Static Analysis of Aspect Interaction and Composition in Component Models

Abdelhakim Hannousse, Rémi Douence, Gilles Ardourel

Static Analysis of Aspect Interaction and Composition in Component Models

Abdelhakim Hannousse
Ascola and Aelos, EMN, INRIA, Lina
abdel-hakim.hannousse@emn.fr

Rémi Douence
Ascola, INRIA
remi.douence@inria.fr

Gilles Ardourel
Aelos, Université de Nantes, Lina
ardourel-g@univ-nantes.fr

Abstract

Component based software engineering and aspect orientation are claimed to be two complementary approaches. While the former ensures the modularity and the reusability of software entities, the latter enables the modularity of crosscutting concerns that cannot be modularized by regular components. Nowadays, several approaches and frameworks are dedicated to integrate aspects into component models. However, when several aspects are woven, interferences may appear which results on undesirable behaviors. The contribution of this paper is twofold. First, we show how aspectualized component models can be formally modeled in Uppaal model checker in order to detect potential interferences among aspects. Second, we provide an extendible catalog of composition operators used for aspect composition. We illustrate our general approach with an airport Internet service example.

1. Introduction

Component based software engineering, or CBSE in short [1], enables the modularization of concerns in terms of separate software entities called components. Each component provides a set of services and may require other services from other components. Components can be assembled in order to construct complex component systems. On the other hand, aspect oriented programming, or AOP in short [2], focus on the modularization of scattered and tangled concerns that cannot be modularized using regular software entities. Crosscutting concerns are not related to a specific paradigm and CBSE is not an exception. However, current works on CBSE focus only on the integration of aspects into component models missing the interferences that may appear when several aspects are woven to a system. Moreover, aspect interferences detection and resolution is still a challenge for AOP. In this paper we contribute by analyzing aspect interactions on component models. In a previous work [3], we introduced a declarative pointcut language VIL for component models. VIL enables the composition of aspects scattered on the architecture by reconfiguring systems. The implementation of aspect composition for Fractal component model [4] is given in [5] with an introduction to aspect interferences. In this paper, we extend our work in [5] by providing an architectural description language (ADL) to describe both structural and behavioral properties of components and aspects (section 3). We show how to formalize component systems using a set of transformation rules from our ADL to networks of automata (section 4). We use Uppaal model checker [6] to detect potential interferences among aspects in the result formal system (section 5). Finally, we present a catalog of composition operators to solve interferences (section 6). The catalog describes seven binary operator templates that can be instantiated for aspect composition. Our proposal is illustrated with a running example: an airport wireless connection.

2. Motivation Example: Airport Internet Access

Our case study is a simplified version of that given in [7, 8]. The example models an airport service for providing a wireless Internet connection for passengers. Free Internet access is granted to passengers owning valid flight tickets. A passenger uses his flight ticket number to login and access the network for an associated time to the ticket.

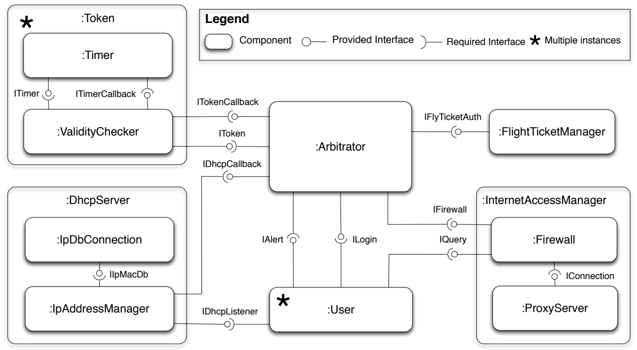


Figure 1. Architecture of the airport wireless access system

The component architecture of this application is depicted in Figure 1. The `User` component represents a passenger in the system. The `User`, first, requests an IP address from the `DhcpServer`, then it asks to login from the `Arbitrator`. Once connected, it sends queries to the `InternetAccessManager`. The `InternetAccessManager` forwards users' requests to the `Firewall` that blocks unauthorized Internet connections. The requests of users with enabled IP addresses are actually sent to the Net `Proxy`. The `User` component has multiple instances (one per customer) as noted with "⋆" in the figure. The `DhcpServer` delegates IP address requests to the `IpAddressManager` that provides dynamic allocation of IP addresses. The allocated IP addresses are managed by the `IpDbConnection` component. The `Token` models a user session. When the `Arbitrator` receives a login request, it retrieves the authorized access time from the `FlightTicketManager`, orders the `InternetAccessManager` to enable communications for the user, and starts a new session by

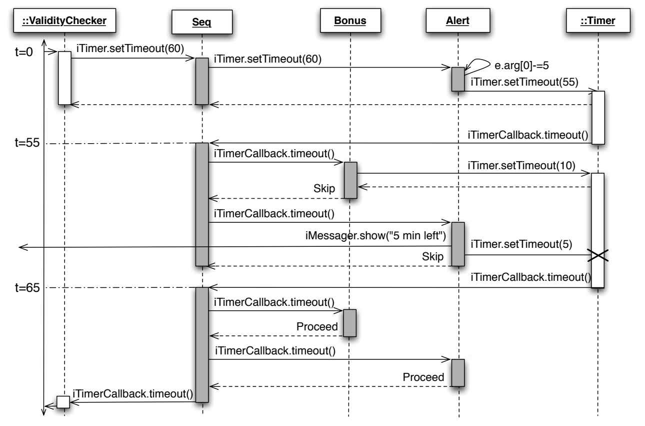


Figure 2. `Seq(Bonus,Alert)` scenario

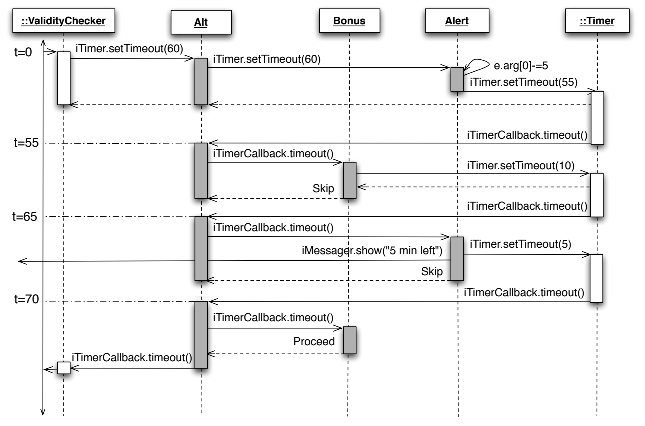


Figure 3. `Alt(Bonus,Alert)` scenario

instantiating a `Token` and starting its `Timer` component. When session time elapses, the `Timer` informs the `ValidityChecker` which in turn informs the `Arbitrator`. The `Arbitrator` closes the session by calling the `InternetAccessManager` to disable the user communications, and the `DhcpServer` to disable its IP address.

We want to alter the behavior of the system by integrating a set of aspects. In this paper we focus on three aspects: `Bonus`, `Alert` and `NetOverloading`. The `Bonus` adds an additional free connection time to first class passengers. The `Alert` warns the connected passengers of the end of their sessions time before five minutes, and the `NetOverloading` blocks the P2P connection queries when the number of connected passengers exceeds a threshold number. In fact, other aspects are proposed and implemented for the example, but for space limitation only these three aspects and their interactions are presented. From the implementation point of view, since the source code of components may not be available to component users, aspects should be integrated in a transparent and a modular way. In [5] we extended Fractal component controllers to implement and compose such aspects.

For better understanding of aspect interferences in component systems, let us consider the original airport system with the bonus and the alert aspects. Let us also assume that the original session duration is 60 minutes, the bonus adds 10 minutes and the alert warns the user 5 minutes before the end of the session. When both aspects are bound to the system, we wish users to get a bonus time and be alerted exactly 5 minutes before the actual end of their sessions (*i.e.* alert at 65 minutes, end of session at 70 minutes).

In AOP [2], the aspect behavior defines a set of advices associated to pointcut expressions defining the join points where the aspect should interfere. Each advice executes extra code and proceed or skip the intercepted join point. Proceeded join points continue their original path or continue to the next advice if any, while skipped join points are blocked. In our approach, when two aspects intercept common join points (*i.e.* services), the advices of the two aspects on those services are executed sequentially, and the service call continue its original path (*i.e.* proceeded) only if at least one of its underlying aspects decides to proceed the call, otherwise, the call is skipped. Figure 2 details the execution of the `Bonus` and the `Alert` behaviors sequentially. At time 0, a user logs in for 60 minutes provided by his flight ticket. The message `setTimeout(60)` sent from a `ValidityChecker` to its `Timer` is intercepted and forwarded to the `Alert` aspect only (*i.e.* the `setTimeout(60)` is not a common intercepted call). The `Alert` subtracts 5 minutes from 60 and proceeds the call with the new parameter value (55). As a result, a `timeout` service call is intercepted at time 55. The `timeout` is common intercepted service so that its call is forwarded to the `Bonus` first then to the `Alert`. The `Bonus` resets the timer for 10 minutes and skips the call. Then, the call is sent to `Alert` that warns the user, resets the timer for 5 minutes and skips the call. This violates the expected behavior: the alert is sent too early (at time 55 instead of the expected 65). Moreover, the `Timer` has been set twice with different values and hence it is inconsistent whatever happens next. This is called an interference since the desired behavior is not satisfied by the default sequential composition of aspects. To solve this interference, another composition strategy is needed: *the first occurrence of `timeout` should only be managed by `Bonus` and the second occurrence should only be managed by `Alert`.* Within this alternate strategy, when a service common to both aspects is intercepted, its occurrences are passed alternately to the left and the right hand side aspects. When a service is not common to both aspects, the call is forwarded to its corresponding aspect only. Figure 3 details the alternate execution of the `Bonus` and the `Alert` scenario. At time 0, a user logs in for 60 minutes provided by his flight ticket. The message `setTimeout(60)` sent from a `ValidityChecker` to its `Timer` is intercepted and forwarded to `Alert`. The `Alert` subtracts 5 minutes from 60 and proceeds the call. As a result the `setTimeout(55)` call is proceeded. At time 55, a `timeout` is intercepted which is a common intercepted message by both aspects. Here, the call is forwarded only to `Bonus` (first occurrence) that resets the timer for 10 minutes and skips the message. Thus, the first `timeout` call is ignored. At time 65, a second `timeout` is intercepted. The call, this time, is forwarded only to `Alert` (second occurrence) that warns the user, resets the timer for 5 minutes and skips the message. Thus, the `timeout` call is ignored again. At time 70, a third `timeout` call is intercepted and forwarded only to `Bonus` (third occurrence) that proceeds the call since the bonus time is consumed. As a result, the `timeout` is proceeded. This ends the session and elaborates the desired behavior.

The above example shows how aspect interact and confirms that the sequential composition of aspects, supported by the most popular AOP implementation models such as AspectJ [9], is not enough. One more example is the case that the airport manger decides to block P2P access when the network is overloaded. This also can be modeled as an aspect that wraps the `Firewall` and the `IpAddressManager`. The `NetOverloading` aspect requires the `IpAddressManager` for its `IIpMacDb` interface that defines `add(IP)` and `remove(IP)` services. Intercepting those services, the aspect will be able to count the number of connected users. In addition, the aspect requires the `Firewall` component for its `IConnection` interface that defines the `connect(IP)` service. Intercepting calls to this service enables the aspect to block calls to P2P addresses when the number of connected users exceeds the indicated threshold number. Now, if we want to prevent associating a bonus time to users when the system is overloaded, the `Bonus` and the `NetOverloading` aspects can be composed using the `Cond` operator. Within `Cond`, the `Bonus` behavior is not executed if the `NetOverloading` detects an overloading state of the system. In the following, we propose a formalization of component systems and aspects as a network of automata and we use Uppaal model checker to check whether the system satisfies the desired behavior and whether the bound aspects are interference-free. In addition, the composition strategies described above to solve aspect interferences are abstracted into binary composition operators that can be instantiated for arbitrary aspects. Uppaal model checker is used to check whether an instantiated operator solves the interference.

3. ADL for Aspectualized Component Systems

In this section we describe an ADL that enables the definition of both structural and behavioral properties of component systems. Our ADL is not a new language, it, rather, extends current ADL(s) with explicit definition of aspects, aspect behaviors, aspect weaving rules, and aspect composition. Table 3 shows the BNF-like grammar of our ADL. In the table, id refers to general identifiers, cId, $itfId$, and $svId$ refer to component, interface and service identifiers. In addition, $asId$, $pctId$, $opId$, and $locId$ refer to aspect, pointcut, operator, and location identifiers, respectively. Finally, we use t to refer to data types.

```
Architecture   ::=   system id ⟨Interfaces⟩ ⟨Components⟩ ⟨Attachments⟩
                              ⟨Aspects⟩ ⟨Weavings⟩
Interfaces     ::=   (interface id {(@(sync | async) svId)+ })+
Components     ::=   primitive | composite | ⟨Components⟩; ⟨Components⟩
Template       ::=   [attributes (t id;)+]
                     [provides (itfId id;)+]
                     [requires (itfId id;)+]
Primitive      ::=   primitive id [(n :≥ 2)] {
                        ⟨Template⟩ computation ⟨Behavior⟩
                     }
Composite      ::=   composite id [(n :≥ 2)] {
                        ⟨Template⟩ internals cId+
                     }
Attachments    ::=   (client=cId1.itfId1 server=cId2.itfId2)*
Behavior       ::=   process id {
                        [(t id)*;] [clock id*;]
                        state locId+;
                        init locId;
                        trans ⟨Transitions⟩;
                     }
Transitions    ::=   (locId − > locId {[⟨Guard⟩] [⟨Sync⟩] [⟨Assign⟩]})+
Guard          ::=   guard bexp
Sync           ::=   sync [(proceed | skip).][asId.]itfId.svId (! | ?)
Assign         ::=   assign exp
Weavings       ::=   (weave
                       asId (pctId, vexp)+
                      | opId ⟨Weavings⟩⟨Weavings⟩)*
Aspects        ::=   (aspect id ⟨Behavior⟩)*
Ops            ::=   (operator id ⟨Behavior⟩)*
```

Table 1. ADL description language for component systems

According to the above ADL specification, a component system is defined as a set of interfaces (*Interfaces*), components (*Components*), attachments (*Attachments*), aspects (*Aspects*), and a set of weaving rules (*Weavings*). Each interface is defined by an identifier, and a set of service signatures, each of which is annotated with (@**sync** or @**async**) to indicate whether the service is synchronous or asynchronous, respectively. We distinguish two kinds of components: primitives and composites. A primitive component is defined with an identifier, a set of attributes, two sets of provided and required interfaces and a behavior indicated with the **computation** keyword. Compared with a primitive, a composite does not have a behavior, instead a set of its internal components are indicated within the **internals** keyword. Since different instances may exist in the component system configuration, an indication of the number of instances for each component is optionally indicated with a natural number ($n :\geq 2$) that follows the component name. The set of attachments defines the configuration of the system by setting down all the connections between components. Inspired from Fractal [4], a connection binds a component required interface (*i.e.* **client**) to a component provided interface (*i.e* **server**). The weaving part of the ADL description, indicates which aspect should be bound to the system and how aspects are composed using binary composition operators. Both aspects and composition operators are defined with an identifier and an abstract behavior. By abstract behavior, aspect pointcuts are denoted with abstract names that should be replaced with concrete join points at the weaving stage (see section 4.6). To describe behaviors we adopt Uppaal XTA-like notation [10]. Accordingly, a behavior is indicated with a (**process**) keyword followed by potential declarations of local variables, clocks and a set of transitions. Each transition indicates the start and the end location, and a transition label. A transition is decorated with a guard, a synchronization channel, and a sequence of assignments. A guard is a predicate (*i.e.* boolean expression $bexp$), its satisfaction enables the transition. For channel labels, we adopt the following notations in the ADL specification: a channel label is a concatenation of the interface and the service identifiers. In addition, a channel label can be prefixed with two predefined keywords (**proceed** and **skip**) to indicate the actions taken by an aspect. Assignments are a sequence of clock and/or variable assignments, it can be also be a call to a C function. Finally, for the weaving, an aspect is associated with a mapping $(pctId, vexp)$ where: $pctId$ is an abstract pointcut identifier used in the aspect behavior specification, and $vexp$ is a VIL expression [3] used to define concrete join points that correspond to $pctId$.

```
 1 system airportInternetAccess
 2   interface ITimer {@sync setTimeout(int)}
 3   interface ITimerCallback {@async timeout()}
 4   interface IToken {@sync startToken()}
 5   interface ITokenCallback {@async tokenInvalidated()}
 6   // other interfaces
 7   primitive Timer (n) {
 8     provides ITimerCallback iTimerCallback;
 9     requires ITimer iTimer;
10     // behavior (see section 4.1)
11   }
12   primitive ValidityChecker (n) {
13     provides ITimer iTimer, IToken iToken;
14     requires ITimerCallback iTimerCallback,
15              ITokenCallback iTokenCallback;
16     // behavior
17   }
18   composite Token (n) {
19     provides ITokenCallback iTokenCallback;
20     requires IToken iToken;
21     internals Timer(1), ValidityChecker(1);
22   }
23   client ValidityChecker.iTimer server Timer.iTimer;
24   // other attachments
25   aspect Bonus //Behavior
26   weave Bonus (pct, {(ValidityChecker,iTimer,timeout)});
```

Listing 1. An excerpt of the ADL of the airport system example

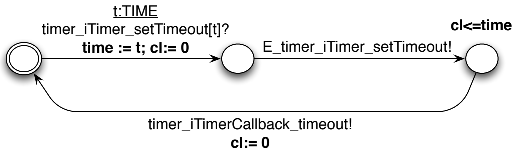
Listing 1 shows an excerpt of the ADL specification of the airport Internet access example. In the listing, four interfaces are declared with the signatures of their services (line 2-5). For example, the `ITimerCallback` interface defines only one asynchronous service named `timeout`. Two primitive components, the `Timer` and the `ValidityChecker`, are described (lines 7-11, 12-17, respectively) with their provided and required interfaces. The `Token` composite component is described (line 18-22) with its interfaces and internals. Since several tokens can be created, the component is parametrized with n (the maximum number of instances). In the internals part declaration, the `Timer` and the `ValidityChecker` are parametrized with the value 1 to indicate that only one instance of each is enabled in each `Token` component instance. The attachment description in (line 23) indicates that the `iTimer` interface of the `ValidityChecker` component is bound to the `iTimer` interface of the `Timer` component. The weaving declaration (line 26) indicates that the `Bonus` aspect (line 25) is bound to the system and its abstract pointcut `pct` should be replaced by the concrete join point: the `timeout`, service of the `iTimer` interface of the `ValidityChecker` component.

4. Aspectualized Component Systems in Uppaal

In this section we overview Uppaal model checker and we describe the transformation scheme of our ADL to Uppaal processes.

4.1 Overview of Uppaal

Uppaal [6] is a toolbox, used to design, simulate and check CTL (Computation Tree Logic) [11] properties for systems that can be modeled as networks of timed automata. A timed automata is a regular finite state machine extended with local variables, data types, and clock variables. Each automaton in Uppaal is called a *template*. A template can be parametrized with constants indicating how that template is instantiated (*e.g.,* how many instances are created). Each instance is called a *process*. Template nodes are called *locations* while the edges are called *transitions*. For describing systems, Uppaal provides both a graphical (XML format) and a textual (XTA format). For better understanding of Uppaal formalism, let us consider the behavior of the `Timer` component shown in Figure 4. The template can be read clockwise from the initial location. The initial location is distinguished with a double circle. The `Timer` waits for a `setTimeout` call with a parameter of type `TIME`[1] declared at the top of the channel label (`time:TIME`). When it receives such an event, it stores the time value in a local variable (`time:=t`), resets a clock variable `cl` to 0 (`cl:=0`) and goes to the next location. Then, the `Timer` sends `E_..._setTimeout` indicating the end of the treatment of the `setTimeout` event (`setTimeout` is synchronous event) and goes to the next location. This latter is decorated with an invariant (`cl<=time`) to indicate that the process should not stay at that location when the invariant becomes false (*i.e.* `cl>time`). When that happens, the `Timer` enables the last transition by triggering a `timeout` event, resets the clock again to 0 (`cl:=0`), and returns to the initial location.



t:TIME
timer_iTimer_setTimeout[t]?
time := t; cl:= 0
E_timer_iTimer_setTimeout!
cl<=time
timer_iTimerCallback_timeout!
cl:= 0

Figure 4. Formal Model for the `Timer` Component

Listing 2 shows the Uppaal textual description (XTA) of the `Timer` component. Data types (*e.g.,* `TIME`) are declared first (line 1). Each template in Uppaal is declared within the `process` keyword followed by the name of the template (`Timer` in this case) (line 2). Clock variables (`cl`) and local variables (`time`) are declared in the top of the declaration (line 3-4). Then the locations are listed (line 5), and the initial location is explicitly indicated (line 6). The transitions comes last following the same syntax we adopted for behaviors specification (line 7-12).

[1] `TIME` is a user defined data type

```
1 typedef int[0,6] TIME;
2 process Timer() {
3 clock cl;
4 TIME time;
5 state l0 {cl<=time}, l1, l2;
6 init l0;
7 trans
8     l0 -> l1 {select t:TIME;
9                 sync timer_iTimer_setTimeout[t]?;
10                assign time:=t, cl:=0;},
11    l1 -> l2 {sync E_timer_iTimer_setTimeout!;},
12    l2 -> l0 {sync timer_iTimerCallback_timeout!;};
13 }
```

Listing 2. The Uppaal-XTA description of the `Timer` component

Uppaal also provides a simulator that enables the exhaustive examination of systems behaviors. Within the simulator, a user can interact with the system, execute the system step by step, decide which transition should be taken if many are enabled, and see how local variables and clocks values change during the execution. The Uppaal model checker is designed to check reachability, safety and liveness properties expressed in a subset of CTL formulae [11]. When a particular property is violated, a counter example in terms of a diagnosis trace is automatically reported to the user. Thus, the user is given a feedback that helps on the detection of potential errors and how to correct them. The use of Uppaal in our proposal is important because of its support of the following features: template instantiation and value passing which are intrinsic properties to component models, and timing support which is important for our case study. In the following we describe how to transform the above ADL specification into Uppaal process.

4.2 Formalization of Primitive Components

Each primitive component is modeled as a Uppaal process. Since primitive components come with their behavior specification, those specifications should be transformed into correct Uppaal-XTA form. In our ADL, the behavior specification is chosen to be a subset of the XTA description of templates in Uppaal and hence minimum adaptations are needed. In particular, each channel label in the ADL behavior specification consists of the interface and the service names which are enough for the local behavior of the component. In our formalization we adopt the following notation for channel labels: each label is a concatenation of the component, the interface and the service identifiers separated with "_" symbol. This notation enables component bindings. Accordingly, all the channel labels in the specification should be prefixed by the component name. In addition, when more than one instance are required, the channel labels are suffixed with "`[id]`" indicating the instance reference of each component; where `id` is a constant that ranges over `[1..n]` and `n` is the indicated number of instances for the component in the specification. In Uppaal XTA, this number should be declared as a parameter of the template modeling the component. The following listing describes the general transformation rule that generates a Uppaal-XTA template for ADL primitive component specification. In the listing, the `clone()` function makes a copy of a behavior (first argument) of a software entity with a given new name (second argument).

```
1 𝒫 : Primitive → UppaalTemplate
2 𝒫⟦primitive cId temp computation cpt⟧ =
3             cpt[cId_itfId_svId / itfId.svId]
4 𝒫⟦primitive cId (n) temp computation cpt⟧ =
5             let p= clone(cpt,cId(const id:[1..n]))
6             in p[cId_itfId_svId[id] / itfId.svId]
```

Listing 3. primitive component transformation rule

4.3 Formalization of Composite Components

A composite is modeled as a set of Uppaal processes, one for each bound interface. Each template of those processes has a central initial location and a set of directed cycles from and to that location. Each cycle describes one service. Asynchronous services are represented by cycles of two transitions: receives a message ($cId_1\_itfId_1\_s_i?$), then forwards it ($E\_cId_1\_itfId_2\_s_i!$) (listing 4 line 17-18). Synchronous services are represented by cycles with four transitions: receives a message ($cId_1\_itfId_1\_s_i?$), forwards it ($cId_2\_itfId_2\_s_i!$), waits for the reply ($E\_cId_2\_itfId_2\_s_i?$), and forwards the reply ($E\_cId_1\_itfId_1\_s_i!$) (listing 4 line 12-15). When the composite may have multiple instances, similar to primitives, we suffix the channel labels of the component in question with "`[id]`" (listing 4 line 21-37). The following is the complete transformation rule of a composite component from the ADL specification of composites where: *attachments()* function returns the set of attachments declared in the architecture, *nbSync()* and *nbAsync()* return the number of synchronous and asynchronous services for a given interface in the architecture, and *synchronous()* function checks whether a given service is synchronous in a given interface.

```
 1 C : Composite → Architecture → UppaalTemplate*
 2 C⟦composite cId1 temp internals cIds⟧ a=
 3   ∀itfId1 ∈ interfaces(cId1),
 4   ∃(client= cId1.itfId1 server= cId2.itfId2) ∈ attachments(a):
 5       process cId1_itfId1() {
 6          state l0,..,lk;
 7          % k =nbSync(itfId1, a) * 3 + nbAsync(itfId1, a)
 8          init l0;
 9          trans
10          ∀ si ∈ services(itfId1, a) {
11          if (synchronous(si, itfId1, a)) {
12             l0  -> li1 {sync cId1_itfId1_si?;},
13             li1 -> li2 {sync cId2_itfId2_si!;},
14             li2 -> li3 {sync E_cId2_itfId2_si?;},
15             li3 -> l0  {sync E_cId1_itfId1_si!;};
16          } else {
17             l0  -> li1 {sync cId1_itfId1_si?;},
18             li1 -> l0 {sync cId2_itfId2_si!;};
19          }
20       };
21 C⟦composite cId1 (n) temp internals cIds⟧ a=
22   ∀(client= cId1.itfId1 server= cId2.itfId2) ∈ attachments(a)
23       process cId1_itfId1(const id : [1..n]) {
24          state l0,..,lk;
25          % k =nbSync(itfId1, a) * 3 + nbAsync(itfId1, a)
26          init l0;
27          trans
28          ∀ si ∈ services(itfId1) {
29          if (synchronous(si, itfId1)) {
30             l0  -> li1 {sync cId1[id]_itfId1_si?;},
31             li1 -> li2 {sync cId2_itfId2_si!;};
32             li2 -> li3 {sync E_cId2_itfId2_si?;};
33             li3 -> l0  {sync E_cId1[id]_itfId1_si!;};
34          } else {
35             l0  -> li1 {sync cId1[id]_itfId1_si?;},
36             li1 -> l0 {sync cId2_itfId2_si!;};
37          }
38       };
```

Listing 4. composite component generation rule

Let us consider the `Token` composite component. This component has two interfaces: `iToken` and `iTokenCallback` (see Listing 1 line 19-20). The interface `iToken` has one synchronous service: `startToken` (Listing 1 line 4) for starting a new session. Figure 5 depicts the generated Uppaal template for `iToken` interface of the `Token` component (one cycle of four transitions). While the `iTokenCallBack` interface has a single asynchronous service: `timeout` (Listing 1 line 3) for signaling that the session time elapsed. Figure 6 depicts the generated Uppaal template modeling the `iTokenCallback` of the `Token` component (a single cycle of two transitions).

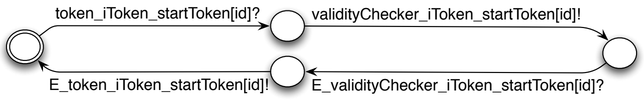

token_iToken_startToken[id]?
validityChecker_iToken_startToken[id]!
E_token_iToken_startToken[id]!
E_validityChecker_iToken_startToken[id]?

Figure 5. Formal Model for `Token`: `iToken` interface

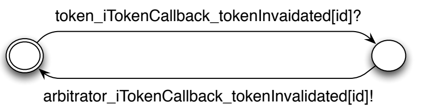

token_iTokenCallback_tokenInvaidated[id]?
arbitrator_iTokenCallback_tokenInvalidated[id]!

Figure 6. Formal Model for `Token`: `iTokenCallback` interface

4.4 Formalization of Bindings

Component bindings can be modeled either as separate Uppaal processes that receive channels from required interfaces and forward them to their bound provided interfaces, or by renaming. In our approach we adopt the second solution for minimum state number generation. By renaming, a bound interface $itfId_1$ of a component cId_1 to an interface $itfId_2$ of a component cId_2 is modeled by replacing each channel label occurrence $cId_1\_itfId_1\_s$ in the template of cId_1 by $cId_2\_itfId_2\_s$, for each service name `s`. This synchronizes the channels between the two bound components. Listing 5 describes the general binding rule. The *name()* function in the rule returns the name of a software entity (*i.e.* component, interface, aspect, behavior or Uppaal template).

```
B : UppaalTemplate → Architecture → UppaalTemplate
B⟦p⟧ a=
  let cId1= name(p)
  in ∀ (client= cId1.itfId1 server= cId2.itfId2) ∈
        attachments(a),
  ∀s ∈ services(itfId1) p[cId2_itfId2_s / cId1_itfId1_s];
```

Listing 5. binding function

Figure 7 depicts the `Timer` template after binding its required interface `iTimerCallback` to that provided by the `ValidityChecker` component (Listing 1 line 23). Thus, the `timer_iTimerCallback_timeout!` is replaced by `validityChecker_iTimerCallback_timeout!`.

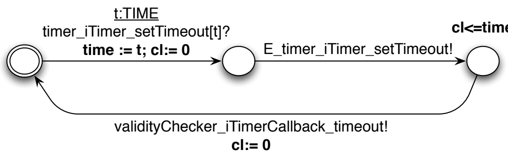

t:TIME
timer_iTimer_setTimeout[t]?
time := t; cl:= 0
E_timer_iTimer_setTimeout!
cl<=time
validityChecker_iTimerCallback_timeout!
cl:= 0

Figure 7. Formal Model for the `Timer` Component after binding

4.5 Formalization of component systems

A complete component system without aspects is modeled as the parallel composition of all the components of the architecture. The primitive components are adapted to follow the Uppaal-XTA syntax and bound to each other using the binding function, while composite templates are automatically generated from the ADL specification. Formally:

$$\mathcal{S}[\![a]\!] = \|_{\forall c \in primitives(a)} \mathcal{B}[\![\mathcal{P}[\![c]\!]]\!] a \ \|_{\forall t \in attachments(a)} \mathcal{C}[\![t]\!] a$$

The complete airport system example is modeled by 20 templates (9 for primitive components and 11 for composite components' interfaces).

4.6 Formalization of Aspects and Aspect weaving

The behavior of aspects is already described in the ADL specification following Uppaal-XTA form. The aspect behavior defines a set of cycles from and to the initial location, each of which describes the behavior of an aspect for an abstract pointcut `pctId`. The proceed and the skip actions taken by an aspect for each join point `e` are explicitly modeled by (`proceed_e`) and (`skip_e`), respectively. However, the behavior is abstract and should be instantiated for concrete join points. In our model, pointcuts are defined using VIL [3] in a declarative style. VIL interprets and transforms the pointcut expressions into tuples of the form *(cId, itfId, svId)* (*i.e.* a component, an interface and a service identifiers). In addition, a mapping *(pctId,e)* from an abstract pointcut to an expression describing the concrete join points is given in the ADL specification for each aspects. In the instantiation process, for each mapping *(pctId,e)*, we use VIL to interpret the expression *e* and returns a set of tuples of the form *(cId, itfId, svId)*. For each tuple, we make a copy of the cycle denoting *pctId* in the aspect abstract behavior. Then, we replace each *pctId* occurrence by `cId_itfId_svId` from the tuple. Listing 6 describes this instantiation process of Uppaal templates from the ADL specification of aspect behaviors. In the listing, the `duplicateTransitions(b,pctId)` function, as its name indicates, duplicates the set of transitions in the behavior specification *b* where *pctId* appears in the channel labels, and $\mathcal{V}$ denotes VIL interpretation of pointcut expressions.

```
ℐ : Behavior→(ID,VExp)⁺ →Architecture→UppaalTemplate
ℐ⟦b⟧ map a =
  process name(b) {
    // the declaration given in b
    trans
      ∀(pctId, e) ∈ map :
      let jps = 𝒱⟦e⟧(a)
      in ∀(cId, itfId, svId) ∈ jps :
      duplicateTransitions(b,pctId)[cId_itfId_svId / pctId]
  }
```

Listing 6. Instantiation rule

In order to synchronize the component processes with the aspect, a set of locations and transitions should be added to component specifications. This extension ensures that each intercepted service call is forwarded to the aspect process, which executes its behavior and returns either proceed or skip. In the former case, the extension ensures that the service call reaches its target and continues its original path. In the latter case, the extension ensures that the service call is skipped by returning to the initial location if the service is asynchronous. If the service is synchronous, all the actions between the begin and the end events of the service call are ignored. For instance, when a bonus is applied to the `ValidityChecker`, its corresponding template must be adapted as detailed in Figure 8. Part *(a)* shows an excerpt of the original `ValidityChecker` template and part *(b)* shows the same excerpt adapted. As a result, when `timeout` is received, it is forwarded to the bonus aspect process. The `ValidityChecker` waits for either `skip` to return to the original location, or `proceed` to forward the `timeout` to the `Token` component. Listing 7 generalizes the process of aspect weaving.

In Listing 7, the aspect process is instantiated first (line 3). Then for each concrete join point tuple, transitions and locations are added to the component process following the example shown in Figure 8. The `process(cId,a)` function (line 7) is used to get the Uppaal template defined for `cId` in a component architecture `a`. In the case of aspect composition, aspects are instantiated following the instantiation rule, then, the composition operator is instantiated.

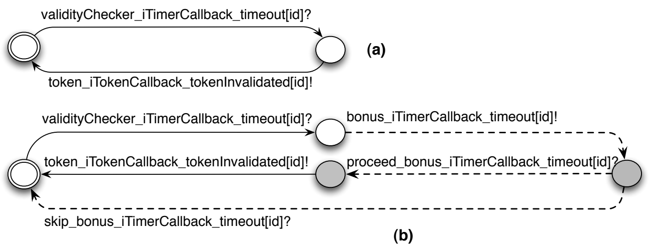

validityChecker_iTimerCallback_timeout[id]?
(a)
token_iTokenCallback_tokenInvalidated[id]!
validityChecker_iTimerCallback_timeout[id]?
bonus_iTimerCallback_timeout[id]!
token_iTokenCallback_tokenInvalidated[id]!
proceed_bonus_iTimerCallback_timeout[id]?
skip_bonus_iTimerCallback_timeout[id]?
(b)

Figure 8. `ValidityChecker` template adaptation (a) original, (b) adapted

Finally, the components owning the join points are adapted so that all the services in the concrete join points are forwarded to the operator process.

```
 1 𝒲 : Weavings → Architecture → UppaalTemplate*
 2 𝒲⟦weave asId (ptcId_i, e_i)⁺⟧ a=
 3   ℐ⟦behavior(asId)⟧ (ptcId_i, e_i)⁺ a;
 4   ∀e_i ∈ (ptcId_i, e_i)⁺ :
 5   let pts= 𝒱⟦e_i⟧(a)
 6   in ∀(cId, itfId, svId) ∈ pts:
 7       let p = process(cId,a)
 8       in
 9         if synchronous(itfId, svId)
10           p[(
11             l_i -> l_i1: {X_1 sync cId_itfId_sId? X_2},
12             l_i1-> l_i2: {sync asId_itfId_svId!},
13             l_i2-> l_j: {sync proceed_asId_itfId_svId?},
14             l_i2-> l_k: {sync skip_asId_itfId_svId?},
15             l_k-> l_l: {Y_1 sync E_cId_itfId_svId! Y_1},
16             )/
17             (
18             l_i -> l_j: {X_1 sync cId_itfId_svId? X_2},
19             l_k-> l_l: {Y_1 sync E_cId_itfId_s! Y_2},
20             )
21           ]
22         else
23           p[(
24             l_i -> l_i1: {X_1 sync cId_itfId_svId? X_2},
25             l_i1-> l_i2: {sync asId_itfId_svId!},
26             l_i2-> l_j: {sync proceed_asId_itfId_svId?},
27             l_i2-> l_0: {sync skip_asId_itfId_svId?},
28             )/
29             (
30             l_i -> l_j: {X_1 sync cId_itfId_svId? X_2},
31             )
32           ]
```

Listing 7. weaving an aspect to a component

As an example of the instantiation of composition operators, Figure 9 depicts the template for the alternate operator `Alt`. In the figure, when a common intercepted service call is intercepted (`e12?`), the function `swap()` is executed. This latter maintains a boolean `isLeft` to indicate which aspect should be applied. If it is the left-hand side aspect turn (`isLeft`), `e12` is forwarded to this aspect (`a1_e12!`), otherwise, `e12` is forwarded to the right-hand side aspect (`a2_e12!`). In both cases, the operator waits for either `proceed_ai_e12` or `skip_ai_e12` and forwards the intercepted command to the caller (`proceed_e12` or `skip_e12`). While the individual intercepted service calls `e1` and `e2` are only forwarded to their corresponding aspects only. The above `Alt` template is instantiated by generating cycles like the one defined for `e12` for each intercepted call by both aspects, while a cycle of that defined for `e1` is generated for each service intercepted by the first aspect only. Similarly, a cycle of that defined for `e2` is generated for each service intercepted by the second aspect only.

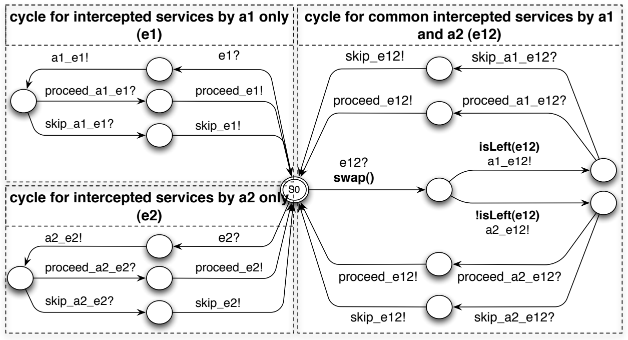


Figure 9. The `Alt` template

5. Interference Detection

For interference detection, we define a component system Γ as a pair $(\mathcal{A}_\Gamma, \mathcal{P}_\Gamma)$ where, $\mathcal{A}_\Gamma$ is the process modeling the component system generated from the ADL description, and $\mathcal{P}_\Gamma$ is a set of CTL formulas describing the behavior of the system.

For instance, our case study is defined as $(\mathcal{A}_{airport}, \mathcal{P}_{airport})$. Where $\mathcal{P}_{airport}$ describes the set of properties the system ensures whenever it is executed. The system is designed to satisfy different (liveness, safety, and reachability) properties. These are given at the top of Table 2 ($\mathcal{P}_{airport}$). In particular, a user can not stay connected forever (*Live 1*), the system is deadlock free (*Safe 1*), a user cannot stay connected more than the validity time indicated in his flight ticket (*Safe 2*), a user can connect to all the Ip addresses when its access is enabled by the firewall (*Safe 3*), and several users can be connected at the same time (*Reach 1*). The formulas rely on different constants, variables and auxiliary functions: `IDS` and `IPS` denote the range for user identifiers and Ip addresses, respectively, `Connected` and `Disconnected` are identifiers denoting particular locations of the user process, the `validity(id)` is a global function that returns the authorized connection time of a user `id`, `currentIp(id)` returns the current Ip address the user want to connect, and `enabled(id)` checks whether a user id access is enabled by the firewall. The `cl` is a local clock associated to the user process, and the `isConnected` is a local variable in the user process that stores the firewall response of the user access to each Ip address.

The properties satisfaction indicates the well-definedness of component systems.

DEFINITION 5.1. *A component system $\Gamma = (\mathcal{A}_\Gamma, \mathcal{P}_\Gamma)$ is well defined if its modeling process satisfies all its desired properties:*

$$\mathcal{D}(\Gamma) \overset{\text{def}}{=} \mathcal{A}_\Gamma \models \mathcal{P}_\Gamma$$

The intent of each aspect should also be given as a set of CTL formulas. The intent describes the set of properties the aspect ensures when it is applied to a system. The satisfaction of these properties when the aspect is bound determines the applicability of the aspect to the base system.

DEFINITION 5.2. *Given a component system $\Gamma = (\mathcal{A}_\Gamma, \mathcal{P}_\Gamma)$ and an aspect $\Lambda = (\mathcal{A}_\Lambda, \mathcal{P}_\Lambda)$ where, $\mathcal{A}_\Lambda$ is the process modeling the aspect behavior, and $\mathcal{P}_\Lambda$ is the set of its intent properties. An aspect is said to be correct with respect to a component system Γ if the following condition holds:*

$$[\![\mathcal{A}_\Gamma]\!]_{\mathcal{W}_\Lambda} \parallel \mathcal{A}_\Lambda \models \mathcal{P}'_\Gamma \wedge \mathcal{P}_\Lambda$$

Where $[\![\mathcal{A}_\Gamma]\!]_{\mathcal{W}_\Lambda}$ denotes the process modeling the system Γ after weaving the aspect Λ, $\parallel$ denotes the parallel composition of processes and $\mathcal{P}'_\Gamma \subseteq \mathcal{P}_\Gamma$ is a subset of the system properties that must be preserved after the weaving of the aspect Λ. These properties should be defined for each aspect to be woven. For our case study, when the bonus is applied to the airport system, the set of intrinsic properties of the system is defined as all the $\mathcal{P}_{airport}$ properties except (Safe 2), since bonus allows the user to connect for more than the time indicated in his flight ticket (see Safe 2'). While in the case of alert $\mathcal{P}'_\Gamma$ is simply $\mathcal{P}_{airport}$. The properties for the bonus ($\mathcal{P}_{bonus}$), the alert ($\mathcal{P}_{alert}$), and the netOverloading ($\mathcal{P}_{netOverloading}$) aspects are shown in the bottom part of Table 2. The bonus aspect ensures that the user can stay connected a bonus time (`BonusTime`) after its authorized time expires (Safe 3'). The alert ensures that the user is always alerted before it is disconnected (Live 2) and the alert is intercepted exactly before a `TimeAlert` of its expiration time (Safe 4). While the netOverloading aspect ensures that a user is unable to access P2P addresses when the system is overloaded (Safe 2'). In the formulas, `Alerted` is an identifier denoting a particular location in the user process, `isAlerted` is a local boolean variable of the user indicating whether a user reached the `Alerted` location, `BonusTime` and `AlertTime` are constants denoting the bonus and the alert time, respectively. Finally, `isP2P(ip)` and `isOverload()` are two predicates defined in the netOverloading aspect to check whether an Ip address is P2P and whether the server is overloaded, respectively.

Extending components with several aspects may give rise to interferences. Two aspects are interference-free with respect to a base program, when both aspects are bound to the system, the result process satisfies all the properties of the underlying aspects as well as the system properties to be preserved by both aspects. Formally:

DEFINITION 5.3. *Given a base system $\Gamma = (\mathcal{A}_\Gamma, \mathcal{P}_\Gamma)$ and two aspects $\Lambda_1 = (\mathcal{A}_{\Lambda_1}, \mathcal{P}_{\Lambda_1})$, $\Lambda_2 = (\mathcal{A}_{\Lambda_2}, \mathcal{P}_{\Lambda_2})$, Λ_1 and Λ_2 are interference-free if the following conditions hold:*

1. $\mathcal{D}(\Gamma)$ *: the base system is well defined*
2. **let** $\alpha = [\![\mathcal{A}_\Gamma]\!]_{\mathcal{W}_{\Lambda_1}}$
   in let $\beta = [\![\alpha]\!]_{\mathcal{W}_{\Lambda_2}}$
   in $\beta \parallel \mathcal{A}_{\Lambda_1} \parallel \mathcal{A}_{\Lambda_2} \models \mathcal{P}_\Gamma^{\Lambda_1,\Lambda_2} \wedge \mathcal{P}_{\Lambda_1} \wedge \mathcal{P}_{\Lambda_2}$ *: the composition is correct w.r.t* Γ

Where $\mathcal{P}_\Gamma^{\Lambda_1,\Lambda_2}$ denotes the set of system properties to be preserved by both Λ_1 and Λ_2.

In our case study, when both bonus and alert are woven to the system and composed using `Seq`, Safe 3 property is violated which reports an interference with a diagnostic trace. The reported trace shows that both aspects reset the `Timer` for two different values and the `Timer` is not designed to accept such kind of behavior. In addition, the alert message is send to the `User` before the addition of bonus. Thus, we used the `Alt` operator that sets the `Timer` once for each intercepted `timeout` event, and it ensures that the alert is send to the `User` after consuming the bonus. The use of Uppaal model checker this time shows that all the desired properties are satisfied which indicates that the interference is solved. In general, a composition operator solves an interference if when it is instantiated for two aspects and composed to the system, the interference disappears. Formally:

DEFINITION 5.4. *Given a base system $\Gamma = (\mathcal{A}_\Gamma, \mathcal{P}_\Gamma)$ and two interfering aspects $\Lambda_1 = (\mathcal{A}_{\Lambda_1}, \mathcal{P}_{\Lambda_1})$, $\Lambda_2 = (\mathcal{A}_{\Lambda_2}, \mathcal{P}_{\Lambda_2})$, a composition operator Θ solves an interferences if the following condition hold:*

$$\mathcal{A}'_\Gamma \parallel \Theta(\Lambda_1, \Lambda_2) \models \mathcal{P}_\Gamma^{\Lambda_1,\Lambda_2} \wedge \mathcal{P}_{\Lambda_1} \wedge \mathcal{P}_{\Lambda_2}$$

where $\mathcal{A}'_\Gamma$ is the process modeling the system after weaving Λ_1 and Λ_2, and $\Theta(\Lambda_1, \Lambda_2)$ denotes the parallel composition of the

Properties for the Airport Base System ($\mathcal{P}_{airport}$)	
Live 1	`User(id).Connected --> User(id).Disconnected`
Safe 1	`A[] not deadlock`
Safe 2	`A[] ∀(id:IDS),∀(ip:IPS) (User(id).Connected ∧ currentIp(id)==ip ∧ Firewall.enabled(id)) ⇒ User(id).isConnected`
Safe 3	`A[] ∀(id:IDS) User(id).Connected ⇒ User(id).cl<=validity(id)`
Reach 1	`E<> User(0).Connected ∧ (∀(id:IDS) id!=0 ⇒ User(id).Connected)`
Properties for the Bonus aspect ($\mathcal{P}_{bonus}$)	
Safe 3'	`A[] ∀(id:IDS) User(id).Connected ⇒ User(id).cl<=validity(id) + BonusTime`
Properties for the Alert aspect ($\mathcal{P}_{alert}$)	
Live 2	`Use(id).Connected --> User(id).Disconnected ∧ User(id).isAlerted`
Safe 4	`A[] ∀(id:IDS) User(id).Alerted ⇒ User.cl== validity(id) - AlertTime`
Properties for the NetOverloading aspect ($\mathcal{P}_{netOveloading}$)	
Safe 2'	`A[] ∀(id:IDS),∀(ip:IPS) (User(id).Connected ∧ currentIp(id)==ip ∧ NetOverloading.isP2P(ip) ∧ NetOverloading.isOverload()) ⇒ !User(id).isConnected`

Table 2. Properties of the airport system

processes of Λ_1 and Λ_2 and the instantiated template Θ for Λ_1 and Λ_2 as described in Section 4.6.

We should mention here that our example is a large case study. It is instantiated with three users for the base system and two users for the extended version. The instantiation of the system with more users leads to state explosion in UPPAAL. However, in our example, merely one user is sufficient to detect the interference of `Bonus` and `Alert`, and two users are sufficient to detect the interference between the `Bonus` and the `NetOverloading` aspects.

6. Catalog of operators for aspect composition

In this section we provide seven abstract composition operators modeled as Uppaal templates. Those templates can be instantiated for aspects in order to solve their potential interferences. Each presented operator is given with a short explanation of its applicability and a discussion of a motivation example to help users to select the right composition strategy needed for their cases. This extendible set of operators forms a first step towards a catalog of patterns for aspect interference resolution. In this paper we describe seven operators, three of them are used for our case study: `Seq`, `Alt` and `Cond`. While `Seq` is used as a default composition strategy of aspects, `Alt` solved the interference between the `Bonus` and the `Alert`, and `Cond` solved the interference between the `Bonus` and the `NetOverloading` aspects.

6.1 Sequential composition operator

The use of the `Seq` operator is restricted to aspects sharing at least one join point. The `Seq` ensures a precedence between aspects, it is common to several aspect-oriented languages such as AspectJ [9]. In our proposal we adopt the use of the `Seq` operator as a default composition strategy for aspects sharing join points. The `Seq` operator forwards the intercepted calls common to both aspects to the first aspect, it waits for its decision to send it to the second aspect. The call in this case is proceeded when at least one of the aspects decides to proceed it, otherwise, the service call is skipped. In addition, the intercepted calls by either aspects are forwarded to their corresponding aspects only. Figure 10 depicts the template modeling the `Seq` operator. In the figure, the intercepted call by both aspects `e12?` is forwarded to the first aspect (`a1_e12?`), the decision of this aspect is saved in a local variable `fstAct`, then the call is forwarded to the second aspect (`a2_e12?`). The call is proceeded when it is proceeded by the second aspect (`proceed_a2_e12?`) or it is skipped by it and proceeded by the first aspect. Two other cycles describes forwarding non common service calls to their corresponding aspects (cycles for `e1?` and `e2?`).

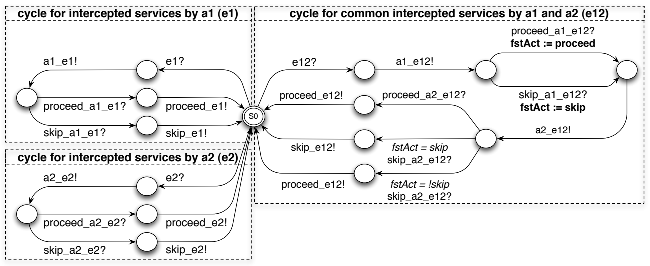


Figure 10. The `Seq` template

6.2 Alternate composition operator

The `Alt` is used for two aspects that need to be executed alternately on shared join points as shown between the `Bonus` and the `Alert` aspects. Thus for each shared join point, only one aspect behavior is executed.

6.3 Conditional composition operator

The `Cond` operator is used for conditional dependency between aspects. That is the case where one aspect relies on state variables or any other effects generated from the execution of another aspect. In that case, both should be applied but the second is executed only when a condition holds after the execution of the first aspect. One additional example of that previously presented, is the case for the `LegalAccess` aspect that blocks the access to illegal IP addresses for minors. Since this addresses can be P2P, the manager may decide to apply the `LegalAccess` aspect behavior for non first class customers. In that case, the `Cond` operator can be used for the `LegalAccess` and the `NetOverloading`.

The `Cond` operator (see Figure 11) forwards each intercepted call to the first aspect and maintains a predicate when the action of the first aspect is received. According to the predicate, the call is forwarded to the second aspect or the action of the first aspect is directly taken. The above structure describes the case of two aspects sharing join points. Another variant of this operator is designed to consider two aspects with no shared join points, it is shown in Figure 12. In that variant, the predicate is maintained and stored when the action of the first aspect is taken on one service call. The predicate is evaluated later when another service call for the second aspect is intercepted.

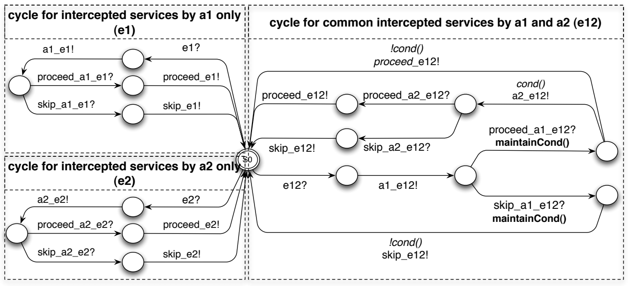


Figure 11. The `Cond` template

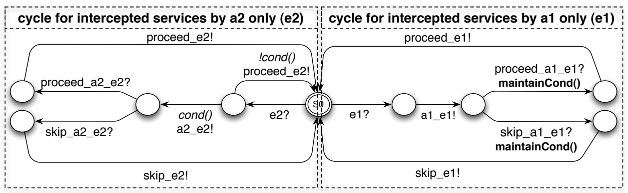


Figure 12. The `Cond` template (variant)

6.4 Fst composition operator

The `Fst` operator can be used when two aspects are mutually exclusive. This appears when two aspects implement two contradictory behaviors or two different algorithms for the same problem. In addition, the `Fst` can be used for inclusion relation between aspects, this appears when all the properties of one aspect are satisfied when another aspect is applied. Take for example the case of another aspect that blocks P2P access for non first class customers when the system is overloaded. This property is satisfied by the `NetOverloading` aspect version presented in this paper. In that case, there is no reason to apply this new aspect when the former is applicable. One more applicability case of the `Fst` operator is to hide some actions of aspects on specified join points.

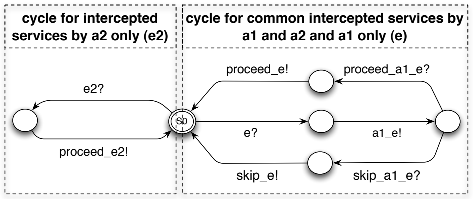

cycle for intercepted services by a2 only (e2)
cycle for common intercepted services by a1 and a2 and a1 only (e)
e2?
proceed_e2!
S0
proceed_e!
proceed_a1_e?
e?
a1_e!
skip_e!
skip_a1_e?

Figure 13. The `Fst` template

The `Fst` operator forwards the intercepted calls common to both aspects and those only intercepted by the first aspect to the first aspect only, and all the calls intercepted by the second aspect are directly proceeded. Thus, only the first aspect is executed and the second aspect is never called. Figure 13 depicts the template modeling the `Fst` operator. In the figure, the intercepted call `e?` represents an intercepted service call by either the first aspect only or by both aspects. These calls are send to the first aspect only `a1_e!` and the operator waits for the decision of the aspect and forwards it. The intercepted call `e2?` which is intercepted by the second aspect only is directly proceeded (`proceed_e2!`).

6.5 And composition operator

The `And` is used for the case where two aspects complement each other and should only be executed in a specific order otherwise a conflict appears. The `And` operator is used to proceed calls only when it is proceeded by both aspects and skips it when one of the aspects skips it. Consider for example the case of a `LimitAccess` aspect that counts the number of users being accessing to specific IP addresses and blocks the access to those addresses of the number exceeds a threshold number. Since the `NetOverloading` aspects block the access to P2P addresses when the system is overloaded, the `LimitedAccess` should not count the skipped requests by the `NetOverloading` aspect, otherwise the result number becomes erroneous.

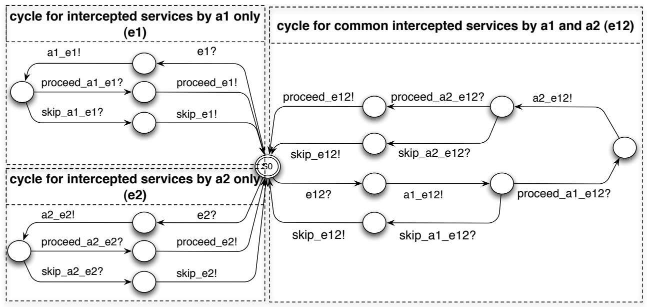


Figure 14. The `And` template

6.6 Cflow composition operator

The `Cflow` is used for strongly dependent aspects. This appears when the execution of one aspect is based on a set of previously intercepted calls and the action taken by another aspect for those calls. Take for example, a `Security` aspect that provides an additional security code for logging and skips the intercepted login request if the entered security code is not correct. Also consider a `CheckLogin` aspect that automatically disconnects the user from the server for three successive rejected login requests. These latter aspect, requires information about the history of the login requests for each user. Thus the execution of the `CheckLogin` aspect is based on the actions taken by the `Security` aspect. The template of this operator is not shown here for space limitation.

7. Related Work

Interference detection and resolution is still a challenge for both features [12] and aspects [13]. However, few works are dedicated to analyze aspect interferences and composition in component models. In particular, JAsCo [14] is an aspectualized component model and it provides an API to compose aspects in a programmatic way. But no interference detection support is provided. LEDA is a component framework and AspectLEDA [15] is its extension with aspects. Aspects in LEDA are represented by regular components and aspect execution is ordered following a predefined priority order.

Current works on features are focussing on domain specific interferences. For example, Gouya et al. [16] propose an algorithm for feature interactions in IP multimedia subsystem (IMS). The algorithm uses a predefined interference rules based upon traces on service calls. Some of these interferences with their solutions are defined in a database, if the interference is not in the database, it is reported to the user.

Several works are dedicated to aspect interference analysis in AOP. For example, Goldman et al. [17] model the base program, the aspects, and the woven system with state machines in order to formally check properties. Their weaving process is implemented by inlining the aspect state machine directly in the base system. Moreover, they focus on LTL and use two kinds of properties. First, they check if the base system satisfies aspect assumptions that enable their weaving. Second, they check if the woven system guarantees the expected behavior of aspects. The limitations of this approach are: they weave an aspect at a time and they only consider weakly invasive aspects. Moreover, when an interference is detected (*i.e.* a property is not satisfied) the programmer is responsible to fix it: they do not provide composition operators. Krishnamurthi et al. [18] also use state machines to model both aspects and base systems. However, the proposed approach defines a state machine for each advice. Moreover, the work is limited to treat aspects that do not modify data variables of base systems. Temporal logic has previously been used by Katz et al. [19] to describe the expected behavior of aspects. In this work, a semi-automatic interactive process is proposed to define the assume-guarantee properties of aspects in LTL formulas. Aspect interferences are checked independently of any base system by checking their guaranties properties. At the weaving stage, another check should be performed to show if the base system satisfies the assumptions of all the aspects to be woven. In [20] advices are annotated with assumptions about their composition. Interferences are detected by matching the assumptions of an advice and all the other advices. Finally, we should mention that our current proposal is a byproduct of our previous work on aspect interference detection and resolution [21] and formalization of aspects in a concurrent context [22]. The first work focuses on interferences at shared join points and introduces composition operators. The second models the woven system as FSP processes and checks properties with LTSA.

8. Conclusion

In this article, we have shown how to formally analyze aspect interferences in the context of component-based systems. First, a system is specified in a architecture description language (ADL) where primitive component behaviors and aspects are specified in UPPAAL. Second, we have detailed a transformation scheme from the ADL to UPPAAL generating a formal model of the complete system that can be model checked. In particular this makes it possible to check whether the base system properties are violated by aspects and whether the desired properties of aspects are violated by aspect interferences. Third, our aspect advices explicitly return either proceed or skip. This enables us to define composition operators for aspects. We have proposed several operators and discussed how our approach makes it easy to define new ones. We have also formally model these operators. This makes it possible to check whether a composition of aspects solves the interferences among them. We have illustrated our approach with different extension scenarios for a wireless Internet system for airports. Note that our approach is not component model dependent. We have already shown how it can be applied to Fractal [5] and we plan to apply it to other models. This may require few adaptations (*e.g.,* Sofa [23] will require to extend our transformations in order to take into account component connectors for enabling several communication styles).

References

[1] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 2nd ed., ser. Component Software Series, C. Szyperski, Ed. ACM Press and Addison-Wesley, 2002.

[2] G. Kiczales and E. Hilsdale, "Aspect-oriented programming," in *Proceedings of the 8th European software engineering conference*. ACM, 2001, p. 313.

[3] A. Hannousse, G. Ardourel, and R. Douence, "Views for aspectualizing component models," in *Proceedings of the Ninth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2010, pp. 21–25.

[4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Software-Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[5] A. Hannousse, R. Douence, and G. Ardourel, "Composable controllers in fractal: Implementation and interference analysis," in *proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE Computer Society, to appear, 2011.

[6] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *SFM-RT*, ser. LNCS, no. 3185. Springer-Verlag, 2004, pp. 200–236.

[7] O. Šery and F. Plášil, "Slicing of component behavior specification with respect to their composition," in *CBSE*, ser. LNCS, vol. 4608. Springer, 2007.

[8] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil, "Component reliability extensions for fractal component model." http://websvn.ow2.org/.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP '01. London, UK, UK: Springer-Verlag, 2001, pp. 327–353.

[10] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, pp. 134–152, 1997.

[11] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," *Information and Computation*, vol. 111, pp. 394 –406, 1992.

[12] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Computer Networks*, vol. 41, no. 1, pp. 115 – 141, 2003.

[13] F. Sanen, E. Truyen, and W. Joosen, "Classifying and documenting aspect interactions," in *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2006, pp. 23–26.

[14] D. Suvée, W. Vanderperren, and V. Jonckers, "Jasco: an aspect-oriented approach tailored for component based software development," in *AOSD*. ACM, 2003, pp. 21–29.

[15] A. Navasa, M. A. Pérez-Toledano, and J. M. Murillo, "An adl dealing with aspects at software architecture stage," *Information and Software Technology*, vol. 51, no. 2, pp. 306–324, 2009.

[16] A. Gouya and N. Crespi, "Detection and resolution of feature interactions in ip multimedia subsystem," *Int. J. Netw. Manag.*, vol. 19, pp. 315–337, 2009.

[17] M. Goldman, E. Katz, and S. Katz, "Maven: modular aspect verification and interference analysis," *Formal Methods in System Design*, vol. 37, pp. 61–92, 2010.

[18] S. Krishnamurthi and K. Fisler, "Foundations of incremental aspect model-checking," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 2, pp. 1–39, 2007.

[19] E. Katz and S. Katz, "Incremental analysis of interference among aspects," in *FOAL*. ACM, 2008, pp. 29–38.

[20] A. Marot and R. Wuyts, "Detecting unanticipated aspect interferences at runtime with compositional intentions," in *RAM-SE*. ACM, 2009, pp. 31–35.

[21] R. Douence, P. Fradet, and M. Südhot, "Composition, reuse and interaction analysis of stateful aspects," in *AOSD*. ACM, 2004, pp. 141–150.

[22] R. Douence, D. L. Botlan, J. Noyé, and M. Südhot, "Concurent aspects," in *GPCE*. ACM, 2006, pp. 79–88.

[23] T. Bures, P. Hnetynka, and F. Plasil, "Sofa 2.0: Balancing advanced features in a hierarchical component model," in *proceedings of the 4th International Conference on Software Engineering Research, Management and Applications*. IEEE Computer Society, 2006, pp. 40–48.