An Event-Based Coordination Model for Context-Aware Applications

Angel Núñez, Jacques Noyé

An Event-Based Coordination Model for Context-Aware Applications

Angel Núñez and Jacques Noyé

OBASCO project, École des Mines de Nantes – INRIA, LINA
4, rue Alfred Kastler. B.P. 20722, 44307 Nantes cedex 3, France
{Angel.Nunez,Jacques.Noye}@emn.fr
http://www.emn.fr/x-info/obasco

Abstract. Context-aware applications adapt their behavior depending on changes in their environment context. Programming such applications in a modular way requires to modularize the global context into more specific contexts and attach specific behavior to these contexts. This is reminiscent of aspects and has led to the notion of context-aware aspects. This paper revisits this notion of context-aware aspects in the light of previous work on concurrent event-based aspect-oriented programming (CEAOP). It shows how CEAOP can be extended in a seamless way in order to define a model for the coordination of concurrent adaptation rules with explicit contexts. This makes it possible to reason about the compositions of such rules. The model is concretized into a prototypical modeling language.

1 Introduction

A *context-aware application* is an application that is able to adapt its behavior in order to best meet its users' need. It does so by taking into account *context* information, *i.e.*, any piece of information *relevant to the interaction between a user and an application* [1]. This typically includes information on the physical environment (*e.g.*, noise level, time of day, location, computer resources) as well as the social environment of the user (*e.g.*, nearby people, previous interactions, objectives, mood). The versatility of this notion of context has lead to a focus on context modelling and structuring [2, 3] against the dynamic aspects of context change.

Some (reactive) context-aware applications [4–8] adapt their behavior using Event-Condition-Action (ECA) rules (first used in the field of reactive databases [9]), also referred to as *adaptation rules*. An ECA rule defines an action to be performed as a reaction to some event under a certain condition. In context-aware applications, the event part refers to context changes, the condition part to the current context and the action part to an adaptive behavior. In spite of the fact that the necessity of coordinating adaptation rules was early proposed (coordinated adaptation [5]), not much work has been done in this regard. As this paper shows, uncoordinated adaptation rules may lead to an inconsistent application state.

Let us illustrate this point with an example (inspired by an example used in the presentation of Fact Spaces [10]). Suppose Bob has a laptop that is able to control the house devices in terms of his location in the house. In the living room there is a video projector and speakers. Both devices support wireless connection to the laptop. Bob has programmed his laptop such that, when he is in the living room, the laptop connects to the video projector and the speakers, and opens a video player to play his favorite music clips in the living room.

Bob has however noted an undesirable behavior. Sometimes he can watch the clips in the video projector but the corresponding sound is not played in the speakers. Instead, he hears the music of his roommate Alice, who has a similar laptop. This is because Alice has programmed her laptop to listen to music in the living room, which only requires an audio connection. So, when Bob arrives after her, she has already a connection to the speakers. To solve this, Bob has reprogrammed his laptop. The context in which the clips have to be played is when he is in the living room and he has access to both the projector and the speakers. Now, when Alice and Bob are in the living room one can listen to music if Alice has arrived first, otherwise one can watch and listen to music clips.

Last week Alice has changed her preferences. Now she has reprogrammed her laptop to also play music clips in the living room (in the same way as Bob). Since then, a problem sometimes arises when Alice and Bob arrive at the same time in the room: nobody can see their clips. The reason is that sometimes Alice's laptop gains access to the speakers, whereas Bob's laptop gains access to the video projector. Since each laptop requires access to both resources in order to play the clips, no clip is played. We can see this situation as a kind of (context-aware) deadlock.

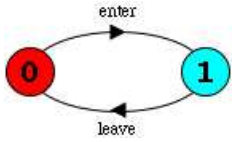
The behaviors programmed by Bob and Alice can be seen as adaptation rules that adapt the video player and the resources to the context of a presence of Bob and Alice in the living room. This is an example of uncoordinated adaptation leading to an inconsistent state in the application. We propose a model for the coordination of concurrent adaptation rules. The model is based on a model of concurrent aspects, CEAOP [11], and is provided in the form of a language for modeling the adaptation of applications to context changes. The language extends Finite State Processes (FSP) [12] proposed by Kramer *et al.*, which is a simple algebraic notation to describe process models. In our language, an application written in plain FSP syntax is enhanced with explicit contexts and adaptation rules. The enhanced application is translated into pure FSP and checked against the LTSA tool [12] to detect concurrency problems.

This paper is structured as follows. Section 2 briefly describes FSP and shows how our running example is modeled in FSP. Section 3 presents our language and at the same time describes how the running example can be enhanced with explicit contexts and adaptation rules. Section 4 describes the model of concurrent adaptation rules. Section 5 discusses related work. Finally, Sect. 6 concludes.

2 Overview of FSP

2.1 Syntax and Informal Semantics

A Labeled Transition System (LTS) is a form of state machine description, such that its transitions are labeled with action names. The left of Fig. 1 models, for instance, the behavior of a person that `enter`s and `leave`s a room. The action `enter` causes a transition from *state(0)* to *state(1)*, and the action `leave` causes a transition from *state(1)* to *state(0)*.




```
Person   = ( enter -> Inside ),
Inside   = ( leave -> Person ).

Person2  = ( enter -> leave -> Person2 ).
```

Fig. 1. Light switch state machine.

Finite State Processes (FSP) is a simple algebraic notation to describe process models. Each FSP description has a corresponding state machine (LTS) description. Figure 2 shows a simplified version of the FSP syntax (the full syntax can be found in [12]). This syntax definition, as well as the other definitions shown in this paper, are based on the syntax definition formalism SDF [13] (close to EBNF) and its implementation with scannerless generalized-LR parsing (SGLR) [14, 15]. An SDF production $s_1 ... s_n$ -> s_0 defines that an instance of non-terminal s_0 can be produced by concatenating elements from symbols $s_1 ... s_n$, in that order. SDF provides notation for optional (*?*) and iterated (**,+*) non-terminals. The notation $\{s\ lit\}^+$ represents a list of s separated by *lit*.

ProcId `=` *ProcBody* `.`	-> *ProcDef*
Proc	-> *ProcBody*
Proc `,` $\{$ *LocalProcDef* `,` $\}^+$	-> *ProcBody*
ProcId `=` *LocalProc*	-> *LocalProcDef*
`(` $\{$ *Branch* `|` $\}^+$ `)`	-> *Proc*
$\{$ *ActLabel* `->` $\}^+$ `->` *ProcId*	-> *Branch*
Label	-> *ActLabel*

Fig. 2. FSP syntax definition.

The definition of a finite state process (*ProcDef*) associates an identifier (*ProcId*) to a body (*ProcBody*). The body consists of a process expression (*Proc*) and an optional list of local process definitions (*LocalProcDef*). A process expression is a choice between one or more branches separated by the choice operator `|`. A branch (*Branch*) is a sequence of action labels separated by the sequence operator `->`, and terminated by a process identifier. Processes defined in the body are

referenced by their identifiers, whose scope is the definition of the finite state process. The non-terminal *ActLabel* is intentionally introduced for further use.

An FSP starts behaving as its process expression. It performs a sequence of actions until a reference to another process (or the same process) is found. Then, the FSP continues behaving as the referenced process. As an example, the code on the right of Fig. 1 illustrates two equivalent FSPs for the person's behavior. `Person` performs the action `enter` and continues behaving as the process `Inside`. Then, it performs the action `leave` and comes back to the initial state. `Person2` is a more compact notation for the same behavior.

Parallel processes can be built by composing sequential or parallel processes using the operator `||`. Parallel composition corresponds to the synchronized product of the corresponding automata. *Shared actions* constrain parallel composition so that an action shared by a set of processes is performed at the same time by all the processes of the set. Relabeling and hiding operations can be used to define which actions are actually shared at composition time. FSP is used to model applications as a set of processes together with their interactions through shared actions.

2.2 Modeling the Motivation Example using FSP

Let us illustrate FSP by modeling the behavior of our running example, without adaptation rules. Bob and Alice enter and leave the living room and both an audio and a video resources are ready to accept connections. Figure 3 shows this model. The basic behavior of a user is modeled by the process `Person` of the previous section. The activity of Alice is modeled by the process `alice:Person`, an "instance" of the process `Person`, whose actions are prefixed by `alice`. Analogously, the activity of Bob is modeled by the process `bob:Person`. The process `Resource` models a resource, which can be `acquire`d by a user and afterward `release`d. The video resource is modeled by the process `{alice.video,bob.video}::Resource`, an instance of the process `Resource`, whose transitions are duplicated and each duplicate is prefixed by `alice.video` and `bob.video`, respectively. This means that there are two distinct actions (`alice.video.acquire` and `bob.video.acquire`) to obtain the video resource and similarly two actions to free it (`alice.video.release` and `bob.video.release`). Analogously the process `{alice.audio,bob.audio}::Resource` models the audio resource. The access to each resource is mutually exclusive. Finally, the process `Application` models the application by composing all the processes using the composition operator `||`.

3 The Language

This section presents our language for modeling the adaptation of applications to context changes. The language extends FSP with *adaptation rules*, *contexts*, and *context rules*. An adaptation rule is used to adapt a base application by

```
Resource      = ( acquire -> release -> Resource ).

||Application = (  alice:Person || bob:Person
                 || {alice.video, bob.video}::Resource
                 || {alice.audio, bob.audio}::Resource ).
```

Fig. 3. Model of the application in FSP.

triggering *reactions* to the occurrence of particular base actions. In our example, an adaptation rule can trigger playing clips as a reaction to the entrance of a user in the living room and can stop playing clips as a reaction to the user's exit. A context is used to abstract a situation, e.g. the presence of a user in the living room. A context rule improves adaptation rules by attaching behavior to abstract contexts rather than to concrete actions. Once defined, context rules can be instantiated with respect to concrete contexts. In our example, a context rule can attach the behavior of playing clips to an abstract context denoting a situation when clips should be played. By instantiating the context rule, it is possible to associate this situation to the presence of a user in the living room. Decomposing adaptation rules into contexts and context rules makes it easier to capture context-awareness by making the notion of context explicit. It also improves modularity and reuse.

The overall language is built as a combination of languages, namely FSP, adaptation rules, contexts, and context rules. Thanks to the use of SDF, we can easily combine the FSP syntax definition presented in Sect. 2 with the other languages. We use *grammar mixins*, i.e. syntax definitions parameterized with the context in which they are used. We explain this notion in the remainder of this section as we present the different parts of the language.

3.1 Adaptation Rules

An adaptation rule is used to adapt an application. The application that is the subject of this adaptation is denoted as the *base* application. Furthermore, we use the term "base" as an adjective to denote an entity that belongs to the base application, *e.g.* base action.

Let us call *events* the base actions. An adaptation rule in our language can be seen as a process that observes events and can optionally react by introducing actions, called *reactions*, where observing an event means synchronizing on it. The definition of an adaptation rule only talks about the events of interest in each state. A complete process model of the rule, which defines what happens for each event in each state, is generated by a compiler, which translates our language to FSP. In particular, new transitions, that we call *waiting loops*, are created in each state for each (shared) label not explicitly taken care of. These transitions simply loop back to their source state.

Figure 4 shows the syntax of an adaptation rule, which is very similar to the FSP syntax. We use grammar mixins in order to reuse the FSP syntax. An adaptation rule is defined as an identifier and a body. The body has the same

syntax as an FSP body (mixin *ProcBody[[RuleCtx]]*), but the mixin parameter is propagated within *ProcBody* and makes it possible to define new productions, here for *ActLabel*, which may now, in the context of an adaptation rule, include a reaction, distinguished through the operator `=>`. The prefix `+` is used to easily recognize adaptation rules.

```
 + ProcId = ProcBody[[RuleCtx]]  .  -> RuleDef
Label => Reaction                   -> ActLabel[[RuleCtx]]
{ Label ; }+                        -> Reaction
```

```
+PlayRule    = ( enter -> Inside),
 Inside      = ( video.acquire -> Video | audio.acquire -> Audio
               | leave -> PlayRule ),
 Video       = ( audio.acquire => play -> Played | leave -> PlayRule ),
 Audio       = ( video.acquire => play -> Played | leave -> PlayRule ),
 Played      = ( leave => stop -> PlayRule ).
```

Fig. 4. Syntax of an adaptation rule (at the top) and example of an adaptation rule that triggers the actions that `play`s and `stop`s clips (at the bottom).

As an example, the process `PlayRule` at the bottom of Fig. 4 illustrates an adaptation rule that plays the clips in the living room. Note the use of the operator `=>` to indicate the reactions `play` and `stop`. This rule can be afterward instantiated for each user using prefixing, *e.g.* `alice:PlayRule` corresponds to the adaptation rule that plays the clips for Alice.

Adaptation rules are expressive enough to observe context changes and react. However, without an independent notion of context, contexts are implicitely embedded in the adaptation rules to the detriment of modularity and easy reasoning.

3.2 Context

Context Modeling A fundamental part of a context-aware application is context modeling. Due to its versatility, the notion of context can be represented in different ways like *key-value* models, *logic-based* models and *ontology-based* models [16]. The choice of a specific model depends on the required level of abstraction. *The exact GPS position of a person might not be of value for an application but the name of the room the person is in, may be* [3]. We aim to abstract the notion of context as much as possible in order to facilitate the definition of adaptation rules and the verification of the adapted systems.

A context represents an environmental state. Adaptation rules adapt an application with respect to such a state. Usually, this adaptation is required as soon as the context changes, which means that adaptation rules are triggered as soon as the change is detected. The application detects context changes through

computations such as the evaluation of a value in a key-value model, or the detection of a new fact in a logic-based model. We consider these computations as *context-switch events* and we model a context using them. In addition, we assume that these events can be extracted from the application model. For example, this is when `alice.enter` occurs that the system can detect that Alice is in the living room. In this setting, we model a context as the tuple *(context, in, out, context provider)*, where *context* is the name given to the context, *in* and *out* are context-switch events, *in* refers to the event making the system detect that it is in *context*, *out* refers to the event making the system detect that it is not in *context* anymore, and *context provider* is a process that defines these events. In the remainder, it will be said that a context is *active* in all the states following the *in* action and preceding the *out* action, otherwise it will be said to be *inactive*.

Context Providers A context is defined in a *context provider*, which has the same name as the context. A context provider is a process that observes events and indicates what are the context-switch ones. A context provider can be either *primitive* or *composite*. A primitive context provider observes events and annotates, with the suffixes `:in` and `:out`, the *in* and *out* actions, respectively.

Figure 5 shows the syntax of a primitive context provider. A primitive context provider is defined as an identifier and a body with the same syntax as an FSP. Furthermore, an action label in the context of a primitive context provider, can be a label followed by the suffix `:in` or `:out`. Figure 5 also illustrates a primitive context provider, namely `LivingRoom`, which defines the context of being in the living room. This context observes the base action `enter` and the sequence `enter -> leave`. As the annotations indicate, the context is activated at `enter` and deactivated at `leave`. In a similar way, we define the context `Connected`, which models a resource connection context. A context provider can be instantiated using prefixing, *e.g.* `video:Connected` represents the context of a connection to the video resource.

```
@ ProcId = ProcBody[[CpCtx]] .   -> PrimCxtDef
Label Suffix                     -> ActLabel[[CpCtx]]
:in | :out                       -> Suffix
```

```
@LivingRoom = ( enter:in -> leave:out -> LivingRoom ).
@Connected  = ( acquire:in -> release:out -> Connected ).
```

Fig. 5. Syntax of a primitive context provider (at the top) and example (at the bottom).

More complex contexts can be defined by composing simple context providers using operators. We provide the three basic logical operators: the conjunction operator `&`, the disjunction operator `|`, and the negation operator `!`. For example, the context `LivingRoom & Connected` is active when both `LivingRoom` and

`Connected` are active, i.e. when in the context `LivingRoom`, `Connected` becomes active, or vice-versa. In an analogous way, the context `LivingRoom | Connected` is active when either `LivingRoom` or `Connected` is active. Finally, `!LivingRoom` is active when `LivingRoom` is inactive, *i.e.* it is a context that is initially active and becomes inactive as soon as `LivingRoom` becomes active.

Figure 6 shows the syntax of a composite context provider. This figure also defines the context `Ready` as the conjunction of the context `LivingRoom` and the context `Connected`, instantiated for each resource. Because of the conjunction, `Ready` models the context in which a user is in the living room and a resource has been connected for him/her.

```
@ ProcId = ! ProcId .              -> CompCxtDef
@ ProcId = ProcId BinOp ProcId .   -> CompCxtDef
& | |                              -> BinOp

@Ready  = ( LivingRoom & video:Connected & audio:Connected ).
```

Fig. 6. Syntax of a composite context provider (at the top) and example (at the bottom).

3.3 Context Rules

We define *context rules* as a means to define adaptation rules that abstract the context away. A context rule can be seen as a parameterized adaptation rule that receives a context as a parameter, and can observe the context-switch events *in* and *out* associated to this context (through actions denoted by the keywords `in` and `out`, respectively). Once defined, a context rule can be instantiated for a concrete context. Figure 7 shows the syntax of a context rule (*CtxRuleDef*) and its instantiation (*CtxRuleInst*).

The rule that plays the clips of our running example can be written in a modular way using a context rule, as shown at the bottom of Fig. 7. `PlayDef` defines such a rule in terms of a generic context. In the same way, we can model a rule, namely `ConnDef`, that attempts to acquire a resource. It observes the beginning of the context and can either react by performing `acquire`, if the resource is free, or observe the deactivation of the context. Afterwards, it releases the resource if it was previously acquired. These rules can be instantiated for the concrete contexts `LivingRoom` and `Ready`. A context rule can be prefixed, *e.g.* `alice:PlayDef` would correspond to the rule that plays clips for Alice. The prefixing of a context rule is such that all the actions of the rule are prefixed, except the context-switch events.

3.4 Composition

Adaptation rules adapt a base application. In terms of FSP, this adaptation means a composition. We provide the operators `+` and `*` for denoting the se-

```
+ ProcId ( ProcId ) = ProcBody[[CrCtx]] .      -> CtxRuleDef
{ ActLabel[[CrCtx]] -> }+ -> ProcId ( ProcId ) -> Branch
Label => Reaction                              -> ActLabel[[CrCtx]]
( in | out ) => Reaction                       -> ActLabel[[CrCtx]]
in | out                                       -> ActLabel[[CrCtx]]
+ ProcId = ProcId( ProcId ) .                  -> CtxRuleInst
```

```
+PlayDef(Cxt)  = ( in => play -> out => stop -> PlayDef(Cxt) ).
+ConnDef(Cxt)  = ( in => acquire -> out => release -> ConnDef(Cxt),
                 | in => out -> ConnDef(Cxt) ).
+ConnRule      = ConnDef(LivingRoom).
+PlayRule      = PlayDef(Ready).
```

Fig. 7. Syntax of a context rule (at the top) and examples (at the bottom).

quential composition of the FSP that represents a base application with one or more (context) adaptation rules, whose syntax is shown in Fig. 8.

```
|| ProcId = BaseExpr SeqOp ProcId .   -> Adaptation
ProcId                               -> BaseExpr
BaseExpr SeqOp ProcId                -> BaseExpr
+ | *                                 -> SeqOp
```

Fig. 8. Syntax of the composition of an application and adaptation rules.

The composition of the base application with a single rule gives as a result a new FSP denoting an adapted base application. This composition is such that a reaction defined for an event takes place after the event is performed by all the base processes synchronizing on such an event. Furthermore, these processes cannot continue until the reaction has been performed.

When applying several rules, these operators are left-associative, i.e. if B is a standard FSP, and *R1* and *R2* are rules, then B + *R1* + *R2* is the same as (B + *R1*) + *R2*. B is first composed with *R1* giving as a result a new adapted application that is afterward composed with *R2*. As a result, if two rules apply to the same event, some form of precedence takes place. Let us consider the example above when considering individual events. The general scheme is that the reaction of *R2* precedes the reaction of *R1*. In other words, the last adaptation has priority. When considering context rules, this general scheme is applied to all the events when using the operator +. The operator * behaves slightly differently with respect to "out" reactions, which are in reverse order: the "in" reaction of *R2* precedes the "in" reaction of *R1*, but the "out" reaction of *R2* comes after the "out" reaction of *R2* in order to obtain a form of nesting.

3.5 Adaptation Rules and Aspects

Aspect-Oriented Programming [17] makes it possible to localize concerns that cannot be encapsulated in standard modularization systems. In our language, an adaptation rule can be seen as a kind of aspect. It observes actions that are scattered in the definition of other processes and introduces behavior. However, a specific property of aspects is that they may prevent the base program from executing some actions, and this is clearly not supported by our adaptation rules.

We include support for aspects in our language by extending the syntax of adaptation rules with *aspect expressions*. The extension allows adaptation rules to observe when a base action is about to happen. Then, it can introduce actions before and/or after the occurrence of the action. In addition, it can prevent the base program from performing such an action. An adaptation rule is equipped with aspect expressions of the form *action* `>` *before; ps; after*, where *action* is a base action, *before* a sequence of actions performed before *action*, *after* a sequence of actions performed after *action*, and *ps* either `skip` or `proceed`. The action `skip` means that the base action must be skipped and the action `proceed` means that this base action must take place. With this extension, expressions of the form *event* `=>` *reaction* are syntactic sugar for the expression *event* `> proceed;` *reaction*. For the sake of simplicity, this paper just deals with the case *ps* is `proceed` and only after actions are defined. Figure 9 shows the way the syntax of (context) adaptation rules is extended.

Label `>` *Advice*	-> *ActLabel[[RuleCtx]]*
Label `>` *Advice*	-> *ActLabel[[CrCtx]]*
(`in` | `out` *)* `>` *Advice*	-> *ActLabel[[CrCtx]]*
(Label `;` *)* PS (* `;` *Label)**	-> *Advice*
`proceed` | `skip`	-> *PS*

Fig. 9. Extension of (context) adaptation rules with aspect expressions.

Including support for aspects in the language allows us to reuse previous work on concurrent event-based aspect-oriented programming (CEAOP) [11, 18, 19]. In this way, we add support for concurrent rules using the model of concurrent aspects as the next section shows.

This section has presented a language to model context-aware applications. Our running example can be modeled in a modular way using explicit context and context rules. The next section is about coordinating rules in order to avoid inconsistent states in an application.

4 Concurrent Adaptation Rules

We denote as concurrent adaptation rules all the rules that are triggered by the same event. In our running example, the rule that attempts to open a video

connection is concurrent with the one for an audio connection. Both rules are triggered by the same event: the entry to the living room. The uncoordinated behaviors of these rules may lead to an inconsistent state in the base application, as mentioned in the introduction of this paper. The necessity of coordinating concurrent rules has already been presented as the necessity of coordinated adaptation [5]. We contribute to this by presenting a model for the coordination of adaptation rules.

Adaptation rules are translated into aspects, as shown in the previous section. Thus, the schemes for aspect coordination introduced by CEAOP can be used for adaptation rules, based on a combination of event renaming and hiding, and the use of specific operators. In the following we present the main points behind the coordination of aspects restricted to adaptation rules, *i.e.* aspects that always proceed and that only define after advices. More details about coordinating full-blown aspects are available in [11]. In our model, a base application is considered as a combination of several processes, and an adaptation rule as an independent process that is coordinated with the base application and the other adaptation rules. Adaptation rules are translated into FSPs, which are composed with the FSPs that model the base application. Some variability is allowed for the coordination, which determines the way the translation is done, as described in the remainder of this section.

4.1 Coordinating the Base Application with Adaptation Rules

An adaptation rule can be composed with the base application following several coordination schemes: (1) the reactions to an event can be performed in the background as soon as the event occurs, while the base application may continue its normal computation, (2) the base application may wait until the reaction has been performed, or (3) the reactions can be performed in parallel with the event. We embody these schemes in the operator `sync(`*base*`,`*rule*`,`*parallel*`,`*yield*`)`, where *base* is the FSP that models the base application, *rule* is an adaptation rule, *parallel* is a boolean value denoting whether reactions are triggered in parallel with events, and *yield* is a boolean value denoting whether the base application has to wait for the end of the reactions. In this setting, an expression B `+` R using the operator `+` of Sect. 3.4 is equivalent to `sync(`B`,`R`,false,true)`, i.e. the reactions occur after the event and the base application waits for the end of them.

The coordination, using the operator `sync`, of an adaptation rule declaring an expression *event* `=>` *reaction* `->` Q is implemented as follows. The reaction is translated into the sequence at the bottom of Fig. 10. The base application is instrumented such that all the occurrences of *event* `->` P are translated into the sequence at the top of Fig. 10. The following *synchronization events* are included in the translations: `pb_`*event* (the event is about to be performed), `pe_`*event* (the event has been just performed), and `e_`*event* (the end of the event scope). Figure 10 illustrates a coordination `sync(`B`,`R`,false,true)`. The labels surrounded with squares correspond to the synchronization events and the vertical lines represent the different rendezvous. After a first rendezvous at `pb_`*event*,

the base application performs *event*. A second rendezvous is at `pe_`*event*. Then, the actions denoted by *reaction* are performed until a last rendezvous at `e_`*event*. The different options of `sync` can be achieved by hiding events. For example, if `pe_`*event* is hidden the reaction is parallel to the event, or if `e_`*event* is hidden the application does not wait for the reaction.

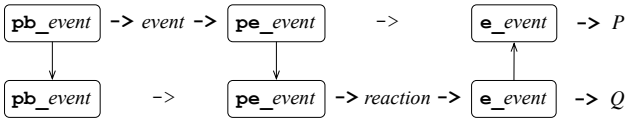


Fig. 10. Coordination of an application and an adaptation rule.

4.2 Coordinating Adaptation Rules

Let us come back to our running example and consider the rules for triggering the video and audio connections. These rules are not coordinated, leading to an inconsistent state in the application. Ordering the way resources are acquired is a known manner to solve this problem. This means coordinating adaptation rules.

Adaptation rules can be composed together using operators. The result of applying an operator is a composite rule that can be afterward composed with the base application. Two concurrent adaptation rules can be coordinated using two different schemes: (1) reactions to a shared event can be performed sequentially, or (2) they can be performed in parallel. We reuse the CEAOP operators to implement these schemes: the operator `Fun` implements sequential reactions, and the operator `ParAnd` parallel reactions.

The implementation is as follows. Let us consider two adaptation rules declaring expressions of the form *event* `=>` *reaction1* `->` Q, and *event* `=>` *reaction2* `->` R, respectively. Analogously to the previous section, the expressions are translated into the sequences on the middle and at the bottom of Fig. 11, respectively. The base application is instrumented such that all the occurrences of *event* `->` P are translated into the sequence at the top of Fig. 11. This figure illustrates the operator `Fun`. This operator uses relabeling to impose a rendezvous (indicated by a vertical line) between `pe_`*event* of the first rule and `e_`*event* of the third one by giving a common name `pe1_`*event*. As a result, the reactions are performed sequentially in the order *reaction2* `->` *reaction1*. Without this renaming, *reaction1* would run in parallel with *reaction2*, which is the behavior determined by `ParAnd`. (see Fig. 12). The operator `Fun` is used to implement the composition of a base application with two rules using the operator `+` as defined in Sect. 3.4.

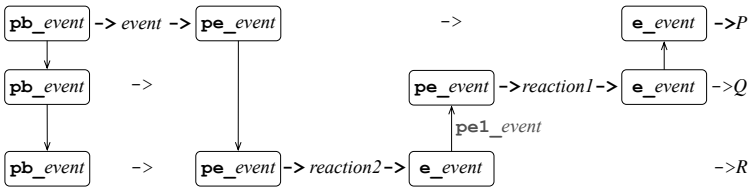


Fig. 11. Coordination of adaptation rules using the operator `Fun`.

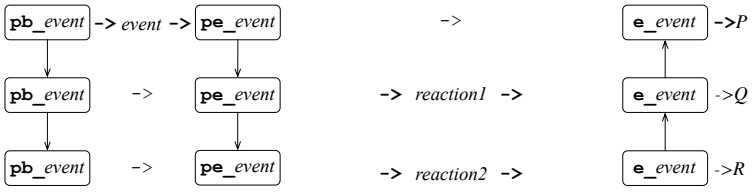


Fig. 12. Coordination of adaptation rules using the operator `ParAnd`.

4.3 Coordinating Contexts and Adaptation Rules

A context provider is a process that observes events and indicates which are the context-switch ones. A context rule uses these context-switch events in order to define reactions that depends on the context. Therefore, the coordination of the base application, context providers and context rules is done through these events.

At the implementation level, some instrumentation is required. Let us consider that a context is activated at the second occurrence of an event `foo`. In this case, it is not possible to say that `foo` is the context-switch event, because the first occurrence of `foo` does not activate the context. It is necessary to define other events "representing" each context-switch event, generated at the same time as the original context-switch event. If the name of the context is *context*, then `in_`*context* and `out_`*context* represent the context-switch events *in* and *out*, respectively.

Finally, a context provider is implemented as an adaptation rule throwing as a reaction to the occurrence of a context-switch event the corresponding pseudo context-switch event. Context rules are translated into adaptation rules by replacing the `in` and `out` events by the corresponding pseudo context-switch events.

This section has shown a model for the coordinated adaptation of an application using explicit contexts and adaptation rules. Coordinating schemes such as ordering reactions make it possible to keep a consistent application state while adapting the application.

5 Related Work

Fact Spaces [10] is a logic-based approach to context-awareness implemented in a concrete language called CRIME. One of the main ideas behind the model of

Fact Spaces, inherited from Linda, is the existence of a distributed data base of knowledge populated with shared facts. Each time a fact is added or removed from the data base, subscribed applications are notified. New facts can be added or actions in applications can be triggered as soon as their required facts are matched. These rules correspond to a list of facts, which have to be satisfied in order to trigger actions or publish other facts. Our language provides support for defining similar rules but using events instead of facts. This allows us to easily model concurrent rules and analyze concurrency properties using pre-existing frameworks, which in a logic-based system is harder.

ContextL [20] is an object-oriented programming language that allows for Context-oriented Programming. It provides means to associate partial classes and method definitions with *layers* and to activate and deactivate such layers in the control flow of a running program depending on the current context. Thus, the behavior of objects is extended with the activated layers. Although our approach is at the modeling level, we can find similarities. The behavior of an object modeled in an FSP can be extended with context rules, which can be seen as a kind of layers. Reactions are triggered depending on whether a given context is active or not. A unique feature of our approach is that we specifically deal with *concurrent* adaptation rules, whereas ContextL has no specific support for concurrent layers.

Shankar *et. al* introduced the ECPAP framework for policy-based management of a pervasive system [21]. The framework manages policies based on the Event-Condition-Action pattern: a policy defines an action to be performed when an event occurs under a given condition. In addition, the action is triggered only if a precondition holds, and it is considered successful if a postcondition is satisfied after its execution. The approach deals with concurrent policies, *i.e.* policies that define actions for a common event. When this event occurs a Petri net (built at compile time) defines an optimal order of execution of the involved actions. The adaptation rules of our approach are comparable with the policies of the ECPAP framework. Indeed, when considering individual events, our adaptation rules are based on a form of ECA pattern. However, our approach is more abstract. We consider events not only as environmental events but also as any kind of action (join point) in the computation of a system. In this way, an action triggered by an adaptation rule can be seen as an event by another rule, thus permitting the detection of possible conflicts between adaptation rules. Furthermore, linking related events using explicit events makes it possible to detect other kinds of conflicts between rules. Finally, we include an explicit notion of context and introduce context rules. The management of pre- and postcondition together with the automatic ordering of reactions is an interesting feature of the ECPAP framework that would be worth including in our approach.

In the area of Aspect-Oriented Programming, Tanter *et al.* introduced *context-aware aspects* [22], as aspects that match base-program joint points depending on whether a given context is active. They stated the necessity of context as a first-class entity in aspect-oriented languages that has to be stateful, compos-

able and parameterized. Our approach augments the proposal of Tanter *et al.* by providing support to concurrent context-aware aspects.

Finally, our approach is based on CEAOP [11] and therefore includes stateful aspects [23]. In our approach, part of a stateful aspect can be factorized out in the notion of context. The instantiation of a context rule with a concrete context can be seen as the completion of the initial state machine of a stateful aspect with the states and transitions defined in the concrete context.

6 Conclusion

This paper has presented an approach to model coordinated adaptation of context-aware applications by enhancing a simple process calculus, FSP, with adaptation rules, context, context rules, and aspects. We have built an extension of the LTSA tool that supports processes written using our language.[1] The extension translates (context) adaptation rules and context providers into FSPs. Then, these FSPs are manipulated with the standard LTSA tool in order to check concurrency properties.

Note that our main contribution is not in expressiveness (everything is translated into FSP), but rather in clarity of the specification and modular reasoning. In this regard, we provide a model that simplifies the analysis of concurrency properties in context-aware applications. Furthermore, our approach can be used as a model for context-aware applications that takes concurrency into account.

We are now interested in the use of this modeling language in two complementary directions: the analysis of existing programs, e.g. programs written using CRIME rules, with respect to their concurrency properties, as well as the synthesis of actual implementations by refining part of the model (e.g. adding parameters to actions) and combining the refined language to a general-purpose programming language (e.g. Java) that could be used to implement the atomic actions. Some work has already been done in this direction as we have previously explored the possibility of using CEAOP as the basis of combining Java components and aspects using behavioral interfaces [19]. When considering aspects, these interfaces have much in common with adaptation rules.

Acknowledgments. We would like to thank the anonymous reviewers for their helpful comments and mention that this work has been partly supported by the project AMPLE: Aspect- Oriented, Model-Driven, Product Line Engineering (STREP IST-033710).

References

1. Dey, A.K., Abowd, G.D.: Towards a better understanding of context and context-awareness. In: Proceedings of the CHI 2000 Workshop on the What, Who, Where, When and How of Context-Awareness, Georgia Tech (April 2000)

[1] Implementation available at http://www.emn.fr/x-info/anunezlo/ltsa-modeling/.

2. Gu, T., Pung, H.K., Zhang, D.Q.: Toward an OSGi-based infrastructure for context-aware applications. IEEE Pervasive Computing **3**(4) (2004) 66–74
3. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. International Journal of Ad Hoc and Ubiquitous Computing **2**(4) (2007) 263–277
4. Rakotonirainy, A., Indulska, J., Loke, S.W., Zaslavsky, A.B.: Middleware for reactive components: An integrated use of context, roles, and event based coordination. In Guerraoui, R., ed.: Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms, Proceedings. Volume 2218 of Lecture Notes in Computer Science., Heidelberg, Germany, Springer-Verlag (November 2001) 77–98
5. Efstratiou, C., Cheverst, K., Davies, N., Friday, A.: An architecture for the effective support of adaptive context-aware applications. In Tan, K.L., Franklin, M.J., Lui, J.C.S., eds.: Mobile Data Management, Second International Conference, MDM 2001, Proceedings. Volume 1987 of Lecture Notes in Computer Science., Hong Kong, China, Springer-Verlag (January 2001) 15–26
6. Ahn, J., Chang, B.M., Doh, K.G.: A Policy Description Language for Context-Based Access Control and Adaptation in Ubiquitous Environment. In Zhou, X., Sokolsky, O., Yan, L., Jung, E.S., Shao, Z., Mu, Y., Lee, D.C., Kim, D., Jeong, Y.S., Xu, C.Z., eds.: EUC Workshops. Volume 4097 of Lecture Notes in Computer Science., Seoul, Korea, Springer-Verlag (August 2006) 650–659
7. Daniele, L., Costa, P.D., Pires, L.F.: Towards a Rule-Based Approach for Context-Aware Applications. In Pras, A., van Sinderen, M., eds.: Dependable and Adaptable Networks and Services, 13th Open European Summer School and IFIP TC6.6 Workshop, EUNICE 2007, Proceedings. Volume 4606 of Lecture Notes in Computer Science., Enschede, The Netherlands, Springer-Verlag (July 2007) 33–43
8. de Ipiña, D.L., Katsiri, E.: An ECA Rule-Matching Service for Simpler Development of Reactive Applications. IEEE DSOnline **2**(7) (2001)
9. Paton, N.W., Díaz, O.: Active Database Systems. ACM Computing Surveys **31**(1) (1999) 63–103
10. Mostinckx, S., Scholliers, C., Philips, E., Herzeel, C., Meuter, W.D.: Fact Spaces: Coordination in the face of disconnection. In Murphy, A.L., Vitek, J., eds.: Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Proceedings. Volume 4467 of Lecture Notes in Computer Science., Paphos, Cyprus, Springer-Verlag (June 2007) 268–285
11. Douence, R., Le Botlan, D., Noyé, J., Südholt, M.: Concurrent aspects. In: Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'06), ACM Press (October 2006) 79–88
12. Magee, J., Kramer, J.: Concurrency: State Models and Java. 2nd edn. Wiley (2006)
13. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
14. Visser, E.: Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam (July 1997)
15. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, Springer-Verlag (April 2002) 143–158
16. Strang, T., Popien, C.L.: A Context Modeling Survey. In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing. (September 2004)

17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Aksit, M., Matsuoka, S., eds.: ECOOP'97 - Object-Oriented Programming - 11th European Conference. Volume 1241 of Lecture Notes in Computer Science., Jyvskyl, Finnland, Springer-Verlag (June 1997) 220–242
18. Núñez, A., Noyé, J.: A domain-specific language for coordinating concurrent aspects in Java. In Douence, R., Fradet, P., eds.: 3ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2007). (March 2007)
19. Núñez, A., Noyé, J.: A seamless extension of components with aspects using protocols. In Reussner, R., Szyperski, C., Weck, W., eds.: WCOP 2007 - Components beyond Reuse - 12th International ECOOP Workshop on Component-Oriented Programming. (July 2007)
20. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: An overview of ContextL. In: DLS '05: Proceedings of the 2005 Symposium on Dynamic Languages, ACM (2005) 1–10
21. Shankar, C.S., Campbell, R.H.: Ordering management actions in pervasive systems using specification-enhanced policies. In: 4th IEEE International Conference on Pervasive Computing and Communications (PerCom 2006), 13-17 March 2006, Pisa, Italy, IEEE Computer Society (2006) 234–238
22. Tanter, É., Gybels, K., Denker, M., Bergel, A.: Context-Aware Aspects. In Löwe, W., Südholt, M., eds.: Software Composition, 5th International Symposium, SC 2006, Revised Papers. Volume 4089 of Lecture Notes in Computer Science., Vienna, Austria, Springer-Verlag (March 2006) 227–242
23. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In Murphy, G.C., Lieberherr, K.J., eds.: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22-24, 2004, ACM (2004) 141–150