



Composable Controllers in Fractal: Implementation and Interference Analysis

Abdelhakim Hannousse, Rémi Douence, Gilles Ardourel

► To cite this version:

Abdelhakim Hannousse, Rémi Douence, Gilles Ardourel. Composable Controllers in Fractal: Implementation and Interference Analysis. the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'11), Aug 2011, Oulu, Finland. pp.99. hal-00606269v1

HAL Id: hal-00606269

<https://hal.science/hal-00606269v1>

Submitted on 5 Jul 2011 (v1), last revised 6 Jan 2012 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

¹In this article, we present a simplified version of the system (only one kind of customer, `FlyTicketManager` is a primitive component...). A more complete Fractal implementation and Uppaal formalization of the system are available by request to the authors.

SessionManager. Once connected, it sends queries to the AccessManager. The AccessManager 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 superimposed boxes. The DhcpServer delegates IP address requests to the IpAddressManager that provides dynamic IP address allocation. The allocated IP addresses are managed by the IpDbConnection component. The SessionManager manages sessions. When it receives a login request, its inner Arbitrator component retrieves the authorized access time from the FlyTicketManager, then the SessionManager instantiates a ValidityChecker. The Arbitrator in turn orders the AccessManager to enable communications for the user. When the authorized time elapses the ValidityChecker asks the Arbitrator to close the session. The Arbitrator in turn orders the AccessManager to disable communications for the user, and the DhcpServer to disable its IP address.

III. COMPONENT EXTENSION IN FRACTAL

Fractal [1] is a hierarchical component model developed by the OW2 consortium. Fractal supports a set of interesting features such as: component sharing and component behaviors extension through component controllers. Each component in Fractal has a content and a membrane. The content encapsulates the business logic, while the membrane exposes the provided and required interfaces and a set of controllers. Components are connected to each other via bindings. A binding connects a required interface of a component to a provided interface on the assumption that the provided interface type is a sub-type of the required interface type. Fractal provides an architectural description language called Fractal-ADL to describe the architecture of systems in a declarative style.

In Fractal, component behaviors are extended by means of controllers. Extensions can either be installed on: (1) a single component by adding a controller to a primitive component, (2) several components by adding a controller to a composite that encapsulates all the components to be extended. Component sharing enables the application of controllers to components belonging to different composites. A regular Fractal controller intercepts messages sent and received by a component and it can alter them, redirect or even discard them by calling or not a method `invoke()`. However, without an explicit discard call event, the composition of controllers has to be made in a programmatic and non-modular way by calling one controller from another. Here, we propose to extend Fractal with controllers that make discarding explicit. We call such controllers *composable controllers* and we provide a set of generic operators to compose them.

A. Composable Controller in Fractal

We define a composable controller as a pair (Dispatcher, Act). The Dispatcher is a regular Fractal controller used to intercept service calls, reify them

```
1 enum Cmd {Proceed, Skip}
2 interface IController {
3   Cmd match(MessageContext c);
4 }
```

Listing 1. IController interface

into message objects and pass them to the Act object. An Act object is a regular object implementing the IController interface (see Listing 1). It implements the behavior of the extension in a `match(MessageContext)` method. The `match` method inspects or modifies messages, executes extra codes and finally decides whether the message should actually be proceeded by returning the command `Proceed` or discarded by returning `Skip`. The Dispatcher controller calls its method `invoke()` to proceed the service call only if the Act object returns a `Proceed` command.

Like regular controllers, composable controllers can be plugged into components that have to be altered by the controller behavior. When the controller needs to be applied to several components that are scattered over the architecture, the system is reconfigured as follows: a new composite is created and the required components are added to that composite as shared components. This way, the controller is plugged into the new composite and the original configuration is preserved.

In the following, we show how composable controllers can be used to extend the airport service example with (1) adding a bonus time to customers and (2) alert customers five minutes before the end of their sessions.

B. Extension 1: Add a Bonus

Let us suppose that the airport decides to offer a bonus time to first class customers. Such an extension can be done by adding a composable controller with an IController, named Bonus, to the ValidityChecker.

The Dispatcher controller intercepts service calls to the ITimerCallback interface that defines a timeout service. This service is called by the Timer to inform the ValidityChecker of the end of the session. The Dispatcher intercepts the call, reifies it into a message object and sends it to Bonus by calling its `match` method. The `match` method of Bonus behaves as follows: when it receives the first occurrence of timeout (*i.e.* the customer session should be closed), it checks if the customer has a first class ticket. If it is the case, it resets the timer for 10 more minutes (by calling the `setTimeout(10)` service on the ITimer interface of the corresponding Timer component), and informs the Dispatcher controller that it wants to skip the call by returning a `Skip` command. That means that the timeout, in this case, is not actually proceeded and the session continues. If it receives a second occurrence of timeout, the `match` method returns a `Proceed` command to Dispatcher. This causes the Dispatcher to call its `invoke()` method which proceeds the call and ends the current session of the user.

In our scenario, the composable controller with the Bonus

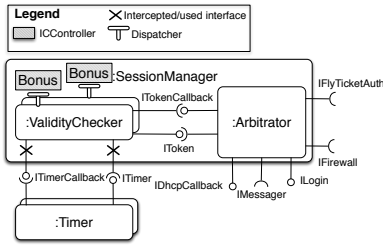


Fig. 2. The airport system extended with Bonus

ICController is attached to each `ValidityChecker` instance. This enables the interception of required services (*i.e.* timeout of the `ITimerCallback` interface) and the use of its provided services (*i.e.* `setTimeout` of the `ITimer` interface). Figure 2 shows the architecture of the airport system after the integration of Bonus.

C. Extension 2: End Session Alert

Now, suppose that the airport decides to add a service that alerts users five minutes before their authorized time connection expires. In this case, a session timer is initialized with five minutes less than the authorized time. When the time expires and a timeout is generated, an alert is sent to the user and the timer is reset for five more minutes. Here, we propose to integrate a composable controller with an ICController `Alert` that behaves as follows: when the `setTimeout` service call is intercepted on the `ITimer` interface of the `ValidityChecker` component, the `Alert` changes the parameter value of the call (*e.g.*, 60 minutes) by subtracting a time alert (5 minutes), and proceeds the call. Thus, the `ValidityChecker` will receive a timeout 5 minutes before the end of the session. When the timeout is intercepted, the `Alert` sends an alert message to the user (by calling the `show("you have only 5 min left")` on the `IMessenger` interface required by the `Arbitrator`), resets the `Timer` for 5 minutes by calling `setTimeout(5)`², and skips the currently intercepted timeout. The `Alert` proceeds the next intercepted timeout to end the session. Note that the controller, in this case, is plugged into the `SessionManager` composite that encapsulates several instances of `ValidityChecker(s)`. Thus, the `Alert` needs to store the identity of the `Timer` triggering the first timeout call, so that it can proceed the second triggered timeout and ends the right user session. Figure 3 shows the airport system after being extended with `Alert`.

D. Controllers Interferences and Compositions

Extending components with several controllers may give rise to interferences. For better understanding of controller interferences, let us consider the original airport system, its bonus and alert extensions. 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.

²A controller intercepts only services of the component it controls (*e.g.*, alert does not intercept its own call to `setTimeout(5)`).

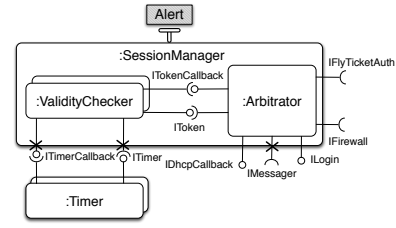


Fig. 3. The airport system extended with Alert

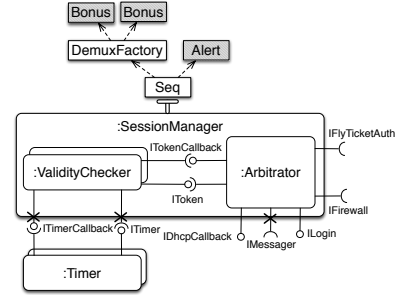


Fig. 4. The default composition of Bonus and Alert with Seq

When both controllers are plugged into 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 our approach, when two controllers intercept common services, the default composition is made using a `Seq` operator. When the `Seq` operator is used and a service call is intercepted, the `match` method of both controllers are called sequentially and the service call is proceeded only if at least one of its underlying ICControllers returns `Proceed`, otherwise, `Seq` returns `Skip` to the `Dispatcher`. However, in Section III-B, we plugged an instance of `Bonus` into each instance of `ValidityChecker`, while in Section III-C we introduced a single instance of `Alert` for all the instances of `ValidityChecker` and we plugged it into the `SessionManager` component. In our approach, two controllers can be composed only if they are plugged into the same component which is not the case for `Bonus` and `Alert`. To tackle such a problem, we define an operator `DemuxFactory` that is both a factory of ICController(s) and a demultiplexer responsible for “routing” the intercepted messages to the right instances. In our example, the `DemuxFactory` is responsible for the instantiation of `Bonus` for each `ValidityChecker` instance, and when a timeout message is intercepted for a `Timer` instance, the `DemuxFactory` redirects the call to the correspondent `Bonus` instance (*i.e.* the one plugged into the `ValidityChecker` bound to the `Timer` making the call). Figure 4 shows the composition of the above controllers where the white boxes refer to composition operators and gray boxes refer to ICController(s). Note that the composition operators in our proposal also implement the ICController interface. This enables our controllers to be composed in a composite pattern way with a single `Dispatcher`.

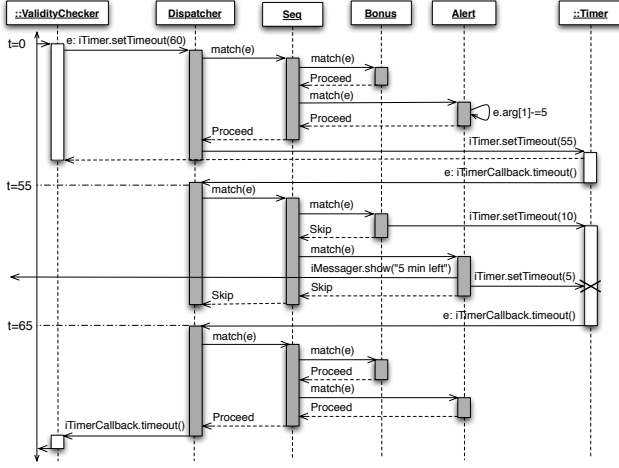


Fig. 5. Seq(DemuxFactory(Bonus), Alert) scenario

Figure 5 sketches the Seq(DemuxFactory(Bonus), Alert) scenario. At time 0, a user logs in for 60 minutes provided by his fly ticket. The message setTimeout(60) sent from a ValidityChecker to its Timer is intercepted by the Dispatcher controller. The Dispatcher invokes the match method of the Seq operator. The Seq then calls the DemuxFactory(Bonus) that instantiates a Bonus and calls its match method that proceeds the call (setTimeout(60) is not of interest to Bonus). The call is then sent to Alert that subtracts 5 minutes from 60 and proceeds the call with the new parameter value (55). As a result, the Dispatcher calls setTimeout(55). At time 55, a timeout is intercepted. The Dispatcher invokes again the match method of the Seq. The Seq sends the message to the DemuxFactory(Bonus) first, that routes the message to the corresponding Bonus instance which resets the timer for 10 minutes and skips the message. Then, the message is sent to Alert that warns the user, resets the timer for 5 minutes and skips the message. 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 ensured by the default sequential composition of controllers. 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*. This strategy can be abstracted with the Alt (alternate) composition operator. This binary operator aggregates two ICController(s) and a set of intercepted services common to both controllers. When a service common to both controllers is intercepted, its occurrences are passed alternately to the left and the right hand side controllers. When a service is not common to both controllers, the Alt operator passes the call to both controllers sequentially and the call is proceeded when both controllers returns Proceed, otherwise, the message is skipped. This works since the ICControllers are

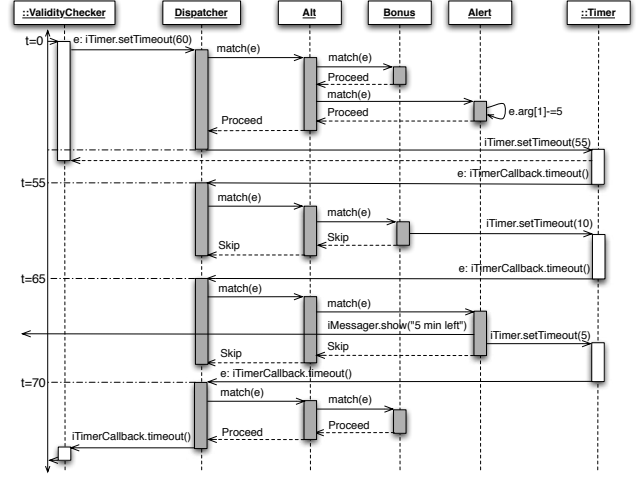


Fig. 6. Alt(DemuxFactory(Bonus), Alert) scenario

designed to return Proceed for irrelevant calls.

Figure 6 sketches the Alt(DemuxFactory(Bonus), Alert, {"timeout"}) scenario. At time 0, a user logs in for 60 minutes provided by his fly ticket. The message setTimeout(60) sent from a ValidityChecker to its Timer is intercepted by the Dispatcher and sent to Alt. The setTimeout is not a common intercepted message, so Alt sends it first to DemuxFactory(Bonus) that instantiates a Bonus and calls its match method that proceeds the call. Second, Alt calls the match method of Alert that subtracts 5 minutes from 60 and proceeds the call. Since both controllers want to proceed the call, Alt returns Proceed which causes the Dispatcher to call setTimeout(55). At time 55, a timeout is intercepted which is a common intercepted message by both controllers. Here, Alt sends it only to DemuxFactory(Bonus) (first occurrence) that routes it to the corresponding Bonus instance which resets the timer for 10 minutes and skips the message. Thus, the Dispatcher ignores this timeout call. At time 65, a second timeout is intercepted. Alt sends it only to Alert (second occurrence) that warns the user, resets the timer for 5 minutes and skips the message (the Dispatcher ignores again the timeout call). At time 70, a third timeout is intercepted. Alt this time, sends it only to DemuxFactory(Bonus) (third occurrence) that returns Proceed. As a result, the Dispatcher calls timeout and the session is ended which is the desired behavior.

IV. FORMAL VERIFICATION IN UPPAAL

Uppaal [4] is a model checker used to design, simulate and verify systems that can be modeled as networks of timed automata. Each timed automaton models a part of the system (e.g., component). A timed automaton is a finite state-machine extended with local variables, data types and clock variables. A clock variable is modeled by float values and all clocks of the same system progress synchronously. An automaton specification is called template and its instantiation is called

process. Besides its time support, other interesting features are also provided by Uppaal. For example, it enables passing data between processes, automatic and multiple instantiation of templates, declaring a set of C functions with loops to be used as transitions guards or state variables assignments. A system in Uppaal is modeled as a parallel composition of timed automata. In Uppaal, properties to be verified are specified in a subset of CTL (computational tree logic). When the verification of a particular property fails, a diagnostic trace is automatically reported by Uppaal. The use of timed automata is important to model the *Timer* component and the timing constraints of our case study. In the following we describe the general modeling process of Fractal components with composable controllers in Uppaal.

A. Formalization of Primitive Components

Each primitive component is modeled as a Uppaal process. Here, we assume that each primitive component comes with its specification in Uppaal with the following notations:

- 1) each transition label is a concatenation of the component, interface and service identifiers. For example `user_iLogin_login` denotes the *login* service of the *ILogin* interface of the *User* component.
- 2) a synchronous communication is modeled with a pair of transitions (and the asynchronous communication by a single transition). For example, the synchronous login message is decomposed into `user_iLogin_login` (for call) and `E_user_iLogin_login` (for return).
- 3) arrays on transition labels are used for passing data between processes. For example, a customer uses his fly ticket number to login and passes it to other processes. This is denoted as: `user_iLogin_login[id]`.
- 4) parametrized templates are used for automatic instantiation of processes. For example, the *User* template is parametrized with `int[0, FLYTICKET_ID_NB]` `id` where `FLYTICKET_ID_NB` is a constant that defines the number of users to be instantiated.

For instance, Figure 7 shows a simplified Uppaal template modeling the *User* component³. The template can be read clockwise from the initial location. The initial location *S0* labeled *Disconnected* is distinguished with a double circle. A user first sends its `mac` address to the DHCP server to request an IP address (`requestIpAddress[mac]!`). When it receives an IP address (`requestIpAddress[ip]?`), it stores it in a local variable `ips`, then it requests to login with his fly ticket `id` (`login[id]!`). When he succeeds to login, he requests connections to web addresses (`connect[ips][wips]!`), that can succeed or fail. When it receives a `timeout[id]?`, it returns to the initial location.

B. Formalization of Composite Components

A composite is modeled as a set of Uppaal processes, one for each bound interface. Each template of these processes has

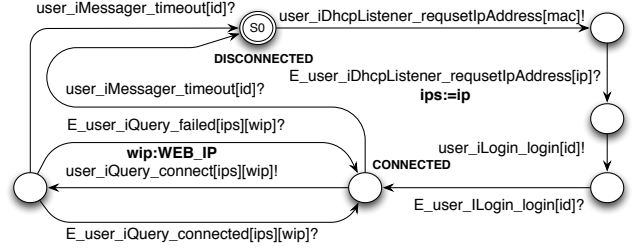


Fig. 7. Formal Model for the User Component

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, then forwards it). Synchronous services are represented by cycles with four transitions (receives a message, forwards it, waits for the reply, and forwards the reply).

C. Formalization of Bindings

We model component bindings with renaming. The binding of a required interface *i* of a component *c*₁ to a provided interface *j* of a component *c*₂ is modeled by replacing each transition label occurrence `c1_i_s` in the template of *c*₁ by `c2_j_s`, for each service name *s*.

D. Formalization of component systems

A component system *B* is a tuple $(\mathcal{A}_b, \mathcal{P}_b, Inv_b)$ where:

- 1) \mathcal{A}_b : is the parallel composition of processes modeling primitive and composite components after applying the binding (see Section IV-C).
- 2) \mathcal{P}_b : is a set of CTL formulas describing the behavior of the system.
- 3) Inv_b : is a subset of \mathcal{P}_b that should always be satisfied even if the system has been extended.

The complete airport system is modeled by 20 templates (9 for primitive components and 11 for composite components' interfaces). The system is designed to satisfy different (liveness, safety, and reachability) properties. These are given at the top of Table I ($\mathcal{P}_{airport}$). 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 fly ticket (*Safe 2*), all users can be connected at the same time (*Reach 1*). The formulas rely on different constants, variables and auxiliary functions: *IDS* denotes the range for user identifiers, *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*, and *cl* is a local clock associated to the user process.

Definition IV.1 A component system $\mathcal{B} = (\mathcal{A}_b, \mathcal{P}_b, Inv_b)$ is well defined if its modeling process satisfies all its desired properties:

$$\mathcal{W}(\mathcal{B}) \stackrel{\text{def}}{=} \mathcal{A}_b \models \mathcal{P}_b$$

³This template has more locations and transitions that are not shown here.

TABLE I
PROPERTIES OF THE AIRPORT SYSTEM

Properties for the Airport Base System ($\mathcal{P}_{airport}$)	
Live 1	$User(id).Connected \rightarrow User(id).Disconnected$
Safe 1	$A[]$ not deadlock
Safe 2	$A[] \text{ forall}(id:IDS) User(id).Connected \text{ imply } User(id).cl \leq \text{validity}(id)$
Reach 1	$E \langle \rangle User(0).Connected \text{ and } (\text{forall}(id:IDS) id \neq 0 \text{ imply } User(id).Connected)$
Properties for the Bonus controller (\mathcal{P}_{bonus})	
Safe 2'	$A[] \text{ forall}(id:IDS) User(id).Connected \text{ imply } User(id).cl \leq \text{validity}(id) + \text{BonusTime}$
Properties for the Alert controller (\mathcal{P}_{alert})	
Live 2	$User(id).Connected \rightarrow User(id).Disconnected \text{ and } User(id).isAlerted$
Safe 3	$A[] \text{ forall}(id:IDS) User(id).Alerted \text{ imply } User.cl == \text{validity}(id) - \text{AlertTime}$

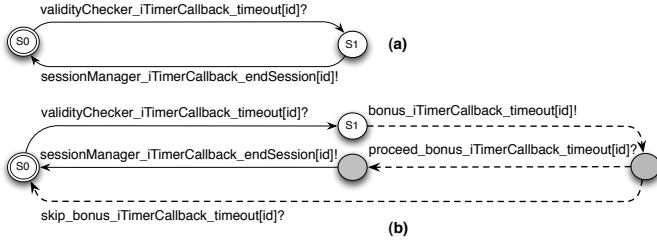


Fig. 8. ValidityChecker template adaptation (a) original, (b) adapted

E. Formalization of controllers

Each composable controller of the form (Dispatcher, IController) is modeled in two steps. First, a Upaal template is generated to model the behavior of the IController. Compared with primitive component templates (see Section IV-A), `proceed_e` and `skip_e` are used as transition labels to denote proceeding and skipping a service call `e`, respectively. Second, the generic Dispatcher is modeled by adapting the templates of the components to be affected by the controller. This enables the synchronization of IController process with the controlled components on the services that have to be intercepted. 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 controller process. The ValidityChecker waits for either `skip` to return to the original location, or `proceed` to forward the `timeout` to the SessionManager component.

In addition to the above specification, a set of intrinsic properties the controller ensures when it is applied to the system should be given. The properties for the bonus (\mathcal{P}_{bonus}) and the alert (\mathcal{P}_{alert}) controllers are shown in the middle and bottom part of Table I, respectively. The bonus controller ensures that the user can stay connected a bonus time (BonusTime) after its authorized time expires (Safe 2'). While 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 3). 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, and `BonusTime` and `AlertTime` are constants denoting the bonus and the alert time, respectively.

Definition IV.2 Given a component system $\mathcal{B} = (\mathcal{A}_b, \mathcal{P}_b, Inv_b)$ and a composable controller $\mathcal{CC} = (\mathcal{A}_{cc}, \mathcal{P}_{cc})$ where, \mathcal{A}_{cc} is the process modeling the controller behavior, and \mathcal{P}_{cc} is the set of its intrinsic properties. A composable controller is said to be correct with respect to a component system \mathcal{B} if the following condition holds:

$$\mathcal{A}'_b \parallel \mathcal{A}_{cc} \models Inv_b \wedge \mathcal{P}_{cc}$$

Where \parallel denotes the parallel composition of processes and \mathcal{A}'_b is the parallel composition of the \mathcal{B} processes after adapting the templates modeling the components affected by \mathcal{CC} .

For our case study, when the bonus is applied to the airport system, the invariant is defined by all the $\mathcal{P}_{airport}$ properties except (Safe 2), since bonus allows the user to connect for more than the time indicated in his fly ticket (see Safe 2'). While in the case of alert the invariant is simply $\mathcal{P}_{airport}$.

F. Formalization of Composition Operators

A composition operator is defined as a template to be instantiated. For instance, Figure 9 details the template for the alternate operator `Alt`. In the figure, when a service call is intercepted (`e?`), the function `swap()` is executed. This latter maintains a boolean `isLeft` to indicate which controller should be applied. If it is the left-hand side controller turn (`isLeft`), `e` is forwarded to this controller (`cc1_e!`), otherwise, `e` is forwarded to the right-hand side controller (`cc2_e!`). In both cases, the operator waits for either `proceed_cci_e` or `skip_cci_e` and forwards the intercepted command to the caller (`proceed_e` or `skip_e`).

The template is instantiated by substituting `e` with the service to be intercepted (e.g., `ITimerCallback_timeout`), `cc1` with the identity of the first controller (e.g., `bonus`), and `cc2` with the identity of the second controller (e.g., `alert`). These substitutions synchronize the composition operator process with its underlying controllers.

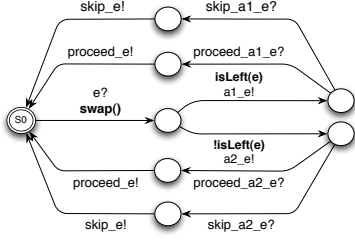


Fig. 9. The Alt template

G. Interference detection and resolution

For interferences detection, two composable controllers are interference-free with respect to a base program, if: (1) each composable controller is correct with respect to the base program, (2) when both composable controllers are added to the system, the result process satisfies all the properties of the underlying controllers and the system invariant. Formally:

Definition IV.3 Given a base system $\mathcal{B} = (\mathcal{A}_b, \mathcal{P}_b, Inv_b)$ and two composable controllers $\mathcal{CC}_1 = (\mathcal{A}_{cc1}, \mathcal{P}_{cc1})$, $\mathcal{CC}_2 = (\mathcal{A}_{cc2}, \mathcal{P}_{cc2})$, \mathcal{CC}_1 and \mathcal{CC}_2 are interference-free if the following conditions hold:

- 1) $\mathcal{W}(\mathcal{B})$: the base system is well defined
- 2) $\mathcal{A}'_b \parallel \mathcal{A}_{cc1} \models Inv_b \wedge \mathcal{P}_{cc1}$: \mathcal{CC}_1 is correct w.r.t \mathcal{B}
- 3) $\mathcal{A}''_b \parallel \mathcal{A}_{cc2} \models Inv_b \wedge \mathcal{P}_{cc2}$: \mathcal{CC}_2 is correct w.r.t \mathcal{B}
- 4) $\mathcal{A}'''_b \parallel \mathcal{A}_{cc1} \parallel \mathcal{A}_{cc2} \models Inv_b \wedge \mathcal{P}_{cc1} \wedge \mathcal{P}_{cc2}$: the composition is correct w.r.t \mathcal{B}

Where \mathcal{A}'_b , \mathcal{A}''_b and \mathcal{A}'''_b , denote the parallel composition of \mathcal{B} processes after adapting the templates affected by \mathcal{CC}_1 , \mathcal{CC}_2 , and \mathcal{CC}_1 and \mathcal{CC}_2 , respectively.

In our case study, when both bonus and alert are added to the system and composed with the default composition operator Seq (by instantiating its correspondent template), Safe 3 property is violated which reports an interference with a diagnostic trace. After analyzing the reported trace, we decided to use the Alt operator that solved the problem. In general, a composition operator solves an interference if when it is instantiated for two controllers and composed to the system, the interference disappears. Formally:

Definition IV.4 Given a base system $\mathcal{B} = (\mathcal{A}_b, \mathcal{P}_b, Inv_b)$ and two interfering controllers $\mathcal{CC}_1 = (\mathcal{A}_{cc1}, \mathcal{P}_{cc1})$, $\mathcal{CC}_2 = (\mathcal{A}_{cc2}, \mathcal{P}_{cc2})$, a composition operator Op solves an interferences if the following condition hold:

$$\mathcal{A}'_b \parallel Op(\mathcal{CC}_1, \mathcal{CC}_2) \models Inv_b \wedge \mathcal{P}_{cc1} \wedge \mathcal{P}_{cc2}$$

where $Op(\mathcal{CC}_1, \mathcal{CC}_2)$ denotes the parallel composition of the processes of \mathcal{CC}_1 and \mathcal{CC}_2 and the instantiated template Op for \mathcal{CC}_1 and \mathcal{CC}_2 as described in Section IV-F.

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 enough to detect the interference of Bonus and Alert.

V. RELATED WORK

Our work can be compared with two categories of works: extending components with aspect-orientation and aspect and feature interactions.

Extending Components with Aspects There are several aspect oriented extensions to Fractal. FAC [5], Fractal-AOP [6], and Safran [7] extend Fractal with aspects. In FAC an aspect is not a controller but several components. The application of an aspect to several components is achieved by creating a so called *aspect domain*. This latter, encapsulates all the components to be managed by the aspect and the components modeling the aspect itself. Fractal-AOP is quite similar to FAC but it provides an explicit controller interface *Proceed* to execute the original services. Safran focuses on adaptation: it proposes to insert a component to intercept service calls instead of their original targets, it executes the adaptation strategy and it possibly proceeds the calls. The main drawback of these approaches is that the extensions (i.e. aspects) are composed in a programmatic way (there are no predefined operators and `skip` is implicit). In addition, no interference detection support is provided.

Other component models have also been extended with aspects. The PRISMA framework [8] comes with an architecture language PRISMA AOADL to define where the extensions should be applied to the system. Aspects are defined as separate architectural elements. However, users are responsible to detect potential interferences among aspects and when one is detected the only composition strategy provided by the model is sequential ordering. LEDA is another component framework and AspectLEDA [9] is its extension with aspects. Aspect behaviors are represented by regular LEDA components. Aspect execution is ordered following a predefined priority order. In particular, JAsCo [10] is not a hierarchical component model and it provides an API to compose aspects in a programmatic way. But no interference detection support is provided.

Interference detection and resolution Interference detection and resolution is still a challenge for features [11] and aspects [12]. Current works on features are focussing on domain specific interferences. For example, Gouya et al. [13] 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. Goldman et al. [14] is the closest related work with respect to our formal verification approach. They 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 the aspect. They weave

an aspect at a time. 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. Note that they only consider weakly invasive aspects. Krishnamurthi et al. [15] 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 as previously been used by Katz et al. [16] 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 [17] advices are annotated with assumptions about their composition. Interferences are detected by matching the assumptions of an advice and all the other advices. However, these approaches focus only on interference detection at shared joinpoints. Our experience shows that controllers may interfere even if they are not applied to common components [18]. Our current proposal is a byproduct of our previous work on aspect interference detection and resolution [19] and formalization of aspects in a concurrent context [20]. The first work focuses on interferences at shared joinpoints and introduces composition operators. The second models the woven system as FSP processes and checks properties with LTSA.

VI. CONCLUSION AND FUTURE WORK

In this article we have shown how to extend Fractal with composable controllers. Our controllers with explicit actions (proceed and skip) are easily combined with composition operators. Composition is not restricted to ordering controllers and it is not restricted to controllers intercepting common services (*e.g.*, Alert intercepts `setTimeout` but Bonus does not). In fact, two controllers with no common intercepted services can be composed together. For instance, an `ICController CC1` (that maintains a predicate) and an `ICController CC2` could be composed with a `Cond` operator that calls the `match` method of `CC2` only if the condition holds. In particular, the composition `Cond(Overload, Bonus)` would add bonus times only when the server is not overloaded. Note that, other operators can also be developed in a similar way. Take for example, the operator `And` that calls the second `ICController` only when the first one proceeds the call, or the `Xor` that calls the second only when the first skips the call. We have also shown how Fractal components and composable controllers can be formally modeled in Uppaal. This way, the properties of the extended system can be checked and traces violating properties help to select the right composition operator.

Our proposal is currently partly supported by tools. The introduction of controllers is fully automated: our tool parses VIL expressions [18] that define the components to be controlled and the services to be intercepted in a declarative style. Then, it transforms accordingly the Fractal ADL definition

by introducing controllers and new composites of shared components if the required ones are scattered in the original architecture. The `Dispatcher` and different composition operators are also implemented. In this article we have identified the transformation scheme required to produce a Uppaal model of the complete extended system. We believe that some parts of the scheme can be automated such as the generation of composites and bindings from Fractal-ADL. We plan to develop such an Uppaal transformer.

REFERENCES

- [1] 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.
- [2] O. Šery and F. Plášil, "Slicing of component behavior specification with respect to their composition," in *CBSE*, ser. LNCS, vol. 4608. Springer, 2007.
- [3] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plášil, "Component reliability extensions for fractal component model," <http://websvn.ow2.org/>.
- [4] 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.
- [5] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye, "A component-based and aspect-oriented model for software evolution," *Int. J. Computer Applications in Technology*, vol. 31, no. 1/2, pp. 94–105, 2008.
- [6] H. Fakihi, N. Bouraqadi, and L. Duchien, "Aspects and software components: A case study of the fractal component model," in *WAOSD'04*.
- [7] P. C. David and T. Ledoux, "Towards a framework for self-adaptive component-based applications," in *DAIS*, ser. LNCS, vol. 2893. Springer, 2003, pp. 1–14.
- [8] J. Pérez, N. Ali, J. A. Carsí, I. Ramos, B. Álvarez, P. Sanchez, and J. A. Pastor, "Integrating aspects in software architectures: Prisma applied to robotic tele-operated systems," *Information and Software Technology*, vol. 50, no. 9-10, pp. 969–990, 2008.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] E. Katz and S. Katz, "Incremental analysis of interference among aspects," in *FOAL*. ACM, 2008, pp. 29–38.
- [17] A. Marot and R. Wuyts, "Detecting unanticipated aspect interferences at runtime with compositional intentions," in *RAM-SE*. ACM, 2009, pp. 31–35.
- [18] 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.
- [19] R. Douence, P. Fradet, and M. Südhof, "Composition, reuse and interaction analysis of stateful aspects," in *AOSD*. ACM, 2004, pp. 141–150.
- [20] R. Douence, D. L. Botlan, J. Noyé, and M. Südhof, "Concurrent aspects," in *GPCE*. ACM, 2006, pp. 79–88.