Modèles comportementaux pour le contrôle de composants logiciels autonomes

Frederico Alvares de Oliveira Jr.[1], Eric Rutten[1] and Lionel Seinturier[2]

[1] INRIA Rhône-Alpes, Montbonnot, France, frederico.alvares,eric.rutten}@inria.fr
[2] University of Lille 1 & INRIA Lille, Villeneuve d'Ascq, France, lionel.seinturier@inria.fr

Résumé

Les systèmes logiciels modernes et leur architecture doivent s'adapter dynamiquement de façon réactive aux événements provenant de l'environnement (p. ex. la charge de travail demandée par les utilisateurs, les modifications apportées aux fonctionnalités) et de la plate-forme d'exécution (p. ex. les ressources disponibles). Les architectures à base de composants ont montré leur adéquation pour l'auto-adaptation, non seulement en raison de leurs caractéristiques intrinsèques, comme la réutilisation et la modularité, mais aussi en raison de leurs capacités de reconfiguration dynamique. Cependant, les solutions existantes sont basées souvent sur des langages de bas niveau, impératifs, et sans modèle comportemental. Cet article présente Ctrl-F, un langage dédié pour le support de haut niveau pour la spécification des comportements et des politiques d'adaptation dans les système à base de composants logiciels. Nous nous appuyons sur la programmation réactive pour la vérification et le contrôle des reconfigurations. Nous intégrons Ctrl-F avec FraSCAti, une plate-forme intergicielle pour les architectures réparties orientées services.

1 Introduction

Les composants logiciels jouent un rôle majeur dans le développement de systèmes logiciels autonomes. L'adaptabilité et la reconfigurabilité sont des attributs qui sont recherchés dans ce paradigme : on veut être capable d'adapter les assemblages de composants (configurations) afin de faire face aux nouvelles exigences et conditions qui se produisent lors de l'exécution. Des boucles de contrôle peuvent être conçues pour prendre des décisions d'adaptation à l'exécution en choisissant la configuration suivante en fonction, d'une part, du passé observé (des événements, des états, des configurations), et d'autre part, de la logique d'adaptation et des propriétés à conserver invariantes tout au long de l'exécution du système contrôlé. Cependant, il peut être non trivial, en particulier pour les architectures complexes, de concevoir des boucles bien maîtrisées, avec des garanties sur la façon dont la navigation à travers l'espace de configurations est effectuée. En effet, cette forme de décision autonome implique non seulement la mise à jour de l'avancement des comportements, mais aussi des prédictions sur les futurs possibles du système.

Ces garanties peuvent être obtenues par des approches de du domaine de l'automatique, où l'utilisation de modèles comportementaux permet de prendre des décisions présentant des garanties. Comme nous nous concentrons sur le contrôle logique pour le choix des configurations, nous nous appuyons sur le contrôle discret. En particulier, nous abordons la conception d'un tel module de décision comme un problème de synthèse de contrôleurs discrets (SCD), qui consiste à générer automatiquement un contrôleur capable de commander un ensemble de variables de telle sorte qu'une propriété soit satisfaite. Pour ce faire, nous proposons la conception de gestionnaires de composants basés sur des modèles comportementaux à automates à états finis (AEF), qui fournissent des connaissances sur les événements et états passés ainsi que sur les futurs possibles, à savoir, l'espace de configurations accessibles. Ainsi, nous sommes en mesure d'éviter d'aller dans les directions menant à de mauvaises configurations.

Nous proposons un langage dédié, appelé Ctrl-F, pour décrire les programmes comportementaux à partir des concepts de haut niveau décrivant des reconfigurations architecturales. Nous donnons la définition comportementale des programmes Ctrl-F par leur traduction vers des AEF. Plus précisément, nous fournissons la traduction intégrale vers le langage réactif Heptagon/BZR, qui permet la compilation vers des outils formels et ainsi de bénéficier de la SCD ou de la vérification. Comme l'exploration des états peut être très coûteuse et exponentielle dans le nombre de configurations possibles, la SCD est faite hors ligne, i.e., compilée. En outre, cela se fait d'une manière maximalement permissive, c'est-à-dire que le contrôleur maintient le maximum de configurations possibles ne violant pas les politiques énoncées, ce qui rend le système autonome maximalement flexible. En conséquence, pour un programme comportemental donné, on produit une fonction exécutable qui, à chaque étape de décision, prend l'état actuel et les événements actuels, et renvoie une valeur de commande qui correspond à la prochaine configuration telle que les politiques énoncées sont appliquées.

La suite de l'article est organisé comme suit : la section 2 présente les principaux concepts et outils nécessaires à la bonne compréhension de notre approche. La section 3 présente le langage Ctrl-F et la section 4 présente sa traduction vers le langage Heptagon / BZR. La section 5 fournit des détails sur l'intégration de Ctrl-F avec une plate-forme à composants. Les travaux connexes sont discutés dans la section 7 et la section 8 conclut cet article.

2 Etat de l'art

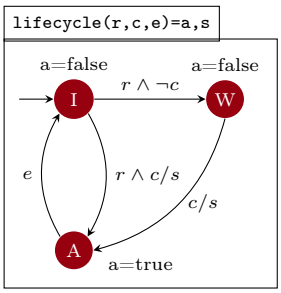
2.1 Architectures à base de composants

Une architecture logicielle définit la structure de haut niveau d'un système logiciel, en décrivant comment il est organisé sous la forme d'une composition de composants [19]. Les langages de description de l'architecture (ADL : *architecture description language*) [22] sont généralement utilisés pour décrire l'architecture d'un système. Malgré la diversité des ADL, les éléments architecturaux proposé dans la quasi-totalité des langages suivent la même base conceptuelle [14].

Un *composant* est défini comme l'unité la plus élémentaire de transformation ou de donnée et il est habituellement décomposé en deux parties : l'implémentation et l'interface. L'implémentation décrit le comportement interne du composant, tandis que les interfaces définissent la façon dont le composant doit interagir avec l'environnement. Un composant peut être défini aussi comme étant simple ou *composite* (composé d'autres composants). Un *connecteur* est le support des interactions entre les composants. Il sert de médiateur lors d'une communication inter-composants. Diverses formes d'interactions peuvent être envisagées.

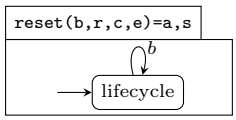
Une *configuration* correspond à un graphe orienté de composants et connecteurs décrivant la structure de l'application et / ou une description de comment les interactions entre les composants évoluent au fil du temps. D'autres éléments comme des attributs, des contraintes ou des styles architecturaux peuvent également apparaître dans les ADL [14].

Aujourd'hui, les applications logicielles nécessitent la possibilité de changement continu de manière à s'adapter aux conditions d'exécution ou de faire face aux nouveaux besoins des utilisateurs. Ainsi, il devient impératif qu'une architecture / configuration, une fois définie, puisse évoluer tout en conduisant à des nouvelles configurations qui sont plus adaptées au contexte actuel.




```
node lifecycle(r,c,e:bool)
       returns(a,s:bool)
let automaton
  state I do
   a=false;s=r & c
   until r & c then A |
         r & not c then W
  state W do
a=false;s=c until c then A
  state A do
   a=true;
   s=false
   until e then I
 end; tel
```

FIGURE 1 – Représentation graphique et textuelle du cycle de vie d'un composant.




```
node reset(b,r,c,e:bool)
     returns(a,s:bool)
let automaton
  state H do (a,s)=
       lifecycle(r,c,e)
          until b then H
  end; tel
```

FIGURE 2 – Exemple de composition hiérarchique.

2.2 Langages réactifs

Les langages réactifs ont été proposés pour décrire des systèmes qui, à chaque réaction, effectuent une étape en prenant des flots de valeurs en entrée, calculant les transitions, mettant à jour les états, déclenchant des actions, et émettant des flots de sortie [16]. Leur définition est souvent basée sur des automates à états finis (AEF), qui constituent le formalisme de base pour représenter les comportements : c'est le cas de StateCharts [17] et des langages synchrones [15].

Heptagon/BZR [10] est un exemple de langage qui permet la définition de systèmes réactifs à l'aide des machines de Moore généralisées, i.e., avec des équations synchrones de flots de données et des automates [6]. Un programme Heptagon est structuré en un ensemble de nœuds. Chaque nœud correspond à un comportement réactif qui prend en entrée et produit en sortie un ensemble de flots. Le corps d'un nœud est constitué d'un ensemble de déclarations qui prennent la forme d'automates ou d'équations. Les équations déterminent les valeurs des sorties, en termes d'expressions sur des valeurs des flots d'entrée ou locaux.

La Figure 1 montre une représentation graphique et textuelle d'un programme Heptagon. Le programme décrit le contrôle du cycle de vie d'un composant qui peut être en état de repos (I), attente (W) ou (A) actif. Le programme prend en entrée trois variables booléennes : `r`, qui représente un signal de demande pour le composant ; `c` : une condition externe (utilisé plus tard comme variable contrôlable) ; et `e` : une notification de terminaison. Il produit en sortie deux valeurs booléennes, indiquant si le composant est actif (`a`) et l'action de démarrage du composant (`s`). Dans l'état initial, suite à une demande (quand r est vrai), l'automate est conduit soit à l'état attente soit l'état actif, selon la condition `c`. Si elle n'est pas vraie, il va d'abord à l'état d'attente, puis à l'état actif lorsque `c` devient vrai. Toutes les transitions entrantes arrivant à l'état actif déclenchent l'action de démarrage (`s`). De cet état actif, l'automate est conduit à l'état de repos suite à la notification de terminaison `e`.

Une caractéristique importante de Heptagon/BZR est le support à la composition hiérarchique et parallèle d'automates. La Figure 2 illustre un exemple de composition hiérarchique, dans lequel un automate invoque le sous-automate *lifecycle*. Il dispose d'une transition qui résulte en la réinitialisation des sous-automates (i.e., *lifecycle*) à chaque occurrence de b. Le Listing 1 illustre la composition parallèle de deux instances du nœud *delayable* (par l'opérateur ';'). Ils s'exécutent en parallèle, d'une manière synchrone, ce qui signifie qu'un pas global correspond à un pas local pour chaque nœud.

2.2.1 Contrats et Synthèse de Contrôleurs Discrets

BZR [10] est une extension de Heptagon avec des contrats comportementaux. Sa compilation utilise les outils formels Sigali [21] ou Reax [3] à des fins de SCD. Un *contrat* est un programme qu'on associe à un nœud, avec deux sorties : e_A, une hypothèse sur l'environnement de nœud ; et e_G, une propriété qui doit être satisfaite par le nœud. Un ensemble $\{c_1, \ldots, c_q\}$ de variables contrôlables locales sont utilisées pour assurer cet objectif. Autrement dit, le contrat signifie que le nœud est contrôlé en donnant des valeurs à $\{c_1, \ldots, c_q\}$ telles que pour n'importe quel flot d'entrée satisfaisant l'hypothèse e_A, la sortie satisfait toujours l'objectif e_G. Lorsque qu'un contrat n'a pas de variables contrôlables spécifiées, une vérification que e_G est satisfaite dans l'espace d'états accessible est réalisée, même si aucun contrôleur n'est généré.

Listing 1 – Exemple de contrat en Heptagon/BZR.

```
1 node twocomponents(r1,r2,e1,e2:bool) returns (a1,a2,s1,s2:bool)
2 contract
3 assume true
4 enforce not(a1 and a2)
5 with (c1,c2)
6 let
7  (a1,s1)=lifecycle(r1,c1,e1);(a2,s2)=lifecycle(r2,c2,e2)
8 tel
```

Le Listing 1 montre un exemple de contrat sur un nœud englobant une composition parallèle de deux instances de *lifecycle* (cf. Figure 1). Il est composé de trois blocs. Le bloc *assume* (ligne 3) déclare qu'il n'y a pas d'hypothèse sur l'environnement (i.e., $e_A = true$). Le bloc *enforce* (ligne 4) décrit l'objectif de contrôle : $e_G = \neg(a1 \wedge a2)$, c'est à dire que les deux composants sont mutuellement exclusifs, et ne peuvent pas être actifs en même temps. Enfin, le bloc *with* (ligne 5) définit deux variables contrôlables qui sont utilisées dans le nœud (ligne 7). En pratique, elles auront des valeurs telles que les variables `a1` et `a2` ne sont jamais vraies au même instant.

2.2.2 Compilation et génération de code

La chaîne de compilation Heptagon/BZR, à partir du code source, produit en sortie un code séquentiel dans un langage de programmation à usage général (par exemple, Java ou C) contenant la mise en œuvre de la logique de contrôle, sous la forme d'une fonction *step* à appeler à chaque réaction dans la boucle autonome. Si le code fourni en entrée contient un *contrat*, le compilateur génère également un code intermédiaire qui sera donné en entrée de la SCD (Sigali ou Reax), qui produira en sortie un code Heptagon/BZR correspondant au contrôleur. Ce dernier est ensuite de nouveau compilé vers un code exécutable.

3 Le language Ctrl-F

Ctrl-F est notre proposition de langage dédié qui étend les langages ADL classiques avec des constructions de haut niveau pour décrire le comportement et les politiques de reconfigurations à appliquer tout au long de l'exécution du système cible.

3.1 Vue d'ensemble et concepts généraux

La syntaxe de Ctrl-F a deux parties : l'une statique, liée à des concepts architecturaux de base tels que les composants, les connexions, les configurations, etc.) ; l'autre dynamique, pour les comportements et les politiques reconfiguration. La partie statique de Ctrl-F partage les

concepts de nombreux ADLs existants (par exemple, Fractal [5], Acme [14]). Un composant (*component*) est constituée d'ensembles d'*interfaces*, de ports d'*événement*, d'attributs (*attributes*) et de *configurations*. Les *interfaces* définissent comment un composant interagit avec d'autres composants, en exprimant des fonctionnalité requises (*client interfaces*), qui peuvent être fournies par un autre composant et / ou en exprimant des fonctionnalités fournies (*server interface*) pouvant être utilisée par d'autres composants. Les *ports* décrivent les événements d'un certain type que le composant est capable d'émettre (*port out*) et / ou d'écouter (*port in*). Une *configuration* est composée d'un ensemble d'*instances* de *composants*, un ensemble de liaisons (*bindings*) raccordant les interfaces de service (*server interface*) aux interfaces clientes (*client interfaces*), et / ou un ensemble d'affectations de valeurs aux attributs.

La partie dynamique est constituée d'un comportement (*behaviour*) et d'un ensemble de politiques *policies* définis au sein de chaque composant. Un comportement prend la forme d'ordres et conditions (en ce qui concerne les événements et valeurs des attributs) sous lesquels les transitions entre les configurations (reconfigurations) ont lieu. Le politiques sont des objectifs ou contraintes de haut niveau, qui peuvent impliquer l'inhibition de certaines transitions.

3.2 Comportements

Une caractéristique particulière de Ctrl-F est sa capacité d'écrire les comportements dans le contexte des applications à base de composants. Nous entendons par le comportement le processus dans lequel les éléments architecturaux sont changés, et plus précisément l'ordre et les conditions dans lesquels les reconfigurations ont lieu.

Les comportements dans Ctrl-F sont définis à l'aide d'un langage impératif de haut niveau, composé d'un ensemble de déclarations comportementales (sous-comportements) qui peuvent être composées ensemble de manière à fournir des comportements plus complexes en termes de séquences de configurations. Dans ce contexte, une configuration est considérée comme étant un comportement atomique, à savoir un comportement qui ne peut pas être décomposé en d'autres sous-comportements. Une reconfiguration se produit lorsque la configuration en cours est interrompue et la prochaine est démarrée. Nous supposons que les configurations ne sont pas en mesure directe de se terminer ou se démarrer elles-mêmes : elles sont explicitement démarrées ou terminées par une déclaration de comportement. Néanmoins, comme les composants sont capables d'émettre des événements, on peut en définir dont l'objectif est d'émettre des événements afin de forcer un comportement souhaité, par exemple une reconfiguration.

La Table 1 résume les déclarations de comportement du sous-language comportemental de Ctrl-F. Lors de l'exécution d'un comportement B, la déclaration *when-do* définit que quand un événement du type e_i se produit la (les) configuration(s) qui compose(nt) B devrait (devraient) être terminée(s) et que celle(s) du comportement correspondant B_i est (sont) lancée(s).

La déclaration *case-then* est similaire à *when-do* avec la différence qu'un comportement donné B_i est exécuté si la condition c_i est vraie : on n'attend pas qu'un événement se produise. La déclaration *parallel* décrit que deux comportements sont exécutés en même temps, c'est à dire qu'à un certain moment, il doit y avoir deux branches d'exécution de comportement indépendantes en parallèle. Cette déclaration est également utile dans le contexte de composants atomiques (i.e., qui ne possèdent pas de configurations ayant des instances d'autres composants), où nous pourrions, par exemple, définir des configurations composées d'attributs orthogonaux comme le mode et la taille de la police : `text || font-huge`.

La déclaration *alternative* permet de décrire des points de choix parmi des configurations ou parmi d'autres déclarations de comportements séquentiels plus élaborés. Ces choix sont laissés libres dans les spécifications locales et seront résolus dans les assemblages de niveau supérieur

TABLE 1 – Résumé des déclarations comportementales.

Déclaration	**Description**
B **when** e_1 **do** B_1, ... , e_n **do** B_n **end**	Pendant l'execution de B à l'occurrence de e_i exécuter B_i
case c_1 **then** B_1, ... , c_n **then** B_n **end**	Exécuter B_i si c_i est vrai
$B_1 \mid B_2$	Exécuter soit B_1 soit B_2
$B_1 \mid\mid B_2$	Exécuter B_1 et B_2 en parallèle
do B **every** e	Exécuter B et le re-exécuter à chaque occurrence de e

(dans les composites), de manière à satisfaire les politiques en les contrôlant.

Enfin, la déclaration *do-every* permet l'exécution d'un comportement B et la ré-exécution de celui-ci à chaque occurrence d'un événement e. Il est important de mentionner que le comportement B est préempté : la (les) configuration(s) active (s) dans B est (sont) terminée (s), et la (les) première(s) dans B est (sont) démarrées.

3.3 Politiques

Les politiques sont exprimées avec des concepts de haut niveau décrivant des contraintes sur les configurations, soit temporelles soit sur des valeurs des attributs. Elles définissent un sous-ensemble de l'espace des configurations possibles, où le système devrait rester invariablement : l'objectif sera atteint en utilisant les points de choix pour contrôler les reconfigurations. Un exemple intuitif est que deux composants dans des branches parallèles peuvent avoir chacun plusieurs configurations possibles, et certaines d'entre elles doivent rester exclusives. Ceci est imposée par le choix de la configuration appropriées lors du démarrage des composants.

Les contraintes ou optimisations sur les attributs prennent la forme de prédicats et / ou primitives d'objectifs d'optimisation (i.e., maximiser ou minimiser) sur les attributs des composants.

Les contraintes temporelles sont des concepts de haut niveau sur l'ordre de configurations, qui peuvent être très utiles quand il y a beaucoup de chemins de reconfiguration possibles (e.g., par composition parallèle ou alternative), auquel cas la spécification déclarative est plus facile.

- $conf_1$ **precedes** $conf_2$: $conf_1$ doit avoir lieu juste avant $conf_2$. $conf_1$ peut ne pas être la seule, mais elle doit figurer parmi les configurations ayant lieu avant $conf_2$.
- $conf_1$ **succeeds** $conf_2$: $conf_1$ doit avoir lieu juste après $conf_2$. $conf_1$ peut ne pas être la seule, mais elle doit figurer parmi les configurations ayant lieu juste après $conf_2$.
- $conf_1$ **during** $conf_2$: $conf_1$ doit avoir lieu avec (en même temps que) $conf_2$.
- $conf_1$ **between** $(conf_2, conf_3)$: dès que $conf_2$ est démarrée, $conf_1$ ne doit pas démarrer et $conf_3$, à son tour, ne doit pas être démarrée tant que $conf_2$ n'est pas terminée.

4 Traduction vers Heptagon/BZR

4.1 Vue globale

Le composant est au cœur de la description Ctrl-F et peut être modélisé comme un nœud Heptagon/BZR, comme le montre la Figure 3. Le nœud prend entrée un signal de demande de démarrage (`r`) et un signal de notification de terminaison (`e`), ainsi qu'un ensemble d'événements $\{v_1, \ldots, v_k\}$, ce qui correspond aux types d'événements que le composant en question (`comp`) écoute. En sortie, le nœud produit un ensemble de signaux de demande de démarrage (resp. terminaison) $\{r_1, \ldots, r_m\}$ (resp. $\{e_1, \ldots, e_m\}$) pour chaque configuration $conf_i$ ($i \in [1, m]$), définie dans le composant concerné. En outre, il renvoie également un ensemble de poids $\{w_1, \ldots, w_l\}$, qui correspondent à l'évaluation de chaque attribut du composant.

Le nœud principal (`comp` dans la figure 3) peut contenir un contrat avec un ensemble de variables contrôlables $\{c_1, \ldots, c_q\}$ (dans le cas où il y a un point de choix, e.g., un comportement avec une déclaration *alternative*) et une référence à l'ensemble des politiques énoncées ($\{p_1, \ldots, p_t\}$) afin qu'elles soient appliquées par le contrôleur résultant de la SCD. Les détails sur la façon dont les politiques sont converties sont donnés dans la section 4.3

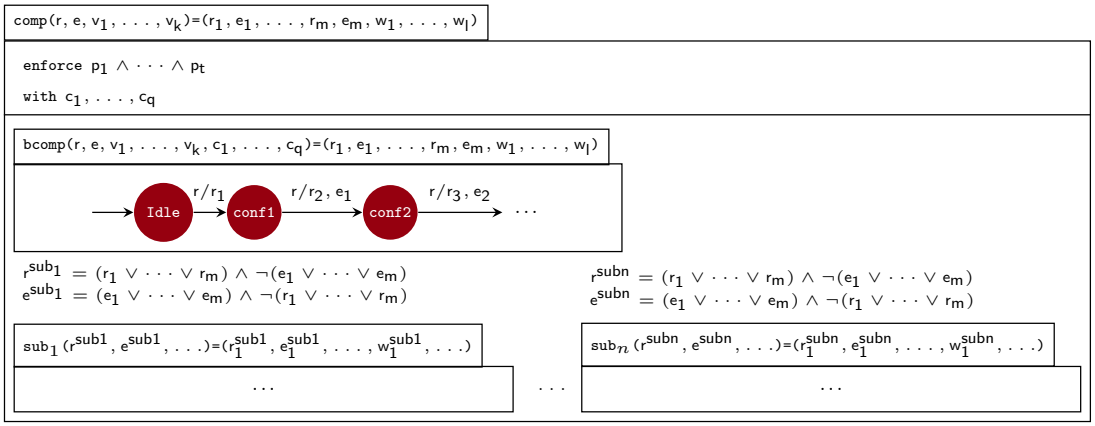


FIGURE 3 – Vue globale de la traduction.

Les comportements de composants sont modélisés comme un sous-nœud (`Bcomp` dans la figure 3), qui est composé d'un automate décrivant l'ordre et les conditions dans lesquelles les configurations se déroulent. A cet effet, il reçoit en entrée les mêmes signaux de demande de démarrage (`r`) et terminaison (`e`) et événements ($\{v_1, \ldots, v_k\}$) que le nœud principal. À la suite de la réaction à ces signaux, il produit les mêmes signaux pour démarrer ($\{r_1, \ldots, r_m\}$) et terminer ($\{e_1, \ldots, e_m\}$) les configurations ainsi que les poids ($\{w_1, \ldots, w_l\}$) correspondant à l'évaluation des attributs dans l'état actuel (configuration) du comportement. Nous fournissons plus de détails sur la traduction des comportements dans la section 4.2.

Enfin, il peut aussi y avoir d'autres sous-nœuds ($\{sub_1, \ldots, sub_n\}$) faisant référence aux composants instanciés dans le composant concerné, à savoir, `comp`. Ceux-ci ont des interfaces et des contenus qui sont structurellement identiques à ceux du nœud principal. Cela veut dire que les sous-nœuds peuvent avoir, à leur tour, un contrat, des sous-nœuds de comportement et un sous-nœud par instance de composant défini à l'intérieur de ce composant. Il faut aussi

noter que les signaux de démarrage ($\mathsf{r}^{\mathsf{subi}}$) et terminaison ($\mathsf{e}^{\mathsf{subi}}$) pour un sous-composant $sub_i \in \{sub_1, \ldots, sub_n\}$ sont définis comme des équations de signaux de demande de démarrage et de terminaison du composant en question. $\{r_1, \ldots, r_m\}$ et $\{e_1, \ldots, e_m\}$ sont respectivement les ensembles de signaux de démarrage et terminaison pour la configuration de $conf_1, \ldots, conf_m$ à laquelle le composant sub_i appartient. Cela signifie qu'un sous-composant sub_i sera demandé si une configuration à laquelle il appartient est également demandé ($r_1 \vee \cdots \vee r_m$) et aucune n'est terminée $\neg(e_1 \vee \cdots \vee e_m)$, ce qui évite l'émission d'un signal de demande de démarrage pour un composant déjà actif ; de même pour le signal de terminaison.

4.2 Comportements

Pour chaque programme en Ctrl-F, nous avons besoin de construire un modèle en AEF, dans Heptagon/BZR, de tous ses comportements possibles. Nous traduisons chaque déclaration de comportement défini dans un autre comportement comme un sous-automate, en décomposant hiérarchiquement le comportement principal, jusqu'aux configurations.

4.2.1 Comportement principal

L'automate modélisant le comportement de l'ensemble consiste en un modèle à deux états, en Figure 4 (a). L'automate est dans l'état *Idle* lorsque le composant ne participe pas à la configuration actuelle. Sur un signal de demande de démarrage (r), il va à l'état *Active*, d'où il retourne à *Idle* sur un signal de fin (e). L'état *Active* comporte une déclaration de comportement, qui est lui-même modélisé comme un sous-automate.

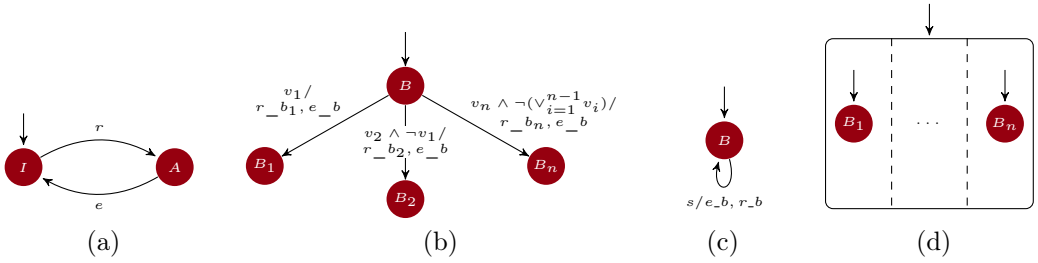
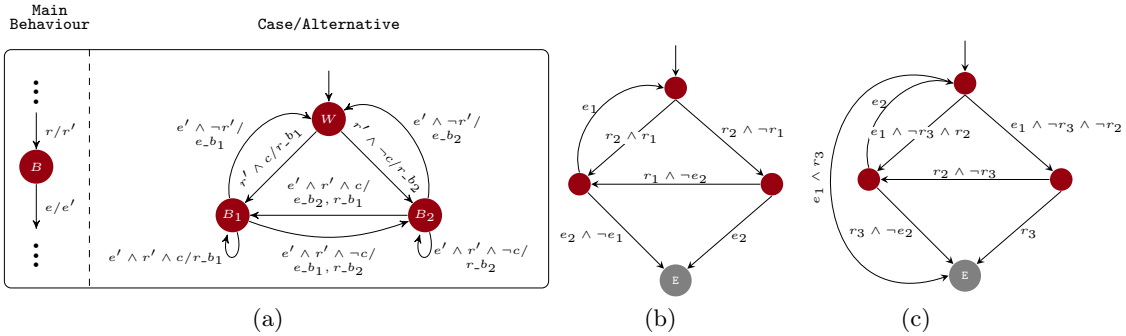


FIGURE 4 – (a) cycle de vie d'un composant ; (b) *When-Do* ; (c) *Every* ; (d) *Parallèle*.

4.2.2 Déclarations

L'automate qui modélise la déclaration *when-do* (Figure 4 (b)) est constitué d'un état initial B correspondant au premier comportement à exécuter. L'automate va à l'état B_i (correspondant à l'exécution du comportement suivant) suite à un signal (événement) v_i tout en produisant des signaux de demande d'initiation du comportement suivant (r_b_i) et de terminaison (e_b) de l'actuel ($1 \leq i \leq n$). Il est important de remarquer que, lorsque deux événements se produisent en même temps, la priorité est donnée en fonction de l'ordre de déclaration des comportements. Par exemple, si v_1 et v_2 déclenchent, respectivement, les comportements B_1 et B_2, B_1 sera donc déclenché s'il avait été déclaré devant B_2.

Les déclarations *case* et *alternative* sont modélisées par l'automate illustré Figure 5. Comme les déclarations de sous-comportement devraient être exécutées au premier instant sur la demande d'initialisation, l'automate doit être composé en parallèle avec l'automate modélisant le



(a) (b) (c)

FIGURE 5 – (a) Composition parallèle du comportement principal et des déclarations *case*/*alternative* ; observateurs modélisant les contraintes temporelles (b) *during* et (c) *between*.

comportement principal (à l'intérieur du nœud `bcomp`, Figure 3). Ainsi, une déclaration *case* ou *alternative* est modélisée comme un simple état du sur-automate dans la hiérarchie qui modélise le comportement principal. Suite à une demande d'initialisation de ces déclarations (le signal r), l'automate principal émet un signal de demande r' qui déclenche une transition de l'état W à l'état suivant (B_1 ou B_2) selon la variable c. Ensuite, il peut aller soit à un autre comportement, au cas où un autre r' est émis et c est vrai ; ou revenir à W si un signal de fin (e') est émis.

Il y a deux différences entre l'utilisation de cet automate pour un *case* ou un *alternative*. Tout d'abord, pour la déclaration *case*, plusieurs (soit plus de deux) branches sont autorisées, donc il pourrait y avoir plusieurs états ($B_1, B_2, \ldots, B_n$) faisant référence à chaque branche ainsi que leur conditions respectives $c_1, c_2, \ldots, c_n$, qui ont été omis ici pour des raisons de lisibilité. Deuxièmement, pour la déclaration *alternative*, les conditions c_i seront considérées comme étant des *variables contrôlables* dans Heptagon/BZR. Ainsi, une SCD doit être effectuée pour garantir que les politiques énoncées ne sont pas violées.

Le modèle de l'automate pour la déclaration *do-every* est illustré Figure 4 (c). Il est composé d'un seul état, ce qui signifie qu'il commence par exécuter directement la déclaration B. Il y a une auto-transition à chaque occurrence du signal s, en émettant les signaux de demande de terminaison (e_b) et d'initialisation (r_b), i.e., le comportement B est ré-exécuté à chaque occurrence de l'événement s. Enfin, la Figure 4 (d) présente le modèle de la déclaration *parallel*, qui correspond tout simplement à la composition parallèle des sous-automates.

4.3 Politiques

Les contraintes sur les valeurs d'attributs correspondent simplement à un ensemble d'équations booléennes dans les nœuds qui modélisent les composants où les politiques sont appliquées. Ensuite, les références aux équations sont utilisés dans le *enforce* d'un contrat afin de les rendre toujours vraies par le contrôle sur les valeurs des variables contrôlables. Bien que la déclaration des objectifs d'optimisation ne soit actuellement pas prise en charge par Heptagon/BZR, on peut modéliser une optimisation en une seule étape directement dans les outils SCD sur lesquels Heptagon/BZR repose [21] [3]. Les détails sur cette approche sont omis ici [12].

Les contraintes temporelles sont modélisées dans Heptagon/BZR par un ensemble d'équations booléennes sur les signaux de demande d'initialisation (r) et terminaison (e) qui sont émis par les automates modélisant les comportements. Pour les contraintes simples comme `conf1`

`succeeds conf2` (resp. `conf1 preceds conf2`), il suffit d'un prédicat $e\_conf2 \Rightarrow r\_conf1$ (resp. $e\_conf1 \Rightarrow r\_conf2$). Cependant, chaque fois qu'il y a besoin de garder la trace de la séquence de signaux (pour demander l'initialisation ou terminaison d'une configuration), l'utilisation des automates observateurs se fait nécessaire. Les automates observateurs sont placés en parallèle avec les automates de comportement, et traduits en Heptagon/BZR au sein du contrat. Le principe est de disposer d'un automate qui observe la séquence de signaux qui conduit à une violation de la politique et détermine que l'état résultant de cette séquence (un état d'erreur) ne devrait jamais être atteint. L'objectif de la SCD sera donc l'invariance de l'ensemble des états privé de ceux où la variable d'erreur est vrai.

La figure 5 (b) illustre un observateur qui modélise la politique *during* (`conf1 durant conf2`), où r_1 et r_2 (resp. e_1 et e_2) correspondent à un signal de demande d'initialisation (resp. terminaison) des configurations `conf1` et `conf2`, respectivement. L'état d'erreur (E) est atteint si `conf2` est terminée avant `conf1` ($e_2 \wedge \neg e_1$) ou si `conf2` est terminé avant que `conf1` ait été initialisé. L'observateur qui modélise la contrainte *between* (`conf1 between (conf2, conf3)`) est représenté dans la figure 5 (c). De même, r_1, r_2 et r_3 (resp. e_1, e_2 et e_3) correspondent aux signaux de demande d'initialisation (resp. terminaison) pour les configurations `conf1, conf2` et `conf3`, respectivement. L'automate passe à l'état d'erreur (E) à chaque fois que la configuration `conf3` a été initialisée (r_3 est émis) après la configuration `conf2` (e_2), sauf lorsque la configuration `conf1` est initialisée et terminée (r_1 et e_1) entretemps.

5 Mise en Œuvre

Comme on peut le voir dans la Figure 6, le processus de compilation peut être divisé en deux parties : (i) la logique de reconfiguration et (ii) le contrôle/vérification de comportement adaptatifs. La logique de reconfiguration est mise en œuvre par le compilateur *ctrlf2fscript*, qui prend en entrée une définition Ctrl-F et génère en sortie un script contenant les procédures permettant de passer d'une configuration à une autre. À cette fin, nous nous appuyons sur des algorithmes de différenciation / match existants pour modèles orientés objet [27].

Le contrôle de comportements est effectué par le compilateur *ctrlf2ept*, qui prend comme entrée une définition Ctrl-F et fournit en sortie un programme réactif Heptagon/BZR. Le résultat de la compilation de Heptagon/BZR est un code séquentiel dans un langage de programmation généraliste (dans notre cas, Java) comprenant deux méthodes : `reset` et `step`. La première initialise l'état interne du programme, tandis que la seconde est exécutés à chaque étape logique pour calculer les valeurs de sortie en fonction d'un vecteur de valeurs d'entrée et l'état actuel.

Ces méthodes sont généralement utilisées en exécutant d'abord `reset` ensuite en appelant `step` dans une boucle infinie, dans laquelle chaque itération correspond à une réaction à un événement. La méthode `step` renvoie un ensemble de signaux correspondant au démarrage ou terminaison des configurations. De ceux-ci, nous pouvons trouver le script approprié qui contient les actions de reconfiguration à exécuter.

Comme illustré Figure 7, nous intégrons la logique de la boucle de contrôle dans trois composants, qui sont contenus dans un composite nommé *Manager*. Le composant *EventHandler* expose un service permettant lui-même d'émettre des événements. La méthode mettant en œuvre ce service est définie comme non-bloquante de sorte que les événements entrants sont stockés dans une file d'attente FIFO. À l'arrivée d'un événement provenant de *Managed System*, le composant *EventHandler* appelle la méthode `step`, mise en œuvre par le composant *ArchitectureAnalyzer*. La sortie de cette méthode est envoyée au composant *Reconfigurator*, qui comprend une méthode pour trouver le bon script de reconfiguration à exécuter.

Dans ce travail, nous nous appuyons sur la plateforme intergicielle FraSCAti [25], une mise

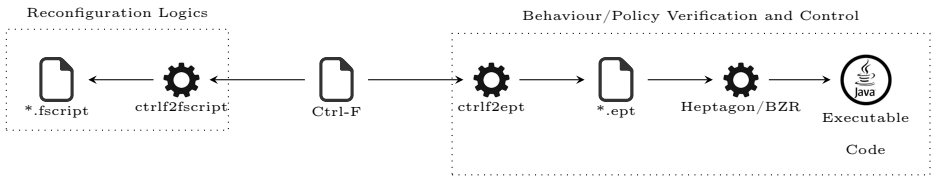


FIGURE 6 – Chaîne de compilation Ctrl-F.

en œuvre en Java du modèle à composants *Service Component Architecture* (SCA). FraSCAti fournit des mécanismes pour les reconfigurations à l'exécution. La plate-forme FraSCAti est elle-même conçue en prenant en compte le modèle SCA, i.e., elle se compose d'un ensemble de composants SCA qui peut être déployé selon les besoins de l'utilisateur. Par exemple, le composant *Manager* instancie le composant *frascati-fscript*, qui fournit des services permettant l'exécution d'une variante de FPath/FScript [8], un langage dédié à l'introspection et la reconfiguration dynamique de composants Fractal. Le composant *frascati-fscript* repose sur d'autres composants intégrant l'intergiciel dans le composite principal (*FraSCAti Composite*), pour effectuer l'introspection et reconfiguration à l'exécution sur les composants du système géré.

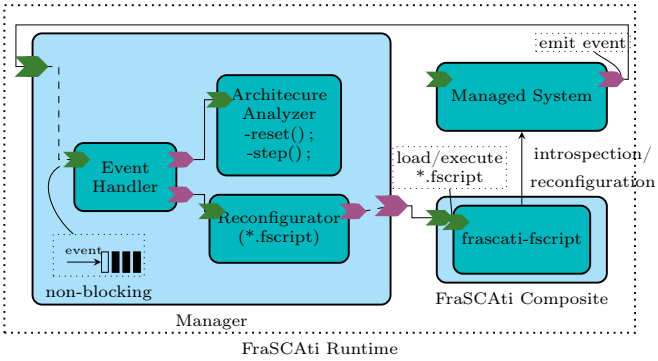


FIGURE 7 – Prototype de gestionnaire englobant la boucle de contrôle.

6 Exemple

L'exemple consiste en une application à six composants (cf. Table 2) ayant chacun comme attributs le coût associé à son exécution ainsi que la qualité de service (QdS) fournie. Seul le composant *B* est capable de s'adapter en définissant un comportement : `do (high | low) every e1`. Donc il passe de *high* à *low* ou vice-versa à chaque occurrence de *e1*. Un composant principal (*Main*) rassemble les instances de ces composants. Pour ce faire, il définit quatre configurations (cf. Table 3), chacune utilisant différentes instances des composants. Le composant *Main*, lui aussi définit un comportement adaptatif : `conf1 when e1 do (conf 2 | conf3 when e1 conf4)`, ainsi que deux politiques : (i) une pour limiter le coût total, c'est à dire la somme des coûts de tous les composants actifs (`cost <= 5`) ; et (ii) une autre pour maximiser la QdS (`maximize qos`).

La Figure 8 montre un scénario d'exécution pour cette application. Le premier graphique montre la valeur actuelle de l'attribut *cost* du composant *Main*, c'est-à-dire le coût total. Les

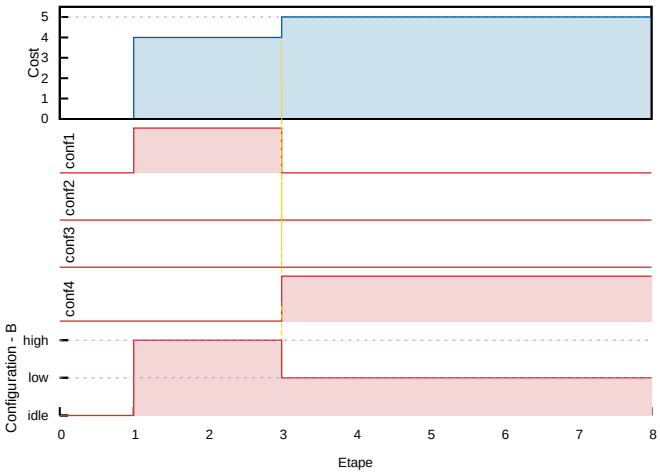


FIGURE 8 – Scénario d'exécution de l'exemple d'application.

Comp.	Conf.	Cost	QoS
A,C	-	1	1
B	high	2	2
	low	1	1
D,E	-	2	2
F	-	3	3

TABLE 2 – Composants, configurations et attributs

	A	B	C	D	E	F
conf1	X	X	X			
conf2	X	X		X		
conf3	X	X				X
conf4	X	X			X	

TABLE 3 – Configurations du composant *Main*.

autres schémas montrent les moments où chaque configuration se fait activer, y compris les configurations (*high* et *low*) de l'instance du composant *B*. Au début, tous les composants sont en repos, à partir de l'étape 1, en suivant le comportement défini dans *Main* et *B*, la configuration *conf1* se fait activer. Lors de l'occurrence de l'événement $e1$ à l'étape 3 (ligne pointillée verticale), nous pouvons observer que la configuration *conf4* se fait activer plutôt que les configurations *conf2* suivie de *conf3*. Ici nous pouvons constater que la reconfiguration se fait contrôler afin de ne pas dépasser la contrainte définie par la politique d'adaptation (`cost <= 5`). Il faut souligner que cette décision est prise en avance, c'est à dire, avant que l'application arrive à l'état violant cette contrainte. Par exemple, même ne violant pas cette contrainte, la configuration *conf2* ne se fait jamais activer car elle peut mener à *conf3*, qui à son tour, a un coût supérieur à 5. Enfin, il faut observer également le changement de configuration du composant *B* (aussi par contrôle), passant de *high* à *low* pour pouvoir respecter la contrainte de coût.

7 Travaux connexes

Classiquement, l'adaptation dynamique pour les architectures logicielles est réalisée d'abord en se fondant sur les ADLs tels que Acme [14] ou Fractal [5], pour une description initiale de la structure et architecture du logiciel, puis en spécifiant les actions de reconfiguration à grain fin avec langages dédiés comme Plastik [2] ou FPath / FScript [8], ou tout simplement en définissant des règles ECA (de l'anglais Event-Condition-Action) pour diriger le système vers l'état souhaité. Une conséquence néfaste de ce type d'approche est que l'espace des états de configurations accessibles est seulement connu comme effet secondaire de ces actions de reconfiguration, ce qui rend difficile de s'assurer que les comportements adaptatifs sont corrects. En outre, un inconvénient des règles ECA est que, contrairement à Ctrl-F, ils ne peuvent pas décrire les séquences de configurations. Les règles ECA, cependant, peuvent être exprimées en Ctrl-F avec un ensemble de déclarations *when-do* (pour la partie E) et *case* (pour les parties C et A) composées en parallèle.

Rainbow [13] est un *framework* autonomique qui est fourni avec Stitch, un langage dédié

permettant la description de l'auto-adaptation des applications définies en Acme. Il dispose d'actions au niveau du système regroupées en *tactiques*, qui à leur tour, sont regroupées au sein d'un chemin arborescent, appelé stratégie. On peut établir une analogie entre la tactique et l'ensemble des actions déclenchées sur un reconfiguration, et entre les stratégies et les comportements dans la langue Ctrl-F. Néanmoins, notre langue fournit un ensemble intéressant de déclarations comportementales y compris les déclarations alternatives et parallèles, ainsi que des comportement dirigés par les événements. Ceux-ci font de Ctrl-F un langage plus expressif. Par ailleurs, le modèle formel de Ctrl-F permet d'assurer des comportements d'adaptation corrects.

Quelques travaux [1] [7] [26] [4] se concentrent sur la façon de planifier un ensemble d'actions qui conduisent en toute sécurité des systèmes à base de composants à une configuration cible. Ces approches sont complémentaires à la nôtre dans le sens où notre accent est mis sur le choix d'une nouvelle configuration et son contrôle. Une fois cette nouvelle configuration choisie, nous nous appuyons sur les mécanismes existants pour déterminer le plan d'action menant effectivement le système de la configuration courante à la prochaine.

Dans [23], les modèles de caractéristiques sont utilisés pour exprimer la variabilité des systèmes logiciels. Lors de l'exécution, un mécanisme de résolution est utilisé pour déterminer les caractéristiques qui devraient être présentes en constituant une configuration. Dans le même sens, Pascual et al. [24] proposent une approche pour la résolution optimale de la variabilité architecturale spécifiée dans le langage commun de variabilité (de l'anglais CVL) [18]. Un inconvénient de ces approches est que, dans la logique d'adaptation spécifiée avec modèles de caractéristiques ou CVL, il est impossible de définir des comportements d'adaptation *stateful*, i.e., des séquences de reconfigurations. La résolution est effectuée en se basant sur l'état actuel et/ou sur des contraintes sur le modèle de caractéristiques. Alors que dans notre approche, dans le modèle réactif basé sur les AEF, les décisions sont prises également en se basant sur l'historique de configurations, ce qui nous permet de définir des comportements et les politiques d'adaptation plus intéressants.

Concernant les méthodes formelles, Kouchnarenko et Weber [20] proposent l'utilisation de logiques temporelles pour intégrer les exigences temporelles aux politiques d'adaptation. Alors que dans cette approche, la vérification est effectuée à l'exécution afin de assurer un comportement correct, nous nous appuyons sur la synthèse de contrôleurs discrets.

Comme dans notre approche, dans [11], les auteurs comptent aussi sur Heptagon/BZR pour modéliser les comportements adaptatifs de composants Fractal. Cependant, il n'y a pas de description de haut niveau (par exemple, un ADL) comme Ctrl-F, et les reconfigurations sont contrôlés au niveau d'actions de reconfiguration de bas niveau, ce qui peut être considéré de coûteux en temps et difficile à passer à l'échelle. Delaval et al. [9] proposent d'avoir des automates modulaires qui peuvent être coordonnés de manière à travailler ensemble de façon cohérente. L'approche est complémentaire à la nôtre dans le sens où ils ne fournissent pas de support langage de haut niveau pour décrire ces gestionnaires, bien que les auteurs fournissent des intuitions intéressantes sur une méthodologie pour le faire.

8 Conclusion

Cet article a présenté Ctrl-F, un langage dédié de haut niveau qui permet la description des comportements et des politiques d'adaptation dans le contexte des applications à base de composants. Une caractéristique unique de Ctrl-F est sa formalisation avec le langage synchrone réactif Heptagon/BZR, qui permet de bénéficier, entre autres, d'outils formels pour la vérification, le contrôle et la génération automatique de code exécutable. Nous avons intégré Ctrl-F

avec Frascati, un intergiciel pour le modèle à composant *Service Component Architecture*.

Pour les travaux futurs, nous avons l'intention d'aborder les questions de la modularité et de la coordination des contrôleurs, ainsi que leur répartition physique. Le language et les modèles réactifs sur lequel nous nous appuyons ont des résults récents qui peuvent être exploités pour déployer les contrôleurs en tenant compte des architectures physiques des systèmes.

Références

[1] Naveed Arshad and Dennis Heimbigner. A Comparison of Planning Based Models for Component Reconfiguration. Research Report CU-CS-995-05, U. Colorado, September 2005.

[2] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *Proceedings of the 2Nd European Conference on Software Architecture*, EWSA'05, pages 1–17, Berlin, Heidelberg, 2005. Springer-Verlag.

[3] N. Berthier and H. Marchand. Discrete controller synthesis for infinite state systems with reax. In *IEEE International Workshop on Discrete Event Systems*, pages 46–53, Cachan, France, May 2014.

[4] Fabienne Boyer, Olivier Gruber, and Damien Pous. Robust reconfigurations of component assemblies. In *Proc. 2013 Int. Conf. on Software Engineering*, ICSE '13, pages 13–22, 2013.

[5] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2003)*, Edinburgh, Scotland, May 2004.

[6] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 173–182, New York, NY, USA, 2005. ACM.

[7] Carlos Eduardo da Silva and Rogério de Lemos. Dynamic plans for integration testing of self-adaptive software systems. In *Proc. 6th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 148–157, 2011.

[8] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. FPath & FScript : Language support for navigation and reliable reconfiguration of Fractal architectures. *Annals of Telecommunications : Special Issue on Software Components – The Fractal Initiative*, 2008.

[9] Gwenaël Delaval, Soguy Mak-Karé Gueye, Eric Rutten, and Noël De Palma. Modular coordination of multiple autonomic managers. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pages 3–12, New York, NY, USA, 2014. ACM.

[10] Gwenaël Delaval, Hervé Marchand, and Éric Rutten. Contracts for modular discrete controller synthesis. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010)*, Stockholm, Sweden, April 2010.

[11] Gwenaël Delaval and Éric Rutten. Reactive model-based control of reconfiguration in the fractal component-based model. In *13th International Symposium on Component Based Software Engineering (CBSE 2010)*, Prague, Czech Republic, June 2010.

[12] Emil Dumitrescu, Alain Girault, Hervé Marchand, and Éric Rutten. Multicriteria optimal reconfiguration of fault-tolerant real-time tasks. In *Workshop on Discrete Event Systems, WODES'10*, pages 366–373, Berlin, Germany, August 2010. IFAC.

[13] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10) :46–54, October 2004.

[14] David Garlan, Robert T. Monroe, and David Wile. Acme : Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[15] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Springer-Verlag, Berlin, Heidelberg, 2010.

[16] D. Harel and A. Pnueli. Logics and models of concurrent systems. chapter On the Development of Reactive Systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[17] David Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, June 1987.

[18] Oystein Haugen, Andrzej Wasowski, and Krzysztof Czarnecki. Cvl : Common variability language. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 277–277, New York, NY, USA, 2013. ACM.

[19] I. Jacobson, M. Griss, and P. Jonsson. *Software reuse : architecture process and organization for business success*. ACM Press books. ACM Press, 1997.

[20] Olga Kouchnarenko and Jean-François Weber. Adapting component-based systems at runtime via policies with temporal patterns. In José Luiz Fiadeiro, Zhiming Liu, and Jinyun Xue, editors, *FACS 2013, 10th Int. Symposium on Formal Aspects of Component Software, Revised Selected Papers*, volume 8348 of *LNCS*, pages 234–253, Nanchang, China, 2014. Springer. Revised Selected Papers.

[21] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System : Theory and Applications*, 10(4) :325–346, October 2000.

[22] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1) :70–93, January 2000.

[23] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.

[24] Gustavo G. Pascual, Mónica Pinto, and Lidia Fuentes. Run-time support to manage architectural variability specified with cvl. In *Proceedings of the 7th European Conference on Software Architecture*, ECSA'13, pages 282–298, Berlin, Heidelberg, 2013. Springer-Verlag.

[25] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software : Practice and Experience*, 42(5) :559–583, 2012.

[26] Matthias Tichy and Benjamin Klöpper. Planning self-adaption with graph transformations. In *Proc. 4th Int. Conf. on Applications of Graph Transformations with Industrial Relevance*, AGTIVE'11, pages 137–152, 2012.

[27] Zhenchang Xing and Eleni Stroulia. Umldiff : An algorithm for object-oriented design differencing. In *Proc. 20th IEEE/ACM Int. Conf. on Automated Software Engineering*, ASE '05, pages 54–65, 2005.