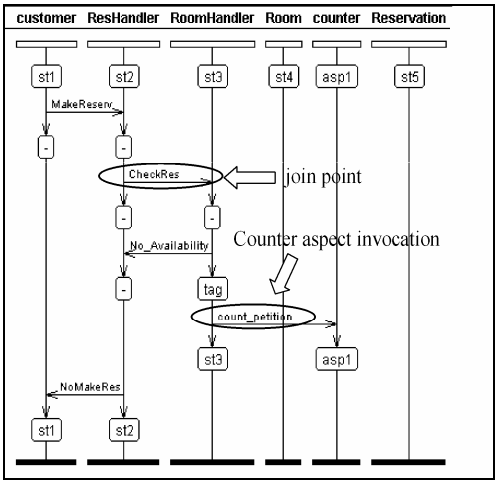
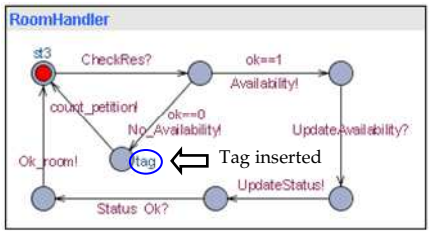


Figure 7: Trace describing the correct Counter aspect invocation.

Besides model simulation and testing, it is advisable to complete the study with model checking operations. The validation, safety, and liveness properties are checked using the CTL query language. The task can be facilitated by including labels in the state machine vertices to test the evolution of the system. Thus, in the example of Figure 8, the label "tag" has been added –in the "RoomHandler" component – to check when the "Counter" aspect is and is not launched. The figure also shows a set of equations designed to check the behaviour and properties of the model. One can now monitor the evolution of components (P0, P4), the possible values of variables (P1), the conditions satisfied during execution (P2, P3), and check whether the elimination of the indeterminism of the CCS equations leads to deadlocks in the construction of the machines (P5). At the side of each CTL equations is an indication of whether or not they have been satisfied by the system being modeled.



P0	E<> ResHandler.st2 and RoomHandler.st3	[True]
P1	A[] ok==1 or ok==0	[True]
P2	ok==1 --> RoomHandler.tag	[True]
P3	ok==0 --> RoomHandler.tag	[False]
P4	RoomHandler.st3 --> RoomHandler.tag	[False]
P5	A[] not deadlock	[True]

Figure 8: Model checking using "tag" inserted within the RoomHandler component

3.7 Adapting the model to aspects

Two of the prerequisites considered in the framework were the scalability of the model and that it should facilitate study of the integration of new aspects. The model of the system is scalable because the application of new aspects only requires their specification and integration into the system. The most complex task is the validation of the system with CCS. Fortunately, new adaptations do not need a complete system validation – it is enough to make partial validations. Partial validations are focused on the components involved by integrating aspects. The partial validation will most often only require temporal model checking, centred on detecting deadlocks or protocol problems among these components. This makes TITAN a tool that is suited to working with large software systems. Nevertheless, the final system may be very large, hindering the aspect adaptation study. In order to reduce this size, we propose grouping state machines together in order to make the simulation easier and to focus the study on the interesting points. Which groups are formed will depend on the developer's specific needs, but the point is that they will allow the creation of traces that exclusively contain the events of interest. A developer can form the groups by consulting the aspect description tables. Later, the tracing aspect can be designed to only monitor the events of a set of state machines, avoiding references to events that occur within grouped machines. This grouping process can be repeated as many times as necessary. It will thus be possible to obtain a minimal set of machines in which only events affected by the declaration of aspects intervene. Furthermore, since the machines are obtained automatically, the model can return to the starting point whenever convenient.

To compose state machines, they have to execute concurrently, and their intercommunication has to be synchronized. Given these premises, the objective of the composition of two machines is to construct a new and equivalent finite state machine which describes behaviour identical to that of the original two machines, while hiding their mutual synchronization aspects ([Blair, 99], [Jones, 99]).

3.7.1 Composition of extended state machines

Let S_1 and S_2 be two extended state machines. Let L be the set of system labels consisting of the set of events of the system (Act), and the set of timed actions (Δ) whose elements are denoted by $\varepsilon(d)$. $\varepsilon(d) \in \Delta$, $\forall d \in R^+$. $L = Act \cup \Delta$. Let S_A be a subset of Act consisting of the set of events that synchronise the execution of the two machines. Let $\otimes$ be the symbol that represents the composition of extended states machines. The result of composing two machines $S_1 = \langle s_1, s_{01}, \rightarrow, I_1 \rangle$ and $S_2 = \langle s_2, s_{02}, \rightarrow, I_2 \rangle$ can be defined as:

$$S_1 \otimes S_2 = \langle S, s_0, \rightarrow, I \rangle$$

where:

- S is the set of states such that $S \subseteq s_1 \times s_2$
- $s_0 \in S$ is the initial state such that $s_0 = (s_{01}, s_{02})$
- I is a function that associates each state with an invariant. This invariant must be satisfied by every variable that operates in that state.

$I(S_1 \otimes S_2) = I_1(S_1) \wedge I_2(S_2)$

- $\rightarrow = \langle l, g, a, r, l' \rangle$ is a transition relation that satisfies the following rules:

1. If there occurs a transition that only affects events from one of the two machines (S_1, S_2), then the resulting transition in the composition will be that which shows the evolution of the affected machine, maintaining the other in the same state. This situation can be represented formally:

$$\frac{S_1 \xrightarrow{g,a,r} S'_1,\ a \notin S_A \wedge a \in Act}{S_1 \otimes S_2 \xrightarrow{g,a,r} S'_1 \otimes S_2} \qquad (\text{Idem } S_2)$$

2. Let a_1 and a_2 be two complementary events. If there occurs a transition that affects a synchronization event, then the resulting transition in the composition will be that which shows the evolution of both machines at the same time. Again, formally this is:

$$\frac{S_1 \xrightarrow{g1,a1,r1} S'_1,\ S_2 \xrightarrow{g2,a2,r2} S'_2, a \in S_A}{S_1 \otimes S_2 \xrightarrow{g,a,r} S'_1 \otimes S'_2}$$

where $a = (a_1, a_2), (g = g_1 \wedge g_2), (r = r_1 \cup r_2)$.

3. Let S1 and S2 be two extended state machines with clocks declared within them. If there exist transitions in both of the machines simultaneously affected by the passage of time, then the resulting transition in the composition will be that which shows the passage of time in both machines:

$$\frac{S_1 \xrightarrow{\varepsilon(d)} S'_1,\ S_2 \xrightarrow{\varepsilon(d)} S'_2}{S_1 \otimes S_2 \xrightarrow{\varepsilon(d)} S'_1 \otimes S'_2}$$

where $S_i = (l_i, u_i), S'_i = (l'_i, u'_i), (l'_i = l_i), (u'_i = u_i + d), \forall d \in R^+$

3.7.2 Adapting the model: an example

After constructing and testing the model, the Hotel Reservation information system can now continue to evolve. New adaptations must be scalable in order to obtain an efficient framework. Nevertheless, as the size of the system increases, so does the number of state machines, making the study of new adaptations more difficult. In order to illustrate this problem, let us suppose that it is necessary to integrate a new component into the Hotel Reservation system. This component ("NewHandler" aspect) will perform a secondary search for rooms in other hotels belonging to the same hotel chain. This action must be performed whenever a room request can not be satisfied in the hotel. The new aspect is modelled and integrated into the original specifications following the procedures explained in Section 3.2. Table 2 is the description of how the new aspect has to be integrated into the system. But the join point and the description of how to apply this aspect coincide with the already existing aspect, and there exists no reference describing the order of application of these two aspects.

Element	Value for the example
Aspect Name	NewHandler
Join point	Check-Res
Aspect Operation	findroom ()
Component	RoomHandler
Event	Received Message Synchronous
Pointcut	NoRoomAvailable=True
When Clause	After

Table 2: Common Items information for the "NewHandler" aspect.

Therefore, following all the specifications could lead to the model constructed of the system being incorrect. Figure 9 is a possible sequence diagram resulting from this action, in which the new aspect has been included after the existing one. But the correct behaviour of the system requires the "NewHandler" aspect to operate before the "Counter" aspect in order to avoid counting room requests that are finally satisfied (in a different hotel).

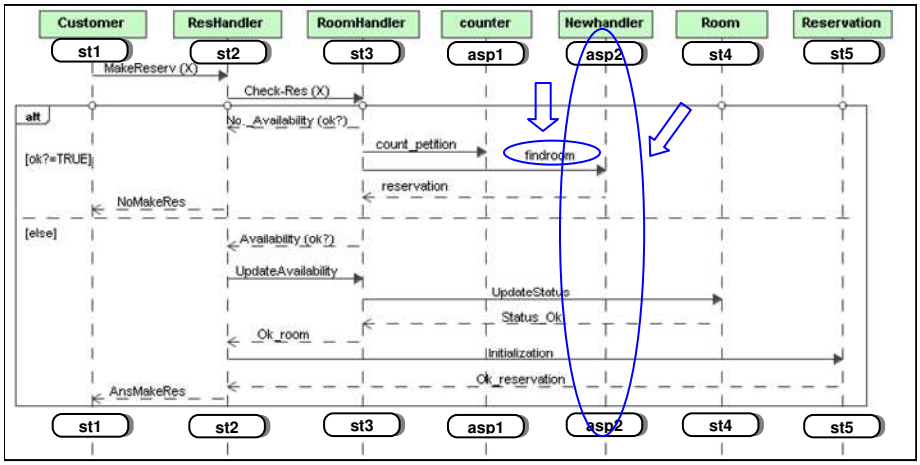


Figure 9: Sequence diagram with the Findroom aspect included.

Of course, this situation can be detected and corrected by simulation and model checking with the model that has been constructed. Nevertheless, as the size of the systems increases –this simple example has 7 extended state machines– performing these operations can be a complex task. TITAN allows machines to be grouped in order to simplify the problem. For the case of the present example, it is possible to consult aspect description tables and to apply the composition rules described in Section 3.7.1 to reduce the size of the system, allowing the study to focus on the "Counter" and "NewHandler" components only. As a first step, one can compose the

descriptions of the "Customer" and "ResHandler" components, leading to a new machine, "ResHandler2" (Figure 10).

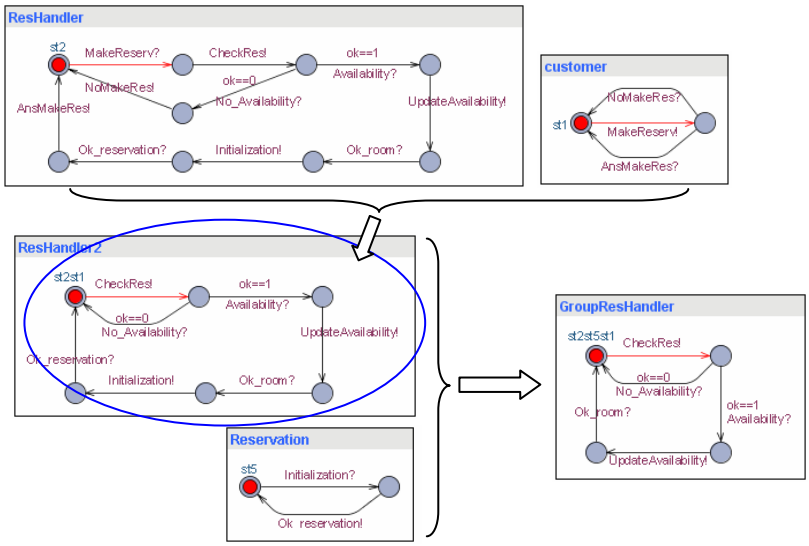


Figure 10: Grouping the machine Customer with ResHandler and Reservation.

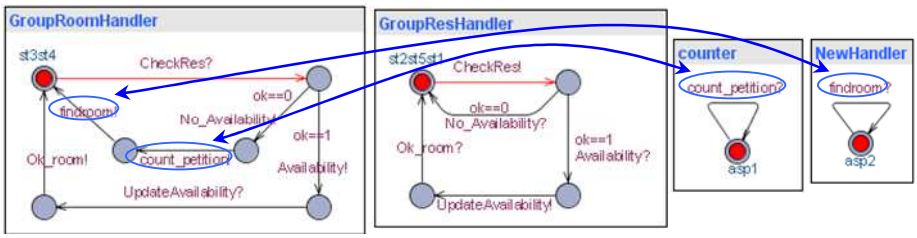


Figure 11: Model resulting from grouping the components Customer with ResHandler, and Reservation with RoomHandler and Room.

This process can be repeated, composing "ResHandler2" with the description of the "Reservation" component, leading to a new machine, "GroupResHandler". This machine will contain the description of all the grouped components, with the complementary messages among them being eliminated. Repeating this process, one can obtain a new machine "GroupRoomHandler" (Figure 11) which groups the "RoomHandler" and "Room" machines. In this way, the model to study has been reduced to the aspect components plus the two machines that launch and receive the requests. The figure shows the resulting model, marking the points at which the aspects are applied. It is now easier to apply the simulation and model checking operations described in Section 3.6, and hence to study the adaptation of new components more efficiently. Furthermore, since the process of constructing the

machines is automatic, after studying the adaptation one can rebuild the machines used before the composition from initial UML specifications.

3.8 Comparing the final code with the model

The study of the constructed model is not enough in itself to ensure the correctness of the code. Having modeled the behaviour produced by the planned adaptations, one then has to check that the code behaves the same. The comparison is based on ensuring that expected properties of the model are matched by the code, and vice versa. Model-checking techniques are used to study the properties of the two systems. We can use Java Pathfinder ([Pathfinder, 03]) to execute model-checking operations within the Java code that we obtain, and UPPAAL to execute model-checking in the model. This study must be completed by simulating the same execution traces in both systems. A trace represents an accepted events sequence. Each event in the sequence is a method invocation. To perform this task, there are two procedures possible:

- Generate tests from state machines. These traces simulate the execution of extended state machines and can be used as inputs in the generated code. To determine whether or not the code produces the same sequence of events as the state machines, a tracing aspect can be applied to the code.
- The same tracing aspect can be used to obtain the event sequence produced by the execution of the code. Such sequences can be simulated on the model.

Comparing traces will allow one to check that the code behaves as expected (Semantic Level) and that the sequence in which the methods must be instantiated is respected (Behavioural Level). Also, comparing properties will allow one to study whether the code's non-functional properties are modified (Service Level).

```
Public aspect Traceaspect {
    pointcut trace(): execution (*.*(..));
    After(): trace(){
    Signature sig = thisJointPointStaticPart.getSignature();
    System.out.println(sig.getDeclaringType().getname()+”.”+
        sig.getName());
     }
 }
```

Figure 12: Tracing aspect to monitor system events.

3.8.1 Comparing the final code with the model: an example

Focusing on the example, we shall use a tracing aspect that logs the trace of events described by the execution of the code in which the aspects have been applied. This trace will be imported into the UPPAAL simulator to check the behaviour of the machines that have been designed. The trace may contain a complete sequence of events or we may limit it to only a specific interesting sequence of events. To

implement either of these options, only the conditions described in the pointcut of the tracing aspect need to be considered. Figure 12 shows the code used in AspectJ ([AspectJ, 02]) to define a tracing aspect that logs the complete sequence of events produced in the example code.

The tracing aspect's output can be converted into an "xtr" format recognized by the UPPAAL simulator. Given that the model used for the simulation has been tested and describes the expected behaviour, each deadlock in the traces reveals some unexpected behaviour in the code which has to be resolved for the final system to match the specifications. In this way, one can study how the adaptation with new components in the code performs. Figure 13 illustrates how to use the trace generated by the tracing aspect in the UPPAAL simulator.

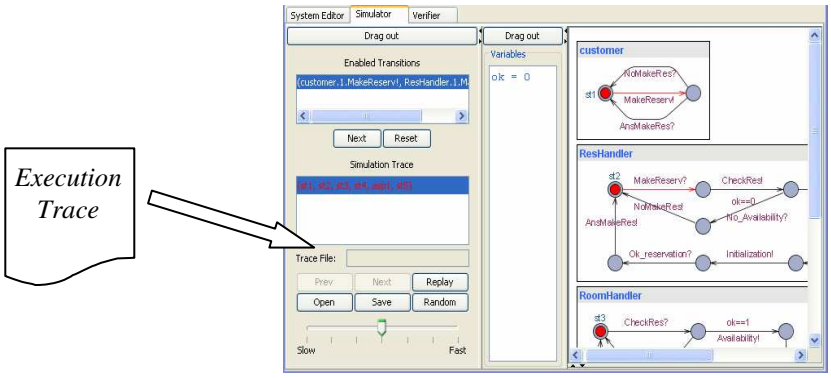


Figure 13: Importing a trace for a simulation in UPPAAL.

4 Related work

There exists no single useful technique to study every type of adaptation that might be applied. Recent research works present how to achieve safe adaptations using moderators [Sibertin, 06], a service-oriented approach [Ocello, 06], or B [Chouali, 2006]. In this section, we shall comment on some studies of the problems caused by the application of aspects.

Several works have considered the effect of adapting aspects to a system by means of restricting some characteristics. Some analyze the properties of the completed system (with the new aspects implemented) by using either model checking [Katz, 03] or static analysis of the code [Clifton, 02]. Others consider the inclusion of one aspect, studying how it affects the properties of a specific woven system [Krishnamurthi, 04] or all the possible ways of weaving it into the code [Deveraux, 03]. There have been studies which analyze how the inclusion of aspects affects a system by comparing the properties before and after their integration [Katz, 04], or which develop a fault model for aspect-oriented programming, including the different types of faults that may occur [Alexander, 04]. Nevertheless, these studies are based on testing the behaviour of already-constructed systems. Our approach is different in that TITAN creates a model of the system. Model-based testing improves the detection of errors and allows testing costs to be reduced because the testing process

can be semi-automated: more test cases can be run, thereby decreasing the number of errors [Beizer, 90]. The model constructed can also be compared to the final system, thus helping to verify the implementation.

Another class of related works consists of those that support system modeling and verification. Motorola Weavr is an add-in for aspect-oriented modeling in Telelogic Tau G2 [Weavr, 06]. This add-in provides support for the system-modeling verification, but with a focus on code generation, and describes system behaviour using statecharts instead of extended state machines. Since its focus is on code generation, it does not perform statechart grouping operations, so that with increasing size of the model, the difficulty in studying it also increases. Theme/UML [Baniassad, 05] is an analysis and design approach that supports the separation of concerns in those two phases of software lifecycle. Aspects are expressed in conceptual and design constructions called themes which are specified using UML packages identified by the stereotype "Theme". On completion of the design process, the result can be validated using Theme/Doc [Baniassad, 04]. TITAN, however, builds the model starting from valid specifications that are more precise than statecharts, allowing one to perform better analyses and simulations of the system than Theme/Doc.

Other approaches to achieving safe adaptations using aspects are Aspect-Oriented Architecture Description Languages (AO-ADLs), such as that developed by our research group [Navasa, 07]. ADLs are based on describing software architectures as a model of components, connectors, and configurations. Aspects are introduced into ADLs in different ways: as components ([Navasa, 07], [Barais, 05], [FuseJ, 07]), as connectors ([AspectualAcme, 06]), or as architectural views ([Katara, 03]). Nevertheless, the construction of architectures adapted to large software systems is a daunting task, whereas TITAN is more flexible since it allows one to study the integration of new aspects using the system's UML specification.

Summarizing, a system model has been built using validated specifications obtained during the analysis and design phases. The advantages of this proposal with respect to the more close related works in the field -[Baniassad, 05], [Weavr, 06], [Navasa, 07]- are the following:

1. The possibility of studying the behaviour of the successive adaptations, before and after they are integrated in the code.
2. The capacity of adjusting the model to each adaptation.
3. The possibility of making model checking operations more precise than in the related works already mentioned.

Taking these into account, we can conclude saying that, our proposal does not aim only at building a formal model of aspect inclusion, but constitutes a practical approach that uses different formal tools to study the behaviour of the system.

5 Conclusions

Aspects encapsulate the functionality of crosscutting concerns, and, thanks to the obliviousness principle, they can be added to or removed from the system at both the design phase and run-time. This characteristic facilitates the use of AOP as a software adaptation tool. However, the application of aspect-oriented techniques to adaptation

raises some new problems: there is not only the difficulty in determining whether or not the adaptations are safe but also whether the aspects are correctly integrated.

With system modeling, it is possible to simulate how a software system will be affected when different aspects are adapted. It is also possible to perform model checking of the properties of the model, and the results can be compared with those belonging to the woven code. Also, by means of trace simulation between the model of the system and the constructed code, one can study the woven code that results from the inclusion of an aspect. Using TITAN, the comparison between the model and the final woven code detects all the problems listed in Section 2 except the failure to preserve the state invariant (Point 6). But this problem too can be detected with TITAN by performing model-checking operations.

In sum, the main contributions of the present work are:

- Automatic generation of CCS algebraic descriptions from system specifications allows the detection of deadlocks and forbidden sequences. Hence, the framework allows one to study the coherence of the requirements described before the final system is actually constructed. These operations can lead to an explosion in the number of states to study, which is mitigated by using compositional LTS construction.
- From the specifications described using UML 2.0 sequence diagrams, the constructed model of the system allows one to code information about fragments, such as critical regions, loops, etc., and hence carry out far more precise operations of simulation and model checking than with other models.
- To facilitate this analysis, we have described how the state machines can be composed to reduce the size of the tests and focus the study on the classes involved. These features make the procedure scalable, and hence a suitable framework for modeling large software systems.
- Finally, state labels, represented in sequence diagrams, can be used to describe the aspects' join points. In this way, the join points can be described depending on the history of the components to which they are going to be applied.

With respect to the limitations of our proposal, TITAN needs to validate specifications manually in order to build a correct model. But this limitation only affects the first time that the system is being developed. Subsequent evolutions only require simpler validations. Finally, the description of every interaction can lead to an explosion in the number of states. This problem is mitigated by packing information and by composing participants in order to reduce the number of states. But the state explosion problem cannot be significantly alleviated without losing some capacity of analysis. In order to avoid incorrect analysis, the designer has to know which properties have been preserved within the reduced state space.

Acknowledgements

Thanks to the anonymous reviewers for their useful comments on the earlier draft of this paper. This research work is partially supported by CICYT project number TIN 2005-09405-C02-02 and 2PR04B011.

References

[Aldawud, 03] Aldawud T, and Bader A., "UML Profile for Aspect-Oriented Software Development". In Proccedings of the Third International Workshop on Aspect Oriented Modeling, 2003.

[Alexander, 04] Alexander R.T., Bieman J.M., and Andrews A.A. "Towards the Systematic Testing of Aspect-Oriented Programs", Technical Report, Colorado State University. http://www.cs.colostate.edu/~rta/publications/CS-04-105.pdf, 2006.

[Araujo, 04] Araujo J., Whittle J., Kim D., "Modeling and Composing Scenario-Based Requirements with Aspects", In Proceedings of the 12th IEEE International Requirements Engineering Conference, 2004, pp. 58-67.

[AspectJ, 02]. AspectJ. http://www.eclipse.org/aspectj/, 2002.

[AspectualAcme, 06] AspectualAcme: http://www.leccomm.les.inf.pucrio.br/aspectualacme, 2006.

[Autili, 07] Autili M., Inverardi P., Navarra A. and Tivoli M.. "SYNTHESIS: a Tool for Automatically Assembling Correct and Distributed Component-Based Systems". In Proceedings of the International Conference on Software Engineering, Minneapolis (U.S.A). May 2007.

[Baniassad, 04] Baniassad E. and Clarke S. "Finding Aspect in Requirements with Theme/Doc". Proceeding Workshop on Early Aspect. Lancaster, UK, 2004.

[Baniassad, 05] Baniassad E., Clarke S. "Theme: Aspect Oriented Analysis and Design". http://www.dsg.cs.tcd.ie/index.php?category_id=353", 2005.

[Barais, 05] Barais O., Duchien, L., Le Meur, A-F. "A Framework to Specify Incremental Software Architectures Transformations". In Proceeding of 31st Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE Computer Society. Sep 2005.

[Becker, 07] Becker S., Canal C., Diakov N., Murillo J.M., Poizat P. and Tivoli M. "Cordination and Adaptation Techniques: Bridging the Gap Between Design and Implementation". Object-Oriented Technology. ECOOP 2006. Workshop Reader. Volume: LNCS 4379. Year: 2007, pp: 72-86.

[Beizer, 90] B. Beizer B.. "Software Testing Techniques". International Thomson Computer Press, 1990.

[Blair, 99] Blair L., Blair G. "Composition in Multiparadigm Specification Techniques". In Proceedings of the 3th International Conference on Formal Method for Open Object-Based Ditributed Systems. Pages:401-417, 1999.

[Canal, 06] Canal C., Murillo J.M., Poizat P. "Software Adaptation". L'Object Journal, Vol. 12, num 1/2006. pp 9-31.

[Chouali, 2006] Chouali S. "Cooperation between the B method and the automata theory to check the component interoperability". In FACS'2006, 3rd Int. Workshop on Formal Aspects of Components Software. Pages 211--227, 2006.

[Clarke, 86] Clarke, E.M., E.A. Emerson and A.P. Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications". In ACM Transactions on Programming Languages and Systems, 8(2). Pag:244- 263. 1986.

[Clifton, 02] Clifton C., Leavens G. "Observers and Assistants: a Proposal for Modular Aspect-Oriented Reasoning". In Proceedings of Foundations of Aspect Languages, 2002.

[CWB-NC, 00] Concurrency Workbench of the New Century (CWB-NC). http://www.cs.sunysb.edu/~cwb, 2000.

[Deveraux, 03] Devereux B. "Compositional Reasoning about Aspects Using Alternating-Time Logic", In Proceedings of Foundations of Aspect Languages, 2003.

[Elrad, 05] Elrad T., Aldawud O. and Bader A. "Expressing Aspects using UML Behaviour and Structural Diagrams". In Proceedings of Aspect-Oriented Software Development (edited by Filman, R.E. et al.). Addison-Wesley, 2005.

[Filman, 05] Fillman R., Friedman D. "Aspect Oriented Programming is Quantification and Obliviousness". In Proceedings of Aspect- Oriented Software Development (edited by Filman, R.E. et al.). Addison-Wesley, 2005.

[France, 04] France R.B., Kim D., Ghosh S., Song E. "A UML-Based Pattern Specification Technique". IEEE Transaction on Software Engineering. March 2004 (Vol. 30, No. 3) pp. 193-206.

[FuseJ, 07] FuseJ. "Unifying Aspects and Components". http://ssel.vub.ac.be/fusej/, 2007

[Inverardi, 03] Inverardi P. and Tivoli M. "Software Architecture for Correct Components Assembly". Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture. Volume: LNCS 2804. Year: 2003, pp: 92-121.

[Inverardi, 04] Inverardi P. and Tivoli M. "Automatic Failures-Free Connector Synthesis: An Example". In Proceedings of the 9th International Workshop, RISSEF 2002. Venice (Italy), October 2002, Volume: LNCS 2941. Year: 2004, pp: 184-197.

[Jacobson, 05] Jacobson, I., Pan-Wei, Ng. "Aspect-Oriented Software Development with Use Cases". Addison Wesley, 2005. ISBN 0-321-26888-1.

[Jones, 99] Jones T, Blair L. "A Tool-Suite for Simulating, Composing and Editing Timed Automata (Lusceta: User Manual Release 1.0". In Proceedings of the Workshop on Aspect-Oriented Programming. ECOOP, Lisbon (Portugal), 1999.

[Katara, 03] Katara M.and Katz S. "Architectural Views of Aspect". In Proceedings of Aspect Oriented Software Development, Boston, USA, 2003.

[Katz, 03] Katz S., Sihman M. "Aspect Validation Using Model Checking". In Proceedings of the Symposium de Verification LNCS 2772, June 2003, pp: 389-411.

[Katz, 04] Katz S. "Diagnosis of Harmful Aspects Using Aspect Languages". In Proceedings of Foundations of Aspect Languages, 2004.

[Krishnamurthi, 04] Krishnamurthi S., Fisler K., Greenberg M. "Verifying Aspect Advice Modularly", In Proceedings of International Conference on Foundations of Software Engineering, 2004.

[McEachen, 05] McEachen N., Alexander R.T. "Distributing Classes with Woven Concerns: an Exploration of Potential Fault Scenarios". In Proceedings of the 4th International Conference on Aspect-Oriented Software Development, 2005, pp: 192 – 200.

[Navasa, 05] Navasa A., Perez-Toledano M.A., Murillo, J.M. "Aspect Modelling at Architecture Design". In Proceedings of the 2nd European Workshop on Software Architecture. LNCS vol. 3527. Italy, 2005, pp 41-48.

[Navasa, 07] Navasa A., Perez-Toledano M.A., Murillo, J.M.: "AspectLEDA: Extending and ADL with Aspectual Concepts". In Proceedings of the First European Conference on Software Architecture. LNCS vol. 4758. Spain, 2007, pp 330-335.

[Ocello, 06] Occello A. and Dery-Pinna A.M. "Capitalizing Adaptation Safety: a Service Oriented Approach". In Proceedings of the Workshop on Coordination and Adaptation Techniques for Software Entities, ECOOP, Nantes (France) July 2006.

[Pathfinder, 03] Java Pathfinder. http://javapathfinder.sourceforge.net, 2003.

[Perez-Toledano, 07] Pérez-Toledano M.A., Navasa Martínez A., Murillo Rodríguez J.M., Canal C. "TiTan: a Framework for Aspect-Oriented System Evolution". In Proceedings of Second International Conference on Software Engineering Advances, IEEE Computer Society Press, 2007.

[Scenarios-Machines, 05] Proceedings of the International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, Saint-Louis (EEUU), May 2005.

[Sibertin, 06] Sibertin C., Mauran P., Padiou G. and Thi Xuan P. "Safe Dynamic Adaptation of Interaction Protocols". In Proceedings of the Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'06), ECOOP, Nantes (France) July 2006.

[Simmonds, 05] Simmonds J., Bastarrica, M. C. "A Tool for Automatic UML Model Consistency Checking". 20th IEEE International Conference on Automated Software Engineering (ASE2005), Long Beach, California, EEUU, pp: 431 - 432, 2005, ACM Press.

[Stein, 02] Stein D., Hanenberg S., and Unland R. "An UML Based Aspect-Oriented Design Notation for AspectJ". In Proceedings of the 1st International Conference on Aspect-oriented Software Development, ACM Press, 2002, pp: 106–112.

[Straeten, 03] Van Der Straeten R., Mens T., Simmonds J. Jonckers V. "Using Description Logic to Maintain Consistency Between UML Models". Lecture Notes in Computer Science 2863, pp: 326-340. Springer-Verlag, 2003

[Uchitel, 04] Uchitel S. "Incremental Elaboration of Scenario-Based Specifications and Behaviour Models using Implied Scenarios". ACM T.O.S.E.M. Volume 13, Issue 1 (January 2004), pp: 37 – 85.

[UPPAAL, 96] UPPAAL. http://uppaal.com, 1996.

[Valmari, 98] Valmari A. "The State Explosion Problem". Lecture Notes in Computer Science, Vol. 1491: Lectures on Petri Nets I: Basic Models, pages 429-528. Springer-Verlag, 1998.

[Weavr, 06] Motorola Weavr. http://www.iit.edu/~concur/weavr, 2006.

[Whittle, 00] Whittle J., Schumann J. "Generating Statechart Designs from Scenarios". In Proceedings of the 22nd International Conference on Software Engineering, 2000, pp: 314 – 323.

[Whittle, 04] Whittle J., Araujo J. "Scenario Modeling with Aspects". IEE Proceedings - Software -- August 2004 -- Volume 151, Issue 4, pp. 157-171.