THÈSE

présentée par

Olivier RATCLIFFE

pour obtenir le diplôme de
DOCTEUR DE L'UNIVERSITÉ DE SAVOIE
(Arrêté ministériel du 30 mars 1992)

Spécialité : Informatique

Approche et environnement fondés sur les styles architecturaux pour le développement de logiciels propres à des domaines spécifiques

Application au domaine de la supervision du redémarrage d'accélérateurs de particules

---

Soutenue publiquement le 16 décembre 2005 devant le jury composé de :

Henri BASSON	Président du jury, Rapporteur	Professeur à l'Université du Littoral Côte d'Opale, Calais
Richard MCCLATCHEY	Rapporteur	Professeur à l'Université de West of England, Bristol, Royaume-Uni
Mario BATZ	Examinateur	Ingénieur – Responsable de la salle de contrôle technique au CERN, Genève, Suisse
Sorana CIMPAN	Examinateur	Maître de Conférences à l'Université de Savoie, Annecy
Flavio OQUENDO	Directeur de thèse	Professeur à l'Université de Savoie, Annecy
Luigi SCIBILE	Co-encadrant	Docteur – Responsable de l'ingénierie des systèmes de contrôle au CERN, Genève, Suisse

Préparée au sein du CERN
Organisation Européenne pour la Recherche Nucléaire, Genève
et du LISTIC
Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance
ESIA – Université de Savoie

Remerciements

Les travaux présentés dans ce mémoire ont été rendus possibles par la collaboration entre l'Organisation Européenne pour la Recherche Nucléaire (CERN) et le Laboratoire de Logiciels pour la Productique (LLP) qui est devenu par la suite le Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance (LISTIC).

Je témoigne ma sincère gratitude à M. Flavio OQUENDO, qui a dirigé les travaux de cette thèse, à M. Luigi SCIBILE, qui l'a secondé assidûment dans cette tâche, ainsi qu'à Mme. Sorana CIMPAN qui a largement participé à mon encadrement et qui m'a apporté une aide précieuse, à la fois par ses conseils sur le plan scientifique et par ses témoignages d'amitié. L'intérêt qu'ils ont porté à mes travaux et l'aide qu'ils ont apportée à mes recherches m'ont été d'un très grand soutien.

Je remercie également M. Mario BATZ, responsable de la salle de contrôle technique du CERN, pour m'avoir intégré dans son équipe, ainsi que pour m'avoir permis de valider mes travaux de recherche dans le cadre du projet qu'il dirigeait. Ses conseils m'ont été d'une aide précieuse pour la conduite de l'étude.

Je remercie vivement M. Henri BASSON, Professeur à l'Université du Littoral Côte d'Opale, et M. Richard MCCLATCHEY, Professeur à l'Université de West of England, pour avoir accepté la charge d'être rapporteurs de thèse. Je remercie également :

- M. Mario BATZ, Responsable de la salle de contrôle technique au CERN ;
- Mme. Sorana CIMPAN, Maître de Conférences à l'Université de Savoie ;
- M. Flavio OQUENDO, Professeur à l'Université de Savoie ;
- M. Luigi SCIBILE, Responsable de l'ingénierie des systèmes de contrôle au CERN,

pour l'honneur qu'ils m'ont témoigné en participant à ce jury, ainsi pour avoir consacré du temps à la lecture de ce travail.

Je remercie M. Pierre NININ, chef du groupe Monitoring and Access (ST/MA) du CERN, ainsi que M. Thomas PETTERSSON, chef du groupe Control, Safety & Engineering Databases (TS/CSE), pour la confiance qu'ils m'ont témoignée en m'accueillant dans leurs groupes. Je tiens également à exprimer ma profonde gratitude à M. Alain HAURAT qui m'a accueilli au LLP alors qu'il en était le Directeur, ainsi que M. Philippe BOLON, son successeur au sein du LISTIC.

J'adresse mes remerciements à tous les membres du LISTIC et du groupe TS/CSE, ainsi qu'aux membres des salles de contrôle du CERN, qui m'ont aidé à obtenir les informations techniques indispensables pour la conduite de ce travail. Je remercie également toutes les personnes impliquées dans la conception et le développement des systèmes de supervision du CERN, notamment M. Peter SOLLANDER, pour avoir collaboré à l'intégration et à la validation des outils développés dans le cadre de ma thèse.

Enfin j'exprime ici ma reconnaissance à mes parents pour leurs encouragements et surtout à mon épouse pour m'avoir soutenu avec patience et amour tout au long de ce parcours.

Abstract

Software development techniques were, at first, dedicated to the design of single applications, satisfying specific requirements. Today it is necessary, for cost and “time to market” reasons, to define and implement a set of methods allowing the development of families of software that share common characteristics.

The issue considered in this thesis concerns the definition of a domain-specific development model, as well as its exploitation and evolution in a dedicated software environment. The research philosophy chosen to reach this goal was the use of architectural development techniques including the definition of architectural styles. An architectural style allows the specification of the common characteristics of software families, and the production of applications satisfying the properties defined at the style level.

Concerning existing works, the classical process used to define and to exploit the architectural styles assumes that the application domain expertise is complete and that the style can be directly and entirely defined and can be used to produce applications that satisfy clearly established requirements. However, in most cases, the domain expertise is available (e.g. prototype applications) but incomplete, and the user requirements are not static, but they are expected to evolve frequently. On the subject of the definition and the formalisation of architectural styles, many techniques and languages are available. However, even if some techniques allow the use of styles in the parameterisation of generic development environments in order to specialise them for domain-specific development, there is no single approach allowing the production development environment from styles.

In this context, this thesis defines:

- an inductive process that allows the definition of an architectural style from prototype applications, and the evolution of the style according to the evolution of the requirements concerning the applications constructed from this style;
- an environment dedicated to the development of domain-specific software that satisfies the constraints of an architectural style;
- a new monitoring software design and production approach (the application domain of this thesis), based on the definition and the use of architectural styles.

The approaches and processes proposed in this thesis have been validated in the implementation of a development environment, SEAM (Software for the Engineering of Accelerator Monitoring), which guides and enables the optimisation of particle accelerator monitoring software production.

Keywords: Software Architectures, Architectural Styles, Domain-Specific Software Architectures, Software Evolution and Monitoring Software.

Table des matières

Chapitre I Introduction

Chapitre I : Introduction

I. 1 Introduction à la problématique

Un des principaux objectifs dans le domaine industriel est l'optimisation du temps de fonctionnement des processus de production. Une supervision efficace des processus constitue une partie de la solution pour augmenter ce temps de fonctionnement. En effet, si le système de supervision permet aux opérateurs des salles de contrôle de déterminer précisément la cause d'une panne du processus, ceux-ci sont capables de résoudre le problème rapidement, le temps de redémarrage s'en trouve alors réduit, et le temps d'opération optimisé. Cependant, la supervision de processus complexes implique l'acquisition, la gestion, et la visualisation d'une grande quantité d'informations. Le développement de logiciels de supervision efficaces est donc une question critique.

Dans le cas particulier des accélérateurs de particules, la complexité des systèmes à superviser est très élevée. Les informations de supervision sont acquises et gérées par de nombreux systèmes différents, et exploitées par des opérateurs de plusieurs salles de contrôle ayant des rôles complémentaires. Il est donc nécessaire de fournir des logiciels de haut niveau permettant la supervision de l'ensemble des processus d'opération des accélérateurs. Ces logiciels de supervision, ainsi que la documentation associée, doivent constituer un support pour coordonner efficacement le travail et les compétences des opérateurs, et ainsi optimiser le temps d'opération des accélérateurs.

Dans ce contexte, la salle de contrôle technique (TCR : Technical Control Room) du CERN (Conseil Européen pour la Recherche Nucléaire) a implémenté un ensemble de prototypes de logiciels de supervision devant servir de base pour les développements futurs. Le premier objectif était d'utiliser l'expérience acquise lors de l'implémentation et l'utilisation de ces prototypes, pour définir un modèle de développement de familles de logiciels spécifiant des règles précises que celles-ci doivent respecter pour être facilement exploitables par les opérateurs des salles de contrôle. Le second objectif est d'utiliser ce modèle pour guider et automatiser l'implémentation de familles de logiciels de supervision uniformisés. Le troisième objectif est de faire évoluer le modèle en fonction de l'évolution des besoins propres au domaine d'application.

Dans ce contexte, même si les outils de développement d'applications de supervision disponibles sur le marché proposent un grand nombre de fonctionnalités intéressantes, ceux-ci ne permettent ni d'automatiser entièrement le processus de développement, ni de spécifier les propriétés complexes que les logiciels doivent satisfaire. Par ailleurs, ils ne peuvent pas être entièrement adaptés à un domaine d'application particulier tel que celui traité dans cette thèse.

La problématique consistant à définir, utiliser, respecter et faire évoluer un modèle de développement de familles d'applications logicielles, n'est pas propre au domaine des logiciels de supervision, il se rapporte au contraire à l'ensemble des secteurs d'activités de l'ingénierie logicielle. En effet, même si dans un premier temps les applications logicielles étaient conçues une par une, pour répondre à des besoins spécifiques, il est aujourd'hui nécessaire, pour des raisons de coût et de temps de production, de définir et de mettre en œuvre des méthodes et des outils supportant le développement de familles de logiciels ayant des caractéristiques communes.

Une approche en plein essor traite efficacement de cette problématique, il s'agit des architectures logicielles. Cette approche permet de spécifier formellement des modèles de développement, ou styles architecturaux, et de les exploiter pour produire des applications logicielles. L'utilisation d'un style architectural permet de garantir la satisfaction de besoins spécifiques à un domaine particulier, et automatise le processus de développement des applications. Toutefois, l'utilisation d'un style architectural dans l'implémentation d'un environnement de développement reste une question ouverte. Par ailleurs, le processus de développement centré-architecture habituellement utilisé est essentiellement déductif : il met en avant l'utilisation du style pour développer de nouvelles applications, et non de sa définition inductive à partir d'applications existantes servant de référence (prototypes de logiciels de supervision dans le contexte de cette thèse). De plus, les travaux menés jusqu'à maintenant dans le domaine des architectures logicielles ne proposent pas de support permettant de faire évoluer les styles architecturaux en fonction de l'évolution des besoins des utilisateurs des applications produites à partir de celui-ci.

Ainsi, les axes de recherche traités dans cette thèse sont les suivants :

- proposer une nouvelle approche de conception et de production de logiciels de supervision basée sur l'utilisation et l'exploitation des styles architecturaux ;
- proposer un processus inductif permettant la définition de style architectural à partir d'applications prototypes, et l'évolution du style en fonction de l'évolution des besoins concernant les applications construites à partir de celui-ci ;
- implémenter un environnement dédié au développement de logiciels propres à des domaines spécifiques respectant les contraintes du style architectural.

Les travaux présentés dans cette thèse sont validés par la mise en place d'un outil, nommé SEAM (Software for the Engineering of Accelerator Monitoring), permettant de produire des familles de logiciels spécifiques à la supervision du redémarrage des accélérateurs de particules. Cet outil exploite un style architectural défini de façon inductive à partir des prototypes de logiciels de supervision préalablement implémentés, et formalisé au moyen des langages développés dans le cadre du projet ArchWare. SEAM permet ainsi de guider et d'automatiser la production de familles de logiciels uniformisés respectant un ensemble de règles de développement prédéfinies, et satisfaisant ainsi les besoins des utilisateurs. De plus, le guide de développement proposé par SEAM peut évoluer en fonction des modifications des besoins et/ou du processus à superviser.

I. 2 Cadre de la thèse

Le travail présenté dans cette thèse a été réalisé dans le cadre du projet GTPM (Gestion Technique de Pannes Majeures) au CERN. Ce projet a pour but d'optimiser le temps de fonctionnement des accélérateurs de particules du CERN en réduisant leur temps de redémarrage après des pannes majeures. Pour atteindre cet objectif, le CERN a décidé d'implémenter un outil logiciel de support au développement d'applications de supervision respectant les règles définies par le projet GTPM, et les utilisant pour guider ses utilisateurs dans la production d'applications uniformisées. Cet outil, SEAM, a été développé dans le contexte de la thèse et valide les approches proposées par celle-ci.

Par ailleurs, cet outil s'intègre au système de contrôle et de supervision développé dans le cadre du projet TIM (Technical Infrastructure Monitoring) réalisé au CERN parallèlement au projet GTPM. Ce projet a abouti à la mise en place d'un système d'acquisition, de traitement, de stockage et de représentation des données relatives à l'état des accélérateurs de particules et de leur infrastructure technique. L'outil réalisé dans le cadre de la thèse a donc dû être développé de façon à produire des logiciels de supervision utilisables par le système TIM.

De plus, cette thèse utilise les résultats du projet européen ArchWare (Architecting Evolvable Software). Ce projet a pour objectif principal de répondre à une demande croissante de systèmes logiciels évolutifs. Pour atteindre ce but, ArchWare a développé un ensemble intégré de langages et d'outils orientés architecture. Le travail effectué au cours de cette thèse a consisté à utiliser les langages ArchWare, ainsi qu'à étudier et adapter le processus de développement centré architecture de façon à formaliser le guide de développement GTPM et à l'intégrer dans l'outil de développement d'applications de supervision.

I. 3 Organisation du document

Le document est organisé en trois parties. La première, incluant les deux chapitres suivant cette introduction, porte sur la problématique abordée dans la thèse ainsi que sur l'état de l'art.

La deuxième partie présente la solution proposée : la définition d'un style architectural propre au domaine du développement de logiciels de supervision d'accélérateurs de particules, et son utilisation pour contrôler l'environnement de développement de ces logiciels.

La troisième partie contient l'implémentation et la validation de la solution proposée. Elle comporte le sixième chapitre qui présente l'implémentation de l'outil SEAM intégrant le style préalablement formalisé, ainsi que le septième qui traite la question de l'évolution des logiciels de supervision développés via cet outil ainsi que de l'évolution de l'outil lui-même. En outre, le chapitre huit présente des études de cas permettant de valider le travail effectué dans le cadre de cette thèse.

La conclusion propose un bilan du travail effectué durant la thèse et un ensemble de perspectives ouvertes par celui-ci.

Voici le contenu plus détaillé des chapitres présentés :

- Le deuxième chapitre introduit la problématique de la disponibilité des accélérateurs de particules. Il explique comment la supervision efficace d'un processus, au moyen d'applications appropriées, peut permettre d'optimiser son temps d'opération. Le chapitre présente le projet GTPM ainsi que les premiers logiciels de supervision qui ont été développés lors d'une phase de prototypage. Enfin, il expose les besoins concernant l'outil de support à la construction de logiciels de supervision, dont la conception et le développement font l'objet de la suite de la thèse ;
- Le troisième chapitre présente les méthodes et les outils utilisés pour le développement d'applications de supervision. En outre, ce chapitre expose les travaux qui se sont intéressés au développement de familles d'applications logicielles possédant des caractéristiques communes, et ce notamment dans le domaine des architectures logicielles ;
- La solution proposée par cette thèse est exposée dans le quatrième chapitre. Celui-ci présente l'approche d'intégration des techniques centrées architecture dans l'environnement de développement de logiciels de supervision, ainsi que le modèle informel qui est le fondement de cet environnement ;
- Le cinquième chapitre présente tout d'abord les différents langages et outils développés dans le cadre du projet ArchWare. Il décrit ensuite la formalisation du style architectural introduit dans le chapitre quatre, ainsi que des exemples d'architectures de logiciels de supervision ;
- L'implémentation de la solution proposée par cette thèse est présentée dans le sixième chapitre. Celui-ci expose la stratégie choisie pour intégrer le style GTPM dans l'environnement de développement SEAM, ainsi que les détails concernant l'implémentation des différents modules de cet environnement ;
- Le septième chapitre explique comment les logiciels de supervision peuvent être mis à jour en fonction de l'évolution des besoins. En outre, il présente une approche permettant de superviser l'évolution des logiciels de supervision, et d'exploiter le résultat de cette supervision en mettant à jour le style et l'environnement de développement associé ;
- Le huitième chapitre évalue la solution proposée par cette thèse au moyen de trois études de cas. La première a pour but de déterminer l'intérêt d'utiliser un environnement de développement spécifique au domaine d'application, la seconde met en évidence l'influence de l'approche centrée style dans ce contexte, et la troisième étudie l'utilité de fournir un support pour l'évolution des architectures et des styles, et comment celui-ci permet de s'assurer que l'environnement de développement s'adapte aux attentes des utilisateurs ;
- La conclusion propose une synthèse et un bilan du travail effectué durant la thèse et un ensemble de perspectives liées à la continuation du travail, aux nouvelles applications, ainsi qu'aux nouveaux thèmes de recherche ;
- L'Annexe 1 présente les phases de développement des prototypes de logiciels de supervision ;
- L'Annexe 2 fournit la formalisation du style GTPM ;
- L'Annexe 3 fournit le schéma complet de la base de données de l'outil SEAM.

Chapitre II Problématique de la supervision des accélérateurs de particules

Chapitre II : Problématique de la supervision des accélérateurs de particules

II. 1 Problématique de la supervision des processus

II. 1. 1 Introduction

Dans le contexte de la production industrielle, peu de choses sont plus critiques qu'un processus étant indisponible lorsqu'il existe un besoin urgent de satisfaire des demandes de clients. Pour respecter les contraintes de productivité et de rentabilité des entreprises, les processus industriels doivent être fiables. En effet, dans la majorité des secteurs d'activité il est nécessaire d'être capable de produire en permanence, et les pertes financières dues aux pannes des systèmes et à leur temps d'arrêt sont considérables.

De nombreuses études (ex : [Bell 2002], [DowntimeCentral]) ont estimé ces pertes et ont proposé des solutions pour les réduire. Par exemple, un projet ayant pour but de déterminer le coût des pannes dans le domaine du chemin de fer (Swedish Rail Administration) [Karl-Olof, 1998] a dénombré plus de cinq cent pannes ayant généré une perte de près d'un million d'euros, soit 7,7% du coût total de leur production. D'un point de vue plus général, une étude menée dans le cadre d'un projet de réduction de coûts dans le domaine de la production alimentaire [Philips et al. 1997] a mis en évidence que les pertes dues à une mauvaise opération et maintenance des processus des industries américaines s'élèvent à des milliards de dollars chaque année. La réduction des pertes résultant de périodes d'indisponibilité des processus est donc une problématique industrielle générale. Dans l'objectif d'optimiser le temps de disponibilité, il existe deux solutions principales et complémentaires : réduire la *fréquence* des pannes, et réduire leur *durée* (et donc leur coût). Ainsi, [Spencer et Rhee 2003] ont adopté une solution consistant, premièrement, à mettre en redondance les systèmes critiques du processus de production qu'ils étudiaient pour réduire la *fréquence* de ses arrêts, et, deuxièmement, à réduire le temps nécessaire à la maintenance corrective de ces systèmes pour réduire la *durée* de ces arrêts. Toujours dans l'objectif de réduire les pertes dues à l'arrêt de la production, des travaux [Philips et al. 1997] ont permis d'établir une liste de cinq domaines à surveiller pour optimiser le rendement des processus : l'opération, la maintenance, l'ingénierie, la formation et l'administration. Pour améliorer la sécurité, la fiabilité et l'efficacité d'un processus, il est nécessaire de mettre en place un programme de formation adéquate et une suite d'outils fournissant :

- une méthode pour comprendre quand et comment appliquer une analyse des causes de problèmes ;
- une stratégie d'amélioration des processus ;
- une compréhension claire de l'opération du processus et de son infrastructure ;
- une technologie de maintenance.

Ces deux derniers points sont indispensables à la réduction de la *durée* des pannes. En effet, pour améliorer le temps de redémarrage d'un processus il est nécessaire de mettre en place une maintenance curative efficace. Cela n'est possible qu'en ayant entre autres une excellente connaissance du processus qui permet de déterminer quelle action entreprendre à quel moment. Les processus complexes sont souvent supervisés depuis des salles de contrôle, et leur opération nécessite une grande efficacité des opérateurs [IFE]. En effet, même si ceux-ci supervisent plus des automatismes qu'ils n'exercent de contrôle manuel, leur tâche reste une tâche de contrôle. Les opérateurs de salles de contrôle doivent donc être bien formés et avoir des outils leur permettant de connaître précisément et à tout moment l'état du processus, en particulier la cause de la panne (sous-processus/système/équipement défaillant) de façon à pouvoir intervenir plus vite et plus efficacement, dans le sens de prendre des bonnes décisions et de mettre en œuvre une séquence optimisée de rétablissement du processus. Cette considération met en évidence le fait que la minimisation du temps d'indisponibilité d'un processus, et donc l'optimisation du temps de son fonctionnement, n'est possible qu'à condition de mettre en place des interfaces graphiques (IHMs, ou vues) de supervision fiables, qui permettent à la fois de prévenir les dysfonctionnements, et d'optimiser le redémarrage lorsqu'ils surviennent. La section suivante exposera les critères permettant d'aboutir à ce résultat.

II. 1. 2 Conception d'interfaces graphiques de supervision

Ces dernières années, les développements effectués dans le domaine de la conception des IHMs ont fait partie des travaux les plus intéressants concernant les systèmes de supervision. Le fonctionnement optimal d'un processus industriel passe par la réduction de la possibilité d'erreurs humaines dues à la mauvaise interprétation des IHMs de supervision de ce processus. Il est donc crucial que les IHMs de supervision soient fiables et bien conçues. Lorsque ce n'est pas le cas, la responsabilité d'une erreur d'interprétation ne revient pas uniquement aux opérateurs mais aussi aux concepteurs du système de supervision [Jambon 1997]. La qualité et l'exploitabilité des IHMs de supervision sont obtenues par l'homogénéisation des représentations graphiques, par l'affichage d'informations pertinentes, par la capture du savoir-faire du domaine d'application, ainsi que par l'association d'une documentation ciblée et facilement accessible.

II. 1. 2. 1 Homogénéisation et standardisation

Les processus complexes sont couramment supervisés par de multiples systèmes hétérogènes. Cette hétérogénéité est rencontrée au niveau des applications d'acquisition et de transmission de données, mais aussi à celui des applications de représentation des informations. Pour connaître l'état global d'un processus il est souvent nécessaire de consulter plusieurs IHMs de supervision affichées par des applications différentes. Cette multiplicité des systèmes augmente le risque d'erreur d'interprétation de la part des opérateurs de salles de contrôle. En effet, les conventions graphiques peuvent largement varier selon les applications, par exemple : deux systèmes ou équipements identiques peuvent avoir des représentations différentes ; inversement, deux objets graphiques semblables peuvent représenter des systèmes ou équipements différents ; les couleurs peuvent avoir des significations différentes. L'origine géographique de l'application peut

en outre influencer les standards utilisés. Par exemple, un interrupteur qui est en position haute est « on » aux Etats-Unis, mais « off » en Europe, tandis que les interrupteurs japonais s'actionnent horizontalement, ayant le « on » à droite et le « off » à gauche [Moray 2002]. Ce constat souligne le besoin d'homogénéiser les applications de supervision utilisées dans une salle de contrôle, et de standardiser les vues. Si toutes les IHMs utilisées dans une salle de contrôle suivent un standard bien défini et parfaitement connu des opérateurs, elles seront plus facilement interprétables, et la probabilité d'erreurs humaines sera donc réduite. Il est donc nécessaire de définir clairement les propriétés graphiques et fonctionnelles que doivent satisfaire toutes les IHM utilisées dans une salle de contrôle.

II. 1. 2. 2 Sélection des informations

Un autre facteur intervenant dans la qualité et l'exploitabilité des interfaces de supervision est la pertinence des informations affichées. Dans le domaine de la supervision des processus, une importante distinction doit être faite entre les données et l'information [Moray 2002]. Les données ne sont que des nombres, alors que l'information est une combinaison de ces nombres ayant une signification précise permettant d'indiquer l'état d'un système avec un minimum de calcul mental et de manipulation de données. Il est nécessaire de faire une distinction entre les valeurs qui sont indiquées explicitement et celles qui doivent être déduites ou calculées par l'opérateur pour identifier l'état du processus. Les problèmes peuvent surgir lorsque l'état du processus ne peut être obtenu que par interprétation de nombreuses valeurs. D'un autre côté, si les informations affichées sont de haut niveau, calculées à partir de valeurs non affichées, il peut être très difficile pour un opérateur de déterminer précisément d'où vient une éventuelle faute. Il s'agit alors de trouver un compromis concernant le niveau de détail des informations affichées. Souhaite-t'on privilégier la vision globale du processus ou la vision détaillée des systèmes intervenant dans ce processus ? Lorsque l'objectif est de fournir une interface de supervision permettant de connaître à tout moment l'état global du processus il est nécessaire de représenter, dans un premier lieu, des informations de haut niveau. Ceci n'empêche pas de fournir des fonctionnalités permettant d'obtenir à la demande les valeurs propres à chaque système intervenant dans le processus.

Dans le cas de processus industriels complexes le nombre de systèmes et d'équipements à superviser est très important. Chacun d'eux rend disponible de nombreuses valeurs indiquant leur état. Pour superviser l'état global du processus il est nécessaire de sélectionner uniquement les valeurs indispensables à l'obtention d'informations pertinentes. Un réseau d'alimentation électrique, par exemple, est constitué de nombreux équipements (interrupteurs, disjoncteurs, transformateurs,…) qui fournissent chacun plusieurs informations concernant leur état. Toutefois, pour connaître l'état de ce réseau, il n'est pas nécessairement indispensable de prendre en compte toutes ces données, une simple mesure de tension peut suffire. Parmi toutes les données disponibles il ne faudrait alors sélectionner que cette mesure. Par contre, si une telle mesure n'est pas rendue disponible par les équipements existants, il sera nécessaire d'effectuer un travail de synthèse sur les données disponibles de façon à obtenir une information résultante calculée à partir de celles-ci.

II. 1. 2. 3 Capture de l'évolution des besoins

Afin de développer des IHMs fiables et exploitables il est indispensable de connaître en détail le processus supervisé. C'est ici qu'intervient le problème de la communication entre les spécialistes des processus, les opérateurs de salle de contrôle, et les développeurs d'IHMs. L'opération des processus et le développement d'IHMs sont deux domaines tout à fait distincts. Toutefois il n'est pas possible aux développeurs de fournir des IHMs représentant correctement un processus sans le connaître au moins globalement. De même, il n'est pas possible aux opérateurs de formuler des besoins exploitables sans connaître un minimum le domaine du développement d'IHMs. Les cycles de développement itératifs, comme l' « eXtreme Programming » (XP) [Beck 1999] peuvent faciliter la communication entre les développeurs et les clients (opérateurs) en impliquant davantage ceux-ci dans le processus de définition des IHMs. En effet, les pratiques telles que « Client sur site » et « Livraisons fréquentes » donnent de nombreuses occasions au client de donner du retour (feedback) sur l'interface, et la démarche itérative d'XP autorise réellement l'amélioration progressive de celle-ci tout au long du projet. Une telle approche peut donc permettre de développer des IHMs représentant correctement le processus au moment de leur mise en service. Toutefois, les processus industriels sont sujets à des modifications, parfois simplement au niveau de leurs équipements, mais parfois aussi à celui de leur fonctionnement global. Les besoins concernant les IHMs peuvent donc évoluer au cours de la durée de vie du processus. La problématique devient donc de capturer l'évolution des besoins pour mettre à jour les IHMs. Le problème majeur est souvent que l'équipe de développement n'est plus disponible pour mettre à jour les IHMs quelques mois ou quelques années après leur première mise en place. En conséquence, il est parfois nécessaire de faire appel à une autre équipe de développement pour implémenter de nouvelles IHMs, ce qui est une démarche coûteuse. Une solution alternative est de disposer d'IHMs pouvant évoluer en fonction des modifications du processus supervisé, soit automatiquement, soit à la demande des utilisateurs (opérateurs), mais sans nécessiter d'interventions de développeurs.

II. 1. 2. 4 Documentation

Pour que des IHMs soient exploitables, il est préférable d'éviter qu'elles soient visuellement surchargées, et donc, de représenter uniquement les informations fournissant l'état du processus en temps réel. Toutefois, lors d'une défaillance sur un système ou équipement particulier, il peut être utile à l'opérateur de salle de contrôle d'accéder à des informations statiques fournissant plus de détails sur celui-ci. C'est pourquoi il est souvent intéressant d'associer aux IHMs représentant l'état d'un processus une documentation pertinente et exploitable concernant les éléments de ce processus. L'IHM doit être conçue pour permettre une navigation intuitive vers les pages de documentation et de consignes/instructions de façon à fournir rapidement et précisément à l'opérateur les informations dont il a besoin pour déclencher une opération de redémarrage ou de maintenance sur un système ou équipement indisponible.

II. 1. 3 Les accélérateurs de particules au CERN

La problématique particulière des accélérateurs de particules est similaire à la problématique générale de la production industrielle. L'objectif des intervenants responsables des différents processus est de maximiser la disponibilité globale des accélérateurs, de façon à produire en permanence les faisceaux de particules nécessaires aux expériences menées par les utilisateurs physiciens (ou clients). Cet objectif ne peut-être atteint qu'en mettant tout en œuvre pour empêcher les pannes des accélérateurs, notamment celles de longue durée, ainsi qu'en assurant un redémarrage rapide et efficace après une telle panne.

La complexité des accélérateurs de particules est très élevée. Au CERN, les données de supervision sont acquises et gérées par de nombreux systèmes différents, et exploitées par des opérateurs de différentes principales salles de contrôle, ayant des rôles complémentaires, et étant souvent géographiquement séparées [Arduini et al. 2002a] (cf. Figure II.1) :

- La « Meyrin Control Room » ([MCR] – Site de Meyrin du CERN – Suisse) est dédiée à l'opération des accélérateurs LINAC, PS Booster et PS (injecteurs SPS) et de plusieurs installations expérimentales (ISOLDE, East Hall, nToF, Décélérateur d'Antiprotons) ;
- La « Prevessin Control Room » ([PCR] – Site de Prevessin du CERN – France) est dédiée à l'opération du SPS (Super Proton Synchrotron) et prochainement à celle du LHC (Large Hadron Collider) ;
- La « Technical Control Room » ([TCR]) est dédiée à l'opération de l'infrastructure technique du CERN (24h/24) comprenant l'électricité, le refroidissement, la ventilation, les systèmes de sécurité (etc.) dont tous les accélérateurs et expériences du CERN dépendent.

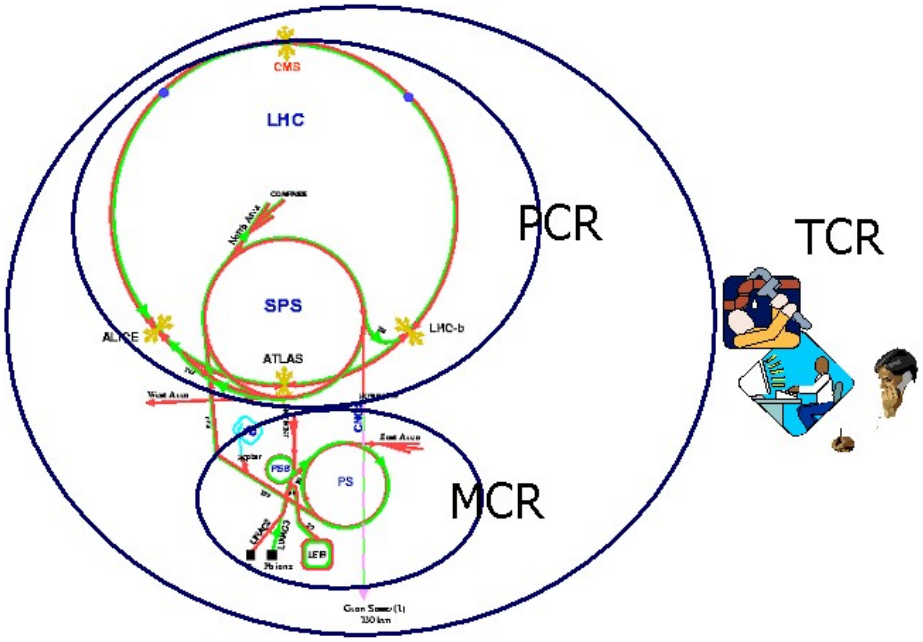


Figure II.1 : Accélérateurs et salles de contrôle du CERN

Dans l'objectif de maximiser le temps d'opération des accélérateurs, une excellente collaboration entre les opérateurs des différentes salles de contrôle, ainsi qu'une connaissance des tâches propre à chaque service sont nécessaires. Il est en outre indispensable de superviser efficacement les processus. En effet, si le système de supervision permet aux opérateurs des salles de contrôle de déterminer précisément le système qui est à l'origine d'une panne du processus, ils sont capables de résoudre le problème rapidement, et la durée de fonctionnement de l'accélérateur est alors optimisée.

Il est par conséquent nécessaire de fournir des logiciels de supervision[*] fiables et exploitables, en tenant compte des différents points développés dans la section II.1.2 :

- **Homogénéisation et standardisation :** les systèmes du CERN sont supervisés par des salles de contrôle différentes ayant des outils différents. Jusqu'à maintenant, il était nécessaire de consulter plusieurs applications pour connaître l'état du processus d'accélération de particules. Afin d'améliorer le temps de redémarrage il est nécessaire de fournir des IHMs standardisées représentant l'état global des accélérateurs, et indiquant précisément quels sont les systèmes qui sont défaillants, ainsi que l'ordre de leur remise en route (exemple : rétablir l'alimentation électrique (TCR), puis les systèmes d'accès (PCR), et enfin la production de particules (PCR)). L'état des systèmes intervenant dans le fonctionnement des accélérateurs ne doit plus être déduit de la lecture de nombreuses alarmes ou de l'association de multiples systèmes de supervision propre à des domaines spécifiques (services techniques : électricité, refroidissement, air comprimé, etc. ; accélérateurs : aimants, équipements de contrôle et de mesure du faisceau, etc.). En outre, les IHMs doivent respecter des conventions clairement définies de façon à être facilement interprétables, même par des opérateurs travaillant dans des salles différentes et ayant une connaissance technique différente ;
- **Sélection des informations :** comme dit précédemment, les accélérateurs de particules sont des machines complexes composées de nombreux systèmes et équipements. Jusqu'à maintenant, les pannes majeures engendraient un très grand nombre d'alarmes, la plupart étant sans intérêt pour le diagnostic de la cause de la panne. Dans le but de développer des IHMs exploitables, il est nécessaire de limiter le nombre d'informations qu'elles affichent. Parmi les nombreuses valeurs rendues disponibles par les équipements, il est donc crucial de ne sélectionner que celles indispensables à l'obtention d'informations pertinentes, notamment la résultante qui signale la disponibilité d'un système complexe pour la production des faisceaux ;
- **Evolution des besoins :** les accélérateurs étant des machines complexes, les personnes intervenant dans leur fonctionnement sont nombreuses. Il est nécessaire d'établir une bonne communication entre celles-ci de façon à synthétiser les besoins concernant les IHMs. De même, les accélérateurs de particules ont une longue durée de vie, et sont soumis à de nombreuses évolutions durant celle-ci. Il est donc nécessaire de fournir un support permettant de capturer les nouveaux

[*] Afin de faciliter la lecture, le terme (réducteur) « IHM » sera utilisé, dans la suite de cette thèse, pour désigner les logiciels de supervision (ayant une forte composante visualisation) devant être produits dans le cadre de cette thèse.

besoins, résultant des modifications au niveau du processus, et de les prendre en compte pour faire évoluer les IHMs en conséquence ;

- **Documentation :** comme les informations représentées sur les IHMs ne sont que celles indispensables pour connaître l'état des systèmes principaux des accélérateurs, il est important de leur associer une documentation pertinente. Cela permettra aux opérateurs d'obtenir rapidement toute l'information nécessaire au redémarrage des opérations des accélérateurs.

Dans ce contexte, le CERN a mis en place une méthode utilisée pour définir les informations qui doivent être supervisées pour permettre le redémarrage rapide des accélérateurs après une panne majeure. Cette méthode est décrite dans la section suivante.

II. 2 GTPM : Gestion Technique de Pannes Majeures

II. 2. 1 Introduction

La méthode GTPM [Arduini et al. 2002b] a été conçue pour développer des systèmes de supervision respectant un modèle orienté tâche, ce qui est particulièrement adapté au fonctionnement des accélérateurs de particules et de leur infrastructure technique. Cette méthode comprend l'analyse, la documentation et la présentation des systèmes et de leurs interactions. Les informations de supervision ainsi définies aident les opérateurs des salles de contrôle à identifier rapidement et correctement les équipements en panne et leurs implications pour le processus, ainsi qu'à établir les priorités pour les interventions des services de dépannage et des spécialistes. Ceci créée un langage commun en améliorant la compréhension et la collaboration entre les différentes salles de contrôle et entre les différents responsables d'équipements, et en rendant disponible des instructions d'opérations, des descriptions techniques, ainsi que des documents de formation d'une manière standardisée et transparente.

Cette méthode sert de base à la création d'une famille de diagrammes de supervision représentant les systèmes et leurs interactions, ainsi que la séquence et la logique de redémarrage des accélérateurs et de leurs systèmes auxiliaires (infrastructure technique). Ces diagrammes sont communs à tous les acteurs et peuvent être implémentés sous forme d'IHM dans leur environnement de contrôle et supervision.

II. 2. 2 Définition de la méthode

Pour chaque mode opérationnel d'accélérateur (ex : faisceau accéléré, production de telle particule pour telle expérience,…), la méthode produit une documentation des systèmes orientée opération, qui est ensuite utilisée pour implémenter des diagrammes de supervision orientés tâche. La documentation est extraite des documentations existantes concernant les systèmes et processus de l'accélérateur concerné et de son infrastructure technique. Cette documentation est alors optimisée pour l'opération, ce qui est rendu possible par une étroite collaboration avec les spécialistes des équipements.

Les tâches de la méthode GTPM sont les suivantes :

- identification des acteurs (opérateurs de salles de contrôle, services de dépannage, spécialistes des équipements, etc.), de leurs rôles et de leurs responsabilités ;
- définition des utilisateurs : cibles fixes, expériences (collisionneurs de particules), spécialistes du faisceau, etc. ;
- définitions des scénarii d'opération (modes opérationnels) possibles satisfaisant différentes classes d'utilisateurs (expériences utilisatrices de particules accélérées) ;
- identification des principaux systèmes/processus (exemples : vide, aimants, instrumentation faisceau) critiques pour le fonctionnement de l'accélérateur ;
- identification de l'infrastructure technique (exemple : électricité, refroidissement) dont dépendent les systèmes des accélérateurs ;
- identification des sous-systèmes/processus de l'infrastructure technique (exemple : tours de refroidissement, production et distribution d'eau déminéralisée) ;
- identification des interdépendances entre systèmes, ainsi que des corrélations de leurs fonctions ;
- construction sous forme de diagrammes de la séquence de redémarrage idéale, basée sur les interdépendances et les corrélations ;
- identification des chemins critiques pour chaque scénario d'opération ;
- représentation de cette information en diagrammes de supervision ;
- standardisation des noms des systèmes et des abréviations.

II. 2. 3 Diagrammes de supervision

En règle générale, un seul niveau de supervision est implémenté pour l'opération des processus : le niveau des équipements, c'est à dire les diagrammes détaillés des équipements et les programmes d'analyse spécialisés. La méthode GTPM ajoute trois niveaux de supervision (cf. Figure II.2).

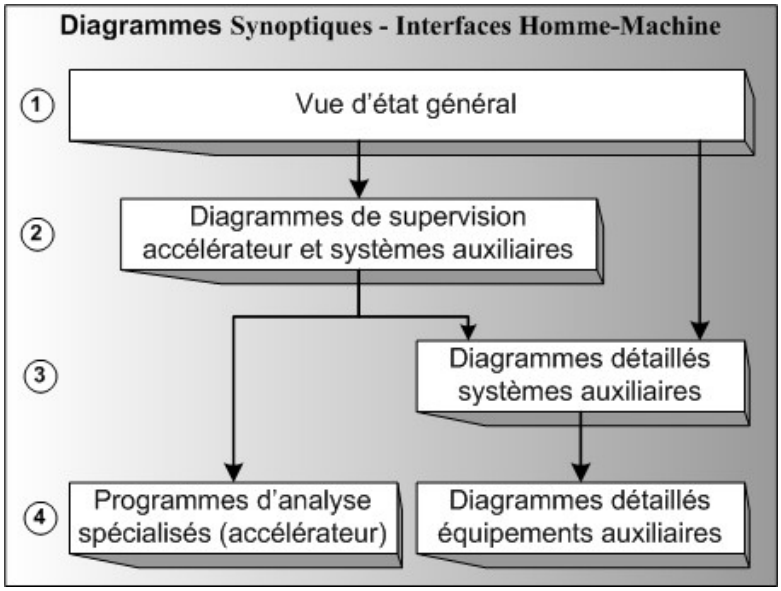


Figure II.2 : Niveaux de supervision GTPM

1. La *Vue d'état général* permet à l'utilisateur d'évaluer la disponibilité des systèmes nécessaires au fonctionnement de l'accélérateur « d'un seul coup d'œil ». Un diagramme de ce type représente tous les équipements propres à un accélérateur ainsi que les équipements auxiliaires qui lui sont directement connectés ;
2. A l'aide du niveau inférieur, c'est-à-dire des *Diagrammes de supervision accélérateurs et systèmes auxiliaires*, l'opérateur peut évaluer rapidement les états des systèmes, vérifier l'exactitude de la procédure standard de redémarrage, et établir des procédures alternatives si nécessaire ;
3. Les *Diagrammes détaillés systèmes auxiliaires* ont la même fonction mais concernent les systèmes de l'infrastructure technique. Ils représentent les systèmes et sous-systèmes auxiliaires requis par l'accélérateur ;
4. Au niveau inférieur, des programmes de diagnostic spécifiques pour l'accélérateur et pour les équipements auxiliaires sont disponibles. Ceux-ci correspondent aux seuls outils d'opération qui étaient disponibles jusqu'à maintenant.

Les diagrammes de supervision (niveau 2 de la figure II.2) contiennent les informations suivantes :

1. Les bâtiments dans lesquels se trouvent les équipements des accélérateurs et de l'infrastructure technique ;
2. L'ordre recommandé de redémarrage « géographique », c'est à dire, l'ordre dans lequel les bâtiments techniques doivent être rendus opérationnels ;
3. La séquence logique de redémarrage, c'est à dire, l'ordre dans lequel les systèmes doivent être redémarrés en fonction des contraintes techniques et des interactions ;
4. L'attribution des systèmes aux différents acteurs impliqués.

Dans chaque diagramme des niveaux 1 à 3, la procédure de redémarrage commence dans le coin supérieur gauche, et l'accélérateur est complètement disponible lorsque le coin inférieur droit est atteint et que tous les éléments intermédiaires sont disponibles.

II. 2. 4 Représentation des états des systèmes

Pour être utilisables, les diagrammes précédemment cités doivent être intégrés dans le système de contrôle/supervision de chaque acteur sous la forme d'IHMs. Pour un affichage correct des états, les valeurs rendues disponibles par chaque système doivent être identifiées et combinées pour fournir son état de disponibilité. Les états suivants sont distingués [Sollander et Camara 2000] :

- Volontairement à l'arrêt, prêt à démarrer (blanc) ;
- Complètement opérationnel (vert) ;
- Partiellement non-opérationnel (jaune) : cela signifie que le système fonctionne dans un mode dégradé et qu'une intervention est requise pour éviter une panne due à la dégradation ;
- Pas de transmission de données (bleu) ;
- Non-opérationnel ou indisponible (rouge) ;

- Non pertinent pour le mode d'opération ou pas d'information disponible pour calculer l'état (gris).

II. 2. 5 Expérimentation de la méthode

La méthode GTPM décrite dans cette section a été expérimentée une première fois pour spécifier, concevoir et implémenter un prototype dédié à la supervision d'un des accélérateurs du CERN, le SPS. Le prototype est composé d'un ensemble d'IHMs respectant les contraintes définies par la méthode. Le développement de ce prototype et son analyse sont l'objet de la section suivante.

II. 3 Développement d'un prototype : supervision du redémarrage du SPS

Le développement du prototype de supervision de l'accélérateur SPS a été effectué conformément aux besoins définis par la méthode GTPM. L'objectif était d'implémenter une famille d'IHMs homogènes et standardisées permettant de connaître à tout moment l'état précis de l'accélérateur, ainsi que la séquence de redémarrage de ses éléments après une panne majeure. La section II.3.1 expose les différentes étapes de développement qui ont été suivies, et la section II.3.2 présente l'application ainsi obtenue.

II. 3. 1 Etapes du développement

Le développement des IHMs de supervision du SPS a fait intervenir et collaborer plusieurs acteurs :

- **Les responsables d'exploitation** : il s'agit des personnes qui maintiennent les équipements, assurent leurs réglages et leur évolution et qui exécutent les dépannages souvent à l'aide d'entreprises sous-traitantes ;
- **Les spécialistes du contrôle et de la supervision de l'accélérateur :** il s'agit des personnes qui possèdent la connaissance du processus et de ses équipements. Ce sont elles qui sont à même de définir les équipements et les données devant être supervisés pour connaître l'état de l'accélérateur. Il s'agit souvent d'opérateurs de salles de contrôle, et sont donc les futurs utilisateurs des IHMs à implémenter ;
- **Les développeurs d'IHM (non informaticiens) :** il s'agit encore d'opérateurs de salles de contrôle qui connaissent le processus à superviser et qui maîtrisent les aspects graphique et ergonomique du développement d'IHM. Ces développeurs ne sont pas des programmeurs, il ne sont donc pas chargés de l'implémentation des traitements effectués par les IHMs ;
- **Les développeurs d'IHM (informaticiens) :** il s'agit de membres de l'équipe de support logiciel. Ils ne connaissent pas le processus à superviser mais sont chargés des développements requérants du codage ;
- **Le responsable de la base de données de contrôle et de supervision :** il s'agit de la personne gérant l'intégration de nouvelles informations dans la base de données de configuration du système de contrôle et d'acquisition de données. Son rôle est de vérifier que ces informations (provenant de systèmes hétérogènes) ont

bien été formatées de façon à être compatibles avec la base du système de contrôle.

Les phases de développement du prototype ont été les suivantes (cf. Annexe 1) :

- **Définition des diagrammes de supervision :** les spécialistes des équipements ont spécifié sous format graphique (feuilles Excel), la structure des IHMs qui allaient être développées. Ces diagrammes ont servi de modèle aux développeurs d'IHMs ;
- **Définition de la logique d'animation des « blocs »[1] et sélection des données à superviser :** les spécialistes des équipements ont spécifié sous format textuel (Word) la logique d'animation des blocs qui allaient être représentés sur les IHMs (exemple : le bloc « 400 V » est vert (opérationnel) si la mesure de tension 18kV est comprise entre 16kV et 19kV, et si le transformateur 18kV-400V n'envoie pas d'alarme indiquant un dysfonctionnement). Ces spécifications ont été utilisées par les développeurs d'IHMs pour coder la logique des blocs. Les spécialistes des équipements ont par ailleurs spécifié et formaté les informations provenant des systèmes à superviser. Ces informations ont dû être intégrées dans la base de données de configuration du système de contrôle ;
- **Intégration des données :** le responsable de la base de données de contrôle et de supervision vérifie le format des données sélectionnées par les spécialistes des équipements et les intègre à la base de données du système de contrôle ;
- **Implémentation des blocs :** les développeurs d'IHMs codent la logique des blocs en fonction des spécifications fournies par les spécialistes des équipements ;
- **Implémentation des IHMs :** une fois que les données de supervision sont intégrées dans la base du système de contrôle et que les blocs permettant de représenter l'état des équipements sont disponibles, les développeurs implémentent les IHMs suivant les diagrammes de supervision fournis par les spécialistes des équipements ;
- **Test des IHMs :** les spécialistes des équipements (et de l'opération de l'accélérateur) testent les IHMs implémentées ;
- **Mise en opération des IHMs :** selon le retour provenant des spécialistes des équipements, les développeurs mettent en opération les IHMs ou corrigent les éventuelles erreurs.

Ce processus de développement a aboutit à la mise en service d'une application prototype pour la supervision du SPS conforme aux contraintes fixées par la méthode GTPM.

II. 3. 2 Présentation du prototype

L'application prototype obtenue est une IHM globale affichant l'état de tous les systèmes nécessaires à l'opération de l'accélérateur [Ratcliffe et al. 2004]. Par son utilisation, les opérateurs de salle de contrôle sont capables de connaître à tout moment

[1] Le mot « bloc » désigne l'élément graphique de base des IHMs prototype. Il s'agit d'un « équipement virtuel » représentant l'état résultant d'une série d'équipements pertinents pour un mode d'opération donné.

l'état exact de l'accélérateur. Si l'opérateur désire concentrer sa supervision sur une sous partie de l'accélérateur, l'interface utilisateur lui donne la possibilité de naviguer de l'IHM globale vers des IHMs spécifiques. Ces IHMs spécifiques sont les IHMs de redémarrage de l'accélérateur, pour chacun de ses modes opérationnels (exemples : faisceau accéléré, faisceau sur cible, …) et zones géographiques (exemples : bâtiment x, point d'accès y, …). Une IHM spécifique affiche l'état des systèmes requis pour rendre disponible une partie spécifique de l'accélérateur. Les IHMs spécifiques sont composées de blocs représentant l'état de systèmes. Un bloc est associé à un ou plusieurs équipements d'un système spécifique, comme par exemple, l'électricité ou le vide. Comme expliqué dans la section précédente, ces blocs contiennent la logique utilisée pour afficher l'état d'un système en fonction de l'état de ses équipements. Ces blocs ont été construits à partir d'un bloc générique contenant les éléments graphiques de base utilisés pour la représentation de tous les systèmes. La figure II.3 représente la structure de l'application développée (l'IHM globale correspond au niveau de supervision 1 de la figure II.2, et les IHMs spécifiques correspondent au niveau de supervision 2).

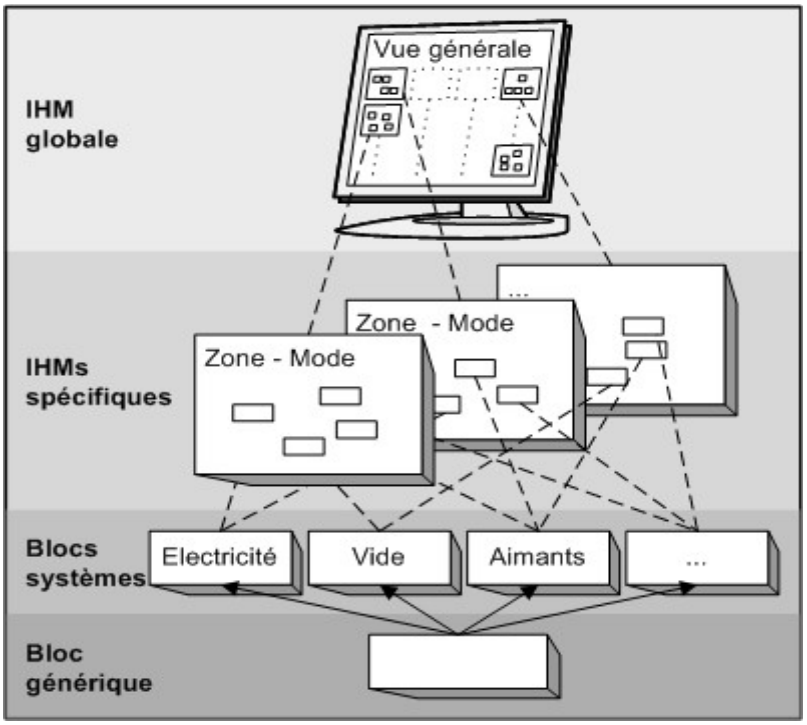


Figure II. 3 : Structure de l'application de supervision du SPS

La figure II.4 représente une capture d'écran de l'IHM globale de l'application de supervision fournie aux salles de contrôle [Barbeau et al. 2002].

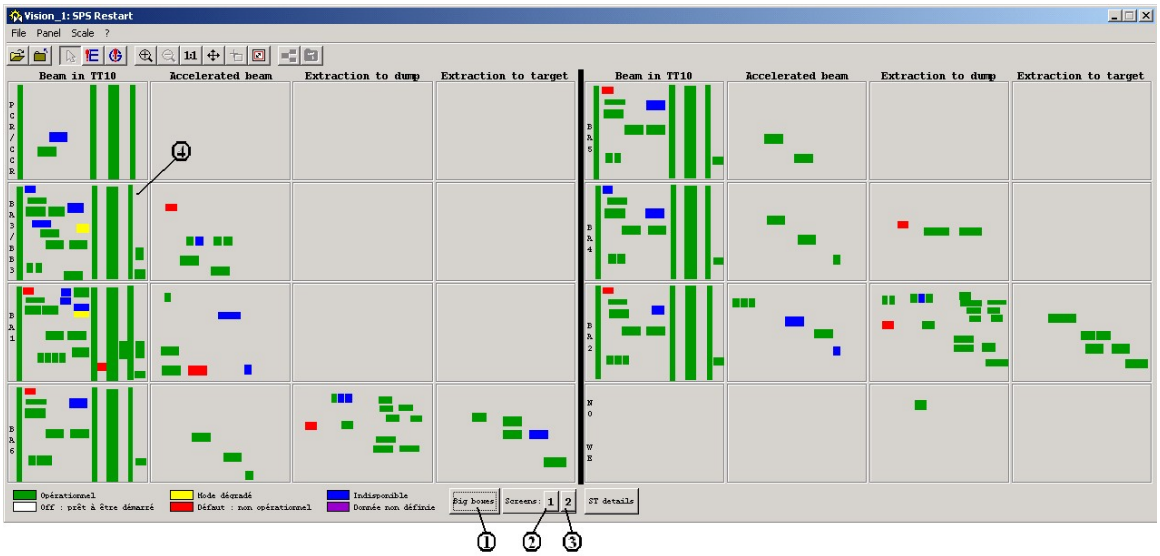


Figure II. 4 : IHM globale

L'IHM globale contient les IHMs spécifiques pour tous les bâtiments où se trouvent des équipements du SPS ainsi que pour ses quatre modes opérationnels : faisceau en ligne de transfert (TT10), faisceau accéléré, extraction vers le « dump », et extraction vers la « target ». Les opérateurs ont la possibilité de changer le niveau de détails des informations représentées①, de changer de mode d'affichage (vue globale affichée sur un ou deux écrans②③), et de zoomer sur les IHMs spécifiques en cliquant sur ④ (cf. figure II.5).

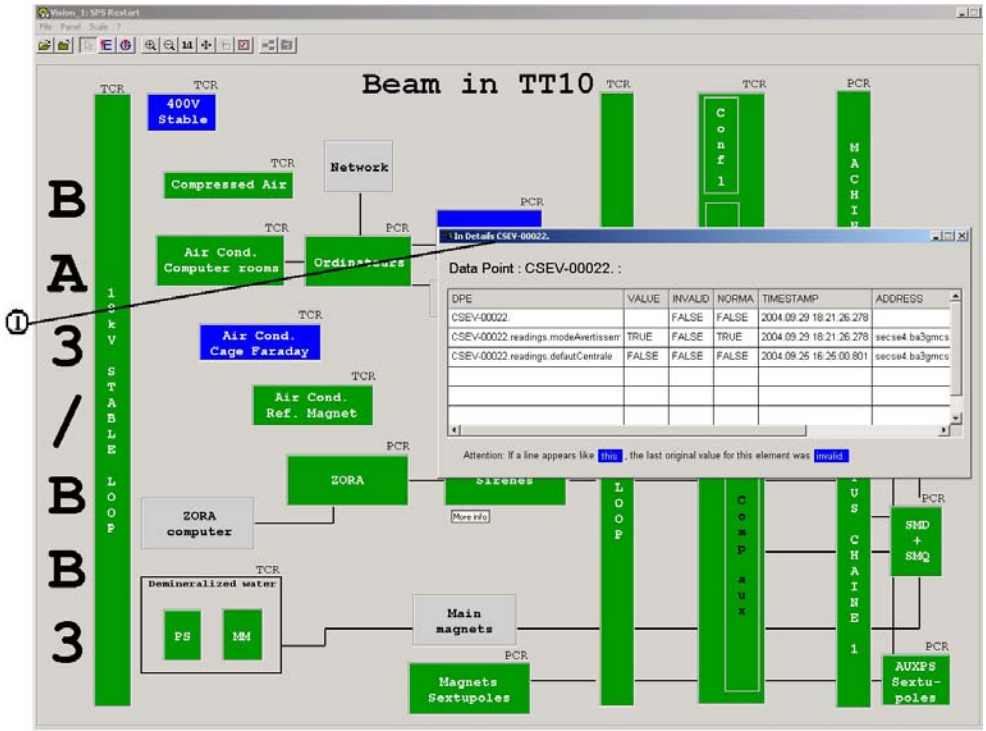


Figure II. 5 : IHM spécifique

Les IHMs spécifiques ne représentent que les systèmes correspondant à une zone géographique pour une phase d'accélération donnée. Ces IHMs permettent en outre d'obtenir des détails concernant les systèmes représentés par les blocs ① (figure II.5) : le nom des variables supervisées, les valeurs courantes, l'horaire du dernier changement de valeur des variables, leur qualité, leur valeur par défaut, …

II. 3. 3 Synthèse

La phase de prototypage présentée dans cette section a montré que :
- il est possible d'implémenter un ensemble d'IHMs correspondant aux critères définis par le projet GTPM ;
- ces IHMs permettent effectivement de réduire le temps de redémarrage des accélérateurs de particules ;
- le processus de développement des IHMs est complexe et source d'erreurs ;
- un travail d'amélioration et de support est nécessaire.

En effet, les IHMs de supervision du SPS développées ont facilité la coordination des membres des différentes salles de contrôle pour le redémarrage de l'accélérateur après ses pannes majeures. L'objectif de la méthode GTPM a bien été atteint, le temps de redémarrage a été réduit par rapport aux précédentes pannes. Ce prototype a permis de valider cette méthode (définie en II.2). Il a donc été décidé de l'appliquer aux autres accélérateurs et zones expérimentales du CERN.

Toutefois, les autres accélérateurs ont tous leurs spécificités et il était clair que des difficultés allaient être rencontrées, notamment au niveau du nombre de systèmes impliqués dans leur opération ainsi qu'à celui de leur hétérogénéité.

Par ailleurs, comme nous l'avons vu, la méthode de développement du prototype a été relativement complexe, impliquant plusieurs catégories d'acteurs et de nombreuses interactions entre eux. D'ailleurs, les problèmes sont majoritairement survenus lors de ces interactions. L'incomplétude ou le mauvais formatage des spécifications fournies par les spécialistes de l'opération ont parfois rendu difficile ou impossible leur implémentation par les autres acteurs. Il a fallu de multiples itérations avant d'obtenir des spécifications exploitables. Par ailleurs, les autres acteurs, n'étant pas assez guidés dans leur implémentation, ont commis à plusieurs reprises des erreurs au niveau du développement des IHMs. Là encore, il a fallu de nombreuses itérations avant que les spécialistes de l'opération obtiennent des IHMs satisfaisantes. Parmi les problèmes les plus souvent rencontrés se trouvent :
- le mauvais choix des équipements à superviser ;
- la mauvaise définition et/ou implémentation de la logique d'affichage des états de blocs ;
- les erreurs au niveau de la standardisation des IHMs ;
- les conflits dus aux développements parallèles ;
- la mauvaise adaptation des données entre leur système source et le système de contrôle du prototype (problème de l'homogénéisation des données) ;
- les erreurs de codage.

Ces constats ont mis en évidence la nécessité d'implémenter un outil logiciel permettant d'améliorer, d'automatiser, et donc de faciliter le processus de développement des IHMs. L'objectif de cet outil est de servir de support à la méthode GTPM en aidant au choix des équipements à superviser, en guidant la spécification ainsi que l'implémentation des IHMs et des blocs correspondants, et en permettant la génération d'IHMs standardisées. La section suivante expose les besoins concernant cet outil ainsi que son environnement de fonctionnement.

II. 4 Cahier des charges de SEAM : Software for the Engineering of Accelerator Monitoring

Cette section expose les besoins, exprimés par les principales salles de contrôle du CERN (TCR, PCR, MCR), devant être satisfaits par l'outil logiciel SEAM. Celui-ci doit être utilisé comme support à la méthode d'ingénierie de supervision des équipements de l'accélérateur et de l'infrastructure technique.

II. 4. 1 Besoins généraux

Afin de proposer des solutions aux problèmes listés en II.3.3, l'outil SEAM doit être conçu pour permettre de :

1. **Faciliter l'implémentation des IHMs :** les opérateurs doivent être capables d'implémenter les IHM sans l'aide de développeurs informaticiens. Le but est de simplifier le processus de développement exposé en II.3.1. Comme le nombre d'intervenants est ainsi réduit, le nombre d'erreurs dues aux problèmes de communication entre eux le sera aussi. Le nombre d'itérations nécessaires à la mise en place d'IHMs complètes et exploitables sera minimisé. SEAM doit aussi réduire le nombre d'erreurs de spécification en fournissant une aide au choix des équipements à superviser et à la définition de la logique des blocs ;
2. **Générer des familles d'IHMs standardisées** qui doivent respecter des propriétés graphiques, structurelles et comportementales communes (définies dans la section II.4.2). SEAM doit fournir un guide de développement des IHMs ;
3. **Contraindre la construction des IHMs** pour qu'elles vérifient des propriétés spécifiées : la standardisation, la complétude et l'exploitabilité des IHMs ne sont possibles que si des « normes » de développement sont respectées (cf. section II.4.2). SEAM doit donc permettre de spécifier ces normes et d'assurer que les IHMs générées les respectent ;
4. **Capturer le savoir-faire du domaine et réutiliser l'expérience acquise :** SEAM doit permettre à ses utilisateurs d'accéder facilement aux sources d'informations concernant le processus à superviser, c'est à dire le redémarrage d'accélérateurs. Il doit par ailleurs être flexible pour pouvoir évoluer en fonction de l'expérience acquise au cours des développements d'IHMs successifs (correspondant à différents accélérateurs). En effet, le savoir-faire propre au redémarrage des accélérateurs va croître au fur et à mesure des développements. SEAM doit se mettre à jour en fonction de l'évolution des besoins concernant les IHMs à générer ;

5. **Gérer l'évolution des IHMs :** les accélérateurs de particules, ainsi que leurs processus de redémarrage, sont soumis à des évolutions durant leur durée de vie. SEAM doit donc permettre de mettre à jour les IHMs en fonction des évolutions des processus.

Techniquement, l'outil SEAM doit principalement proposer les fonctionnalités suivantes [Ratcliffe 2002] :

- proposer une interface utilisateur permettant de spécifier (de façon graphique) les éléments essentiels des séquences de redémarrage d'un accélérateur (Cf. section II.4.4.1);
- automatiser le plus possible les différentes étapes de la méthode de sélection des informations de supervision pertinentes (Cf. section II.4.4.2);
- comporter un module d'interfaçage avec le système de contrôle et de supervision (Cf. section II.4.4.3), de façon à faciliter au maximum l'implémentation de plusieurs niveaux hiérarchiques d'IHMs supervisant le redémarrage de l'accélérateur.

L'outil doit être en relation avec plusieurs bases de données afin de consulter et de sélectionner les diverses informations concernant les équipements intervenant dans les séquences de redémarrage de l'accélérateur. L'outil doit posséder sa propre base de données pour regrouper et homogénéiser l'ensemble de ces informations. Cette base doit permettre d'enregistrer sous un format électronique et de maintenir les informations qui étaient jusqu'à maintenant traitées par l'intermédiaire de fichiers (Word et Excel, cf. section II.3.1). D'autre part, il doit être étroitement lié au système de supervision chargé de l'acquisition des états fournis par les équipements et de l'affichage des vues de conduite. SEAM doit fournir à ce système le code des IHMs concernant le redémarrage de l'accélérateur. La section II.4.2 liste les caractéristiques que devront respecter les IHMs produites par SEAM. La section II.4.3 présente l'environnement de fonctionnement de cet outil, c'est à dire les éléments des systèmes de contrôle et de supervision du CERN avec lesquels il doit interagir. Enfin, la section II.4.4 détaille les fonctions que doivent remplir les interfaces de SEAM avec cet environnement.

II. 4. 2 Caractéristiques des IHMs à produire

SEAM doit permettre de construire des IHMs facilement exploitables. Pour cela elles doivent respecter des caractéristiques et propriétés communes. Les IHMs doivent premièrement respecter certaines propriétés graphiques. Ces propriétés correspondent à la « norme » définie dans le cadre du projet GTPM et raffinée après le développement et l'analyse des IHMs prototype. Il s'agit des propriétés des éléments graphiques composant les IHMs ainsi que les blocs représentant l'état des systèmes, comme par exemple :

- leur dimension ;
- leur couleur ;
- leur position absolue et relative ;
- leur plan d'affichage ;
- la possibilité ou l'interdiction de superposition ;
- les propriétés des zones de texte.

En plus de ces propriétés qui ont pour but de rendre les IHMs « lisibles », SEAM doit imposer le respect de contraintes de représentation des informations en fonction de leur rôle ou de leur signification, comme par exemple :

- la position horizontale des blocs sur les IHMs spécifiques dépend de l'ordre de redémarrage des systèmes qu'ils représentent ;
- les contraintes de cardinalité des blocs présents sur les IHMs spécifiques, comme leur nombre maximum, leur présence obligatoire ou optionnelle, ou encore l'interdiction de la redondance d'information (deux blocs représentant l'état du même système ne peuvent pas se trouver sur la même IHMs) ;
- le positionnement horizontal des IHMs spécifiques sur l'IHM globale dépend de leur position des systèmes associés dans l'ordre de redémarrage de l'accélérateur ;
- le positionnement vertical des IHMs spécifiques sur l'IHM globale dépend de la position géographique des systèmes dont elles supervisent l'état.

Les IHMs spécifiques permettent de représenter les dépendances existant entre les systèmes représentés. Celles-ci doivent aussi obéir à certaines règles, comme :

- deux dépendances identiques ne doivent pas se trouver sur la même IHM ;
- un bloc ne doit pas être dépendant de lui même ;
- il ne doit pas exister de boucles de dépendances qui représenteraient des situations d'autoblocage de la séquence de redémarrage.

Toutes les propriétés citées jusqu'à maintenant concernent la représentation des informations, mais les IHMs de supervision contiennent aussi du code permettant de traiter les données reçues des systèmes avant de pouvoir les afficher. Le traitement des données doit respecter certaines contraintes, comme par exemple :

- les blocs représentant l'état de systèmes doivent traiter toutes les valeurs que ceux-ci peuvent prendre. En d'autres termes, aucune valeur de variable supervisée par un bloc ne doit pouvoir entraîner un état indéterminé ;
- les blocs doivent uniquement superviser des variables dont ils supportent le type ;
- chaque bloc doit, indépendamment du nombre et du type des variables qu'il supervise, posséder une logique interne permettant de fournir un état résultant respectant un format standard (ex : les valeurs résultantes des blocs doivent être des entiers entre 0 et 3, chacune d'elles correspondant à une couleur parmi celles définies dans la section II.2.4).

De plus, l'une des propriétés les plus importantes des IHMs est que celles-ci doivent pouvoir évoluer. Les IHMs doivent être conçues pour être modifiables et améliorables. Pour cela, il doit être possible (sans codage, via une interface graphique) d'ajouter, supprimer ou modifier des blocs, des IHMs ainsi que des liens de dépendance. Toutefois, ces modifications ne doivent pas entraîner de violation des contraintes définies lors de la création des IHMs, c'est à dire celles présentées ci-dessus.

II. 4. 3 Intégration dans l'environnement

La figure II.6 représente l'outil SEAM ainsi que les principaux composants de son environnement :

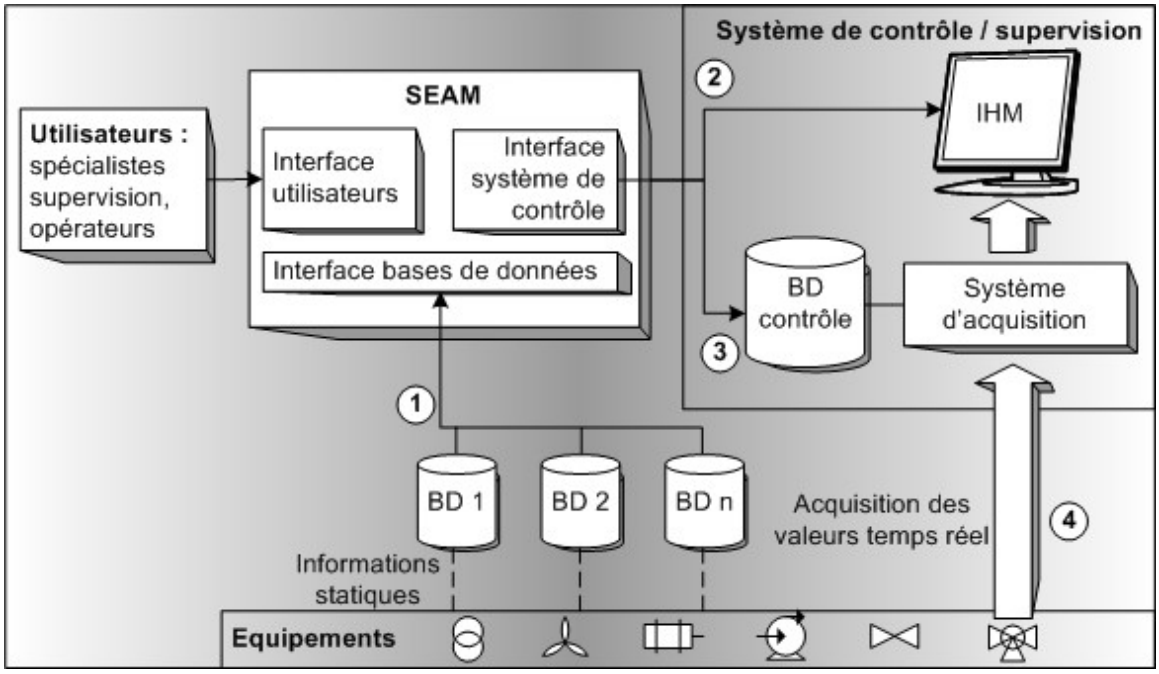


Figure II. 6 : Environnement de l'outil SEAM

Afin de pouvoir construire des IHMs, SEAM doit accéder aux informations concernant les équipements intervenant dans le fonctionnement des accélérateurs ①. Les informations nécessaires à ce niveau sont statiques, en effet, elles ne doivent pas changer durant le fonctionnement de l'accélérateur, mais uniquement en cas de modification matérielle appliquée sur celui-ci. Les bases de données contenant ces informations font l'objet de la section II.4.3.1. Lorsque SEAM a sélectionné les données nécessaires à la construction des IHMs, il doit les intégrer au système de contrôle et de supervision (cf. section II.4.3.2). Pour cela il fournit le code des IHMs à la couche d'affichage graphique du système ② et il configure la base de données de contrôle ③. Les IHMs sont alors exploitables pour superviser les accélérateurs et affichent en temps réel les états des équipements acquis par le système de contrôle ④.

II. 4. 3. 1 Bases de données sources

L'outil SEAM doit permettre la sélection et l'utilisation d'informations concernant des équipements gérés par différents services et différents systèmes. Ces informations sont réparties dans plusieurs bases de données (Oracle) selon le type, la localisation, et la fonction des équipements correspondants. En règle générale, chaque salle de contrôle du CERN possède sa (ses) propre(s) base(s) de données. Comme les IHMs générées par SEAM doivent inclure des équipements sous la responsabilité de chacune des salles de contrôle, il est nécessaire qu'il puisse accéder à leurs bases. Il s'agit majoritairement de

bases de données de systèmes de contrôle. Celles-ci contiennent des informations concernant les alarmes et les valeurs émises par les équipements. Il peut aussi s'agir de bases données spécifiques à des catégories d'équipements (alimentations, câbles, …) contenant leurs caractéristiques détaillées, ou encore de bases de données de GMAO (Gestion de Maintenance Assistée par Ordinateur).

II. 4. 3. 2 Système de contrôle et de supervision

Le système de contrôle utilisé pour la supervision du redémarrage des accélérateurs du CERN acquiert les valeurs rendues disponibles par les automates des équipements via un middleware orienté messages (MOM : Message Oriented Middleware) utilisant la technologie JMS (Java Message Service [Monson-Haefel et Chappelle 2002]). Celui-ci est composé d'un « broker » ([SonicMQ]) et d'un serveur d'application ([Oracle9iAS]). Le middleware, configuré au moyen de la base de donnée de contrôle, transmet les informations acquises à la couche graphique du système de contrôle. Cette couche, qui assure l'affichage des IHMs, est constituée du logiciel [JViews] basé sur la technologie Java.

SEAM doit permettre de spécifier les données à acquérir par le middleware ainsi que les règles à appliquer sur celles-ci pour que l'affichage de l'état des accélérateurs soit possible. Ceci est fait via la base de données de contrôle. Par ailleurs, SEAM doit fournir le code des IHMs au logiciel JViews (cf. section II.4.4.3 pour plus de détails).

II. 4. 4 Interfaces

II. 4. 4. 1 Interface utilisateurs

L'interface utilisateurs est une interface graphique permettant de saisir les informations telles qu'elles sont représentées dans les diagrammes de redémarrage de l'accélérateur et selon les mêmes conventions (ex : lecture de la séquence de redémarrage de gauche à droite et du haut vers le bas ; un lien entre deux systèmes doit être modélisé lorsque ceux-ci ont une relation de dépendance ; etc.).

Elle doit principalement permettre :

- de guider les utilisateurs dans la définition des IHMs. L'interface doit réduire la liberté des développeurs de façon à assurer l'homogénéité des IHMs construites en imposant le respect des propriétés et contraintes définies par la méthode GTPM. Il est à noter que ces contraintes peuvent évoluer. En effet, les évolutions matérielles des accélérateurs ainsi que la prise en compte de nouvelles machines peut provoquer l'ajout de nouvelles contraintes ou la suppression d'anciennes. L'interface graphique de développement doit donc être paramétrable en fonction de l'évolution du jeu de contraintes ;
- la saisie de symboles appelés « blocs » représentant des systèmes ou des ensembles de systèmes auxquels pourront être associées diverses propriétés : identifiant, nom (ex : *Sirènes*), description (ex : *centrale évacuation BA1*), responsable supervision, aspect graphique, équipements associés, valeurs associées, liste des états (couleurs) que pourra prendre le bloc, équation permettant de calculer l'état du bloc en fonction de l'état des équipements qui lui

sont associés, localisation physique, localisation fonctionnelle, liens vers de la documentation, … L'interface doit proposer la sélection des valeurs de certaines de ces propriétés parmi des listes existantes, directement issues de bases de données ;

- la modélisation des interactions ou corrélations entre ces blocs par des liens graphiques orientés (ex : *la présence du 400V est conditionnée par la présence du 18kV*) ;
- le stockage des informations saisies graphiquement (positionnement, taille, corrélations, propriétés…) dans une base de données propre à l'application.

II. 4. 4. 2 Interface bases de données

L'objectif de ce module est de permettre de compléter les informations entrées via l'interface graphique. Pour cela, il doit fournir à l'utilisateur un support pour sélectionner les informations pertinentes dans les bases de données de supervision du CERN, ou quand c'est possible, les sélectionner automatiquement. Ce module doit donc utiliser des informations, présentes dans des bases existantes, qui sont nécessaires au développement des IHMs, c'est à dire les informations concernant tous les organes essentiels au fonctionnement de l'accélérateur pour un mode d'opération donné.

L'accent doit particulièrement être mis sur la maintenabilité du logiciel en fonction des éventuelles modifications des structures de données sources. Ces modifications devraient être les plus transparentes possible pour l'outil. La conception du module de consultation de données doit prévoir un certain nombre de changements pouvant avoir lieu dans son environnement de fonctionnement, afin de réduire leurs influences et ainsi de minimiser et de faciliter les opérations de maintenance.

Les informations consultées par ce module complètent celles saisies par l'utilisateur et sont référencées par le logiciel de façon à constituer une base de connaissance concernant la supervision du redémarrage des accélérateurs.

II. 4. 4. 3 Interface système de contrôle

L'interface utilisateurs et l'interface de sélection de données sont utilisées pour définir les informations nécessaires à la construction des IHMs qui seront affichées en salle de contrôle. Le module d'interfaçage avec le système de supervision doit quant à lui permettre d'exploiter ces informations. Ceci comporte deux aspects :

- **Génération du code des IHMs** dans le langage propre au système de supervision. Le code des IHMs comporte en particulier les caractéristiques graphiques des blocs affichés (éléments, taille, position, couleur…), ainsi que les variables qui animent ces blocs. Par exemple, le code correspondant à un bloc « Basse tension bâtiment 1 » doit inclure les informations suivantes :
  - Nom : *BT Bât 1* ;
  - Taille : *100 x 40 pixels* ;
  - Position : *200, 150* ;
  - Variable d'animation : *400Vbat1* ;
  - Couleurs : *vert si 400Vbat1 = 0, jaune si 400Vbat1 = 1, rouge si 400Vbat1 = 2, …* ;

  - Liens vers des pages de documentation.
- **Insertion des informations** dans la base de données de configuration du système de supervision. Il s'agit des informations concernant les équipements supervisés, les conditions de changement d'état des blocs graphiques, ainsi que les informations concernant ce qu'ils représentent. Pour l'exemple précédemment cité, les informations seraient :
  - Le bloc *BT Bât 1* affiche l'état résultant des équipements suivants : *Haute tension bâtiment 1*, *Transformateur HT-BT* et *Disjoncteur bâtiment 1*. Pour ces trois équipements il faut spécifier leurs localisations précises, les noms de leurs responsables, les noms (ex : *mesure18kVbat1*, *etatTransfoHTBT*, *etatDisjonteurBat1*), les adresses des valeurs à lire, etc. ;
  - Les règles concernant le bloc, comme par exemple : la variable *400Vbat1* est à 0 si *mesure18kVbat1* > 18000 et *etatTransfoHTBT* = 0 et *etatDisjonteurBat1* = 0, etc.

Lorsque le code des IHMs est généré et les informations concernant ce qu'elles représentent sont insérées dans la base de donnée de contrôle, le système de supervision est capable d'acquérir l'état des équipements, d'évaluer les règles fournissant l'état des blocs associés, et de représenter ceux ci sur les écrans des salles de contrôle.

II. 5 Conclusion

Ce chapitre a présenté les différents aspects de la problématique de cette thèse. Il a décrit la méthode GTPM, qui constitue la solution choisie par le CERN pour répondre au besoin d'augmentation de la productivité des accélérateurs de particules par la réduction du temps de redémarrage après une panne. Des prototypes d'IHMs dédiés à la supervision du redémarrage de l'accélérateur SPS ont été développés à partir de cette méthode. Les différentes difficultés rencontrées lors de ce prototypage, ainsi que le besoin d'étendre cette approche à tous les accélérateurs de particules du CERN, ont souligné la nécessité de mettre en place un outil logiciel servant de support à la méthode GTPM. Cet outil, SEAM, doit faciliter et améliorer le processus de développement de familles d'IHMs standardisées. Les besoins le concernant constituent la base de la problématique à résoudre, et son implémentation va permettre la validation de l'approche proposée dans cette thèse.

Les chapitre suivant présente les outils développés, et les travaux menés, dans le contexte de problématiques similaires : le développement d'interfaces graphiques pour la supervision des processus, et le développement de familles de produits.

Chapitre III L'état de l'art

Chapitre III : L'état de l'art

III. 1 Introduction

Les méthodes de développement logiciel en général, et d'IHMs* en particulier, sont de plus en plus abouties et proposent un ensemble d'outils pour les prendre en charge. Ce chapitre présente les travaux réalisés dans plusieurs domaines apportant des solutions à notre problématique. Afin de définir les domaines à étudier, il est nécessaire d'analyser les besoins concernant l'outil SEAM exposés dans le chapitre précédent (section II.4.1) :

1. Faciliter l'implémentation des IHMs ;
2. Générer des familles d'IHMs standardisées ;
3. Contraindre la construction des IHMs ;
4. Capturer le savoir-faire du domaine et réutiliser l'expérience acquise ;
5. Gérer l'évolution des IHMs.

La figure III.1 présente les liens entre les besoins exprimés par les utilisateurs de l'outil SEAM et les critères d'évaluation des solutions étudiées dans ce chapitre.

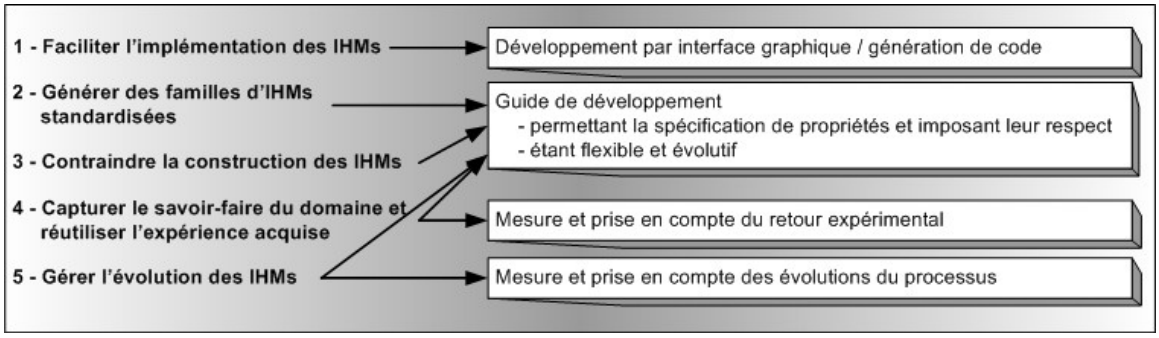


Figure III. 1 : Besoins et critères d'évaluation

SEAM étant un outil de développement d'IHMs, il a tout d'abord été nécessaire d'étudier l'état de l'art dans le domaine des méthodes et des outils dédiés à la conception et à l'implémentation d'IHMs pour la supervision des processus. C'est l'objet de la section III.2 de ce chapitre.

Par ailleurs, le besoin de développer des familles d'IHMs possédant des caractéristiques communes clairement définies, a suggéré l'étude de l'état de l'art dans le domaine des techniques de développement d'architectures permettant de guider

* Le terme « IHM » est employé dans cette thèse pour désigner des logiciels de supervision ayant une forte composante graphique.

l'implémentation de familles d'applications logicielles. Cette étude fait l'objet de la section III.3.

Les critères d'évaluation des méthodes, outils et techniques présentés dans ce chapitre sont ceux mis en évidence par la figure III.1 :

- Possibilité de développement par interface graphique suivie d'une génération de code ;
- Possibilité de spécifier et de respecter un guide de développement flexible et évolutif ;
- Possibilité de mesurer et prendre en compte le retour expérimental ;
- Possibilité de mesurer et prendre en compte les évolutions du processus.

III. 2 Méthodes et outils de développement d'interfaces graphiques pour la supervision des processus

III. 2. 1 Standards et modèles existants

La majorité des travaux effectués dans le domaine du développement d'IHMs soulignent la nécessité d'utiliser des méthodes et modèles de conception. L'objectif est de capturer le savoir-faire du domaine pour faciliter le développement de nouvelles IHMs efficacement exploitables par les utilisateurs. De nombreux « conseils de développement », ou « guidelines », ont ainsi vu le jour depuis le début des années 80. Certains de ces principes, comme « ne pas utiliser plus de cinq à sept couleurs sur une IHM » [Brown et Cunningham 1989], sont très spécifiques, d'autres au contraire, comme « essayer d'obtenir la cohérence », sont beaucoup plus généraux [Shneiderman 1992]. Brown [Brown 1988] a défini une collection d'environ 250 principes de développement. La plupart de ces « conseils » sont utilisables par tout type d'application, comme les « Macintosh Human Interface Guidelines » [Apple Computer Inc. 1992] et « The Windows Interface: An Application Design Guide » [Microsoft Corporation 1987], mais d'autres sont plus ciblés vers le développement d'IHMs de supervision, comme les standards ISO 11064 [ISO 2000] et NUREG-0700 [U.S. Nuclear Regulatory Commission 2002]. Toutefois, l'utilisation de ces « instructions » de développement comporte des difficultés, notamment au niveau de leur sélection, de leur validité et de leur applicabilité [Welie et al. 2000]. En effet, ces guides sont si nombreux qu'il est difficile de sélectionner celui qui parmi eux est le plus approprié pour résoudre un problème particulier. De plus, ceux-ci peuvent parfois se contredire entre eux.

Pour palier à ces problèmes, la communauté des concepteurs d'IHMs tend à remplacer l'utilisation de ses « conseils » et « instructions » par un type de guide de développement emprunté au domaine de l'ingénierie logicielle, il s'agit des patrons (ou « design patterns ») [Griffiths et Pemberton 2000]. Inspirés par les travaux d'Alexander [Alexander et al. 1977], les patrons sont largement utilisés pour la conception logicielle [Gamma et al. 1995]. Ceux-ci, définis par [Love 1991] comme étant des règles de conception permettant une conception réussie, ciblent le contexte d'utilisation et comportent les informations permettant au concepteur de savoir quand, comment et pourquoi la solution peut être appliquée. Un patron est décrit en termes de problème, de contexte, et de solution, cette dernière devant être une solution éprouvée. Les patrons

expriment les « choses à faire » et sont parfois combinés aux anti-patrons [Browne et al. 1998], ou « anti-patterns », qui eux expriment les « choses qui ne vont pas bien » et proposent des solutions pour les éviter. Le terme « patron » est utilisé pour décrire des guides de quatre différents niveaux [Skogseid et Sptring 1995] :

- composants, conventions et interfaces (ex : composants logiciels, outils, et APIs) ;
- styles, combinaisons et règles (ex : l'ordre des informations des menus doit correspondre à l'ordre des actions dans l'application) ;
- métaphores et modèles (ex : rendre les choses visibles et concrètes plutôt qu'invisible et abstraites) ;
- approches et écoles (ex : réalité virtuelle).

Un patron est une abstraction qui peut être instanciée. Il est décrit par une phrase simple et correspond à un aspect observable ou discernable de l'interface. Un patron :

- a une signification pour les utilisateurs, les concepteurs, et les programmeurs ;
- aide à résoudre les conflits de conception en améliorant les communications entre les phases de développement du système ;
- est relativement à l'abri des changements de technologie ;
- et peut être appliqué dans des situations multiples à des niveaux aussi bien spécifiques que généraux.

En outre, un patron doit être facile à instancier et doit fournir un guide spécifique concernant ce qui doit être fait.

Ainsi, les patrons peuvent constituer des outils plus puissants que de simples « instructions » de développement. L'intérêt de la notion de patron pour la conception d'interfaces utilisateurs n'est devenu significatif que récemment [Borchers et Thomas 2001]. Depuis lors, de nombreux patrons de développement d'IHMs ont été proposés. Ceux-ci ont souvent été construits à partir de « conseils de développement » existants. [Skogseid et Spring 1995], par exemple, ont analysé les 250 « conseils » de Brown, les ont groupés, et les ont classés en neuf patrons :

- information consistante ;
- information expressive ;
- interface informative ;
- commandes universelles ;
- séquence expressive ;
- données exploitables ;
- information ordonnée ;
- objets reconnaissables ;
- information appropriée.

Ces patrons, pour être exploitables, ont été décrit selon un même format. Ils sont composés d'un titre, d'un paragraphe décrivant le contexte d'utilisation, d'un paragraphe indiquant la logique du patron, d'exemples d'utilisation, d'une phrase indiquant comment l'utiliser, et des références aux « conseils » constituant le patron. Le tableau III.3 présente l'exemple du patron « séquence expressive » :

Titre	Séquence expressive
Contexte	Une boîte de dialogue doit avoir une séquence expressive. La séquence des opérations peut être essentielle pour la tâche exécutée.
Logique	Les humains ont tendance à organiser les informations en listes et les classer selon certains principes logiques. Il est donc important que les applications informatiques offrent la même possibilité, et présentent les instructions et les informations d'une manière expressive pour l'utilisateur.
Exemples	Utiliser des principes d'organisation existants s'ils sont disponibles (chronologique, alphabétique, séquentiel, fréquence d'utilisation, …)
Utilisation	La séquence de dialogue dans le domaine d'application doit être identifiée et réutilisée dans le système.
Références	Brown (1988) 2.21, 2.26, 2.39, 3.26, 6.10, 6.19, 6.36, 7.25

Tableau III. 3 : Patron « séquence expressive »

Certains travaux ont organisé les patrons de développement d'IHMs en « langages de patrons » [Bayle 1998][Coram et Lee 1996]. Ces langages fournissent les éléments de base utiles pour la conception de la majorité des IHMs (menus, barres d'outils, boîtes de dialogue, palettes, …).

L'utilisation des patrons et des langages associés est très répandue dans le domaine de la conception d'IHMs, et celle-ci permet aux concepteurs de fournir des interfaces facilement exploitables par les utilisateurs. Il est clair que les patrons possèdent de nombreux avantages par rapport aux « conseils » et « instructions », ils permettent de capturer la connaissance prouvée et validée propre au domaine de la conception d'IHMs, et de la réutiliser. Toutefois, ceux-ci sont souvent très génériques, c'est-à-dire utilisables dans le développement d'IHMs pour tout type d'application. Ils ne sont pas conçus pour répondre à des besoins propres à des applications spécifiques, comme par exemple, la supervision des processus. De plus, si les travaux sont nombreux dans le domaine de la définition et l'organisation des patrons, ils le sont moins au niveau de leur exploitation. En effet, les patrons constituent des guides efficaces permettant au développeur d'IHMs de savoir ce qu'il doit faire, mais il n'existe pas d'outil logiciel permettant de les exploiter pour obtenir de façon automatique (au moins partiellement) le code de base des IHMs. La nature informelle de ces patrons ne permet pas d'avoir d'outil vérifiant automatiquement qu'une IHM vérifie tel ou tel patron. Par ailleurs, même s'il existe des méthodes de définition de patrons, celles-ci relèvent d'une démarche intellectuelle humaine, les patrons ne sont pas définis et mis à jour automatiquement à partir de « mesures » faites sur leur domaine d'application.

Les différentes caractéristiques des techniques présentées jusqu'ici sont synthétisées et évaluées dans la section II.2.3.

III. 2. 2 Outils de développement

Il existe de nombreux outils de développement de vues de supervision. Ceux-ci sont basés sur des technologies diverses et ont chacun leurs propres avantages et inconvénients par rapport à notre problématique. Le projet de supervision de redémarrage des accélérateurs de particules du CERN a impliqué l'évaluation de plusieurs de ces

outils. Deux alternatives au niveau de la mise en place du système de supervision ont été évaluées par [Sollander et al. 2003] :

- l'utilisation d'un logiciel commercial de type SCADA (Software for Control and Data Acquisition) qui inclue tout le système de contrôle et de supervision : acquisition et transmission des valeurs provenant des équipements, base de données de configuration, et représentation des informations ;
- l'utilisation d'une approche « composant » : le système de contrôle et de supervision est alors composé de plusieurs couches logicielles distinctes, un middleware pour l'acquisition et la transmission des données, un outil d'édition et d'exécution des IHMs pour leur représentation, et un système de gestion de base de données pour leur configuration.

Les logiciels de type SCADA (comme [PVSS] ou [PcVue]) sont très complets et couvrent la majorité des besoins communs à la supervision de tout processus. Toutefois, ils ont l'inconvénient d'être peu flexibles et donc peu adaptables aux besoins spécifiques à un domaine particulier. La seconde approche permet de combiner des outils dédiés à des aspects spécifiques de la supervision des processus. Au niveau de la couche de représentation des informations, les outils dédiés à l'édition et à l'exécution d'IHMs (comme [JViews] ou [SL-GMS]) sont le plus souvent flexibles, paramétrables, et adaptables aux besoins propres à la supervision des accélérateurs de particules.

Tous les outils actuels de modélisation d'IHMs proposent des interfaces graphiques de développement. Les IHMs de supervision peuvent donc être partiellement développées sans codage. Toutefois il n'est encore pas possible à des développeurs non formés à la programmation de les concevoir entièrement. En effet, même si les aspects graphiques ne nécessitent pas de codage, et même si certains outils (comme PVSS) proposent des guides de type « wizard » pour spécifier quelques comportements basiques des IHMs en fonction d'interactions avec les utilisateurs (saisie d'informations, actions sur des boutons) et des changements de valeurs des équipements supervisés (règles de changement d'état des objets graphiques), le codage reste nécessaire pour la spécification de comportements et règles complexes. Il ne peut-être supprimé que si des outils dédiés à ce type de spécification, dans un domaine d'application particulier, sont développés pour étendre les fonctionnalités des logiciels disponibles sur le marché.

La majorité des outils possèdent des fonctionnalités permettant de guider le développement des IHMs. Cela est rendu possible par la personnalisation des éditeurs graphique en fonction des besoins propres au domaine d'application. La plupart des outils permettent en effet la définition et l'utilisation de palettes d'objets graphiques spécifiques, et certains (comme JViews) autorisent l'ajout ou la suppression de fonctionnalités dans les menus et les barres d'outils. Le développeur est guidé dans le sens que son espace de développement est réduit. L'éditeur d'IHMs est paramétré par suppression des fonctionnalités et objets graphiques inutiles pour le domaine d'application, et par ajout de ceux qui lui sont indispensables. Toutefois de tels guides ne permettent pas d'assurer que toutes les IHMs construites par son intermédiaire seront standardisées et respecteront des propriétés complexes prédéfinies. Hormis au niveau des palettes d'objets, qui peuvent être régulièrement misent à jour, ces guides de développement d'IHMs ne sont pas flexibles et évolutifs. Même si des outils comme JViews permettent lors de leur mise en place l'ajout, la suppression ou la modification de

fonctionnalités, il n'est pas facile de les faire évoluer ultérieurement. En effet, la personnalisation d'un tel outil nécessite une nouvelle phase de codage et donc l'intervention d'une équipe de développement.

Concernant les deux derniers besoins exprimés dans la section précédente, c'est à dire la nécessité de mesurer les évolutions du processus ainsi que la satisfaction des utilisateurs des IHMs implémentées, ceux-ci ne sont pas traités par les outils de développement. La mesure des évolutions du processus nécessite d'avoir accès aux informations concernant ses systèmes et équipements, ce qui n'est pas l'objet de la couche graphique d'un système de supervision. Il est nécessaire de mettre en place un outil supplémentaire permettant d'accéder aux bases de données du processus et de détecter les modifications devant avoir un impact au niveau des IHMs. La mesure de la satisfaction des utilisateurs nécessite de garder une trace des modifications ayant dues être appliquées aux IHMs après leur première implémentation. Il ne s'agit pas d'un besoin satisfait par les outils de développement. Il est nécessaire d'utiliser un outil d'archivage des différentes versions d'IHMs, et d'en mettre en place un autre permettant de mesurer la fréquence et la nature des modifications. C'est seulement lorsque ces différentes versions sont analysées qu'il est possible de mesurer la satisfaction des utilisateurs des IHMs et de modifier si nécessaire le guide de développement qui a servi à leur première implémentation. Ce besoin souligne à nouveau l'importance d'utiliser un outil de développement permettant de guider la conception des IHMs, et que ce guide soit évolutif. Toutefois, il s'agit d'un point qui n'est pas couvert par les outils commerciaux.

III. 2. 3 Synthèse

Les outils et les méthodes utilisées pour développer des IHMs sont nombreux et très aboutis. Les outils sont performants, ils facilitent la construction d'IHMs en permettant la programmation graphique et en minimisant les phases de codage. Certains d'entre eux sont particulièrement flexibles et permettent ainsi leur adaptation aux besoins propres à des domaines spécifiques. En outre, ils proposent parfois des fonctionnalités permettant de guider le développeur d'IHMs dans son travail. Ces outils sont donc intéressants sous plusieurs aspects, mais ont toutefois certaines limites :

- il n'est pas encore possible à des développeurs non formés à la programmation de concevoir entièrement ces IHMs. La spécification de comportements et règles complexes n'est pas possible sans codage si des extensions spécifiques ne sont pas développées ;
- même si des guides élémentaires sont proposés (par personnalisation de l'outil), ceux-ci ne permettent pas d'assurer que toutes les IHMs construites par leur intermédiaire seront standardisées et respecteront des propriétés complexes prédéfinies. De plus, ces guides ne peuvent pas évoluer sans de nouvelles phases de codage ;
- l'évolution de l'outil en fonction de l'évolution du processus ainsi que de la satisfaction des utilisateurs des IHMs implémentées n'est pas possible.

En ce qui concerne les guides de développement et modèles existants, ceux-ci fournissent aux développeurs des conseils leur permettant de savoir ce qu'il doivent faire et ce qu'ils doivent éviter. La notion de « patron » est maintenant largement utilisée dans

le domaine de la conception d'IHMs, et permet la mise à disposition de solutions validées pour résoudre des problèmes fréquemment rencontrés. Les patrons représentent des connaissances de conception éprouvées dans ce contexte plus riche que les simples « instructions » de développement. Toutefois, ceux-ci sont très nombreux ce qui rend difficile le choix de l'un d'entre eux dans une situation spécifique. De plus, ils sont souvent très génériques, et non conçus pour répondre à des besoins propres à des domaines d'applications spécifiques. Par ailleurs, il est à noter que les travaux effectués dans le domaine des patrons ne traitent pas de leur exploitation et de leur mise à jour par des outils de développement d'IHMs. Inversement, les outils existants ne proposent pas de fonctionnalités aidant les développeurs à construire des IHMs respectant des patrons présélectionnés. Il existe de nombreux patrons utiles aux développeurs d'IHMs, il existe aussi des outils de développement puissants, mais le lien entre ces deux domaines n'est pas traité. En conséquence, le travail du développeur d'IHMs n'est pas simplifié autant qu'il pourrait l'être, et le résultat obtenu n'est pas toujours celui attendu. C'est toujours le développeur d'IHMs qui sert d'interface entre les patrons et l'outil : il implémente les IHMs à l'aide de l'outil en veillant à ce que les patrons soient respectés.

Le tableau III.2 résume l'intérêt des travaux effectués dans le domaine de la conception et du développement d'IHMs (outils, standards, modèles) par rapport aux critères d'évaluations définis dans l'introduction de ce chapitre.

	Outils de développement	**Standards et modèles (conseils, patrons, …)**
Développement par interface graphique	Oui, partiellement	Non
Guide de développement - spécification - conformité - flexible et évolutif	 Oui, partiellement Oui, partiellement Non	 Oui Non Non
Prise en compte du retour expérimental	Non	Non
Prise en compte des évolutions du processus	Non	Non

Tableau III. 2 : Evaluation des outils et modèles de développement d'IHMs

Ce tableau montre que les outils et les modèles existants, pourraient, une fois combinés, répondre de façon satisfaisante aux besoins exprimés pour le logiciel SEAM sous deux aspects :

- le développement d'IHMs de supervision par interface graphique ;
- la spécification et le respect de guides de développement.

Le besoin de faire évoluer l'outil de développement, ainsi que les guides qu'il utilise, en fonction des évolutions du processus à superviser et de la satisfaction des utilisateurs, n'est quant à lui pas traité. Les techniques propres au domaine du développement d'IHMs n'apportent donc pas de solution complète à la problématique de cette thèse. Toutefois, les travaux étudiés soulignent la nécessité d'utiliser un guide de développement pour obtenir des interfaces utilisateurs performantes et exploitables par les utilisateurs. Ces

guides (modèles, patrons…) doivent être combinés à des environnements et outils de façon à faciliter le développement d'IHMs et à garantir le résultat obtenu.

La section suivante a pour but de présenter les approches utilisées pour produire des familles de produits logiciels possédant des aspects communs, ce qui constitue une partie des besoins exprimés dans le cadre de cette thèse (points 2 et 3 de la section II.4.1). Les travaux étudiés dans cette section ne concernent pas le développement d'IHMs en particulier mais le développement logiciel en général. L'objectif de l'étude est donc de mettre en évidence les méthodes et outils de développement de familles de logiciels applicables au domaine du développement d'IHMs de supervision.

III. 3 Définition d'architectures pour le développement de familles de produits

Comme cela a été mentionné dans la section II.4.1, l'outil SEAM doit fournir un guide flexible et évolutif permettant de générer des familles d'IHMs. En effet, la supervision d'un seul accélérateur nécessite la mise en place simultanée d'un ensemble d'IHMs possédant des caractéristiques communes. La production de familles d'applications logicielles possédant des caractéristiques semblables n'est pas une problématique propre au domaine des IHMs, mais elle est au contraire commune à tous les secteurs d'activités. Ce thème, les lignes de produits, fait donc l'objet de nombreux travaux. La section suivante présente les bénéfices et les concepts relatifs au développement de lignes de produits et introduit l'intérêt de l'utilisation d'architectures logicielles dans ce contexte.

III. 3. 1 Introduction aux lignes de produits

Les méthodologies de conception logicielles sont généralement construites pour supporter le développement d'applications individuelles. La définition de méthodes permettant le développement de lignes de produits, ou familles d'applications logicielles apparentées, est le résultat de motivations économiques et pratiques : il est trop coûteux de construire tous les membres possibles d'une famille ; il est plus rentable de construire des composants et de développer les membres de la famille à partir de ceux-ci [Lopez-Herrejon et Batory 2001]. En effet, les entreprises devant introduire de nouveaux produits, ou ajouter de nouvelles fonctionnalités à des produits existants en respectant de très cours délais, ne peuvent se permettre de les re-développer un par un.

Une ligne de produits logiciels est un ensemble de systèmes logiciels partageant des fonctionnalités communes qui satisfont les besoins spécifiques d'un segment de marché particulier, et qui sont développés à partir d'éléments de base communs en suivant un processus prédéfini [Clements 2002]. Chaque produit de la ligne de produits est formé en prenant des composants de la base d'éléments communs, en les adaptant si nécessaire via des mécanismes de variation prédéfinis (paramétrage, héritage, …), en ajoutant les nouveaux composants qui peuvent être nécessaires, et en les assemblant conformément aux règles de l'architecture de la ligne de produit [Bergey et al. 2003].

La manière classique de développer une famille de produits est la suivante : construire un programme qui fonctionne, puis prendre des décisions de conception, et

modifier le programme fonctionnant pour créer le membre suivant de la famille. Chaque décision de conception peut soit contraindre soit étendre l'ensemble des programmes possibles de la famille. Il s'agit d'une technique où les programmes membres de la famille sont développés successivement et sont donc des ancêtres directs les uns des autres. Ceci implique que les nouveaux programmes peuvent hériter d'aspects indésirables de leurs ancêtres. Une variante de cette approche permet d'obtenir des sous-familles au sein d'une famille principale. Les membres de la famille ne sont pas nécessairement des descendants directs, mais sont basés sur un ancêtre commun. Il existe de plus des solutions alternatives comme le « raffinement pas à pas » ou encore la « spécification de modules » [Parnas 2001]. Ces méthodes se concentrent sur l'évolution des applications logicielles dans le temps. Dans ce contexte, la famille d'applications est considérée comme une suite de versions bâtie sur un programme initial. Toutefois, dans la problématique de cette thèse, une famille d'IHMs est un ensemble d'IHMs développées et déployées simultanément, et non pas une série de versions d'une même IHM. L'évolution reste un aspect crucial de cette problématique, mais elle concerne l'évolution de la famille d'IHMs, et non de celle d'une seule IHM de départ (dont les versions successives formeraient une famille).

Un aspect intéressant des lignes de produits est la nécessité de définir une « architecture » de ligne de produits. Une architecture de ligne de produits (PLA : Product-Line Architecture) est un plan pour créer une famille d'application apparentées [Batory 1998]. De nombreuses méthodologies ont été mises en place pour créer des architectures de lignes de produits (ex : [America et al. 2000], [Batory et Geraci 1997], [Bosch 1999], [DeBaud et Schmid 1999], [Gomaa et al. 1994], [Weiss et Lai 1999]).

Les outils logiciels jouent un rôle important dans la mise en place des lignes de produits. Un sondage effectué par [Cohen 2002] auprès d'un échantillon d'entreprises développant des lignes de produits indique que ceux-ci sont principalement utilisés aux niveaux de la gestion des besoins ([Requisite Pro], [Doors], …), de la modélisation de la conception ([Rational Rose], [Rhapsody], …) et de la gestion de configuration ([ClearCase], [CCC][*], …). En outre, ce sondage met en évidence les critères de succès dans le développement de lignes de produits, parmi ceux-ci figurent :

- l'expertise du domaine : les entreprises doivent avoir une expérience significative dans le développement de produits avant d'adopter une approche ligne de produits ;
- la coopération dans le processus de développement : les développeurs doivent coopérer dans l'implémentation des éléments constituants les produits et dans celle des produits eux-mêmes ;
- l'excellence architecturale : le succès d'une ligne de produits implique qu'elle soit bâtie sur une architecture qui supporte les variations découvertes dans la compréhension des domaines relatifs à la ligne de produits. L'architecture ne doit pas être simplement un « plan » de construction des produits, mais doit aussi inclure un mécanisme pour organiser le développement logiciel ;
- l'engagement de la direction de l'entreprise : l'équipe de direction (de l'entreprise ou du projet) doit considérer les demandes individuelles de livraison de produits sans sacrifier les concepts de la ligne de produits.

[*] CCC : Change and Configuration Control

Si plusieurs aspects sont cruciaux pour le succès de la mis en place d'une ligne de produits, la priorité doit être accordée à la définition d'une architecture de base [Cohen 2002]. Celle-ci doit être basée sur la compréhension du domaine d'application et des besoins. Elle permet de définir l'espace solution et donne un cadre pour le développement des composants et des produits. Un investissement insuffisant dans ce domaine mènerait à l'implémentation de composants ne pouvant pas fonctionner correctement ensemble, à la non satisfaction des critères de qualité, et a une mauvaise maintenabilité. La définition d'une architecture fiable est donc essentielle pour développer une famille, ou ligne, de produits logiciels. [Van der Hoek 2002] souligne cet aspect et traite dans ses travaux de l'utilisation de langages de description architecturale (ADL : Architectural Description Languages) pour la représentation d'architectures de lignes de produits (les principaux ADLs sont présentés en III.3.4). La section III.3.2 décrit l'état de l'art dans le domaine de la description d'architectures logicielles. La section III.3.3 présente ensuite le sous domaine des architectures logicielles dédié au développement de familles d'applications, il s'agit de la description et l'utilisation des styles architecturaux. Enfin, les sections III.3.5 et III.3.6 présentent les outils existants permettant la définition des architectures et des styles, leur exploitation, et la gestion de leur évolution.

III. 3. 2 Architectures logicielles

Comme l'a montré le paragraphe précédent, le développement d'une famille de produits à l'aide d'un guide de développement utilisant l'expertise d'un domaine d'application, implique la considération d'une solution orientée architecture. Les formalismes de description d'architectures logicielles fournissent des moyens de décrire les propriétés fonctionnelles, structurelles et comportementales que les applications doivent satisfaire (caractéristiques des IHMs dans le contexte de cette thèse : cf. section II.4.2).

Les paragraphes suivants présentent les concepts de base du développement architectural ainsi que les bénéfices généraux que leur utilisation apporte.

III. 3. 2. 1 Définition

Un problème critique dans la conception et la construction de tout système logiciel complexe est son architecture, c'est à dire l'organisation des éléments qui composent le système. Une bonne architecture peut aider à garantir que le système va satisfaire les besoins primordiaux dans des domaines tels que la performance, la fiabilité, la portabilité, et l'interopérabilité. Au contraire, une mauvaise architecture peut avoir des conséquences désastreuses sur le système [Garlan 2000]. C'est pourquoi, au cours de la dernière décennie, les architectures logicielles ont fait l'objet d'une attention croissante, elles sont ainsi devenues un important sous domaine de l'ingénierie logicielle.

Plusieurs définitions du terme « architectures logicielles » sont proposées dans la littérature [Boasson 1995][Bass et al. 1997][Garlan et al. 1992][Perry et Wolf 1992], mais il est largement accepté qu'une architecture soit définie par un ensemble de composants (ex : filtres, objets, bases de données, serveurs, etc.) ainsi que par la description des interactions entre ceux-ci (ex : appels de procédures, envoi de messages,

émission d'événements, etc.) [Garlan et Shaw 1993]. Celle-ci permet de spécifier les caractéristiques d'un système en définissant : quels types de modules sont des composants du système ; combien de composants de chaque type peut il y avoir ; comment les composants interagissent [Rapide 1997] ; quelles propriétés structurelles et comportementales doivent être respectées.

La description architecturale d'un système informatique permet en outre de spécifier :

- sa structure : éléments de traitement et leurs interactions (pas de détails d'implémentation) ;
- son comportement : fonctionnalités et protocoles de communication, dynamisme, évolution ;
- ses propriétés globales (exemples : sécurité, vivacité…).

Le paragraphe suivant présente comment sont spécifiés ces différents aspects des architectures logicielles.

III. 3. 2. 2 Formalisme

Les architectures de la plupart des systèmes logiciels ont longtemps été décrites informellement, au moyen de diagrammes représentant les composants logiciels par des « boîtes » et leurs interactions par des « lignes ». La description informelle d'architectures induit des difficultés au niveau de son interprétation et possède donc un certain nombre de limitations [Abowd et al. 1995]. En effet, pour être exploitable, une architecture doit clairement définir :

- quels traitements les « boîtes » et leurs annotations représentent ;
- est-ce que toutes les « boîtes » ont des comportements similaires ;
- quelles relations de contrôle/flux de données sont indiquées par les « lignes » ;
- comment le comportement global du système est déterminé par celui de ses composants.

De simples diagrammes comportant des annotations ne sont pas suffisamment expressifs pour spécifier une architecture en exprimant toutes ces informations. C'est pourquoi les concepteurs de logiciels qui utilisent de tels diagrammes les interprètent en fonction du contexte d'utilisation. Par exemple, pour décrire une architecture de type « pipe-filter » un diagramme utilisera les « boîtes » pour représenter des « filters » et les « lignes » pour représenter des « pipes ». Pour un autre système, les « boîtes » pourraient représenter des objets et les « lignes » des appels de méthodes. Toutefois, l'interprétation reste imprécise, il est donc impossible de donner à ces diagrammes des significations non ambiguës. Il est alors difficile de déterminer si l'implémentation du système satisfait sa description architecturale. De même, ce manque de précision empêche l'application d'analyses formelles : il est impossible de raisonner formellement sur les descriptions architecturales des systèmes, ou de faire des comparaisons efficaces entre différentes descriptions architecturales. Ainsi, afin de définir plus précisément et de mieux comprendre l'architecture des systèmes logiciels, de nombreux travaux ont proposé d'utiliser une structure permettant d'associer des sémantiques formelles aux diagrammes architecturaux. C'est dans cet objectif que des ADLs ont été développés. Les caractéristiques des principaux ADLs utilisés actuellement seront exposées dans la section III.3.4. L'approche de conception formalisée rendue possible par l'utilisation de

ADLs a de nombreux avantages par rapport aux méthodes informelles, comme par exemple la précision, la capacité à prouver des propriétés, et la possibilité d'analyser la structure.

III. 3. 2. 3 Bénéfices généraux

Les architectures logicielles jouent un rôle important dans au moins six aspects du développement logiciel [Garlan 2000] :

1. *Compréhension :* les architectures logicielles rendent plus facile la compréhension du fonctionnement de systèmes complexe en les représentant à un haut niveau d'abstraction ;
2. *Réutilisation :* les descriptions architecturales supportent la réutilisation à de multiples niveaux. Les travaux actuels dans le domaine de la réutilisation se concentrent généralement sur l'utilisation de bibliothèques de composants. La conception orientée architecture supporte, en plus, la réutilisation de composants complexes et des structures dans lesquelles ces composants peuvent être intégrés. Ceci est prouvé par de nombreux travaux existants dans les domaines des DSSA (Domain-Specific Software Architectures), des architectures de référence, ou des patrons de conceptions [Mettala et Graham 1992][Buschmann et al. 1996] ;
3. *Construction :* une description architecturale fournit un plan de développement en indiquant les composants principaux et les dépendances entre eux ;
4. *Evolution :* l'architecture d'un système peut décrire comment celui-il est censé évoluer. La définition explicite des limites d'évolutions d'un système permet de faciliter sa maintenance et d'estimer plus précisément les coûts des modifications. De plus, les descriptions architecturales distinguent les aspects fonctionnels des composants de la façon dont ces composants interagissent entre eux. Cette séparation permet de modifier facilement les mécanismes de connexion, ce qui favorise l'évolution en termes d'interopérabilité et de réutilisation ;
5. *Analyses :* les descriptions architecturales fournissent des moyens d'analyse, tels que la vérification de la consistance d'un système [Allen et Garlan 1994] [Luckham et al. 1995], la vérification de la conformité aux contraintes imposées par un style architectural [Abowd et al. 1993], la vérification de la conformité à des attributs qualité [Clements et al. 1995], l'analyse de dépendance [Stafford et al. 1998], ainsi que des analyses spécifiques au style à partir les architectures ont été construites [Coglianese et Szymanski 1993][Magee et al. 1995][Garlan et al. 1994] ;
6. *Gestion :* l'expérience a montré que la définition précise d'une architecture logicielle est un facteur clé dans la réussite d'un processus de développement logiciel. L'évaluation d'une architecture mène a une meilleure compréhension des besoins, des stratégies d'implémentation, et des risques potentiels [Boehm et al. 1994].

De plus, l'utilisation d'une conception orientée architecture comporte des avantages pour toutes les personnes intervenant dans le cycle de vie d'un projet logiciel [Upchurch 1995] (Cf. figure III.2).

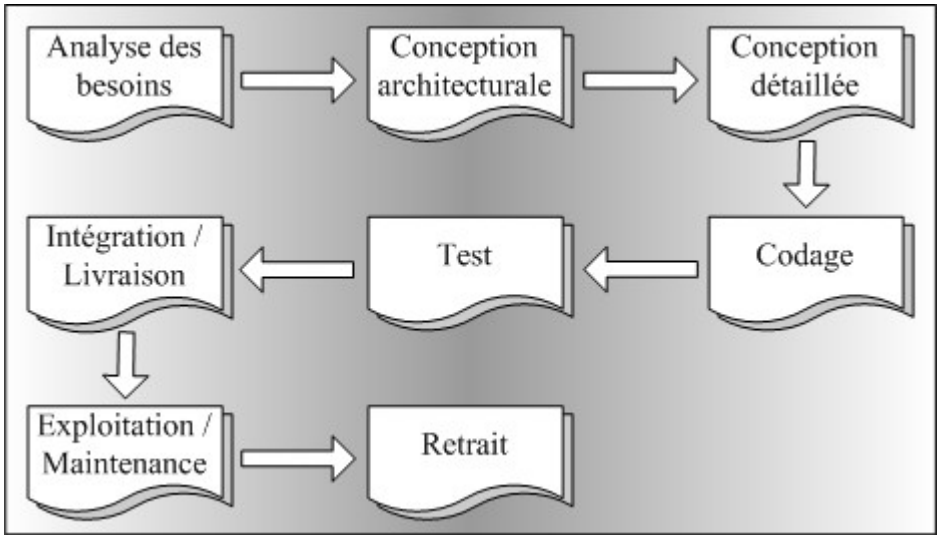


Figure III. 2 : Cycle de vie d'une application logicielle

- **client** : estimation du budget et du temps de développement ; évaluation de la faisabilité et du risque ; traçabilité des besoins ; suivi de progrès.
- **utilisateur** : satisfaction des besoins ; scénarii d'utilisation ; adaptation possible aux évolutions des besoins ; performance, fiabilité, interopérabilité, etc.
- **architecte** : traçabilité des besoins ; support des analyses ; complétude ; consistance de l'architecture.
- **développeur** : suffisamment de détails pour la conception ; référence pour la sélection et l'assemblage des composants ; interopérabilité avec les systèmes existants.
- **mainteneur** : guide pour les modifications logicielles ; guide pour les évolutions de l'architecture.

III. 3. 2. 4 Processus de développement

La figure III.3 représente le processus de développement centré architecture ainsi que les acteurs qui interviennent directement dans celui-ci : l'architecte, le développeur, ainsi qu'éventuellement l'analyste (cette tâche peut être automatisée) [Leymonerie 2004].

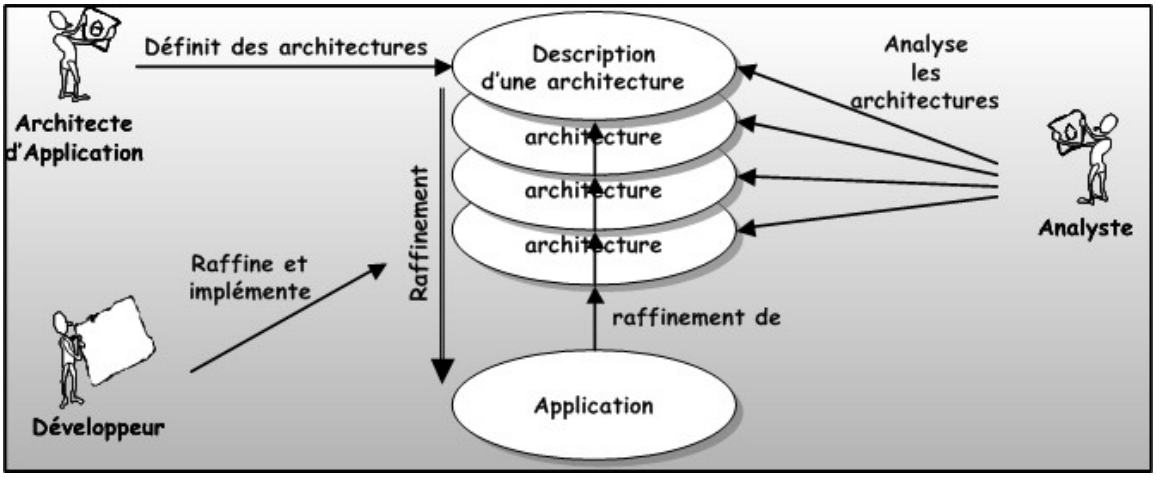


Figure III. 3 : Processus de développement architectural

L'architecte a pour rôle de définir l'architecture qui servira de base au développement de l'application. Le développeur raffine cette architecture de façon à s'approcher petit à petit de l'application finale. Pour cela il implémente les composants ainsi que leurs interactions (les connecteurs) en respectant la structure et les propriétés définies par l'architecture de départ[*]. A chaque étape de raffinement l'analyste est en mesure de vérifier que l'architecture raffinée est conforme à l'architecture du niveau d'abstraction supérieur. Ce processus permet de garantir que l'application obtenue respecte les propriétés fonctionnelles, structurelles et comportementales définies par l'architecte en accord avec le client et les utilisateurs.

L'étude des techniques de développement architectural menée dans cette section indique que celles-ci ont le potentiel d'améliorer le développement des IHMs de supervision en fournissant le cadre nécessaire pour spécifier leur structure et leurs propriétés, ainsi que celles des éléments qui les composent. Ces techniques permettent en outre de s'assurer que les applications obtenues respectent cette structure et ces propriétés. De même, elles rendent possible la gestion de l'évolution des IHMs développées. Toutefois, pour répondre pleinement aux besoins exprimés dans la section II.4, l'outil SEAM doit permettre la spécification graphique des IHMs, la génération automatique de leur code, ainsi que l'utilisation d'un guide de développement. Afin d'évaluer la solution centrée architecture dans ces domaines, les prochaines sections présentent les techniques de définition et d'exploitation des descriptions architecturales. Pour commencer, le paragraphe III.3.3 présente un aspect central de la conception architecturale : l'utilisation des styles architecturaux pour construire des familles d'applications possédant des caractéristiques communes. Il s'agit d'un aspect crucial pour atteindre un des principaux objectifs de l'outil SEAM, la génération de familles d'IHMs de supervision.

[*] Dans les approches formelles, le processus de raffinement peut-être totalement ou partiellement automatisé

III. 3. 3 Styles architecturaux

III. 3. 3. 1 Définition et bénéfices généraux

Un style architectural caractérise une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques [Abowd et al. 1993] (exemples : pipe-filter/client-serveur/RPC). Le but principal des styles architecturaux est de simplifier la conception des logiciels et la réutilisation, en capturant et en exploitant la connaissance utilisée pour concevoir un système [Monroe et al. 1997]. Un style architectural est moins contraignant et moins complet qu'une architecture spécifique. Il spécifie uniquement les contraintes les plus importantes, au niveau par exemple de la structure, du comportement, de l'utilisation des ressources des composants et des connecteurs dans un système…[Abd-Allah 1996].

Un style architectural fournit [Garlan 1995] :

- un vocabulaire : pour spécifier les types spécifiques de composants utilisables ;
- des règles de conceptions ou contraintes : pour spécifier les compositions d'éléments qui sont permises ;
- une interprétation sémantique : pour définir la signification des compositions d'éléments contraintes par les règles de conception ;
- les analyses qui peuvent être appliquées sur les systèmes construits dans ce style.

D'une façon générale, les styles architecturaux permettent à un développeur de réutiliser l'expérience concentrée de tous les concepteurs qui ont précédemment fait face à des problèmes similaires [Klein et Kazman 1999].

En outre, l'utilisation des styles architecturaux comporte des intérêts précis :

- elle promeut la réutilisation au niveau de la conception (et non seulement au niveau du code, comme c'est le cas de la plupart des techniques de réutilisation) [Monroe et Garlan 1996] ;
- elle peut donc aussi mener à une réutilisation significative de code ;
- elle permet de normaliser une famille d'architectures, ce qui améliore la compréhension de l'organisation d'un système ;
- elle permet l'utilisation d'analyses spécifiques au style concerné [Ciancarini et Mascolo 1996].

III. 3. 3. 2 Formalisme

Les travaux concernant les styles architecturaux sont étroitement liés à ceux effectués dans le domaine des patrons de conception (ou « design patterns ») introduits dans la section III.2.2. Les concepteurs de logiciels utilisent fréquemment des patrons informels pour décrire les architectures de leurs systèmes [Shaw 1994]. [Shaw 1995] a identifié plusieurs patrons de haut niveau (ex : architecture en couche, programme principal / sous-routines, invocation implicite, …) guidant la conception de la structure générale de systèmes logiciels. La description de chacun de ces patrons inclue des informations concernant le problème qu'il résout, son contexte d'utilisation, sa solution (c'est à dire le modèle proposé par le patron), le diagramme correspondant, les variantes possibles, ainsi que des exemples d'utilisation. Les informations sont exprimées de façon narrative et se

concentrent particulièrement sur les motivations de l'utilisation du patron. Tout comme les styles architecturaux, de tels patrons permettent de caractériser des familles d'applications ayant des caractéristiques communes. Toutefois, ces patrons ne sont pas exploitables par les langages de programmation conventionnels. De même, les styles architecturaux ont longtemps été définis et utilisés comme des guides de conception informels, n'ayant pas de langages dédiés à leur exploitation. Les travaux menés depuis lors ont souligné l'intérêt de formaliser les styles [Abowd et al. 1995], et de nombreux langages dédiés à cette tâche ont été développés. Ces langages, appelés ASLs (Architectural Style Langages), sont des ADLs proposant des formalismes et des outils permettant de définir et d'exploiter efficacement les styles architecturaux. Leur utilisation comporte de multiples bénéfices : le fait de permettre de décrire formellement l'architecture de systèmes logiciels rend possible l'utilisation de mécanismes (cf. section III.3.3.4) pour représenter les architectures, leur analyse, leur raffinement, et la génération automatique du code correspondant. L'utilisation de styles architecturaux formalisés dans le contexte de l'outil SEAM est donc plus appropriée que celle de simples patrons, même si, comme le soulignent [Shaw et Clements 1996], il est intéressant de combiner les avantages des deux approches. Il est par exemple bénéfique d'ajouter aux descriptions formelles des styles certaines informations permettant de les organiser en bibliothèque pour les réutiliser plus efficacement en connaissant quels problèmes ils résolvent, dans quels contextes ils doivent être utilisés, etc.

III. 3. 3. 3 Processus de développement

La figure III.4 présente un processus de développement général incluant la définition formelle et l'exploitation de styles architecturaux [Leymonerie 2004]. Ce schéma reprend et étend le processus de développement architectural représenté par la figure III.3.

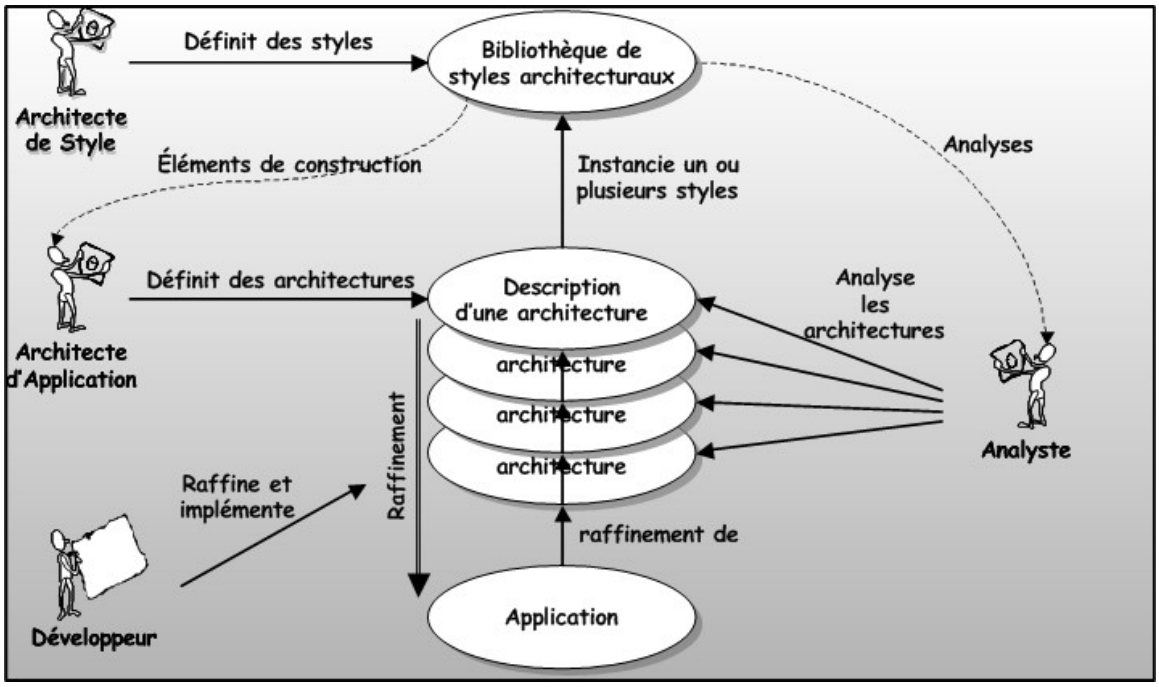


Figure III. 4 : Processus de développement architectural utilisant les styles

La première étape de ce processus est la formalisation du style architectural ou des styles architecturaux. Au fur et à mesure des définitions de styles une bibliothèque peut être définie. L'architecte d'application instancie alors le ou les styles qui correspondent à ses besoins spécifiques et les utilise comme éléments de construction pour définir une architecture de base. La suite du développement est identique au processus précédemment étudié : le développeur raffine petit à petit l'architecture de base en architectures de plus en plus concrètes jusqu'à l'obtention de l'application finale. Il est toutefois à noter que l'utilisation d'un tel processus de développement permet à l'analyste d'appliquer aux différentes architectures des analyses définies par les styles utilisés. Les analyses disponibles ne sont donc plus uniquement des analyses génériques, applicables à toutes les architectures. Elles peuvent aussi être des analyses spécifiques à un domaine d'application particulier, applicables uniquement aux architectures définies à l'aide des styles correspondant à ce domaine.

III. 3. 3. 4 Bénéfices des styles dans la production d'IHMs

Un processus de développement centré sur la définition et l'exploitation d'un style architectural pour développer une famille d'IHMs de supervision possède donc plusieurs bénéfices. En effet, l'utilisation des styles architecturaux simplifie la construction des systèmes, réduit le coût d'implémentation en promouvant la réutilisation, et par dessus tout, améliore l'intégrité des systèmes au moyen d'analyses et d'outils spécifiques au style utilisé [Monroe et al. 1997]. Le développement architectural permet de réutiliser des concepts déjà validés, comme par exemple les prototypes d'IHMs présentés en II.3, et garantit que les futures applications vont respecter les propriétés précédemment validées [Leymonerie et al. 2001]. De plus, l'utilisation d'un style architectural guide le développement d'une famille de systèmes logiciels (ex : IHMs) en fournissant un vocabulaire architectural commun utilisé pour décrire les structures des systèmes, et en contraignant l'utilisation de ce vocabulaire [Allen 1996]. L'adoption d'une approche utilisant les styles architecturaux apporte donc des solutions pour au moins quatre des cinq besoins listés dans l'introduction de ce chapitre, en effet, un tel processus de développement permet de faciliter l'implémentation des IHMs en :

- contraignant la construction de ces IHMs (par la spécification et la vérification de propriétés comme celles listées dans la section II.4.2) ;
- produisant des IHMs standardisées partageant des propriétés graphiques, structurelles et comportementales communes ;
- capturant le savoir faire du domaine et en réutilisant l'expérience acquise (réutilisation des concepts utilisés pour les prototypes d'IHMs).

D'autres aspects de ces besoins, comme la spécification des IHMs au moyen d'une interface graphique ou encore la génération automatique de code, peuvent être satisfaits par certains ADLs accompagnés de leurs outils. L'étude des principaux ADLs existants et le choix du plus approprié dans le contexte de cette thèse sont exposés dans la section III.3.4.

III. 3. 3. 5 Principaux styles existants

L'intérêt de l'utilisation des styles architecturaux ayant été mis en évidence, il est maintenant nécessaire d'étudier les styles déjà existants afin de déterminer si certains d'entre eux résolvent des problématiques semblables à celle exposée dans cette thèse. C'est l'objet du prochain paragraphe.

[Garlan et Shaw 1993] ont répertorié plusieurs styles architecturaux parmi ceux qui ont été les plus fréquemment utilisés, avant même que des formalismes efficaces aient été disponibles pour faciliter leur définition et leur utilisation. La liste obtenue comprend les styles suivants :

- Systèmes événementiels, invocation implicite ;
- Filtres et tuyaux (pipes and filters) ;
- Systèmes orientés objet ;
- Systèmes en couches ;
- Processus distribués ;
- Programme principal – sous routines ;
- Systèmes de type état/transition.

De même, [Shaw et Clements 1997] ont identifié plusieurs styles génériques très couramment utilisés et les ont classés en deux catégories :

- styles du type « flux de données » : dominés par la notion de transmission de données à travers un système (ex : réseaux de transmission de données, pipelines, « filtres et tuyaux », etc.) ;
- styles du type « processus interagissants » : dominés par les modèles de communication entre des processus indépendants et/ou concurrents (« processus communicants », « client-serveur », « broadcast », etc.).

Cette classification, ainsi que la plupart des travaux effectués dans ce domaine, se basent sur le fait que la grande majorité des styles est descriptible en termes de composants et de connecteurs. La catégorie des styles de type « flux de données » correspond aux styles ayant des composants de type « convertisseur de données » connecteurs de type « transmission de données ». La catégorie des styles du type « processus interagissant » correspond quant à elle aux styles ayant des composants de type « processus » et des connecteurs de type « protocole de transmission de message ». Une telle classification permet de déterminer rapidement quelles sont les caractéristiques de tel ou tel style et facilite donc le choix du style approprié pour résoudre un problème donné.

Les exemples de styles cités jusqu'à maintenant permettent de satisfaire des besoins génériques, ils caractérisent de très grandes familles de logiciels. En conséquence, même s'ils fournissent un guide de développement basique, ils ne sont pas suffisamment précis et contraignants pour pouvoir servir de guide de développement de logiciels spécifiques à un domaine d'application particulier (comme le développement d'IHMs de supervision). C'est pourquoi de nombreux architectes ont défini des architectures et des styles propres à leur secteur d'activité, adaptés à leurs propres besoins (DSSA : « Domain-Specific Software Architectures »). En outre, certains styles ont été définis dans les domaines particuliers des systèmes de contrôle de processus et des interfaces homme-machine.

Concernant le développement de systèmes de contrôle de processus [Savigni et Tisato 1999] ont défini une architecture de référence, nommée Kaléidoscope, dédiée au développement de systèmes de contrôle et de supervision. Il s'agit d'une architecture en couches basée sur la séparation de la communication, du traitement, et de la stratégie dans le but de favoriser la modularité et la réutilisation [Savigni 2000]. Cette architecture de référence est elle aussi décrite en terme de composants et de connecteurs. Les « images conceptuelles » sont les composants d'information qui modélisent les concepts du domaine d'application, et sont spécialisés en « images concrètes », comme l'acquisition, le traitement et la représentation des données. L'approche a été expérimentée avec succès dans la construction de deux systèmes importants. Partant du constat que tous les systèmes de contrôle et de supervision présentent un certain nombre de caractéristiques communes, les auteurs considèrent que cette architecture de référence, conçue au départ pour le développement de systèmes de contrôle de trafic routier [Savigni et Tisato 2000], peut être appliquée pour concevoir des systèmes de contrôle et de supervision dans un large panel de domaines d'applications. Toutefois, cette architecture de référence, ou style architectural, est conçue pour permettre le développement de systèmes de supervision complets, alors que la problématique de cette thèse se situe uniquement au niveau de la couche de représentation des données. Kaléidoscope fournit donc un style de trop haut niveau d'abstraction pour répondre précisément aux besoins de construction des IHMs de supervision d'accélérateur de particules.

En conséquence, il est intéressant de considérer les travaux architecturaux effectués dans le domaine du développement d'IHMs. [Taylor et al.] ont défini un style architectural (Chiron-2 , ou C2) de type « composant et message » dédié à la construction d'interfaces graphiques. Constatant que, dans ce domaine, la réutilisation était souvent limitée à celle d'outils (« toolkits ») existants, ils ont mis en place un style flexible supportant une réutilisation de plus « gros grain ». Ce style supporte la conception d'applications distribuées et concurrentes. Les composants qu'il définit interagissent au moyen de messages asynchrones de requête et de notification. Ce style a été éprouvé dans la construction de plusieurs applications. Il s'agit d'un style très général utilisable dans de nombreux domaines d'application. Toutefois son objectif est différent de celui décrit dans cette thèse. En effet, ce style est dédié à la description d'architectures d'interfaces utilisateurs complexes, comportant de nombreux composants (comme des fenêtres de dialogue) interagissant avec plusieurs utilisateurs et utilisant de multiples medias. L'objectif du style devant être défini pour prendre en charge la construction des IHMs GTPM est à la fois plus simple et plus ciblé : il s'agit principalement de contraindre la disposition et le comportement des éléments graphiques représentant l'état d'un processus. Le style C2 ne correspond donc pas précisément à la problématique décrite dans le chapitre 2.

Les besoins concernant les IHMs de supervision à générer étant très spécifiques, il est normal qu'aucun des styles développés dans d'autres contextes, pour répondre à d'autres besoins, ne satisfasse entièrement la problématique de cette thèse. Il s'avère donc nécessaire de formaliser un style dédié à la construction des IHMs de supervision des accélérateurs du CERN. Comme cela a déjà été mentionné précédemment, une telle définition se fait au moyen d'un ADL. La section suivante présente les principaux ADLs existants et les évaluent en fonction des besoins exprimés.

III. 3. 4 Langages de Description d'Architecture (ADL)

Les Langages de Description d'Architecture (ADL) et leurs outils soutiennent le développement centré architecture [Medvidovic et Taylor 1997]. Afin de choisir celui qui, parmi eux, est le plus adapté à la formalisation d'un style architectural décrivant une famille d'IHMs évolutive, il est nécessaire de classer et de comparer les principaux ADLs existants. La classification présentée dans cette section est basée sur les travaux de [Leymonerie et al. 2002].

Pour effectuer cette classification il est important de considérer pour chaque ADL :

- le contexte dans lequel a été développé ;
- les concepts d'architecture et de style architectural qu'il exploite ;
- la façon dont il gère les mécanismes de styles tels que l'instanciation, l'héritage ou le raffinement.

Il est tout d'abord à noter que chaque ADL supporte au moins un style, celui-ci représentant la sémantique et les mécanismes de l'ADL [Leymonerie et al. 2002]. Ce style est plus ou moins générique selon le domaine de l'ADL, par exemple : META-H est un ADL spécifique aux architectures des systèmes multiprocesseurs temps réel pour l'aviation [Binns et al. 1996] ; ACME [Garlan et al. 1997] et π-SPACE [Chaudet et Oquendo 2001] ne sont pas spécifiques à un domaine, leur style propre est plus générique. Afin d'augmenter le niveau de réutilisation et de s'adapter à n'importe quel domaine, quelques ADLs permettent aux utilisateurs de formaliser leurs propres styles.

Concernant l'instanciation des styles, il est intéressant de considérer notamment si les ADLs offrent la possibilité d'instancier plusieurs styles, et s'ils prennent en charge l'héritage multiple. Le raffinement peut être considéré de deux façons : la première est d'ajouter de l'information ; la seconde est de développer la sous structure des composants.

Un style définit les propriétés ou les contraintes qui fixent des règles et des limites de construction. Il existe trois principales classes de contraintes : les contraintes ***topologiques***, ***comportementales*** et les contraintes d'***attribut*** :

- les contraintes ***topologiques*** précisent les éléments de construction utilisables, leur nombre d'occurrences possibles au sein de l'architecture, ainsi que les règles de configuration contraignant leurs interactions. Par exemple, avec AML [Wile 1999] on peut écrire : si *t* est *n-top*, *b* est *n-bottom* et *t* est « *au-dessus de* » (*above*) *b*, alors ces éléments peuvent être attachés :

```
-relationship attached-to (b : n-bottom, t : n-top)
   assume above(t,b)
```

- les contraintes ***comportementales*** concernent l'évolution temporelle de l'architecture et le comportement des éléments architecturaux. Concernant l'évolution temporelle de l'architecture, un style peut spécifier comment l'architecture peut évoluer dynamiquement – quelle sorte d'élément peut apparaître ou disparaître – et quelles contraintes les architectes doivent respecter en modifiant une architecture à la volée. Les styles peuvent aussi spécifier des propriétés de mobilité propres à des éléments architecturaux, et des propriétés

comportementales (exemple : pas d' « étreinte fatale[1] ») – ces propriétés peuvent être celles du comportement d'une configuration entière ou seulement d'un élément architectural. Par exemple, avec σπ-SPACE [Leymonerie et al. 2001] on peut spécifier qu'un serveur reçoit parallèlement des requêtes qu'il traite et envoie des réponses :

```
-specification
 { Serveur [port_any] =
  %<|>
  port_name(requete)●traitement[in(requete),out(reponse)
  ]●port_name<reponse>
 %}
```

- les contraintes d'***attributs*** concernent les aspects non structurels et non fonctionnels d'une architecture. Les attributs apportent des informations sur les éléments architecturaux. Ils peuvent être contraints sur leur type, leur nom et leur gamme de valeurs. Les styles peuvent spécifier comment les valeurs des attributs sont liées avec d'autres aspects architecturaux (la structure et le comportement). Par exemple, avec ARMANI [Monroe 1998] on peut écrire :

```
-Property buffersize : int;
-Invariant buffersize >= 0;
```

Les ADLs permettent de spécifier d'autres informations au sein des formalisations de styles afin de pouvoir les exploiter. Ces informations peuvent être : des analyses d'architecture, des règles de traduction pour la génération automatique de code, une syntaxe spécifique au style, une visualisation des éléments architecturaux, des outils spécifiques ou de la documentation.

Dans ce contexte, les ADLs étudiés sont les suivants :

- ACME : c'est un langage générique d'« interchange » qui permet de décrire les structures architecturales et d'ajouter une sémantique à ces structures. Il décrit des composants, des connecteurs, des ports, des rôles et des configurations. ACME supporte aussi la définition des styles et permet d'assurer le respect des contraintes de conception à l'aide de ses outils. ARMANI [Monroe 1998] est une extension d'ACME conçu pour modéliser les contraintes architecturales. Dynamic ACME [Wile 2001] est une extension permettant de décrire la dynamicité des architectures ;
- ADAGE [Coglianese et Szymanski 1993] : ce langage permet la description de structures architecturales pour le guidage et la navigation dans le domaine de l'aviation ;
- AESOP [Garlan et al. 1994] : il permet de construire des architectures avec des environnements spécifiques générés à partir de description de style ;
- AML (Architectural Meta-Language) [Wile 1999] : il s'agit d'un métalangage pour la spécification de la sémantique des ADLs. Il décrit une architecture comme un ensemble d'éléments (*element*) liés par des relations (*relationship*) soigneusement décrites et contraintes ;
- ArchWare-ADL / C&C [Oquendo 2003][Cimpan et al. 2003] : ArchWare-ADL fournit la structure de base et les constructions comportementales pour décrire les

[1] Dead-lock en langue anglo-saxone

architectures logicielles dynamiques. C'est un langage formel de spécification conçu pour être exécutable et pour supporter l'analyse et le raffinement automatique des architectures. ArchWare-ADL spécialise le π-calcul [Milner et al. 1992] pour permettre la description de comportements dynamiques. L'extension C&C, construite à partir de ArchWare-ADL, constitue une base pour la définition d'autres styles (Cf. section V.2) ;

- DARWIN [Magee et al. 1995] : il supporte l'analyse de systèmes de transmission de messages distribués ;
- META-H [Vestal 1992] : il fournit un guide pour décrire, analyser, et implémenter les architectures de systèmes de contrôle temps réel embarqués dans le domaine de l'aviation ;
- π-SPACE [Chaudet et Oquendo 2001] est un ADL pour les architectures dynamiques permettant la description de comportements architecturaux. σπ-SPACE [Leymonerie et al. 2001] est son extension pour la description des styles architecturaux. π-SPACE et σπ-SPACE utilisent le π-calcul pour la description de comportements dynamiques ;
- RAPIDE [Luckham et al. 1995] : ce langage décrit les interfaces des composants et les communications entre celles-ci ainsi que les modèles événementiels. Il permet de simuler les architectures et propose des outils pour analyser les résultats de ses simulations. Il ne présente pas de mécanisme propre à la formalisation des styles ;
- SADL [Moriconi et al. 1995][Moriconi et Riemenschneider 1997] : il permet la description d'architectures spécifiques et d'architectures paramétrées. Il fournit en outre une base formelle pour le raffinement architectural ;
- UNICON [Shaw et al. 1995] : c'est un ADL qui permet la construction de systèmes à partir de la description de leur architecture. Il possède un compilateur de haut niveau qui supporte l'utilisation de types de composants et de connecteurs hétérogènes. UNICON-2 [DeLine 1996] est une extension pour la prise en charge des styles architecturaux. Il permet la génération de code pour une grande variété de styles ;
- WRIGHT [Abowd et al. 1993][Allen et Garlan 1997] : il s'agit d'un modèle formel pour les architectures logicielles, il est largement utilisé pour l'analyse des protocoles d'interaction entre les composants architecturaux. Cet ADL permet la définition des comportements au moyen d'une algèbre formelle, le CSP [Hoare 1985]. Le CSP permet de décrire des comportements en terme de patrons d'évènements. En comparaison au π-calcul utilisé dans σπ-SPACE et ArchWare-ADL, il ne permet pas la description de comportement dynamique. Les extensions de WRIGHT [Allen 1997][Allen et al. 1998] permettent d'une part, la définition des styles architecturaux d'une manière similaire à ARMANI, et d'autre part, la description (limitée) du dynamisme des architectures.

Le paragraphe III.3.4.1 présente les caractéristiques de ces ADLs concernant la description d'architectures, le paragraphe III.3.4.2 expose les aspects liés à la description des styles architecturaux, et le paragraphe II.3.4.3 présente les outils en environnements de développement associés aux ADLs.

Il existe d'autres ADLs, qui, ayant contexte plus éloigné de celui de cette thèse, ne sont pas étudiés ici. Par exemple, [Inverardi et Wolf 1995] proposent de modéliser les architectures avec des Machines Chimiques Abstraites, CHAM (Chemical Abstract Machine). Une CHAM permet de spécifier un comportement, en s'appuyant sur la comparaison avec une succession de réactions chimiques qui peuvent se produire au sein d'une solution. Les solutions peuvent représenter des architectures ou des configurations architecturales alors que les molécules représentent des éléments architecturaux. Par ailleurs, un certain nombre de styles architecturaux ont été caractérisés en Z [Spivey 1989], un langage de spécification pour la description de modèles formels. [Abowd et al. 1995] ont élaboré une méthode structurée pour définir les styles architecturaux en Z. Ainsi, les styles sont définis en trois parties : le vocabulaire spécifique au style, le modèle sémantique, et un ensemble d'interprétations sémantiques permettant le passage de la syntaxe au modèle sémantique.

III. 3. 4. 1 Formalisation d'architectures

Ce paragraphe résume par les tableaux III.3a et III.3b les différentes notions associées aux principaux ADLs dans le cadre de la définition d'architectures. L'objectif est de présenter ce qu'est une architecture, et de quoi elle est composée, pour chacun de ces ADLs. Il est a noter que pour la plupart des ADLs étudiés, l'architecture est considérée comme une **configuration** de **composants** (représentant des fonctionnalités ou des données du systèmes) et de **connecteurs** (représentant les protocoles de communication et permettant la connexion entre les composants). Les tableaux indiquent par ailleurs si les ADLs permettent la description de comportements et d'attributs, et s'ils gèrent la dynamicité et la mobilité des éléments architecturaux au sein de l'architecture. L'analyse des différents aspects des principaux ADLs décrite dans ce paragraphe ainsi que dans le suivant, va servir de base à leur comparaison présentée paragraphe III.3.4.4.

ADLs	**ACME / ARMANI**	**AESOP**	**AML**	**ArchWare C&C**	**META-H**
Architecture	*system*	*system*	*element*	*component* composite	*macros*
Elément de base (englobant la fonctionnalité)	*component*	*component*	*element*	*component*	*process*
- interface	*port*	*port*	Un type (*kind*) de port peut être défini	*port*	*in port* *out port*
Elément de connexion	*connector*	*connector*	Un type (*kind*) de connecteur peut être spécifié	*connector*	*connection*
- interface	*role*	*role*	Un type (*kind*) peut être défini	*port*	
Attachement	*attachment*	Méthode *attach*	*attached-to*	*attachement* *Bind*	*connection* <-
Comportement			Quantificateurs temporels : *always, sometimes*	*behaviour* basé sur le π-calcul	
Attributs	Annotation		Prédicats logiques	*attributes*	*attributes*
Implémentation	Annotation	*representation*		*wrappers*	*implementation subprogram* et langage Control-H
Composition	*system / representation*	*configuration / representation*	Crée une relation « *has-part* »	*component* composite	*mode-macros*

Dynamicité	Dynamic ACME utilise *open* pour déclarer les éléments pouvant être instanciés dynamiquement			Les attachements peuvent être créés dynamiquement. Les éléments architecturaux peuvent être instanciés à l'exécution.	
Mobilité				Les components, connectors, et ports peuvent transiter via les attachements.	

Tableau III. 3a : Caractéristiques des ADLs concernant la définition d'architectures

ADLs	**σπ-SPACE**	**SADL**	**RAPIDE**	**UNICON-2**	**WRIGHT**
Architecture	*composite*	*architecture*	*architecture*	*component*	*system*
Elément de base (englobant la fonctionnalité)	*component*	*component*	*component*	*component*	*component*
- interface	*port*	*interface*	*interface*	*player*	*port*
Elément de connexion	*connector*	*connector*	*connections*	*connector*	*connector*
- interface	*port*	*interface*		*role*	*role*
Attachement	*attach…to…*	*connection*	*connections*	*connect*	*attachment*
Comportement	*behavior* avec π-calcul		*behavioural constraints*		*computation (comp), glue(conn)* avec CSP
Attributs				Annotation	
Implémentation	*operation*			*primitive implementation*	
Composition	*composite*	*configuration*	*architecture*	*composite implementation*	*configuration*
Dynamicité	Tous les éléments sont instanciables dynamiquement		Rapide permet la définition de règles décrivant des connections dynamiques		Dynamic WRIGHT, permet de simuler la dynamicité par l'ajout d'évènements *control* dans les composants
Mobilité					

Tableau III. 3b : Caractéristiques des ADLs concernant la définition d'architectures

III. 3. 4. 2 Formalisation des styles architecturaux

Au travers des ADLs précédemment cités, les styles sont formalisés de façons différentes, comme **types génériques**, comme **ensemble de classes** dans une approche orientée objet, comme **constructeurs**, ou encore comme **ensemble de types**. Les tableaux III.4a et III.4b présentent les mécanismes de définition des styles propres à chacun des ADLs, ainsi que les contraintes pouvant être exprimées. Ils indiquent par ailleurs si ces langages gèrent la dynamicité et la mobilité des éléments et s'il proposent des mécanismes de génération de code et des outils d'exploitation (ce dernier point est détaillé dans la section III.3.5). Certains des ADLs présentés dans la section III.3.4.1.1 ne figurent pas dans ces tableaux car ils ne proposent pas de mécanismes particuliers permettant à un utilisateur de définir des propres styles (ex : META-H propose un seul style pour décrire uniquement les systèmes de contrôle dans le domaine de l'aviation).

ADLs	AESOP	ACME	ARMANI	AML
Style	*style* = classe	*family* = liste de *type* + structure requise	*style* = liste de *type* + structure requise + contraintes + analyses	*kind* = élément générique
Vocabulaire	Ensemble de classes	Ensemble de *types*	Ensemble de *types*	Ensemble de *kinds*
Contraintes	Exprimées par des méthodes de classe	Inclure structure requise à l'instanciation	*invariants* et *heuristics* [2](prédicats)	Exprimées par des relations (*relationship*)
- Topologiques	Supportées par des méthodes de classe	*template* (constructeurs)	*invariants* et *heuristics*	Cardinalités dans *relationship* (« *has-part* » et « *attached-to* »)
- Comportementales				Quantificateurs temporels
- D'attributs			*invariants* et *heuristics*	
Instanciation	Architecture créé avec l'environnement	Par des *systems*	Par des *systems*	Par un *element*
Héritage	Seulement ajouter des causes d'erreur	Ajout de nouveaux types et extension de la structure requise – héritage multiple possible		Ajout de relations respectant les relations existantes
Dynamicité		Dynamic ACME	ArchStudio le permet	
Mobilité				
Génération de code				
Outils d'exploitation	Génération d'environnements de développement, visualisation	Librairies, environnement de développement, analyses, …	Vérificateur de contraintes, environnement de développement	

Tableau III. 4a : Caractéristiques des ADLs concernant la définition de styles

ADLs	UNICON-2	σπ-SPACE	ArchWare ADL
Style	*duty* de *player*, *role*, *interface*, *protocole*, ou *configuration*	*style* de *port*, *component*, *connector*, *composite*	*style*
Vocabulaire		Ensemble de *styles*	*archetypes* et *styles* définis par l'utilisateur
Contraintes	3 clauses : *requires*, *provides* et *closes*	Mécanismes propres à chaque type de contrainte	*constraints* définies par des propriétés AAL[3]
- Topologiques	3 clauses : *requires*, *provides* et *closes*	- cardinalités - attachements possibles	Propriétés structurelles
- Comportementales		Spécification générique du comportement	Propriétés comportementales
- D'attributs	3 clauses : *requires*, *provides* et *closes*		Valeurs des propriétés
Instanciation	Par un *type*	Par un *type*	Par un élément architectural
Héritage	Clause *includes*	Ajout de contraintes	Un style peut-être étendu pour obtenir un sous-style (ajout de contraintes, d'éléments architecturaux, de constructeurs)
Dynamicité		Tous les éléments sont instanciables dynamiquement	Tous les éléments sont instanciables dynamiquement, le nombre d'instanciations possibles pouvant être limité en utilisant des contraintes
Mobilité			Les connections, ports, composants et connecteurs sont mobiles
Génération de code	Génération automatique de code pour l'implémentation	Génération automatique de code pour l'architecture	Mécanismes de raffinement, génération de code, hypercode
Outils d'exploitation	Compilateur		Environnement de développement, analyses, visualisation, …

Tableau III. 4b : Caractéristiques des ADLs concernant la définition de styles

[2] Exprimés par des prédicats, les *invariants* sont des contraintes strictes et les *heuristics* sont des propriétés qu'il est conseillé de suivre.
[3] Architecture Analysis Language

AML, σπ-SPACE et UNICON-2 et ArchWare-ADL proposent de définir un style comme un **type générique**. En AML, un élément architectural suivant un *kind* doit respecter les relations que ce dernier établit. En σπ-SPACE, on peut avoir des styles pour les composants, les connecteurs, les composites et les ports ; ils définissent des contraintes topologiques et comportementales régissant la structure interne de leurs instances. Les mécanismes de définition de contraintes sont différents selon leur type. UNICON-2 propose, à travers des *duties*, des structures requises et des patrons de construction à respecter. Ces styles peuvent être spécialisés sachant qu'un sous-style peut seulement ajouter des contraintes à son parent.

Le modèle d'AESOP est basé sur l'**approche objet**. Les éléments de constructions sont définis par des classes et les contraintes par des méthodes de classe qui retournent des erreurs lorsqu'elles ne sont pas respectées. AESOP génère des environnements à partir de styles pour la conception d'architectures spécifiques à ses styles. Les instances de style sont les architectures construites avec l'environnement généré. Les mécanismes d'héritage sont ceux de l'approche objet, avec un sous-typage strict concernant les conditions de succès des méthodes (représentant les contraintes).

Une première version d'ACME propose de formaliser les styles comme un ensemble de « *template* », qui sont des **constructeurs paramétrables**. La dernière version présente le style comme une « **famille** » (*family*) qui consiste en un **ensemble de types** et une **structure requise**. Une famille peut être instanciée par une architecture ou un système. Leurs définitions peuvent utiliser les types proposés et doivent contenir la structure requise. Il est possible d'instancier plusieurs familles, dans ce cas seule la structure requise d'une famille est prise en compte. Le mécanisme d'héritage consiste en l'extension d'une famille avec de nouveaux types ou une structure requise plus détaillée. ARMANI y ajoute la possibilité de définir des contraintes à travers des invariants et des heuristiques (logique des prédicats). En outre, ARMANI propose l'aspect « élément générique » et l'aspect « bibliothèque ».

III. 3. 4. 3 Outils et environnement architecturaux

Etant donné que la conception d'architectures et de styles architecturaux est devenue une discipline importante de l'ingénierie logicielle, il a été nécessaire de développer des outils et des environnements permettant la description, l'analyse et l'exploitation d'architectures.

Les outils architecturaux sont principalement des :

- éditeurs et visualisateurs graphiques et textuels ;
- outils d'analyse statique ;
- outils d'analyse dynamique, utilisés pour l'exécution d'architectures (simulation et supervision) ;
- outils d'implémentation (interfaces ou générateurs de code) ;
- outils permettant l'évolution d'architectures ;
- outils de génération de documentation ;
- outils de « traduction » de descriptions architecturales entre différents ADLs.

La plupart des langages décrits dans la section précédente possèdent des outils ou des environnements de développement permettant de définir et d'exploiter des descriptions

architecturales. Une équipe de développement en a répertorié les principaux afin de définir une boîte à outils complète ([ADL Toolkit]) pour la prise en charge des ADLs.

Ainsi, le langage [META-H], dédié à la description d'applications d'aviation temps réel embarquées [Vestal 1994], est associé à des outils permettant :

- la visualisation et l'édition graphique et textuelle d'architectures (outil DoMe) ;
- la vérification des propriétés décrites ;
- l'analyse de fiabilité et l'analyse de sûreté. La première analyse les dysfonctionnements potentiels d'une architecture, et la seconde vérifie si les objets déclarés moins « sûrs » peuvent affecter le bon fonctionnement des objets déclarés plus « sûrs » ;
- la génération du code gérant les processus temps réel, la communication, et la synchronisation des ressources.

Le langage [RAPIDE] propose des outils permettant :

- l'édition graphique et textuelle (outil RapArch) ;
- la compilation et l'exécution des descriptions architecturales ;
- l'animation de l'exécution de l'architecture dans un environnement graphique temps réel (outil Raptor);
- la vérification de contraintes.

La boite à outils d'[UNICON] comporte un compilateur et un générateur de code pour l'infrastructure de communication des architectures écrites en UNICON.

ARMANI possède quant à lui :

- un vérificateur de contraintes architecturales pour régir l'évolution dynamique (à l'exécution) de systèmes logiciels décrits dans le style C2 ;
- un environnement de conception graphique interactif (ArchStudio) permettant la spécification, la visualisation, et l'évolution dynamique d'architectures logicielles suivant le style C2.

Plusieurs outils et librairies ont été développés pour prendre en charge la description architecturale avec le langage ACME, parmi ceux-ci :

- une librairie (AcmeLib, disponible en Java et en C++) de classes réutilisables pour représenter et manipuler les architectures ACME ;
- un environnement de développement incluant un éditeur graphique et un vérificateur de règles ([AcmeStudio]). Cet environnement permet en outre d'éditer et de créer des styles ;
- un outil de génération de documentation au format HTML ;
- un outil permettant de manipuler, d'analyser et de générer des descriptions d'architectures avec PowerPoint. Cet éditeur permet lui aussi la spécification de styles et des propriétés associées ;
- un outil d'analyse d'architectures (ACME PowerPoint Analysis Package) qui communique avec la description PowerPoint ce qui permet l'analyse dynamique de l'architecture créée ;
- un outil d'analyse dynamique (ACME PowerPoint Dynamic Analysis Package) permettant la supervision et la simulation d'architectures décrites en « Dynamic Acme ».

Concernant les environnements de développement, la plupart d'entre eux ont été créés pour construire des architectures à partir de styles spécifiques (ex : l'environnement Armani pour le style C2 cité précédemment). Ces environnements fournissent des outils servant de support à des concepts particuliers de conception architecturale et aux méthodes de développement associées. Par exemple, des environnements ont été développés pour mettre en œuvre des architectures basées sur les flux de données [Mak 1992], la conception orientée objet [Rumbaugh et al. 1991], les systèmes de contrôle [Binns et Vestal 1993], et l'intégration réactive [Fromme 1989].

Malheureusement, ces environnements spécifiques sont construits les uns indépendamment des autres. Une telle approche a un coût élevé, car chaque environnement de développement architectural ne supporte qu'un seul style. Afin de palier à ce problème [Garlan et al. 1994] ont conçu le langage [AESOP] qui permet de générer des environnements de développement architectural à partir de la description d'un style architectural. Dans ce langage, les descriptions architecturales sont représentées par un modèle objet générique, les styles sont obtenus par spécialisation de ce modèle. Lorsque l'architecte a créé le style correspondant à son domaine d'application particulier, il peut utiliser les outils d'AESOP pour obtenir un environnement de développement correspondant. Ces outils incluent d'ailleurs une interface graphique permettant aux utilisateurs de visualiser, d'éditer, et d'utiliser les descriptions architecturales.

[ArchWare]-ADL possède un environnement de développement complet (Cf. section V.2.4) incluant notamment :

- un compilateur et un moteur d'exécution d'architectures ;
- un éditeur graphique (basé sur UML) et un éditeur textuel permettant la définition et la manipulation d'architectures ;
- un animateur graphique ;
- des outils de vérification de propriétés (statiques et dynamiques) ;
- un raffineur et un générateur de code ;
- un outil (customizer) permettant de vérifier que les architectures définies à partir de styles architecturaux spécifiques, via l'environnement ArchWare, respectent bien les contraintes spécifiées.

Le paragraphe suivant a pour but de synthétiser les caractéristiques des différents ADLs étudiés et des outils associés de façon à pouvoir sélectionner celui qui est le plus adapté à la problématique de cette thèse.

Les différents ADLs ainsi étudiés peuvent maintenant être comparés suivant les deux aspects cruciaux pour la conception et la génération de familles d'IHMs. Il s'agit tout d'abord de la capacité à formaliser des contraintes, et ensuite, de l'utilisation des styles formalisés.

III. 3. 4. 4 Comparaison des ADLs

L'objet de cette section est de comparer ces ADLs par rapport aux besoins relatifs à la création de familles d'IHMs possédant des caractéristiques communes (Cf. chapitre 2). Comme cela a été expliqué précédemment, l'utilisation de styles architecturaux est l'aspect essentiel du développement architectural permettant la construction de familles d'applications logicielles. Pour pouvoir permettre de développer des familles d'IHMs, l'ADL choisi doit donc impérativement permettre à un utilisateur de formaliser ses propres styles architecturaux, il s'agit là du premier et principal critère de sélection. Certains des ADLs étudiés (ex : RAPIDE, META-H, …) ne seront pas considérés dans cette section car ils ne satisfont pas cette condition.

Le style architectural devant être développé doit contraindre la construction des IHMs. Les contraintes (ou propriétés) à formaliser sont de plusieurs types :

- contraintes concernant la structure des IHMs ;
- contraintes à appliquer aux valeurs des attributs, notamment celles des attributs graphiques ;
- contraintes concernant le comportement et le traitement des IHMs (acquisition de données, traitement des états, …).

L'objet du paragraphe suivant (III.3.4.4.1) est de positionner les ADLs en fonction de leur capacité à formaliser ces trois types de contraintes.

Un autre aspect crucial dans le choix de l'ADL le plus approprié est celui des outils et des environnements de développement associés. En effet, pour être exploitable, l'ADL choisi doit posséder de tels outils, permettant notamment les analyses des architectures et la génération de code. Par ailleurs, les IHMs étant en permanente évolution, il est important que l'ADL propose des mécanismes prenant en charge la dynamicité au sein des architectures. Le paragraphe III.3.4.4.2 positionne donc les différents ADLs en fonction de ces aspects.

III. 3. 4. 4. 1 Formalisation des contraintes

Concernant les contraintes structurelles, tous les ADLs sont capables de les formaliser, mais ARMANI est le plus puissant grâce à l'utilisation d'une logique des prédicats pour l'expression des contraintes. Concernant le comportement, seuls σπ-SPACE, AML et ArchWare-ADL sont adaptés. σπ-SPACE et ArchWare-ADL offrent un meilleur support grâce à l'utilisation d'une algèbre formelle pour la description des processus. Concernant les contraintes sur les attributs, ARMANI est encore une fois le plus adapté grâce à l'utilisation des prédicats. UNICON-2 et ArchWare-ADL sont eux aussi efficaces pour l'expression de ce type de contraintes. La figure III.5 permet de situer les ADLs les uns par rapport aux autres dans un espace tridimensionnel où les axes sont associés aux trois types de contraintes. Sur cette figure, ainsi que sur la III.6, les axes sont gradués de 0 à 1 : la valeur 0 indique que le langage concerné ne rempli pas les fonctionnalités requises ; la valeur 0,5 indique que seule une partie ce ces fonctionnalités est disponible ; la valeur 1 indique que le langage fournit la totalité des fonctionnalités nécessaires à la formalisation et à l'exploitation d'un style architectural propre au domaine d'application de cette thèse.

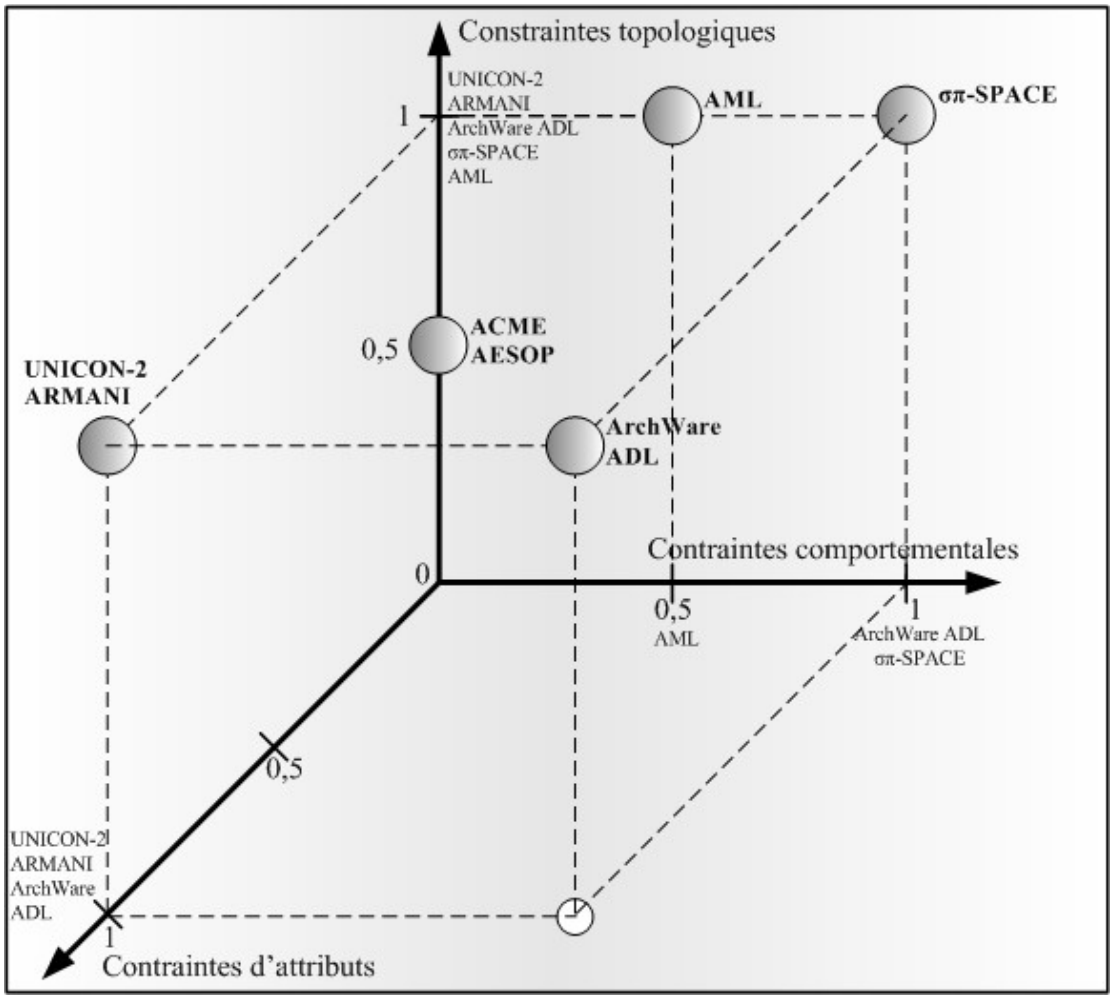


Figure III. 5 : Positionnement des ADLs pour la gestion des contraintes

III. 3. 4. 4. 2 Dynamicité et exploitation des styles formalisés

On peut exploiter de plusieurs façons un style formalisé. La manière de formaliser les styles est plus ou moins adaptée à chaque type d'utilisation. Un style peut être utilisé pour générer un environnement spécifique, comme c'est le but d'AESOP. Un style peut permettre de générer un langage ; AML semble être le plus adapté pour cela. Un style peut servir de bibliothèque, contenant des éléments plus ou moins génériques et réutilisables. Enfin, un style peut être utilisé comme un patron structurel associé à un attribut qualité, comme proposé par les « Attribute-Based Architectural Styles » (ABAS) [Klein et Kazman 1999]. Dans ce cas l'architecte construit son architecture en choisissant les styles suivant les qualités requises par les différentes parties du système à modéliser. Ce dernier point n'est pas pris en compte par les ADLs étudiés. La section III.3.4.3 a présenté les outils et les environnements d'exploitation des différents ADLs. La figure III.6 positionne les différents ADLs en fonction de ce critère ainsi que de la possibilité de générer une partie du code des applications et de gérer la dynamicité des architectures.

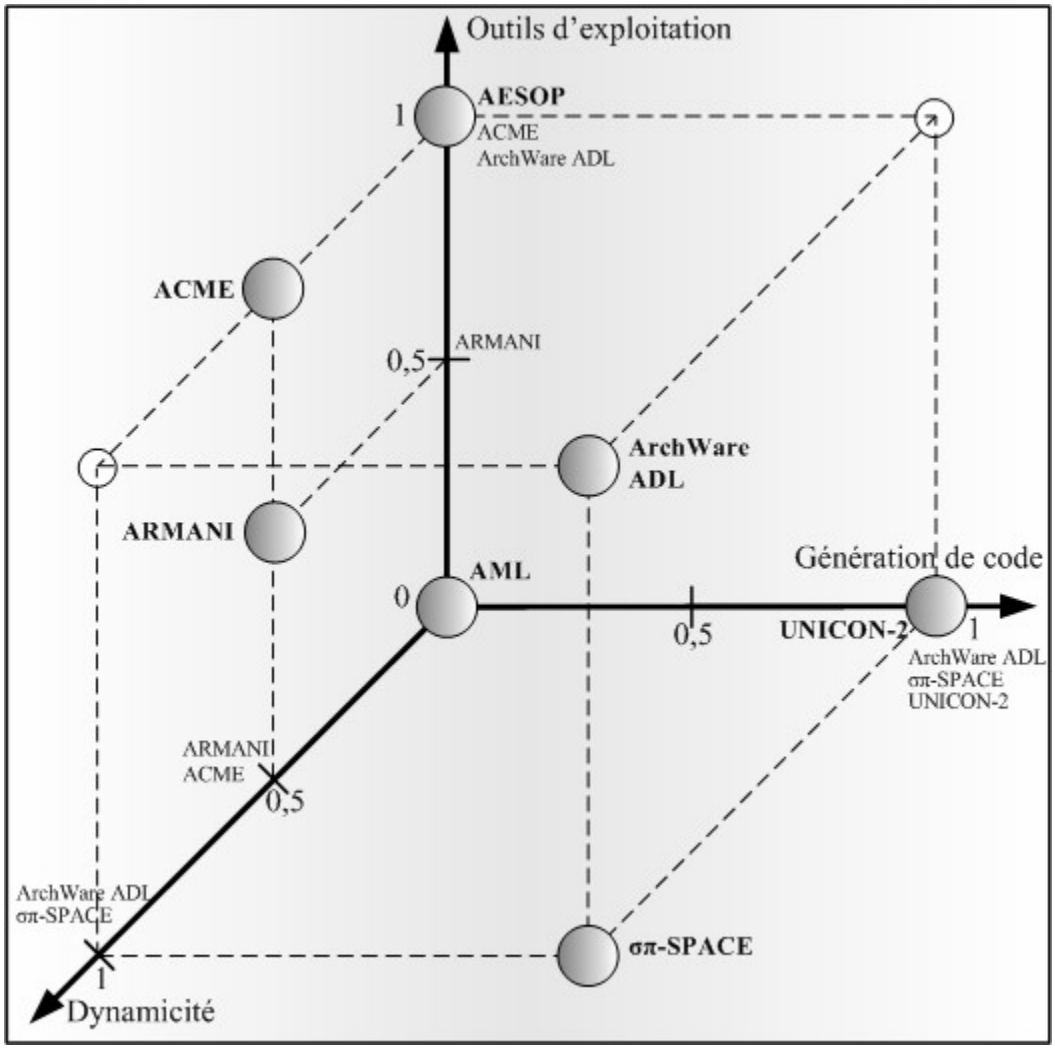


Figure III. 6 : Positionnement des ADLs pour l'exploitation des styles et la dynamicité

III. 3. 4. 4. 3 Conclusion

Les figures III.5 et III.6 donnent les indications permettant de conclure que ArchWare-ADL est le langage le plus adapté pour la construction de familles d'IHMs de supervision.

Tout d'abord, celui-ci possède les mécanismes appropriés pour décrire efficacement les styles architecturaux, notamment au moyen de son extension C&C.

Concernant la gestion des contraintes, même si ARMANI est le plus performant concernant les contraintes topologiques et d'attributs, ArchWare-ADL est le seul qui, en plus d'être efficace dans ces deux domaines, permet la spécification de contraintes comportementales.

La figure III.6 indique par ailleurs que ACME, ARMANI, AESOP et ArchWare-ADL possèdent des environnements d'exploitation complets, mais aussi que, parmi ceux-ci seul ArchWare-ADL propose des outils de raffinement et de génération de code. Par ailleurs, ACME et ARMANI possèdent des extensions et outils permettant de simuler et

visualiser la dynamicité des architectures, mais ArchWare-ADL, de par sa construction sur le π-calcul, est le plus efficace dans ce domaine.

III. 3. 5 Evolution des architectures et des styles

Une fois qu'une application logicielle a été mise en service, elle entre dans une phase d'exploitation plus ou moins longue au cours de laquelle elle devra vraisemblablement être maintenue et évoluer. La mise en place de techniques permettant la gestion de l'évolution des architectures joue donc un rôle important dans la majorité des domaines d'application, et notamment dans celui de cette thèse. De plus, lorsque ces architectures sont construites à l'aide d'un guide de développement (style architectural, ligne de produits), il est nécessaire d'assurer l'évolution de ce guide en fonction des nouveaux besoins qui pourraient apparaître, afin de pouvoir faire évoluer l'ensemble de la famille d'applications associée. Toutefois les travaux menés sur ce thème sont relativement rares. Le but de cette section est de présenter les contributions proposant des solutions pour gérer l'évolution des architectures et des styles architecturaux.

III. 3. 5. 1 Evolution des architectures

Le thème de l'évolution des architectures logicielles peut se diviser en deux aspects principaux : permettre aux architectures d'évoluer, et superviser et mesurer leur évolution.

Concernant le premier point, [Zenger 2002] explique que l'extensibilité est un besoin à respecter pour permettre l'évolution d'architectures. Il fournit une liste de conseils à suivre dans le développement des composants d'une architecture pour s'assurer que celle-ci sera extensible et donc évolutive.

Les concepteurs de langages de description architecturale se sont aussi penchés sur ce besoin d'évolution des architectures. Certains ADLs actuels, notamment ArchWare-ADL, permettent la description d'architectures évoluant dynamiquement.

Le deuxième aspect, la supervision de l'évolution des architectures, est encore une discipline émergeante. La plupart des travaux concernent la supervision des architectures, et non de l'évolution des architectures. Ainsi, [Blazer 1997] présente une méthode pour superviser les architectures au moyen de l'animation. Cette technique permet la supervision du comportement des architectures au moyen de l'instrumentation de leurs connecteurs.

Cependant, même si la supervision de l'évolution des architectures est un thème encore peu traité, de nombreux travaux ont abouti au développement d'outils dédiés à l'analyse d'architectures [Luckham et al. 1995][Shaw et al. 1995][Naumovich et al. 1997]. Ces outils constituent une base pour superviser les architectures en fournissant des informations sur leur structure et leurs propriétés. Par exemple, une technique semi-formelle a été développée pour comparer des descriptions architecturales dans le but de mettre en évidence leurs différences et leurs similarités [Inveraldi et Wolf 1995]. Une telle technique peut-être exploitée pour superviser l'évolution des architectures par comparaison de leurs versions successives. Ainsi [Jazayeri 2002] a utilisé des métriques et des outils d'analyses appropriés pour analyser rétrospectivement les versions successives d'architectures, et déterminer comment celles-ci évoluent.

De plus, la supervision est souvent combinée au contrôle. Le contrôle de l'évolution des architectures implique la possibilité de modifier, d'ajouter, ou de supprimer des éléments qui les composent. Ceci amène la question de l'analyse d'impact de telles modifications. Certains résultats intéressants sont disponibles à ce sujet, comme par exemple la technique de chaînage, qui génère une matrice de dépendance permettant l'analyse de l'impact d'une modification architecturale [Stafford et al. 1998][Stafford et al. 1997].

Ainsi, les techniques actuelles permettent aux architectures d'évoluer. L'évolution dynamique (à l'exécution) des architectures d'IHMs n'entre pas, dans un premier temps, dans la problématique de cette thèse. Toutefois, les IHMs devront évoluer par développements itératifs, et il est essentiel de pouvoir maîtriser, mesurer et analyser cette évolution, ce qui est rendu possible par les travaux présentés dans ce paragraphe.

III. 3. 5. 2 Evolution des styles architecturaux

Les travaux concernant l'évolution des styles architecturaux sont encore très peu nombreux. Toutefois, un style architectural étant un modèle permettant de construire des familles d'applications ayant des caractéristiques communes, son objectif est similaire à une ligne de produits, domaine dans lequel l'évolution a été traitée. D'ailleurs, la plupart des travaux concernant les lignes de produits soulignent le besoin de faire évoluer leurs architectures. Parmi ceux-ci, [Akash et al . 2003] proposent un environnement, nommé « Ménage », gérant l'évolution des architectures de lignes de produits. Ses fonctionnalités principales sont la spécification de points de variation, et le traçage de l'évolution de l'architecture de la ligne de produits et de ses éléments constituants par la gestion de leurs différentes versions. [Svahnberg et Bosch 1999] ont analysé les résultats de deux études de cas menées dans des entreprises utilisant les architectures de lignes de produits depuis plusieurs années. Ils ont identifié et classé en catégories les évolutions des besoins, des architectures des lignes de produits, et des composants de celles-ci. Ces travaux soulignent le fait que l'évolution d'une ligne de produits logiciels est conduite par les changements des besoins concernant les produits de la famille. En effet, dans la situation la plus courante, les utilisateurs des produits expriment de nouveaux besoins, et c'est une équipe de développement logiciel qui prend en compte ceux-ci et implémente une nouvelle version d'un produit ou fait évoluer la ligne de produit. S'il existe des conseils et des guides pour prendre en compte les besoins concernant les produits au niveau de la ligne de produits, il n'existe pas d'outil ou de mécanisme prenant en charge ce travail.

Dans le contexte de cette thèse, l'architecture de la ligne de produits (style architectural permettant la création d'IHMs), est fournie aux utilisateurs qui s'en servent pour créer eux-mêmes leurs IHMs. Lorsque les besoins des utilisateurs évoluent, ceux-ci possèdent les outils pour modifier eux-mêmes les IHMs. Afin de garantir la cohérence et la pertinence de la ligne de produits (ou style), il est nécessaire de fournir un support permettant de la faire évoluer en fonction des modifications appliquées aux IHMs. Les travaux existants dans le domaine de l'évolution des styles architecturaux ne permettent pas de répondre à ce besoin.

III. 3. 6 Bénéfices de l'approche centrée architecture

L'adoption d'une approche de développement centrée architecture apporte de nombreuses solutions à la problématique de création de familles d'IHMs de supervision de redémarrage d'accélérateurs de particules. En effet, l'utilisation d'un ADL pour définir un style architectural servant de guide à la construction des IHMs a le potentiel de répondre à la plupart des besoins exprimés dans l'introduction.

Le paragraphe III.3.4.4.3 a indiqué que l'ADL le plus adapté au contexte de cette thèse est celui proposé par le projet ArchWare. En effet, le langage ArchWare comporte les caractéristiques lui permettant d'être utilisé pour définir un style propre à un domaine spécifique, comme celui de la supervision des accélérateurs. L'intégration de ce style dans un environnement de développement facilitera la construction d'IHMs en permettant de les développer de façon entièrement graphique. En outre, l'environnement de développement associé au style permettra de guider les développeurs d'IHMs dans leur travail, et de garantir que les IHMs obtenues satisfont l'ensemble des propriétés définies par le style. Même si les techniques actuelles ne permettent pas de faire évoluer facilement le style en fonction de l'évolution du processus à superviser et des nouveaux besoins, certains outils architecturaux constituent une base solide pour arriver à ce résultat.

Le tableau III.5 résume l'intérêt de cette solution par rapport aux critères d'évaluations définis dans l'introduction de ce chapitre.

Développement par interface graphique	Oui, au moyen de l'environnement de développement paramétré par le style architectural.
Guide de développement - spécification - conformité - flexible et évolutif	 Oui. Oui. Oui, partiellement.
Prise en compte du retour expérimental	Retour mesurable par analyse de l'évolution des architectures. La prise en compte de ce retour au niveau du style reste à développer.
Prise en compte des évolutions du processus	

Tableau III. 5 : Evaluation de l'approche centrée architecture pour le développement d'IHMs

III. 4 Conclusion

Ce chapitre a permis de présenter l'état de l'art dans le domaine des méthodes et outils de développement d'interfaces graphiques et dans celui de l'utilisation des architectures logicielles dans le développement de familles de produits.

L'étude a montré que les méthodes et outils de développement traditionnels possèdent de nombreux aspects intéressants mais sont néanmoins limités dans certains domaines. En effet, ceux-ci permettent partiellement le développement d'IHMs de supervision par interface graphique et la spécification et le respect de guides de développement, mais ils sont peu flexibles et ne peuvent garantir la satisfaction de propriétés complexes. La

spécificité du domaine d'application de cette thèse rend difficile l'utilisation directe d'un outil existant. De même, les patrons habituellement utilisés dans le domaine du développement d'IHMs sont trop génériques pour satisfaire les besoins relatifs à la supervision du redémarrage d'accélérateurs de particules. Par ailleurs, les travaux effectués sur le thème des lignes de produits ne correspondent pas entièrement à la problématique traitée. Toutefois, ceux-ci soulignent que le développement efficace de familles d'applications logicielles est conditionné par la définition d'une architecture de référence adaptée.

La solution choisie est donc de construire l'outil SEAM autour d'une architecture de référence spécifique (DSSA : Domain-Specific Sofware Architecture) au développement d'IHMs satisfaisant les besoins décrits dans le cadre du projet GTPM. Le moyen choisi pour aboutir à ce résultat est l'utilisation des techniques architecturales incluant notamment la définition de styles. En effet, même si les techniques de développement architectural n'ont pas été spécifiquement conçues pour supporter le développement d'IHMs de supervision, le large champ d'application de certains ADLs, tel que celui adopté (ArchWare-ADL), permet de développer des langages (ou styles) spécifiques à ce domaine particulier. L'utilisation d'un tel style permet de garantir la satisfaction des besoins concernant les IHMs, et son intégration dans un environnement adapté autorise une construction entièrement graphique, et donc réalisable par des développeurs non informaticiens.

Ainsi, les objectifs de la thèse sont les suivants :

- proposer une nouvelle approche de conception et de production de logiciels de supervision basée sur l'utilisation et l'exploitation des styles architecturaux ;
- proposer un processus inductif permettant la définition de style architectural à partir d'applications prototypes, et l'évolution du style en fonction de l'évolution des besoins concernant les applications construites à partir de celui-ci ;
- implémenter un environnement dédié au développement de logiciels propres à des domaines spécifiques, respectant les contraintes du style architectural

Les chapitres suivants décrivent la mise en place, l'exploitation et la validation d'un processus de développement utilisant et adaptant les techniques architecturales présentées dans cet état de l'art. Le chapitre 4 présente un concept d'environnement de construction d'IHMs contrôlé par un style architectural dont la formalisation est décrite chapitre 5. Les chapitres 6 et 7 concernent l'implémentation et l'évolution (à partir des techniques présentées section III.3.5) de cet environnement. Le chapitre 8 a pour but de valider l'approche à partir d'études de cas.

Chapitre IV Approche pour des environnements de conception contrôlés par des styles

Chapitre IV : Approche pour des environnements de conception contrôlés par des styles

IV. 1 Introduction

L'étude de l'état de l'art dans le domaine du développement de logiciels de supervision[1] présentée dans le chapitre précédent à permis de mettre en évidence les bénéfices de l'approche orientée architecture, ce qui a conduit à son adoption.

Ce chapitre a pour but de présenter la construction d'une solution intégrant cette approche, en utilisant notamment les styles architecturaux. Le chapitre est structuré en deux parties principales : la première vise à introduire l'utilisation faite des styles architecturaux dans la construction de la solution ; la deuxième présente de façon informelle le style architectural spécifique au développement d'IHMs de supervision d'accélérateurs de particules[2] qui a été défini. Le chapitre 5 présente la formalisation de ce style.

IV. 2 Le rôle des styles architecturaux dans la conception des IHMs

L'utilisation des styles architecturaux d'une manière « explicite » est relativement récente. Il est toutefois possible d'identifier certains processus qui sont le plus souvent adoptés quand les styles architecturaux sont utilisés.

Avant de présenter l'utilisation des styles architecturaux choisie dans le cadre de cette thèse, la section IV.2.1 rappelle brièvement le processus de conception centré style le plus « classique », qui est basé sur une approche déductive, les architectures étant générées à partir de styles reconnus. La section IV.2.2 explique ensuite comment le contexte de cette thèse conduit à l'adoption d'un processus inductif. Puis, la section IV.2.3 détaille davantage le rôle du style dans l'environnement solution, et, enfin, la section IV.2.4 aborde les aspects liés à l'évolution du style et des architectures.

[1] Afin de faciliter la lecture, le terme (réducteur) « IHM » sera utilisé, dans la suite de ce chapitre, pour désigner les logiciels de supervision (ayant une forte composante visualisation) devant être produits dans le cadre de cette thèse.

[2] Afin de faciliter la lecture, le terme « Style GTPM » sera utilisé, dans la suite de cette thèse, pour designer le style architectural spécifique au développement d'IHMs de supervision d'accélérateurs de particules.

IV. 2. 1 Processus centré architecture classique

Le développement architectural permet de formaliser les besoins des utilisateurs en une architecture qui est ensuite utilisée pour obtenir le code d'applications satisfaisant ces besoins. Il existe principalement deux types de processus de développement permettant d'obtenir ce résultat. Dans le premier cas (Cf. Figure IV. 1), les besoins sont directement formalisés en architecture avec la possibilité d'utiliser des composants existants. Ceci est fait à l'aide d'un ADL (Architecturale Description Language). En fonction du langage utilisé, différentes propriétés peuvent être spécifiées et analysées sur l'architecture. Le code des applications (IHMs dans le contexte de cette thèse) est alors obtenu directement à partir de l'architecture. Plusieurs étapes de raffinement peuvent être employées pour rendre l'architecture de plus en plus concrète, jusqu'à rendre possible la génération du code. Chaque étape de raffinement garantit la préservation des propriétés d'un niveau à l'autre. La figure IV.1 présente une vision simplifiée du développement centré architecture.

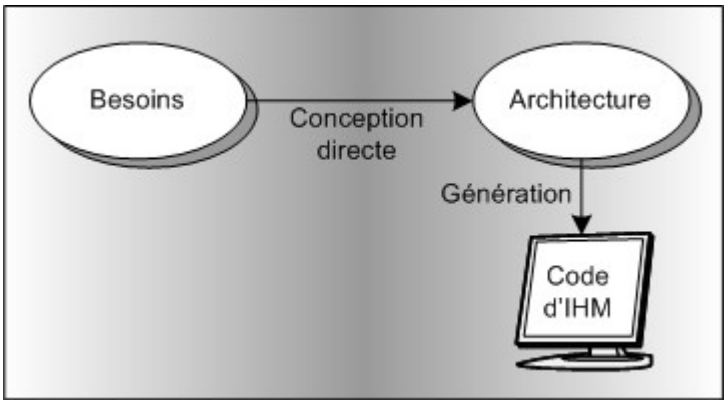


Figure IV. 1 : Processus simplifié du développement centré architecture

Dans le cas du deuxième type de processus de développement (Cf. figure IV. 2), les styles architecturaux sont utilisés comme support à la conception d'architectures. Le style correspondant le plus aux besoins concernant le problème donné est choisi dans une bibliothèque de styles. Le style est alors instancié en architectures respectant les contraintes qu'il impose. Le code des IHMs est ensuite généré à partir de ces architectures. Ce dernier processus de développement favorise la réutilisation de l'expertise de conception et permet d'implémenter une famille d'applications logicielles qui partagent des règles et propriétés communes (Cf. section IV.3).

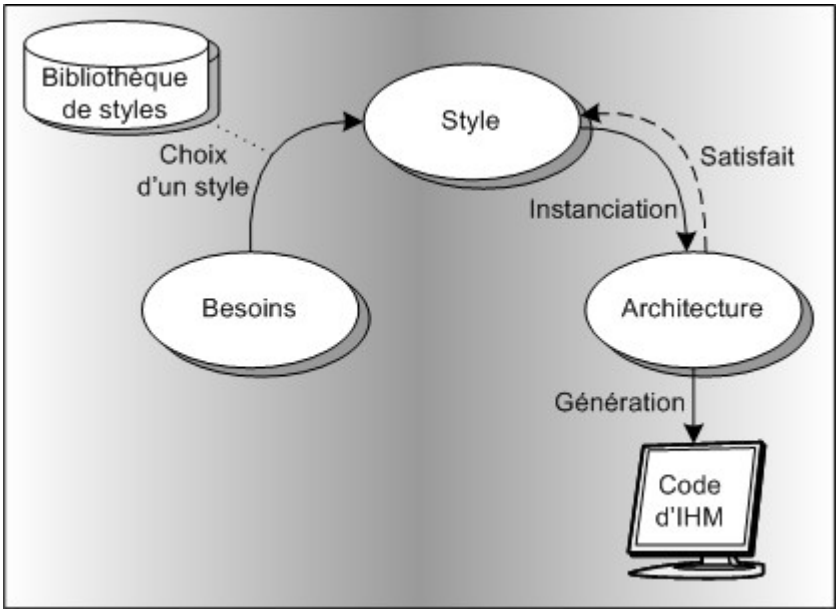


Figure IV. 2 : Processus simplifié du développement orienté style

Dans la pratique il est souvent difficile, voir impossible, de trouver dans la bibliothèque de styles celui qui correspondra aux besoins propres au domaine d'application. Il est alors nécessaire de définir un nouveau style spécifique à ce domaine, soit directement, soit par composition ou raffinement de styles existants. Ceci est particulièrement adapté si le domaine d'application est bien connu et n'évolue pas fréquemment.

Ce processus de développement, essentiellement déductif, suppose toutefois une phase inductive en amont, où l'expertise de conception a été capturée et formalisée sous forme de style. Historiquement parlant, c'est parce que l'existence de l'expertise de conception a été reconnue que des supports pour sa formalisation ont été proposés. Ainsi, actuellement, il existe un certain nombre de styles génériques connus et utilisés selon l'approche déductive présentée.

IV. 2. 2 Définition inductive du style

Dans le domaine du développement d'IHMs pour la supervision d'accélérateurs de particules, aucun style architectural n'était disponible. Il a donc été nécessaire de définir un style propre à ce domaine, permettant de répondre aux besoins présentés dans le chapitre 2.

Comme cela a déjà été mentionné, un style formalise l'expertise de conception acquise dans un domaine particulier. Dans le contexte de cette thèse, cette expertise n'avait pas encore émergé de façon naturelle. Autrement dit, la phase inductive avant le processus déductif présenté dans la section précédente n'avait pas eu lieu. Il a donc été nécessaire de formaliser cette phase inductive pour faire émerger l'expertise de conception.

Le processus inductif pour la définition du style est présenté sur la figure IV.3. La première étape a consisté à spécifier directement à partir des besoins, les architectures des IHMs pour la supervision d'un accélérateur particulier du CERN. Cette étape a été réalisée lors de la phase de prototypage présentée dans la section II.3. Les architectures d'IHMs obtenues ont ensuite été utilisées dans la deuxième étape, qui a consisté à produire une première famille d'IHMs à l'aide d'un outil de développement traditionnel (PVSS). Les IHMs ainsi développées ont été utilisées en salle de contrôle, et, après plusieurs améliorations successives, leurs architectures ont été validées (ex : complétude des données représentées, exploitabilité des informations affichées, conventions graphiques). Il a donc été possible de définir un premier style architectural (dont les caractéristiques sont présentées dans la section suivante) décrivant les propriétés et règles communes à toutes ces architectures. Pour cela il a été nécessaire d'analyser les architectures d'IHMs pour réutiliser l'expertise de conception qu'elles représentaient en la formalisant au niveau du style.

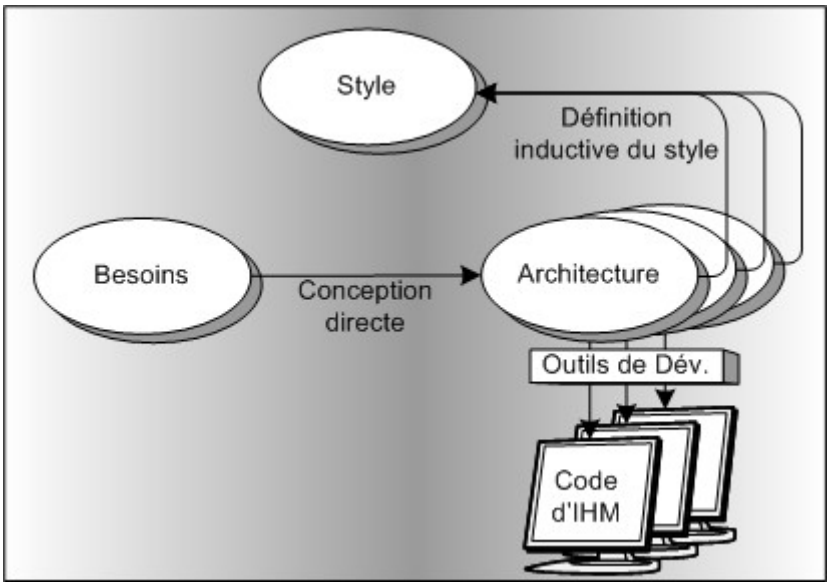


Figure IV. 3 : Processus de développement centré architecture avec description inductive du style

Par ailleurs, étant donné la complexité d'un accélérateur de particules, il était difficile de prévoir dès le début du projet tous les nouveaux besoins qui allaient apparaître au cours de son cycle de vie. Il était encore plus difficile d'avoir une vision globale des besoins communs aux IHMs de tous les accélérateurs de particules du CERN. De plus, les accélérateurs de particules ont une très longue durée de vie au cours de laquelle ils évoluent en permanence. Ceci conduit à de fréquentes modifications des IHMs dédiées à leur supervision. Considérant ces difficultés, il n'était pas envisageable d'utiliser l'approche purement déductive, car il était impossible de définir un style architectural complet utilisable pour tous les accélérateurs du CERN et prenant en compte les possibles évolutions des IHMs.

IV. 2. 3 Environnement de conception contrôlé par le style

Le style obtenu suite au processus présenté figure IV.3 devait être utilisé en tant que support, ou guide pour la création des futures IHMs. Dans cet objectif, une approche possible consiste à utiliser ce style, formalisant l'ensemble des contraintes que doivent respecter les IHMs, pour paramétrer les outils centrés architecture développés dans le cadre du projet ArchWare. Ces outils, une fois paramétrés, permettent d'instancier des architectures qui suivent le style de référence. Cependant, l'approche choisie ne consiste pas à paramétrer des outils mais à définir un environnement dédié au développement de logiciels de supervision, ce qui permet de prendre en compte de façon plus précise les caractéristiques du domaine. Cet environnement, SEAM, intègre les contraintes du style GTPM, de façon à ce que les logiciels qui sont développés par son intermédiaire respectent ces contraintes. Une approche centrée architecture a été adoptée dans la construction de l'environnement de développement. Une fois l'architecture de SEAM définie (décrivant les composants de l'environnement ainsi que les relations entre ceux-ci), l'environnement peut être obtenu après plusieurs raffinements. L'environnement obtenu à partir de l'architecture ne permet le développement que de logiciels de supervision respectant le style. Le style GTPM, une fois conçu et formalisé après l'étude des prototypes d'IHMs, a donc servi de base à la construction de l'environnement SEAM (détaillé dans le chapitre 6). La figure IV.4 permet de comprendre comment l'architecture de SEAM et le style GTPM ont été utilisés pour construire SEAM, et comment lui-même est utilisé pour développer des IHMs conformes aux besoins. Tout d'abord le style GTPM, formalisant les contraintes mises en évidence lors de la phase de prototypage, a été intégré à l'architecture de SEAM. Puis, suite à de nouveaux besoins, les utilisateurs ont pu utiliser SEAM pour construire des architectures d'IHMs. Il est à noter que toutes les propriétés formalisées par le style GTPM n'ont pas, pour des raisons de flexibilité, été spécifiées par SEAM, il n'est donc pas possible de garantir à ce niveau que les architectures d'IHMs satisfont entièrement le style. Toutefois, le respect du style par les architectures d'IHMs doit impérativement avoir été vérifié avant que le code des IHMs soit généré. C'est pour cette raison qu'il a été nécessaire d'ajouter une étape (étape 3) de vérification de conformité style-architectures avant celle de génération de code. La description de l'utilité et du fonctionnement du vérificateur de conformité fait l'objet de la section VI.3.4.

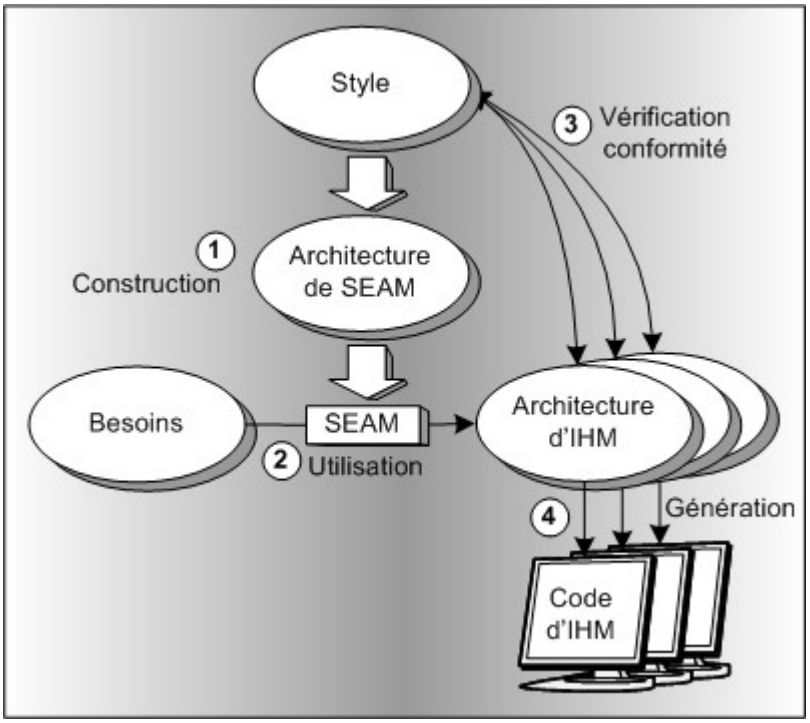


Figure IV. 4 : Construction et utilisation de SEAM

La figure IV.5 reprend le processus de développement présenté figure IV.4, en ciblant plus particulièrement les aspects concernant la construction architecturale, et en écartant ceux concernant l'implémentation. Le processus représenté est appliqué dans le contexte de SEAM et de la production d'IHMs de façon à correspondre aux informations de la figure IV.4, mais il s'agit d'un processus générique, applicable pour tout environnement de développement utilisé pour construire tout type de logiciel.

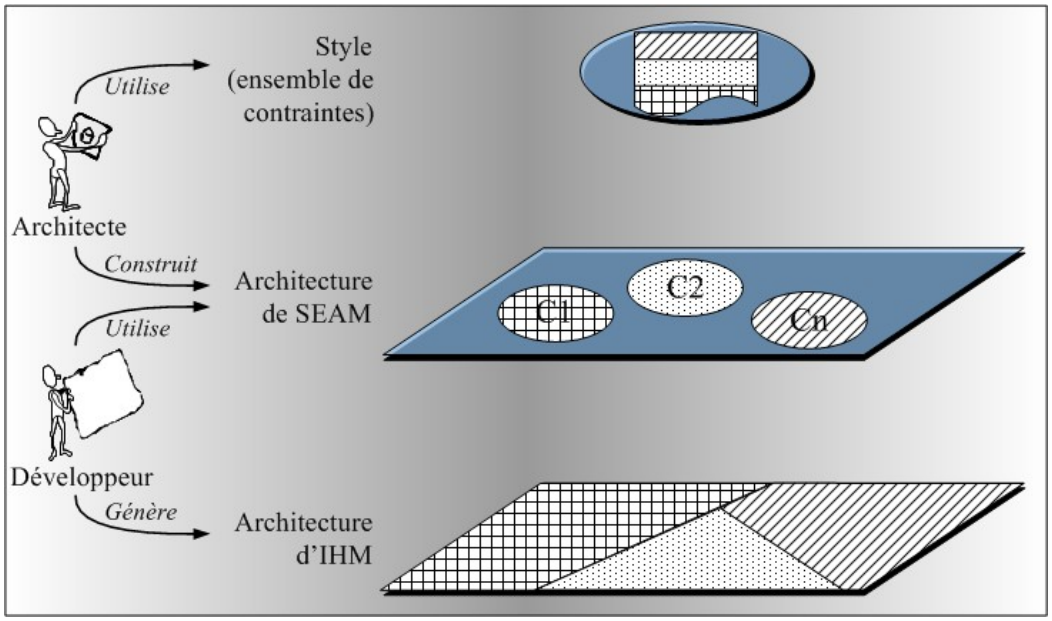


Figure IV. 5 : Construction d'IHMs conformes au style

Comme le montre cette figure, le style formalise différentes classes de contraintes qui sont intégrées dans différents composants de l'architecture de l'environnement de développement (SEAM). Ces composants intègrent les contraintes du style, de façon à ce que les architectures de logiciels qu'ils permettent de produire (IHMs) vérifient, par construction, toutes ces contraintes. Ainsi, ce processus permet de garantir que les architectures produites respectent bien toutes les contraintes spécifiées au niveau du style.

IV. 2. 4 Evolution des architectures et du style

Etant donné que ce style architectural a été défini à partir d'architectures prototypes, il était clair que celui-ci ne pourrait être complet dès sa création. En effet, l'objectif était de l'utiliser pour construire de nombreuses architectures pour lesquelles de nouveaux besoins allaient vraisemblablement apparaître. Il a donc été nécessaire d'étendre le processus de développement de façon à permettre une évolution du style de référence en fonction de l'évolution des besoins.

Une question s'est alors posée : comment mesurer l'évolution des besoins ? L'expérience acquise lors de la phase de prototypage a permis d'y répondre. En effet, depuis la mise en opération de la première version des prototypes d'architectures d'IHMs, de nombreux besoins additionnels sont apparus. Ces nouveaux besoins ont été répercutés au niveau de la conception des architectures et du codage des IHMs. L'évolution du style doit donc être le résultat de l'évolution des architectures des IHMs (le principe de mesure de ces évolutions, ainsi que celui de la mise à jour du style, sont détaillés dans le chapitre 7). Ceci implique notamment que certaines propriétés contraintes par le style, et respectées par les architectures lors de leur instanciation, pourront être violées en cas d'évolution des besoins.

La figure IV.6 représente le processus de développement permettant l'évolution des architectures. La première étape de ce processus est la génération d'une première famille d'IHMs à partir d'architectures vérifiant le style de référence. Par la suite, les architectures peuvent être découplées du style et évoluer conformément à de nouveaux besoins. La troisième étape consiste à observer l'évolution des architectures, les analyser, et décider d'une éventuelle mise à jour du style.

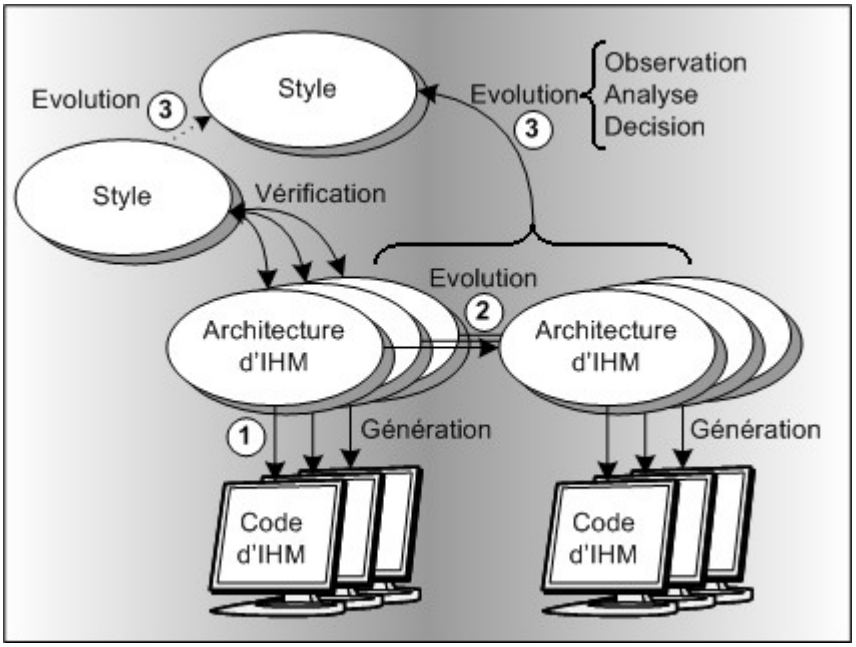


Figure IV. 6 : Processus de développement avec évolution des architectures

Lorsqu'une nouvelle version du style est obtenue, l'architecture de SEAM est revue (afin d'intégrer les modifications au niveau des contraintes) et raffinée vers l'obtention d'un nouvel environnement de développement qui permet de produire des versions améliorées des architectures d'IHMs (Cf. figure IV.7). Ce processus de développement peut être appliqué itérativement de façon à mettre à jour le style progressivement selon l'apparition de nouveaux besoins.

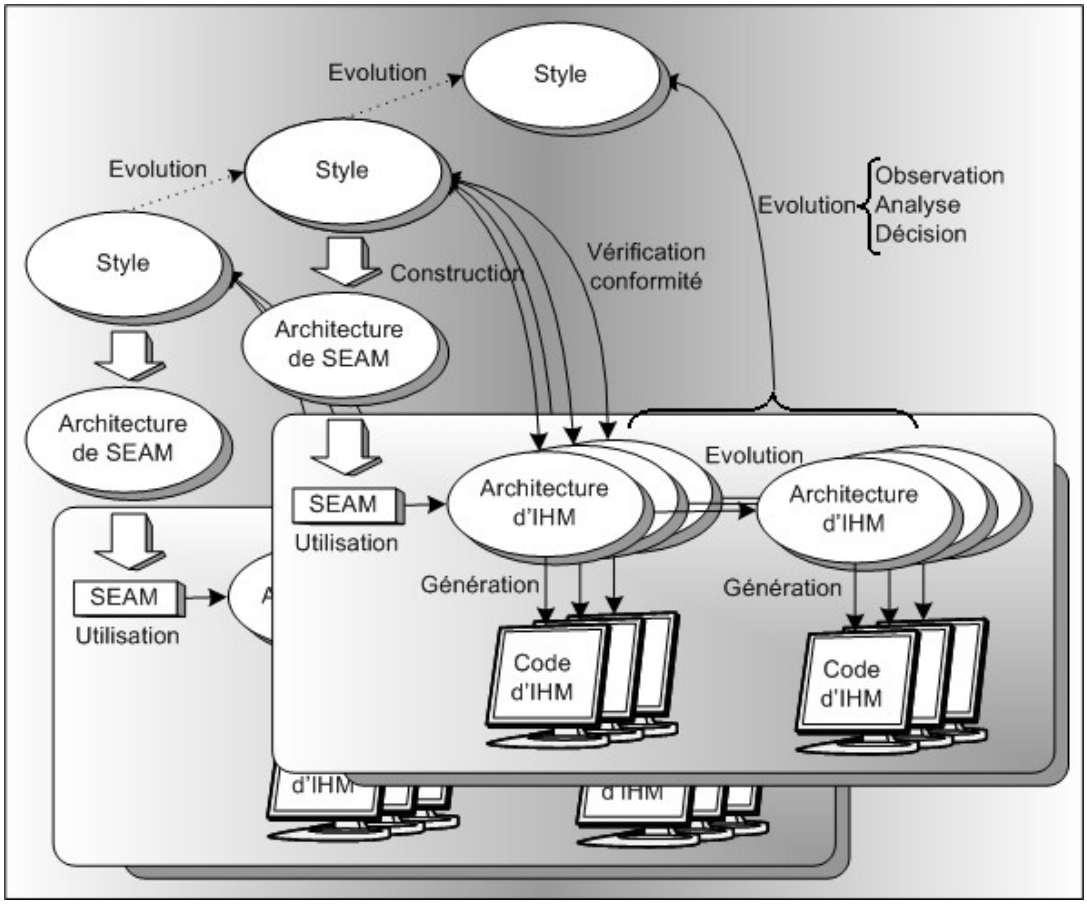


Figure IV. 7 : Processus de développement avec évolution du style

La figure IV.8 récapitule toutes les étapes du processus de développement orienté style suivi dans le cadre de cette thèse.

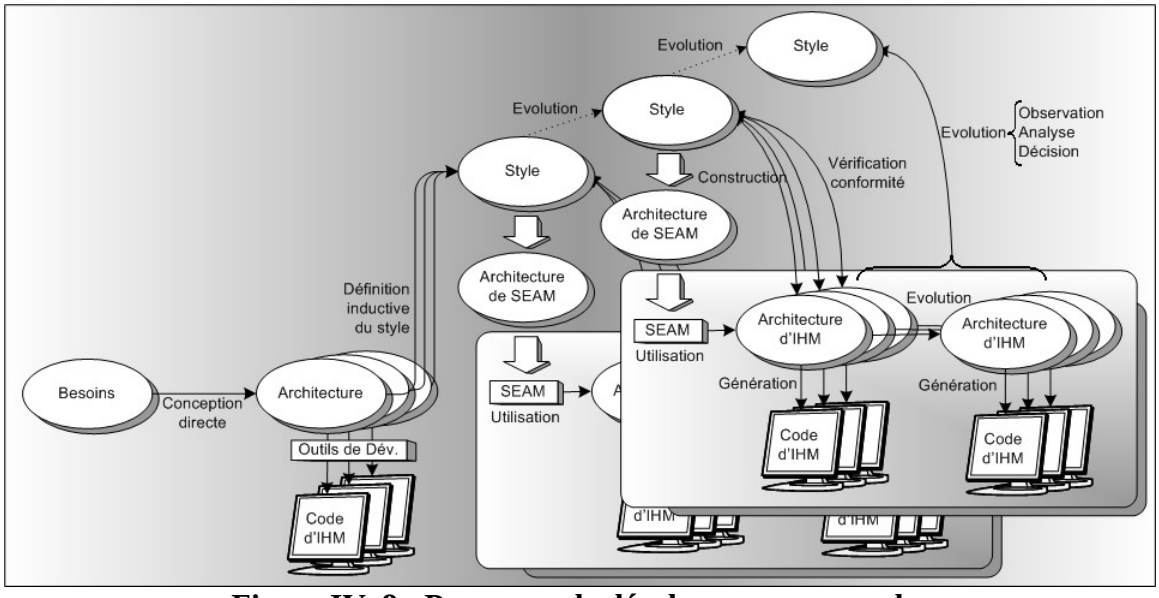


Figure IV. 8 : Processus de développement complet

Comme l'indique cette figure, la première étape, après la phase de prototypage décrite dans le chapitre 2, a été de concevoir et de formaliser le style GTPM. La phase de conception du style est décrite dans la section IV.3, et la phase de formalisation fait l'objet du chapitre 5. La seconde étape du processus représenté figure IV.7 consiste à intégrer ce style à l'architecture de SEAM pour construire l'environnement de développement des familles d'architectures d'IHMs. Cette étape est exposée dans le chapitre 6. La troisième étape, l'observation et l'analyse de l'évolution des architectures, ainsi que la mise à jour du style, est décrite dans le chapitre 7.

IV. 3 Conception du style

Un style architectural fournit un vocabulaire ainsi qu'un ensemble de contraintes. C'est pourquoi, avant de passer à la définition formelle du style, il a été nécessaire de passer par une phase d'analyse ayant pour but l'identification du vocabulaire métier, ainsi que des principales contraintes.

Ayant adopté une approche inductive, cette phase d'analyse a pris la forme d'une étude des vingt prototypes d'IHMs préalablement développés. Ainsi, l'expertise accumulée lors des premiers prototypages est capturée dans ce vocabulaire et ces contraintes. La phase d'analyse a tout d'abord permis de mettre en évidence la structure des IHMs. Ainsi, comme le montre la section II.3.2 et notamment la figure II.3, les IHMs sont composées des éléments suivants :

- **IHM globale :** cet élément permet l'affichage de l'état de tous les systèmes nécessaires à l'opération de l'accélérateur. Il s'agit de l'élément du plus haut niveau de l'architecture, il se compose des éléments de niveau inférieur, les IHMs spécifiques ;
- **IHM spécifique :** cet élément permet l'affichage de l'état des systèmes correspondant à un mode d'accélération ou une zone de l'accélérateur spécifique ; Une IHM spécifique se compose des éléments de niveau inférieur, les blocs systèmes ;
- **Bloc système :** cet élément affiche l'état résultant d'un ou plusieurs équipements d'un système spécifique.

De plus, comme les logiciels de supervision à construire (désignés par le terme « IHM ») ont de fortes composantes graphiques, ces éléments comportent de nombreux attributs graphiques, dont les principaux sont les suivants :

- **pour les IHMs spécifiques :**
  - positionnement sur l'IHM globale (coordonnées X et Y) ;
  - dimensions (largeur et hauteur).
- **pour les blocs d'affichage d'état des systèmes :**
  - positionnement sur l'IHM spécifique (coordonnées X et Y) ;
  - dimensions (largeur et hauteur) ;
  - texte indiquant le nom du système supervisé (police, taille, positionnement) ;
  - texte indiquant le responsable du système supervisé (police, taille, positionnement) ;

- texte indiquant une information additionnelle (exemple : mesure de tension, mesure de température) ;
- niveaux de zoom pour lesquels les textes apparaissent / disparaissent ;
- couleurs utilisées pour représenter l'état du système supervisé.

Le style GTPM peut proposer des valeurs par défaut pour les attributs des blocs et des IHMs énumérés ci-dessus, et les utilisateurs doivent toujours avoir la possibilité de paramétrer ceux-ci.

La figure IV.9 représente, en notation ADL-UML [Alloui et Oquendo 2003], le modèle de construction d'IHMs conçu à partir de l'étude des prototypes. Cette notation utilise les notions de composant et connecteur détaillées dans le chapitre 5. Les composants sont les éléments de base de l'architecture et les connecteurs décrivent les relations entre ceux-ci, les ports étant utilisés comme interfaces. Les éléments composites sont construits à partir de plusieurs éléments atomiques (composants/connecteurs).

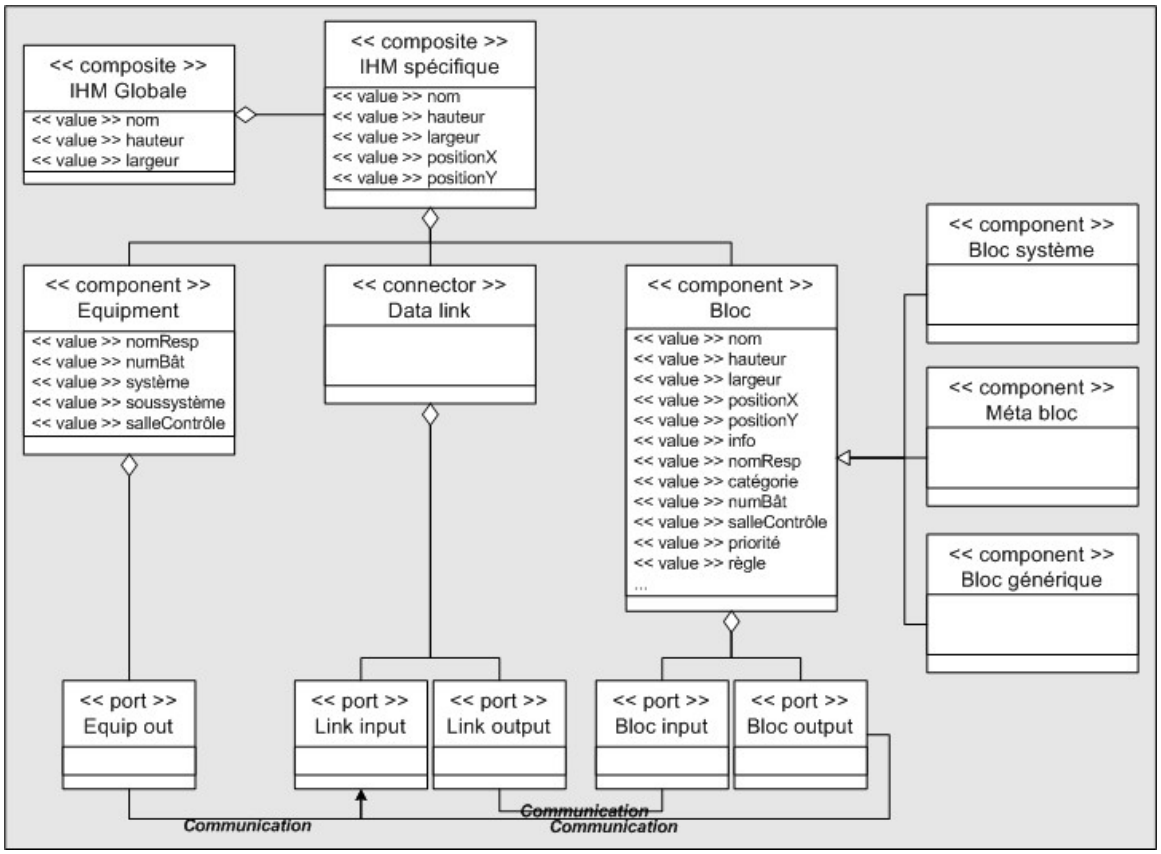


Figure IV. 9 : Modèle ADL-UML pour la construction des IHMs

Cette figure spécifie qu'une IHM globale est composée de plusieurs IHMs spécifiques elles-mêmes composées de plusieurs blocs. Les prototypes d'IHMs n'utilisaient qu'une catégorie de blocs, les blocs systèmes. Ceux-ci sont directement connectés à des équipements dont ils reçoivent une valeur et calculent un état résultant (contraintes sur le comportement). Toutefois, les utilisateurs des prototypes d'IHMs ont jugé intéressant d'introduire plusieurs niveaux de blocs pour les développements futurs.

C'est pourquoi le diagramme UML représente un bloc « générique » qui est spécialisé en « Méta bloc » et « Bloc système ». Ces blocs ont les mêmes attributs graphiques et possèdent une méthode permettant de calculer un état résultant. La différence se situe dans le type de données qu'ils supervisent :

- un **bloc système** représente l'état d'un système physique : il doit superviser directement un ou plusieurs équipements d'un même système (vide, électricité). Les valeurs envoyées par les équipements et reçues par les blocs systèmes peuvent être de type booléen, entier, flottant, ou chaîne de caractères ;
- un **méta bloc** supervise le processus à un niveau supérieur : il ne peut pas superviser directement des équipements mais il supervise uniquement d'autres blocs. Leur objectif est de donner aux opérateurs des salles de contrôle la possibilité d'avoir une vue plus globale de l'état des systèmes complexes, en augmentant le niveau d'abstraction/regroupement des systèmes dans des unités fonctionnelles plus larges. Ainsi, un méta bloc peut superviser l'état d'un bloc système, d'un autre méta bloc, ou d'un bloc « générique ». Par ailleurs, étant donné que tous les blocs fournissent des valeurs résultantes de type entier (Cf. section IV.3.7), les méta blocs sont conçus pour ne superviser que des valeurs de ce type ;
- un **bloc générique**, peut, comme son nom l'indique, superviser tout type d'élément (équipement, méta bloc, bloc système) et peut donc recevoir et traiter tout type de donnée élémentaire. Le bloc générique fournit de la flexibilité aux développeurs d'IHMs qui peuvent, en l'utilisant, superviser à la fois des données de haut niveau (résultantes fournies par des blocs) et des données provenant directement des équipements.

Par ailleurs, comme cela a été exposé dans le chapitre présentant la problématique, il est capital que les IHMs construites à partir du style GTPM soient uniformisées. Par conséquent, il est nécessaire que le style spécifie, en plus des propriétés structurelles citées précédemment, un certain nombre d'autres contraintes s'appliquant notamment aux attributs graphiques. L'analyse des prototypes a permis de répertorier ces contraintes, ainsi que les propriétés et les données utilisées par les IHMs déjà développées. Celles-ci correspondent aux besoins exprimés par les utilisateurs et listés dans la section II.4.2. Les propriétés et contraintes se répartissent dans les catégories suivantes :

- contraintes graphiques générales (taille, couleur, position (…) des éléments) ;
- contraintes de validité des informations ;
- contraintes de positionnement des éléments en fonction de leur rôle dans la séquence de redémarrage de l'accélérateur ;
- contraintes de cardinalité des éléments ;
- contraintes d'interdépendance entre éléments ;
- contraintes d'acquisition de données ;
- contraintes de traitement de données ;
- contraintes concernant l'évolution des éléments architecturaux.

Les paragraphes suivants présentent les propriétés correspondant à chacune de ses catégories.

IV. 3. 1 Contraintes graphiques générales

Le style GTPM permet d'assurer que les valeurs de paramétrage des attributs graphiques correspondent aux « standards » définis dans le cadre du projet GTPM, ce qui implique par exemple les contraintes suivantes :

- **superposition des blocs** : pour éviter la surcharge des IHMs spécifiques et pour faciliter leur lecture, la superposition de blocs est interdite (contrainte sur les positions et dimensions des blocs) ;
- **taille et police** : pour standardiser les IHMs spécifiques, tous les blocs présents sur celles-ci ont des zones de texte de même taille et des polices de caractères identiques ;
- **zoom** : pour standardiser le comportement de IHMs, toutes les zones de texte présentes sur celles-ci apparaissent / disparaissent aux mêmes niveaux de zoom.

IV. 3. 2 Contraintes de validité des informations

En plus des attributs relatifs à l'affichage graphique, les blocs possèdent plusieurs informations statiques servant de documentation aux opérateurs des salles de contrôle. Ces attributs, que sont notamment le nom du responsable du(des) système(s) supervisé(s), le numéro de bâtiment permettant de localiser ce(s) système(s), la catégorie du bloc, ainsi que la salle de contrôle responsable de sa supervision, sont eux aussi soumis à des contraintes :

- **nom de responsable** : le nom de responsable choisi pour le bloc doit bien correspondre à un employé du CERN enregistré dans la base de données des ressources humaines, et il doit être cohérent avec le nom de responsable associé aux équipements qu'il supervise :
    - si, par exemple, tous les équipements supervisés par le bloc B1 ont pour responsable M. Dupont, le responsable du bloc B1 sera M. Dupont ;
    - dans les cas pour lesquels le bloc supervise des équipements ayant des responsables différents, le créateur du bloc devra choisir celui qui, parmi eux, est susceptible de connaître le mieux l'ensemble des équipements supervisés ;
    - dans un cas très complexe, le plus approprié est de choisir comme responsable le nom du créateur du bloc.
- **numéro de bâtiment** : le numéro de bâtiment doit être existant (donc présent dans la base de données du patrimoine) et être cohérent avec le numéro des bâtiments dans lesquels se trouvent les équipements qu'il supervise :
    - si, par exemple, tous les équipements supervisés par le bloc B1 sont localisés dans le bâtiment Bât1, le nom de bâtiment associé au bloc B1 sera Bât1 ;
    - si les équipements supervisés par le bloc se trouvent dans des bâtiments différents, le créateur du bloc devra choisir la localisation la plus pertinente parmi celles proposées.
- **catégorie** : la catégorie des blocs correspond à la classification des équipements utilisée au CERN : chaque équipement appartient à un « système » et à un « sous-

système » auquel l'équipement en question appartient. Le système et le sous-système choisis pour un bloc doivent être existants dans la base de données de contrôle du CERN et cohérents avec ceux des équipements supervisés :

- si, par exemple, tous les équipements supervisés par le bloc B1 font partie d'un circuit d'eau déminéralisée (système « eau » et au sous-système « demi »), la catégorie associée au bloc B1 sera « eau/demi » ;
- si les équipements supervisés par le bloc font partie de systèmes ou sous-systèmes différents, le créateur du bloc devra choisir une catégorie plus générale ou celle qui regroupe la majeure partie des équipements supervisés.

- **salle de contrôle** : à chaque bloc est associé le nom de la salle de contrôle responsable de la supervision des équipements qu'il représente. Ce nom doit, lui aussi, correspondre à une salle de contrôle existante.

IV. 3. 3 Contraintes de positionnement des éléments

Le projet GTPM présenté dans le chapitre 2 a défini un sens de lecture des IHMs de façon à faciliter et à accélérer leur interprétation par les opérateurs des salles de contrôle. Il a donc été nécessaire d'énoncer des contraintes de développement pour que les positions relatives des blocs et des IHMs respectent ce sens de lecture (du coin supérieur gauche au coin inférieur droit). Les contraintes sont les suivantes :

- **positionnement relatif des blocs** : les éléments graphiques sont positionnés en fonction, d'une part, de la priorité de leur remise en route dans la séquence de redémarrage des accélérateurs, et d'autre part, de la catégorie de systèmes à laquelle ils appartiennent. A chaque bloc ont donc été affectées une valeur caractérisant sa priorité ainsi qu'une information concernant sa catégorie (système / sous-système) :
  - par exemple, si l'on considère que pour la remise en opération d'un accélérateur le premier système à redémarrer est l'alimentation en électricité, et qu'ensuite viennent en parallèle les redémarrages des aimants et des systèmes de sécurité, on obtiendrait :

Bloc	Priorité	Catégorie
Alimentation électrique	1	électricité
Alimentation des aimants	2	aimants
Aimants	3	aimants
Détection d'incendie	2	sécurité
Détection de fuite de gaz	2	sécurité

Ces informations doivent ensuite être utilisées pour contraindre le positionnement des blocs les uns par rapport aux autres. La figure IV.10 représente l'IHM qui pourrait être construite en fonction des données ci-dessus.

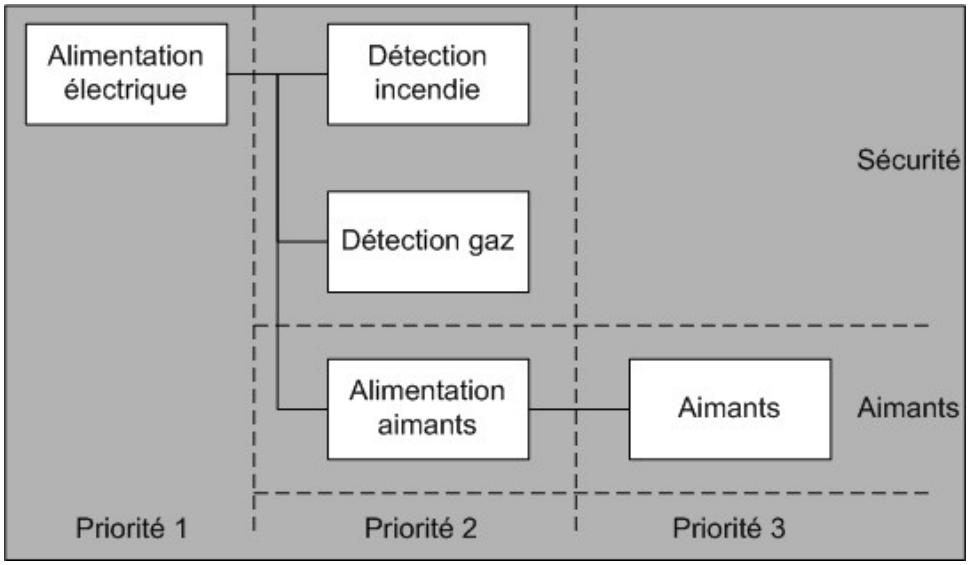


Figure IV. 10 : Illustration de positionnement des blocs sur les IHMs

- **positionnement relatif horizontal des IHMs** : le positionnement horizontal des IHMs spécifiques dépend de la position des systèmes associés dans l'ordre de redémarrage de l'accélérateur :
    - par exemple, les particules devant être produites avant de pouvoir être accélérées, les IHMs représentant l'état des systèmes intervenant dans la production de particules sont positionnées à gauche des IHMs représentant l'état des systèmes intervenant dans leur accélération.
- **positionnement relatif vertical des IHMs** : le positionnement vertical des IHMs spécifiques sur l'IHM globale dépend de la position géographique des systèmes dont elles supervisent l'état.

IV. 3. 4 Contraintes de cardinalité des éléments

Le style GTPM doit formaliser plusieurs contraintes de cardinalité. Par exemple, il spécifie qu'une IHM contient au moins un bloc et qu'un bloc système supervise au moins un équipement. De plus, dans le style GTPM, certaines contraintes de cardinalité sont conditionnées par des attributs de blocs.

Les blocs système utilisés sur les IHMs supervisent des équipements d'une même catégorie. Cette catégorie fait l'objet d'un attribut de bloc qui conditionne les contraintes de cardinalité. Le style GTPM va ainsi permettre de formaliser les contraintes suivantes :

- **présence unique obligatoire** : chaque IHM doit, par exemple, inclure :
    - un et un seul bloc de la catégorie « électricité 18kV » ;
    - un et un seul bloc de la catégorie « électricité 400V ».
- **présence obligatoire** : chaque IHM doit inclure :
    - entre un et cinq blocs de la catégorie « aimants ».
- **présence unique optionnelle** : chaque IHM doit inclure :
    - au maximum un bloc de la catégorie « ordinateurs ».

IV. 3. 5 Contraintes d'interdépendance entre éléments

L'IHM représentée partiellement figure IV.10 comporte des liens d'interdépendance entre les blocs. Ces liens, représentés graphiquement par des traits, indiquent que le fonctionnement d'un système aval est conditionné par le fonctionnement d'un système amont. Il s'agit d'informations purement graphiques qui ont pour but d'aider les opérateurs des salles de contrôle dans l'interprétation du processus, ils ne représentent aucune acquisition ou transmission de données. Ainsi, dans l'exemple de la figure, les blocs « Détection incendie », « Détection gaz » et « Alimentation aimants » ne peuvent fonctionner que si le bloc « Alimentation électrique » est opérationnel. De même, le fonctionnement du bloc « Aimants » dépend de celui du bloc « Alimentation aimants ».

Ces relations de dépendances respectent, elles aussi, certaines contraintes, comme par exemple les suivantes :

- **redondance de relation de dépendance** : le style GTPM spécifie qu'il ne peut exister sur une même IHM deux relations identiques (deux relations liant deux mêmes blocs) ;
- **boucle de dépendance** : le style GTPM interdit toute relation de dépendance indiquant qu'un bloc est dépendant de lui-même. Toute boucle de dépendance est interdite :
    - dans l'exemple de la figure IV.10 une boucle de dépendance apparaîtrait si le concepteur de l'IHM définissait un lien graphique entre le bloc « Aimants » et le bloc « Alimentation électrique ». Pour éviter cette situation, qui se traduirait par un autoblocage dans la séquence de redémarrage de l'accélérateur, le style GTPM permet de parcourir « l'arbre » de dépendances de façon à détecter et empêcher la spécification de telles boucles.

IV. 3. 6 Contraintes d'acquisition de données

Les blocs représentés sur les IHMs acquièrent les valeurs fournies par les équipements, ou, dans certains cas, des valeurs résultantes calculées par d'autres blocs. Les valeurs des équipements peuvent être de différents types élémentaires (booléen, entier, flottant, chaîne de caractères). Les outils de développement utilisés pour construire les prototypes d'IHMs n'empêchaient pas au développeur d'utiliser, par erreur, des blocs incompatibles avec les types des données qu'ils devaient superviser. Considérant ce point, ainsi que celui de l'intégrité de la chaîne d'acquisition de données, il a été nécessaire d'énoncer les contraintes suivantes :

- **typage des blocs compatibles** : le style GTPM assure que les blocs sont associés à des données dont ils supportent les types :
    - par exemple, les bloc conçus pour superviser des données de type flottant (ex : mesures de tension) ne doivent pas être associés par erreur à des données de type booléen (ex : alarmes) sans que l'outil de développement ne génère d'avertissement.

- **boucle d'acquisition** : tout comme pour les dépendances entre blocs, le style interdit toute boucle dans la chaîne d'acquisition* et de transmission de données entre blocs :
    - il ne doit pas être possible, par exemple, qu'un « bloc2 » acquière l'état résultant d'un « bloc1 » lui-même calculé à partir de l'état résultant du « bloc2 ».

IV. 3. 7 Contraintes de traitement de données

L'exploitation des prototypes d'IHMs en salle de contrôle a permis de mettre en évidence un dysfonctionnement au niveau du traitement des données : les blocs étaient régulièrement dans l'état « invalide » (couleur bleue), qui indique une défaillance de l'un des systèmes de la chaîne d'acquisition des données, sans que la moindre panne ne soit effectivement constatée. La raison la plus fréquente de ce phénomène résidait dans le fait que la plage de valeur des données fournies par les équipements n'était pas traitée entièrement. La contrainte suivante a donc été énoncée :

- **traitement de toute la plage de valeurs** : le style GTPM impose que les blocs traitent l'ensemble de l'échelle des valeurs des données qu'ils acquièrent. En d'autres termes, aucune valeur de variable supervisée par un bloc ne doit pouvoir entraîner un état indéterminé :
    - un bloc de mesure de tension peut, par exemple, prendre la couleur verte pour une mesure comprise entre 380V et 420V, la couleur rouge pour une tension supérieure à 420V, et il doit lui être interdit de ne pas traiter les valeurs inférieures à 380V. Pour une telle valeur, l'état du bloc ne doit pas être indéterminé.

Par ailleurs, chaque bloc possède une logique interne, ou règle, permettant de fournir un état résultant calculé à partir des valeurs acquises. Cette règle satisfait la contrainte suivante :

- **état résultant standard** : le style GTPM impose que l'état résultant respecte un format standard (entiers compris entre 0 et 3, correspondant aux états vert/marche, blanc/arrêt, jaune/avertissement, rouge/défaillance), et ce, quelque soient le nombre et le type des variables qu'il supervise.

IV. 3. 8 Contraintes d'évolution

Les IHMs construites à partir du style GTPM doivent pouvoir être mises à jour régulièrement sans que de nouvelles phases de codage soient nécessaires. L'interface graphique de l'outil SEAM doit donc rendre possible l'application de modifications sur la majorité des éléments composant les architectures d'IHMs :

- les attributs des blocs et des IHMs ;
- les associations entre blocs associés aux IHMs ;
- les associations entre blocs et équipements ;

* Les chaînes d'acquisition de données ne sont pas graphiquement représentées sur les IHMs telles que l'exemple donné figure IV.9. Elles sont toutefois indiquées sur le diagramme UML (figure IV.8) par la relation supervise / est supervisé.

- les liens de dépendance entre blocs ;
- les règles permettant de calculer l'état des blocs.

Toutefois, ces modifications ne doivent pas entraîner de violation des contraintes définies lors de la création des IHMs, c'est à dire celles présentées dans les paragraphes précédents. Dans le cas contraire, la modification entraînerait une violation du style, ce qui, selon le cas, doit être interdit, ou, comme cela sera décrit dans le chapitre 7, peut indiquer qu'une évolution du style est nécessaire.

IV. 4 Conclusion

Le processus de développement présenté dans ce chapitre constitue une solution pour répondre aux besoins cités en introduction. En effet, celui-ci a permis de concevoir un style architectural regroupant l'ensemble des contraintes mises en évidence suite à l'étude des qualités et des limites des prototypes d'IHMs. Le processus proposé utilise le style ainsi obtenu en l'intégrant dans la construction d'un environnement permettant le développement de familles d'IHMs respectant les contraintes du style et répondant aux besoins des opérateurs de salles de contrôle.

La suite de cette thèse va maintenant présenter la mise en œuvre de cette approche. La première étape est la formalisation du style jusqu'alors exprimé de façon informelle. L'étape suivante est la conception et l'implémentation de l'environnement SEAM permettant d'exploiter ce style pour concevoir des IHMs respectant les contraintes définies dans ce chapitre. La dernière étape est la mise en œuvre d'un processus permettant l'évolution des IHMs, mais aussi du style et de son environnement.

Chapitre V Formalisation du style architectural

Chapitre V : Formalisation du style architectural

V. 1 Introduction

Les chapitres précédents ont montré l'intérêt d'adopter une approche orientée architecture dans le cadre du développement d'un outil de développement de logiciels de supervision d'accélérateurs de particules. Le processus de développement choisi implique la définition inductive d'un style architectural spécifique à ce domaine d'application. En outre, l'étude de l'état de l'art a montré que le langage de description architectural réalisé dans le cadre du projet ArchWare s'avérait le plus approprié pour la problématique de cette thèse, c'est donc celui-ci qui a été adopté.

Le présent chapitre décrit la formalisation du style architectural obtenue à partir des contraintes précédemment listées, ainsi que des exemples d'architectures de logiciels de supervision. La section suivante va préalablement introduire le projet ArchWare, décrire les langages développés au cours de celui-ci, leurs rôles, leurs principales fonctionnalités, ainsi que l'environnement de développement associé.

V. 2 Fondements

L'objectif du projet ArchWare est de répondre à une demande toujours croissante de systèmes logiciels qui peuvent évoluer au cours de leur durée de vie. Pour atteindre ce but, ArchWare a développé un ensemble intégré de langages et d'outils orientés architecture qui comprend [Oquendo et al. 2004] :

- des langages formels (avec des notations textuelles et graphiques) pour modéliser des architectures dynamiques (incluant l'expression de leur structure, de leur comportement, et de leurs propriétés sémantiques d'un point de vue composant-connecteur), ainsi que pour exprimer leurs analyses et raffinements ;
- un environnement de développement personnalisable pour l'ingénierie logicielle centrée architecture, incluant des processus et outils servant de support à la description d'architectures, à leur analyse, à leur raffinement, à la génération de code, ainsi qu'à leur évolution.

Les quatre langages proposés par ArchWare sont : un langage de description architecturale (ADL[1]) ; un langage d'analyse d'architectures (AAL[2]) ; un langage de description des styles architecturaux (ASL[3]) ; un langage de raffinement d'architectures (ARL[4]). Parmi ces langages, ne sont présentés dans cette section que ceux qui ont été

[1] ADL : Architecture Description Language
[2] AAL : Architecture Analysis Language
[3] ASL : Architecture Style Language
[4] ARL : Architecture Refinement Language

utilisés pour la description du style architectural, c'est-à-dire l'ADL, l'AAL et l'ASL. Les paragraphes V.2.1 et V.2.2 ne font qu'introduire les concepts de base de ces langages. Cependant, leur utilisation et leur syntaxe seront présentées de façon plus détaillée dans le paragraphe concernant le modèle composant-connecteur (V.2.3), une couche formelle construite à partir de l'ADL dédiée à la description d'architectures et de styles. Ces langages sont utilisables et exploitables via un environnement de développement brièvement présenté dans la section V.2.4.

V. 2. 1 ADL ArchWare

Dans ArchWare, la description d'architectures possède deux aspects : la description des styles et des architectures ainsi que la vérification des propriétés sur les architectures, cette dernière pouvant être menée afin de déterminer si une architecture suit un style. L'approche pour la description des architectures proposée dans le cadre du projet fournit un langage noyau générique, π-ADL*, et des mécanismes d'extension et d'adaptation à de domaines particuliers. En outre, l'ADL ArchWare [Oquendo et al. 2002][Oquendo 2003] la structure de base et les constructions comportementales pour décrire les architectures logicielles dynamiques. C'est un langage formel de spécification conçu pour être exécutable et pour supporter l'analyse et le raffinement automatique des architectures.

L'ADL a pour fondement formel le π-calcul typé d'ordre supérieur [Sangiorgi 1992], un langage de description de systèmes mobiles et communicants. L'ADL est lui-même un langage formel défini comme une extension du π-calcul pour la description d'éléments architecturaux mobiles et communicants.

Les principes généraux suivants ont guidé la conception de cet ADL :

- formalisme : cet ADL est un langage formel, au sens mathématique, permettant la description d'architectures logicielles dynamiques et le raisonnement sur elles ;
- point de vue exécution : l'ADL se concentre sur la description d'architectures d'un point vue exécution : leur structure à l'exécution, leur comportement à l'exécution, et comment ils peuvent évoluer au cours du temps ;
- exécutabilité : il s'agit d'un langage exécutable : une machine virtuelle exécute les spécifications des architectures logicielles ;
- ergonomie : cet ADL supporte différentes syntaxes concrètes avec des notations textuelles [Cimpan et al. 2002][Verjus et Oquendo 2003] et graphiques [Alloui et Oquendo 2003][Alloui et Oquendo 2004] facilitant son utilisation par les architectes.

Les termes de l'ADL, les ***behaviours***, expriment le comportement des éléments architecturaux. Un élément architectural à l'exécution est représenté à travers un ***behaviour***. Une ***abstraction*** est une définition de comportement pouvant être appliquée plusieurs fois. Elle représente une définition d'élément réutilisable et paramétrable.

Les ***behaviours*** peuvent être connectés au travers de connexions visibles (***free connections***). Ces connexions représentent les interfaces des éléments architecturaux et permettent la transmission de valeurs (***values***). Un élément architectural peut être

* Dans la suite de ce chapitre, le terme « ADL » est utilisé pour désigner π-ADL (langage noyau)

composé de plusieurs autres éléments interagissant nommés ***constituents***. L'architecture est elle-même un élément architectural.

Le diagramme suivant représente une architecture décrite en utilisant la notation UML de l'ADL. Les ***behaviours*** sont associés par des ***connections***. Les ***connections*** de deux ***behaviours*** différents peuvent être liées ce qui rend possible l'interaction entre ces ***behaviours***. Dans l'exemple de la figure V.1, l'élément A (***behaviour***) possède un point d'interaction CA1 (***connection***) qui lui permet de communiquer avec l'élément B via son point d'interaction CB1.

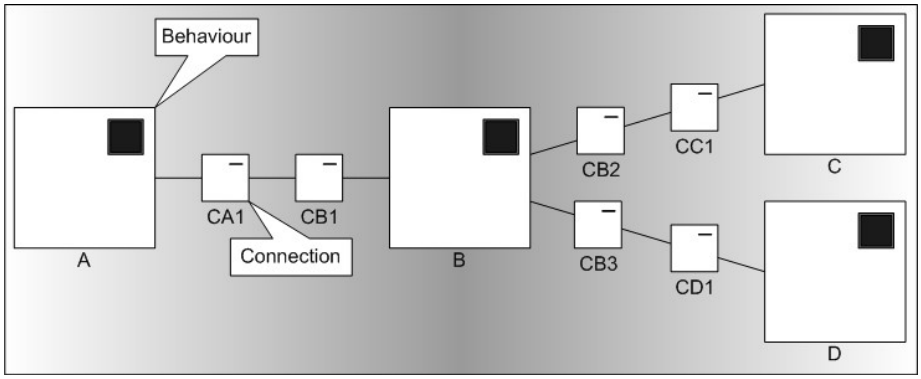


Figure V. 1 : Architecture décrite avec la syntaxe de l'ADL basée sur UML

La couche immédiatement supérieure de l'ADL ArchWare donne la possibilité de construire des styles. En outre, ASL (Architectural Style Langage) est un langage de plus haut niveau, formellement construit à partir de l'ADL et de l'AAL (AAL ArchWare, présenté section V.2.2), et constitue le lien entre ces deux langages en permettant la définition de styles d'éléments architecturaux représentés par des abstractions de comportement gardés par des propriétés. Ce langage, qui a été utilisé dans le cadre de cette thèse, permet de définir des extensions de l'ADL propres à des domaines d'application particuliers, où les propriétés du domaine peuvent être explicitement définies et préservées. Ces extensions sont définies comme des styles architecturaux et introduisent des notations spécifiques au domaine d'application au moyen de notations « mixfix » (Cf. section V.2.3.5.5) et de techniques de réécriture. Parmi ces extensions se trouve le style de base composant-connecteur [Cimpan et al. 2003] qui fournit des éléments de construction facilitant la définition de styles plus spécifiques. Ce style est présenté dans la section V.2.3.

V. 2. 2 AAL ArchWare

L'AAL ArchWare [Alloui et al. 2003] fournit un cadre pour spécifier les propriétés pertinentes des architectures. Ces propriétés sont de différentes natures : elles peuvent être structurelles (ex : cardinalité des éléments architecturaux) ou comportementales (ex : sûreté, vivacité). L'AAL complémente l'ADL en fournissant des fonctionnalités permettant aux architectes d'exprimer et de vérifier les propriétés des architectures d'une façon naturelle. Les analyses sont appliquées au moyen de trois approches : vérification de modèle (model-checking), preuves (theorem proving), et utilisation d'outils externes spécifiques.

L'AAL est un langage formel d'expression de propriétés conçu pour supporter la vérification automatique. Il permet de vérifier mécaniquement si une architecture exprimée en ADL satisfait une propriété exprimée en AAL.

L'AAL a pour fondement formel le μ-calcul modal [Kozen 1983], un langage d'expression de propriétés de systèmes à transitions. L'AAL est lui-même un langage formel défini comme une extension du μ-calcul pour l'expression de propriétés structurelles et comportementales d'éléments architecturaux communicants et mobiles.

Quelques clauses fournies par l'AAL ainsi qu'un exemple de propriété formalisée à l'aide de celles-ci sont présentés dans la section V.2.3.5.4.

V. 2. 3 Style composant-connecteur

Les architectures de type composant-connecteur sont bien connues dans le domaine des architectures logicielles. Ces architectures sont souvent considérées comme la base de toute construction architecturale. En effet, il est généralement accepté (Cf. chapitre 3) que les architectures sont faites de composants (éléments de traitement) et de connecteurs (éléments de communication).

Le style composant-connecteur, ou C&C, fournit un fondement aidant les utilisateurs à formaliser leurs propres architectures logicielles et styles architecturaux à partir de celui-ci. Ce style permet la formalisation d'architectures logicielles et de styles architecturaux basés sur les concepts de composants et de connecteurs. Il est écrit en ASL et utilise les syntaxes concrètes de l'ADL et de l'AAL présentés précédemment. Tous les concepts qu'il propose sont définis en ASL comme des styles d'éléments architecturaux.

Le style C&C est défini comme une base pour la définition d'autres styles. Il a notamment été utilisé pour définir une bibliothèque de styles génériques incluant les styles ***« Pipe&Filter »***, ***« Client-Server »***, ***« Data Indirection »*** et ***« Layered Style »***.

La présente section expose les notions et les mécanismes utilisés pour la description d'architectures et de styles à partir du style C&C. Ceux-ci sont tirées de [Cimpan et al. 2003] où des informations plus détaillées sont disponibles.

V. 2. 3. 1 Présentation des architectures en C&C

Cette section présente les principaux concepts concernant les aspects structurels des architectures et les mécanismes proposés par le style C&C.

V. 2. 3. 1. 1 Structure des architectures

Une architecture est un ensemble d'éléments architecturaux connectés. La vue que l'architecture a sur ses éléments architecturaux correspond aux interfaces que ces derniers proposent. Chaque élément est une « boîte noire » du point de vue des autres éléments constituant l'architecture. Ils peuvent communiquer au travers de leurs interfaces qui proposent un ensemble de services.

Un élément architectural comprend :

- un comportement (***behaviour***) qui définit sa fonctionnalité ;
- un ensemble de ***ports*** qui représentent l'interface de l'élément ;
- un ensemble d'attributs (***attributes***) qui contiennent l'information additionnelle concernant l'élément.

Un élément architectural peut avoir différents rôles dans une architecture, et il peut être structuré de différentes façons.

Concernant le rôle qu'il joue dans une architecture, l'élément architectural peut être un composant (***component***), un connecteur (***connector***), ou un ***choréographe*** (Cf. figure V.2). Ces éléments architecturaux sont présentés en détail dans les sections V.2.3.3.7 à V.2.3.3.9.

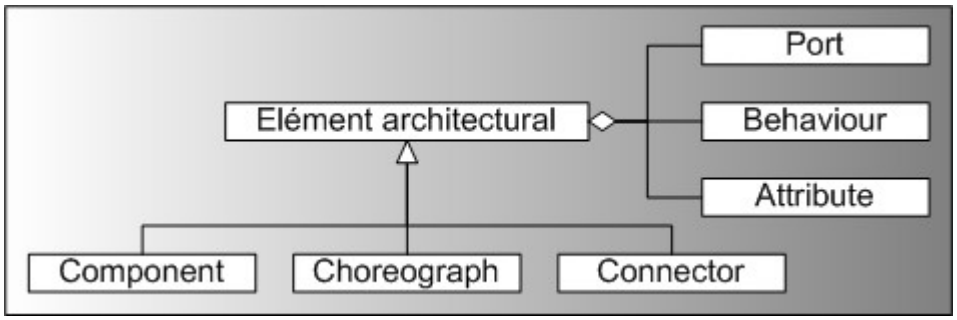


Figure V. 2 : Constituants et rôles des éléments architecturaux

Du point de vue structurel, un élément architectural peut être atomique (***atomic***) ou composé par d'autres éléments architecturaux (***composite***) (Cf. figure V.3).

Un élément composite encapsule une ***configuration*** dans laquelle sont définies les relations (***attachments***) entre les différents éléments architecturaux constituants. Cette configuration représente le comportement de l'élément composite. Elle est associée aux ports du composite au moyen de liens (***bindings***) avec les constituants. Dans le style C&C, les architectures sont des éléments composites.

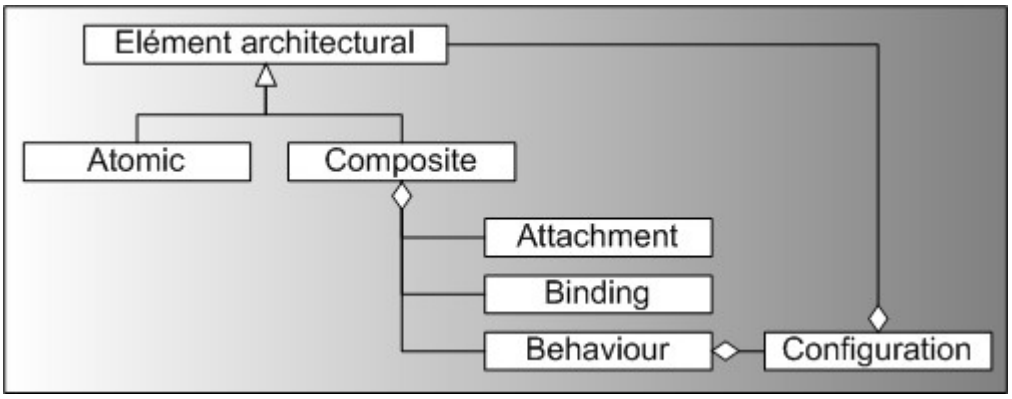


Figure V.3 : Structure des éléments architecturaux

V. 2. 3. 1. 2 Dynamicité

La dynamicité est la possibilité d'instanciation à l'exécution. Toute connexion, port, composant, connecteur, ou attachement d'une architecture peut être instancié dynamiquement autant de fois que nécessaire.

La dynamicité est gérée par le comportement (***behaviour***) dans le cas d'élément atomique. Dans le cas d'un élément composite celle-ci est gérée par le choréographe. Les actions pouvant être réalisées dynamiquement sont :

- une création d'instance pour une connexion, un port, un composant, ou un connecteur ;
- une création d'attachement ou de lien ;
- une suppression d'attachement ou de lien.

V. 2. 3. 2 Description d'architectures en C&C

Cette section présente les différentes techniques utilisées pour définir des architectures qui respectent le style C&C. Elles concernent la façon dont les éléments architecturaux sont déclarés, nommés (ciblage), ainsi que la réutilisation de définitions.

V. 2. 3. 2. 1 Déclarations dans les architectures C&C

Dans le style C&C, les déclarations des types réguliers (types de base et types composés) et des valeurs doivent être faites explicitement. Ces déclarations peuvent concerner les types d'éléments architecturaux (***archetypes***) et les valeurs d'éléments architecturaux.

La déclaration d'un élément architectural consiste à fournir une définition et un identifiant. La définition est soit fournie à la déclaration, soit réutilisée au moyen de l'utilisation d'***archetypes***.

V. 2. 3. 2. 2 Réutilisation de définitions

Le style C&C permet la réutilisation de définitions au moyen des ***archetypes*** et des ***méta-éléments***.

Les ***archetypes*** sont utilisés pour réutiliser des définitions génériques, indépendantes de leur contexte d'utilisation. Un ***méta-élément*** n'est pas indépendant de son contexte mais est au contraire déclaré dans une architecture précise. Les ***méta-éléments*** permettent la réutilisation de définitions paramétrées par l'utilisation de valeurs d'instanciation. L'exemple suivant définit deux ***archetypes***, *Client* et *One_Client_One_Server*. *One_Client_One_Server* est constitué de deux ***méta-éléments***, dont *client* qui utilise la définition de *Client*.

```
archetype Client is component with{…}

archetype One_Client_One_Server is component with{
    …
    consituents
        client : Client,
        server : Server;
    …
}
```

V. 2. 3. 2. 3 Ciblage

Le ciblage est en rapport avec deux aspects des éléments architecturaux : la composition et l'instanciation dynamique.

Les architectures suivant le style C&C utilisent la composition des éléments architecturaux. En effet, les connexions appartiennent aux ports, et les ports appartiennent aux composants et connecteurs. L'opérateur ~ a été introduit pour permettre de cibler un

élément architectural en fonction des éléments auxquels il appartient. L'exemple suivant fournit la référence d'un port d'un constituant d'une architecture :

port_reference **:=** constituent_reference**~**port_name

Le style C&C permet la création de plusieurs instances d'un élément. L'opérateur # est utilisé pour cibler des instances spécifiques d'un même élément. L'exemple suivant permet de cibler le client nommé *toto* d'une architecture client-serveur :

client_occurrence **:=** client**#**"toto"

V. 2. 3. 3 Formalisation d'architectures en C&C

Cette section présente les concepts utilisés pour formaliser des architectures à partir du style C&C.

V. 2. 3. 3. 1 Connexion

Une connexion est une interface atomique permettant l'interaction entre éléments architecturaux par l'envoi et la réception de messages. Une connexion est typée en fonction du type de message qu'elle transite. Par exemple, ***in*** est une connexion transitant des valeurs du type ***String*** et ***out*** est une connexion transitant des valeurs des types ***Integer*** ou ***String***.

in: **connection[String]**
out: **connection[Integer, String]**

Une connexion est un constituant de port. Elle est donc nommée et définie dans une définition de port et elle est utilisée dans la description d'un comportement.

V. 2. 3. 3. 2 Port

Un port est un point d'interaction d'un élément. Il structure l'interface de celui-ci en regroupant ses connexions. La définition du port contient la description du protocole à utiliser pour interagir avec l'élément auquel il appartient. L'exemple suivant est la déclaration d'un port qui contient sa définition. ***InOut*** est un port comprenant deux connexions, ***in*** et ***out***. Le protocole signifie qu'une réception sur ***in*** doit avoir lieu avant un envoi sur ***out***. ***Replicate*** indique que la séquence comportant une réception et une émission peut être répétée de manière indéfinie.

```
InOut is port with {
  connections
    in: connection[Any],
    out: connection[Any];
  protocol
    replicate
      via in receive.
      via out send;
}
```

V. 2. 3. 3. 3 Attachement

Les attachements permettent la communication entre deux éléments architecturaux. Les attachements entre éléments sont matérialisés au niveau des connexions. Celles-ci

sont liées par des « attachements atomiques ». Le code suivant définit l'attachement entre un « ***pipe*** » et deux « ***filters*** » nommés respectivement ***p*** et ***f1*** et ***f2***. Les ports des éléments sont nommés ***inputport*** et ***outputport*** et l'exemple suppose que ***f2*** possède au moins deux ports d'entrée basés sur le même modèle.

attach f1~outputport~out **to** p~inputport~in**.**
attach f2~outputport#2~in **to** p~outputport~out

V. 2. 3. 3. 4 Lien

Un lien, ou ***binding***, permet à un constituant d'avoir des points d'interaction avec l'environnement dans lequel son composite s'exécute. Les liens se trouvent donc entre les connexions d'un composite et celles de ses constituants, ce qui leur permet d'interagir avec l'extérieur. Le code suivant définit un lien dans un composite construit en « ***pipeline*** » (i.e. une série de « ***pipes*** » et « ***filters*** » intercalés). Deux des « ***filters*** » ont des points d'interaction avec l'environnement externe de la « ***pipeline*** » : ***f1*** et ***fn***.

bind inputport~in **to** f1~inputport~in**.**
bind outputport~out **to** fn~outputport~out

V. 2. 3. 3. 5 Attributs

Les attributs stockent les informations concernant les éléments architecturaux. Ils permettent en outre de configurer les éléments à leur initialisation. Le code suivant déclare deux attributs : un temps de latence ainsi que des informations concernant un contrat.

latency**: real default value is** 10.0
contract_info**: view[**date**: String,** author**: String,** object**: String]**

V. 2. 3. 3. 6 Comportement

Le comportement définit la fonctionnalité d'un élément architectural. La description d'un comportement est composée d'un ensemble d'actions et indique comment celles-ci sont planifiées. Comme les éléments peuvent être atomiques ou composites, il existe deux sortes de description de comportement associées avec ces deux types d'éléments. La description du comportement d'un élément atomique est directement composée d'actions alors que celle d'un élément composite indique la composition des comportements de ses constituants. La formalisation des comportements est faite par l'utilisation d'opérateurs, comme l'opérateur de séquence « **.** ». L'exemple suivant indique que *clause3* ne peut s'exécuter qu'après l'exécution de *clause1* ou *clause2* :

choose{clause1 **or** clause2**}.**clause3

L'opérateur « ***compose*** » permet de composer des comportements. L'exemple indique que lorsque tous les comportements composés sont terminés, il est possible de commencer une nouvelle action :

compose until all done {clause1 **and … and** clausen**}**

Le comportement des éléments architecturaux peut inclure des actions de communication, d'instanciation, et de gestion des attachements.

L'instanciation consiste à créer une occurrence d'un ***méta-élement***. A l'instanciation, l'occurrence peut être nommée et configurée (clause ***initialised with***) :

new element_main_name
named element_name
initialised with {value **as** attribute_name**}**

Les actions de communication sont des envois et des réceptions. L'exemple suivant indique que la connexion *connection1* est utilisée pour recevoir une séquence d'informations du type *data_type* :

via connection1 **receive** seq**: sequence[**data_type**]**

Les actions de gestion d'attachements permettent d'attacher (***attach***) et de détacher (***detach***) des éléments, et ce, statiquement ou à l'exécution. L'exemple suivant attache *port1* à *port2* :

attach port1 **to** port2

V. 2. 3. 3. 7 Composant

Un composant est un élément architectural qui fournit des fonctionnalités au système dans lequel il est intégré. Son comportement est nommé ***computation***. Comme un composant doit pouvoir communiquer pour fournir des services, il doit inclure au moins un port. La définition générale d'un composant atomique comporte une section pour la définition des types de données utilisés (***types***), une section de déclaration des ports qu'il comporte (***ports***), une autre pour la déclaration de ses attributs (***attributes***), une pour la configuration des ports (***configuration***), et enfin celle définissant son comportement (***computation***). La définition d'un composant composite comporte les sections ***types***, ***ports***, ***attributes***, ***constituents*** et ***configuration***.

L'exemple suivant définit un archetype de composant composite. Il s'agit d'une architecture avec deux « ***filters*** » ***f1*** et ***f2*** connectés par un « ***pipe*** » ***p*** et lié aux ports du composite. Ce composite possède un attribut spécifiant le temps requis pour le traitement.

archetype One_P_Two_F **is component with {**
ports
inputport**:** Input_port**,**
outputport**:** Output_port**;**
attributes
processing_time**: integer default value is** 0
constituents
f1**:** Filter**,**
f2**:** Filter**,**
p**:** Pipe**;**
configuration
new f1**. new** f2**. new** p**. new** inputport**. new** outputport**.**
bind inputport~in **to** f1~inputport~in**.**
bind outputport~out **to** f2~outputport~out**.**
attach f1~outputport~out **to** p~inputport~in**.**
attach f2~inputport~in **to** p~outputport~out
}

V. 2. 3. 3. 8 Connecteur

Un connecteur est un élément architectural dont le rôle est de connecter les éléments architecturaux pour transmettre des informations. Son comportement est nommé ***routing***. Un connecteur sert à caractériser le type de communication utilisé (synchronisé ou non, …), à assurer la connexion de plusieurs éléments, ainsi qu'à convertir les formats de données entre éléments si nécessaire. Un connecteur atomique contient les clauses ***types***, ***ports***, ***attributes***, ***configuration*** et ***routing*** et un connecteur composite possède les clauses ***types***, ***ports***, ***attributes***, ***constituents*** et ***configuration***. L'exemple suivant concerne la définition d'un « pipe ». Celui-ci transmet simplement le message reçu sans lui appliquer de transformation.

```
archetype simple_pipe is component with {
  ports
    inputport: Input_port,
    outputport: Output_port;
  configuration
    new inputport. new outputport;
  routing
    replicate
      via inputport~in receive x: Any.
      via outputport~out send x;
}
```

V. 2. 3. 3. 9 Choréographe

Un choréographe a pour rôle de gérer les composants et les connecteurs constituants d'un élément composite (architecture), notamment leur mobilité et leur dynamicité. Définir un choréographe consiste à fournir la définition de son comportement. L'exemple ci-dessous concerne une base de données composée d'emplacements contenant des informations. Cette base a un point d'interaction pour chaque emplacement, donc pour chaque information. Il est possible de demander à la base d'ajouter un nouvel emplacement. La base de données est modélisée comme un composant composite ayant un port capturant les événements concernant la création d'emplacements d'informations. Un choréographe gérant la création des nouveaux emplacements traite ces événements. Un emplacement d'information est aussi modélisé comme un composant. Le comportement du choréographe est le suivant : après avoir reçu un événement de création (comportant un nom), il crée et lie un nouvel emplacement et un nouveau port. Ces nouveaux éléments sont ciblés avec le nom reçu avec l'événement.

```
choreographer {
  replicate
    via creation_port~create receive slot_name: String.
    new data_slot named slot_name.
    new data_slot_port named slot_name.
    bind data_slot#slot_name~data_slot to data_slot_port#slot_name
}
```

V. 2. 3. 4 Présentation des styles en C&C

Un style formalise la connaissance architecturale d'un domaine particulier. La définition d'un style comporte plusieurs parties distinctes :

- un ensemble de contraintes spécifiant les propriétés que doit satisfaire une architecture ou un élément qui suit le style ;
- une bibliothèque contenant un ensemble de définitions qui peuvent être réutilisées dans la formalisation de styles ou architectures ;
- un vocabulaire spécifique au style qui permet d'améliorer la compréhension des architectures faisant partie d'un même domaine ;
- une documentation informelle fournissant un support supplémentaire aux architectes qui utilisent le style.

V. 2. 3. 4. 1 Contraintes

Les contraintes constituent la partie centrale de la définition des styles. Elles permettent de définir des sous-espaces de conception architecturale spécifiques aux domaines d'application. Ces contraintes, ou propriétés caractérisent des familles d'architectures. Le style C&C distingue deux catégories de contraintes : la partie « commune » et la partie « variante ». Les contraintes faisant partie de la catégorie « commune » sont respectées par toutes les architectures qui suivent le style. Celles faisant partie de la catégorie « variante » représentent des aspects pour lesquels des différences entre architectures suivant le même style peuvent apparaître. Par ailleurs, les contraintes peuvent être soit topologiques, soit comportementales, soit non fonctionnelles. Les premières concernent la structure des éléments architecturaux, les secondes définissent les séquences d'actions autorisées et interdites, et les dernières concernent les attributs des éléments.

V. 2. 3. 4. 2 Bibliothèque

La bibliothèque d'un style contient un ensemble de définitions réutilisables. Elle fournit un support à la définition et à l'analyse des architectures ainsi qu'à la définition d'autres styles. Les définitions contenues dans la bibliothèque du style peuvent concerner des types, des archetypes de ports et d'éléments architecturaux, des styles de ports et d'éléments architecturaux, des constructeurs et des analyses.

V. 2. 3. 4. 3 Analyses

Les analyses donnent le moyen d'évaluer des architectures ou des éléments architecturaux. Celles-ci sont souvent liées aux contraintes des styles. Une analyse peut par exemple évaluer la performance d'une architecture. Il n'est possible d'appliquer des analyses sur une architecture que si celle-ci vérifie certaines caractéristiques (topologiques, comportementales, ou au niveau des attributs) permettant l'évaluation.

V. 2. 3. 4. 4 Vocabulaire

Un style fournit aux architectes un vocabulaire spécifique à un domaine d'application particulier. Ce vocabulaire peut désigner des éléments architecturaux spécifiques (en général définis comme des extensions du style C&C), des relations entre éléments architecturaux, ou des actions comportementales. Par exemple, un style ***client-serveur***

fournirait deux éléments architecturaux spécifiques (***client*** et ***serveur***) qui seraient des extensions de l'élément « ***composant*** » du style C&C. Il existe ensuite deux options pour utiliser le style client-serveur :

- utiliser la syntaxe du style C&C : l'architecte va définir des composants qui sont soit des clients soit des serveurs ;
- utiliser la syntaxe introduite par le style client-serveur : l'architecture va directement définir des clients et des serveurs.

V. 2. 3. 4. 4 Documentation

Un style peut être accompagné d'une documentation. Celle-ci explique de façon informelle à quoi sert le style et comment l'utiliser, et fournit éventuellement des conseils sur les outils et les techniques à mettre en œuvre pour construire des architectures à partir du style.

V. 2. 3. 5 Formalisation des styles en C&C

La définition d'un style en C&C comporte six sections permettant la définition des types, des ports, des éléments, des contraintes, des constructeurs et des analyses utilisés par le style. Les paragraphes suivants détaillent et illustrent ces différentes définitions.

V. 2. 3. 5. 1 Types

La définition des types en C&C est similaire à celle utilisée dans l'ADL. L'exemple suivant déclare le type ***request*** comme un type composé d'un champ ***number*** de type entier et d'un champ ***message*** de type chaîne de caractères.

```
request is view[number: Integer, message: String];
```

Les types déclarés et définis dans la section ***types*** d'un style peuvent être utilisés dans les définitions d'archetypes, d'attributs et de contraintes du même style.

V. 2. 3. 5. 2 Ports

La bibliothèque d'un style contient des définitions d'***archetypes*** de ports. La définition d'un tel archetype est spécifiée dans la clause ***ports*** et respecte la syntaxe présentée en V.2.3.3.2. Un archetype de port introduit par un style peut être utilisé plus loin dans la définition du style ou par des architectures qui suivent le style.

La définition d'un ***style*** de port permet d'ajouter la spécification des contraintes que les ports qui le suivent doivent respecter. Une définition de style de port contient les clauses ***types*** et ***constraints***. L'exemple suivant formalise le style de port ***OutputPort*** définissant un type de connexion nommé ***Data***. Ce style impose aux ports qui le suivent d'avoir une ou deux connexions du type ***Data***. Toutes les connexions doivent être utilisées pour envoyer des messages. La virgule séparant les deux contraintes correspond à un « et » logique.

```
OutputPort is port style where
  types
    Data is connection[Any]
  constraints
    to connections apply {
      exists ([1..2]c | c of type Data),
```

```
        forall (c | every sequence {true*.via c receive any} leads to state
        {false})
    }
```

Les styles de ports peuvent être utilises pour définir des sous-styles, des archetypes de ports, ou des ports.

V. 2. 3. 5. 3 Eléments

La clause ***elements*** d'un style est celle où les éléments constituants des architectures/éléments architecturaux sont définis. Ces éléments peuvent être définis comme des archetypes (Cf. syntaxe sections V.2.3.3.7 et V.2.3.3.8) ou comme des styles. Lorsqu'un style est défini comme une extension du style composant-connecteur l'architecte doit préciser pour chaque élément s'il s'agit d'un composant ou d'un connecteur. L'élément va alors hériter soit du style composant, soit du style connecteur. La définition d'un style d'élément peut posséder les six clauses cités précédemment : ***types***, ***ports***, ***elements***, ***constraints***, ***templates*** et ***analysis***.

Les styles d'éléments peuvent être utilisés de plusieurs façons, notamment pour la définition des contraintes du style qui les utilise, et pour écrire d'autres définitions (archetypes ou styles d'éléments qui suivent ou utilisent le style en question).

V. 2. 3. 5. 4 Contraintes

Comme exposé précédemment, les contraintes peuvent être topologiques, comportementales ou non fonctionnelles, et elles peuvent être incluses dans la partie « commune » ou la partie « variante » de l'espace de conception fourni par le style. La notation de description des contraintes de la partie « variante » est définie par le langage d'analyse architecturale (AAL). Ces contraintes sont définies pour être appliquées à des ensembles d'éléments architecturaux référencés par les mots clefs suivants :

- ***connections*** : ensemble des connexions d'un élément ;
- ***ports*** : ensemble des ports d'un composant ou d'un connecteur ;
- ***attributes*** : ensemble des attributs d'un composant ou d'un connecteur ;
- ***elements*** : ensemble des éléments d'un composite ou d'une architecture ;
- ***components*** : ensemble des composants d'un composite ou d'une architecture ;
- ***connectors*** : ensemble des connecteurs d'un composite ou d'une architecture ;
- ***occurrences*** : ensemble des occurrences d'un élément comme une connexion, un port, un composant ou un connecteur.

Ces prédicats permettent de cibler les éléments sur lesquels s'appliquent les contraintes. Par exemple, si un style possède un composant « c », c**.ports** permettra de cibler l'ensemble des ports de ce composant.

La clause ***apply*** permet d'appliquer de vérifier une propriété sur un ensemble d'éléments. Deux quantificateurs, ***forall*** et ***exists*** sont utilisés pour spécifier quels éléments de l'ensemble sont concernés par la vérification. L'exemple suivant exprime que deux « ***filters*** » d'une architecture « ***Pipe&Filter*** » ne peuvent être attachés :

```
to elements apply {
  forall (x, y | x in style Filter and y in style Filter implies
              not (x attached to y))}
```

Certains prédicats ont été ajoutés pour faciliter la description de contraintes sur les architectures et styles définis en C&C. Par exemple, le prédicat ***of_archetype*** permet de vérifier si un élément est bien d'un archetype donné. De même le prédicat ***in_style*** vérifie si un élément respecte bien un style donné.

V. 2. 3. 5. 5 Constructeurs

Les ***templates*** fournis par un style sont des constructeurs réutilisables et paramétrables dont les architectes se servent pour définir partiellement automatiquement des architectures qui suivent le style. La définition d'un constructeur comprend la définition des types, des paramètres, des constituants, et de la configuration qu'il utilise ainsi que sa notation ***mixfix*** (notation spécifique permettant d'utiliser le constructeur de façon intuitive).

L'exemple de constructeur suivant fournit un patron comportemental pour expédier des informations d'une source vers plusieurs récepteurs. Les paramètres sont la connexion utilisée par la source pour envoyer les informations (***source***), et un ensemble de connexions pour les récepteurs (***receivers***).

```
dispatch is template with {
  parameters as
    source: connection[Any ],
    receivers: set[connection[Any ]];
  configuration
    via source receive data: Any.
    iterate receivers by c:connection[Any]
    do via c send data;}
  mixfix
    dispatch from source to receivers
```

Les exemples suivants montrent les deux notations possibles permettant d'utiliser ce constructeur pour définir un connecteur expédiant des informations d'une source vers trois récepteurs.

- notation classique :

```
connectorA is connector with {
  …
  routing
  …
    dispatch(inputport~in, set(outputport#1~out, outputport#3~out)).
  …
}
```

- notation ***mixfix*** :

```
connectorA is connector with {
  …
  routing
  …
    dispatch from inputport~in to
                    set(outputport#1~out, outputport#3~out).
  …
}
```

V. 2. 3. 5. 6 Analyses

Les analyses permettent d'évaluer les architectures ou les constituants des architectures. Celles-ci peuvent être définies soit par langage d'analyse architecturale (AAL) soit par le langage spécifique ou modèle (MSL). Les analyses définies en AAL sont des prédicats retournant ***true*** ou ***false*** lorsqu'ils sont appliqués. Elles comportent une clause ***input*** où sont définis les paramètres, une clause ***body*** où l'analyse est décrite, ainsi qu'optionnellement une clause ***mixfix*** permettant de définir une notation spécifique pour appliquer l'analyse. Les analyses définies en MSL sont des fonctions retournant une valeur typée lorsqu'elles sont appliquées. Il est donc nécessaire de rajouter dans leur définition une clause ***output*** indiquant le type de la valeur retournée. L'exemple suivant, défini en AAL, permet de vérifier si deux composants sont attachés à un même connecteur. Ses paramètres sont deux instances du style composant.

```
direct_connection is AAL_analysis
  input
    c1 in style component,
    c2 in style component;
  body {
    to connectors apply {
      exists ([1]c | c1 attached to c and c2 attached to c)}}}
  mixfix c1 connected to c2
```

Les analyses font partie de la bibliothèque du style et peuvent donc être utilisées plusieurs fois, soit directement pour analyser des architectures, soit au niveau du style dans la définition des contraintes.

V. 2. 4 Environnement de développement ArchWare

ArchWare fournit un environnement de développement intégré (IDE : Integrated Development Environment) pour prendre en charge la mise en œuvre de processus de développement centrés architecture. L'IDE ArchWare comprend :

- le noyau de l'environnement, qui fournit un compilateur de l'ADL et une machine virtuelle qui permet l'exécution des descriptions architecturales ;
- les modèles de méta-processus, qui fournissent le support pour les processus logiciels utilisés pour construire et faire évoluer les applications logicielles ;
- les composants de l'environnement, qui fournissent les outils qui supportent la description, l'analyse, et le raffinement d'architectures.

La machine virtuelle de l'ADL ArchWare utilise un stockage persistant pour contrôler l'exécution des applications et les processus utilisés pour leur développement [Greenwood et al. 2003a]. Les architectures décrites avec l'ADL ArchWare peuvent être exécutées après avoir été compilées en code exécutable par la machine virtuelle.

Le principal modèle de méta-processus ArchWare est le processus pour l'évolution des processus (P2E) [Greenwood et al. 2003b]. Il fournit un mécanisme pour sélectionner les abstractions de processus à partir d'une bibliothèque d'abstractions existantes pour produire de nouvelles abstractions si une adaptée n'est pas disponible.

Les composants de l'environnement sont implémentés soit comme des descriptions ADL exécutables soit comme des outils extérieurs intégrés à ArchWare [Seet 2002]. Les

outils de l'environnement incluent un éditeur visuel basé sur la notation UML [Alloui et Oquendo 2004] ainsi qu'un éditeur « hyper-code » [Zirintsis 2000] dans lequel le code source est directement lié aux données et valeurs qui existent dans l'environnement persistant. L'hyper-code permet d'unifier le code source avec le code exécutable. L'environnement ArchWare fournit par ailleurs un animateur graphique, un vérificateur de propriétés [Catry et al. 2003][Garavel et Mateescu 2003], un raffineur [Oquendo 2003], un générateur de code, et un *customizer* permettant la compilation des styles ainsi que leur instanciation.

V. 3 Formalisation du style

Le but de cette étape est de formaliser le style présenté de façon informelle dans le chapitre 4, résultant de l'analyse des prototypes de logiciels de supervision. Le style formel ainsi obtenu doit permettre de faciliter et d'accélérer le processus de développement ainsi que d'augmenter la qualité des logiciels produits.

Une première version du style GTPM a été réalisée directement dans le langage ADL. Toutefois, la solution la plus adaptée s'est avérée être d'utiliser le style de fondation C&C comme base pour la construction du style GTPM. En effet, ce style de fondation a été créé pour faciliter la formalisation de styles propres à des domaines d'application particuliers. De plus, les logiciels de supervision devant être produits à partir du style GPTM étant composées de « blocs » entre lesquels transitent des données, l'adoption d'une approche composant-connecteur était intuitive.

Le style GTPM ainsi construit constitue un langage propre à la définition de logiciels de supervision d'accélérateurs de particules. Dans ce langage, les composants sont *GlobalHCI*, *IndividualHCI*, *StatusBloc* et *Equipment*. De la même manière, les connecteurs sont des *DataLink* permettant la transmission de données entre les équipements, et les ports sont des ports d'entrée (*InputPort*) et des ports de sortie (*OutputPort*). Dans l'objectif d'augmenter la possibilité de réutilisation, tous ces éléments ont été décrits en terme de styles. En effet, la formalisation a consisté à définir un style *GlobalHCI*, un style *IndividualHCI*, un style *Equipment*, un style *StatusBloc*, ainsi que ses sous-styles (*MetaStatus* et *SystemStatus*). Cette solution a été choisie dans le but d'être aussi flexible que possible, et d'être capable de spécifier les contraintes à différents niveaux de la description formelle. Ainsi, les éléments architecturaux suivant le style *GlobalHCI* sont composés d'éléments qui suivent le style *IndividualHCI*, eux-mêmes composés d'éléments suivant le style *StatusBloc*, qui eux-mêmes permettent d'afficher l'état d'éléments qui suivent les styles *Equipment*, *SystemStatus*, *MetaStatus*, et *StatusBloc*.

Les éléments suivant le style *SystemStatus* peuvent uniquement superviser des informations fournies par des d'éléments qui suivent le style *Equipment*. Au contraire, les éléments suivant le style *MetaStatus* peuvent superviser des informations provenant de tous les composants autres que *Equipment*, c'est à dire de ceux qui retournent un état calculé (*SystemStatus*, *StatusBloc*, *MetaStatus*) (Cf. figure V.4).

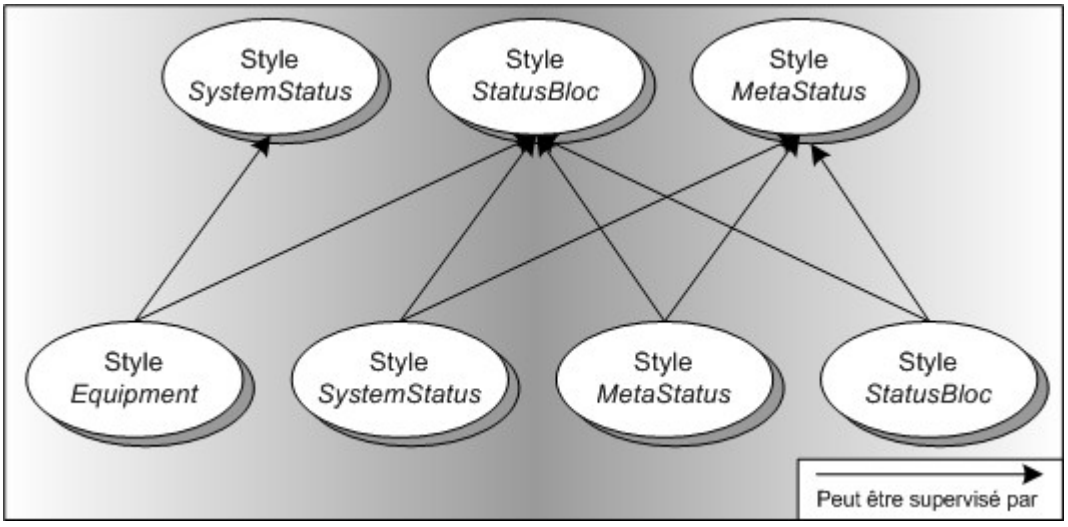


Figure V. 4 : Relations entre les styles de blocs

Cette section présente des extraits des styles formalisés (Cf. formalisation complète en Annexe 2). Ces extraits concernent principalement les contraintes structurelles des architectures ainsi que celles qui s'appliquent aux attributs graphiques.

V. 3. 1 Formalisation des styles de ports de communication

Les ports spécifiques au style GTPM sont décrits en premier lieu car ils sont utilisés par plusieurs autres styles qui seront présentés par la suite. Ceux-ci sont des spécialisations du port de base du style composant-connecteur.

Pour commencer, voici la formalisation du style *InputPort*. Ce port est utilisé par les autres composants comme interface de réception de données.

```
InputPort is style extending Port where {
    types {
        Data is connection[Any];
    constraints {
        exists([1]c | c of type Data),
        to connections apply {
            forall(c | every sequence{true*.via c send
            any} leads to state{false})
        }
    }
}
```

- Le style *InputPort* est construit à partir du style *Port* ;
- Cette contrainte spécifie qu'un *InputPort* possède une et une seule connection de type Data ;
- Cette contrainte spécifie qu'un *InputPort* possède des connections permettant seulement de recevoir des données (toute séquence incluant un envoi de données est impossible).

Le style *OutputPort* est défini de manière analogue. Il est utilisé comme interface d'envoi de données, ce qui implique que, dans son cas, c'est la réception de données que la contrainte doit interdire.

```
OutputPort is style extending Port where {
    types {
        Data is connection[Any];
    constraints {
        exists([1]c | c of type Data),
        to connections apply {
            forall(c | every sequence{true*.via c receive
            any} leads to state{false})
        }
    }
}
```

V. 3. 2 Formalisation du style de connecteur

L'extrait de formalisation suivant correspondant à la définition d'un connecteur répondant aux besoins du style GTPM à partir du connecteur de base proposé par le style composant-connecteur. Il s'agit du style *DataLink*. Celui-ci est utilisé pour transmettre les informations concernant l'état des équipements et systèmes d'un élément architectural à un autre. Le style *DataLink* définit deux ports (*Inport* et *Outport*) et empêche, par les deux dernières contraintes, l'ajout de tout autre port. En effet, il ne doit pas être possible de rajouter de port lors de l'instanciation du style en architectures, où lors de l'exécution d'une architecture qui respecte ce style, car les *DataLink* transmettent des informations entre un port *OutputPort* d'un élément et un port *InputPort* d'un autre élément.

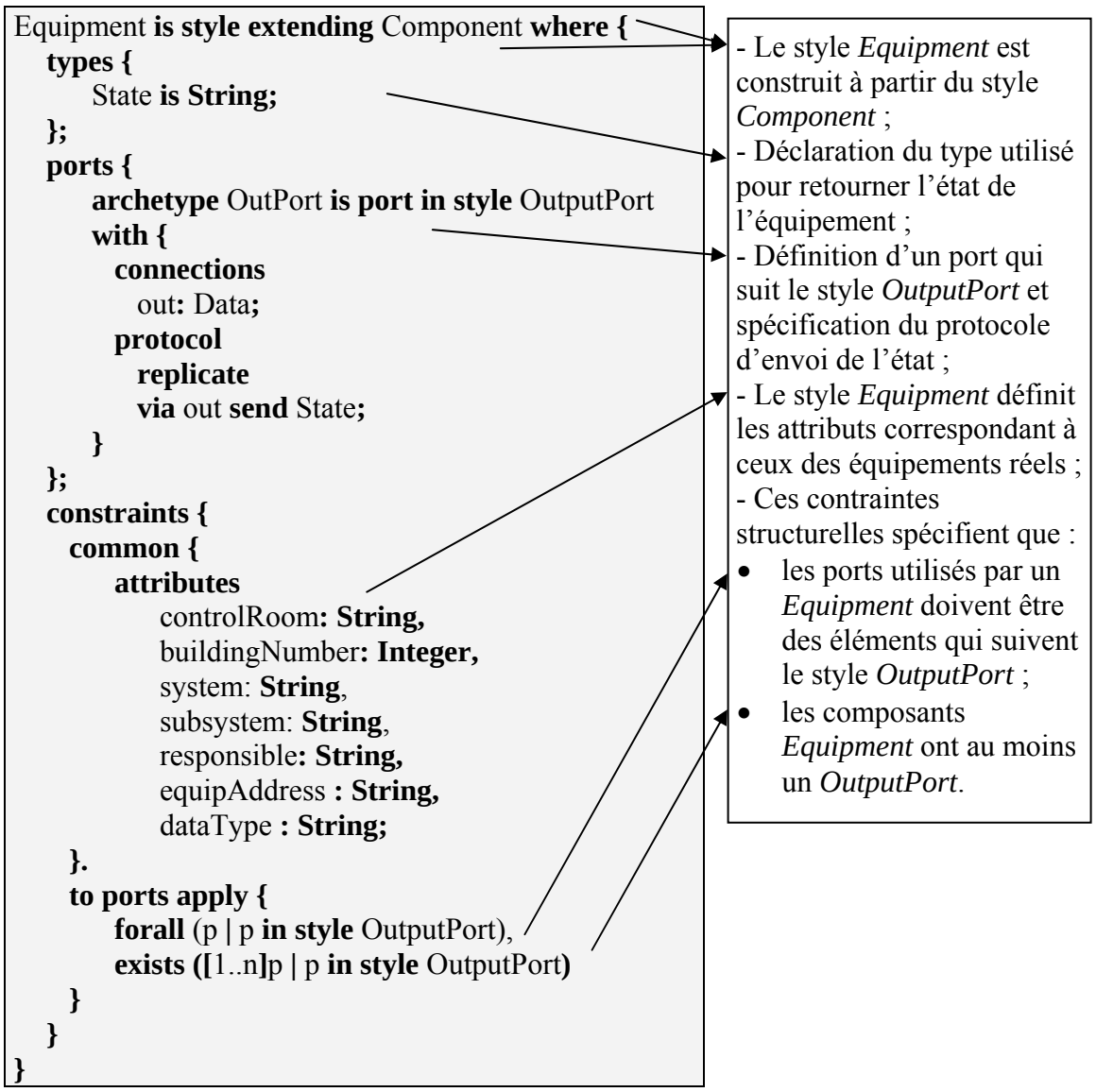
```
DataLink is style extending Connector where (
    ports {
        archetype InPort is port in style InputPort with
        {
          connections
            in: Data;
          protocol
            replicate
            via in receive;
        },
        archetype OutPort is port in style OutputPort
        with {
          connections
           out: Data;
          protocol
            replicate
            via out send;
        }
    };
    constraints {
        to ports apply {
          forall (p | p in style OutputPort xor p in style
          InputPort),
          exists ([1]p | p in style OutputPort),
          exists([1]p | p in style InputPort)
        }
    }
}
```

- Le style *DataLink* est construit à partir du style *Connector* ;
- Le style *DataLink* définit deux archetypes de port qui suivent les styles *InputPort* et *OutputPort*. C'est à ce niveau que sont définis les connections (du type préalablement défini *Data*) et les protocoles d'émission et de réception de données ;
- Ces contraintes structurelles spécifient que :
  - les ports utilisés par un *DataLink* doivent être des éléments qui suivent les styles *OutputPort* ou *InputPort* ;
  - les connecteurs *DataLink* ont un seul *OutputPort* ;
  - les connecteurs *DataLink* ont un seul *InputPort*.

V. 3. 3 Formalisation des styles de composants

V. 3. 3. 1 Formalisation du style de composant Equipment

Ce composant est un élément architectural permettant l'envoi de données. Le but n'est pas de formaliser le comportement réel d'un équipement ou d'un système mais uniquement la génération de valeurs correspondant à son état. En effet, du point de vue d'un logiciel de supervision, un équipement équivaut à une source de données. Ces composants *Equipment* sont très utiles pour simuler le comportement global de l'architecture du logiciel en fonction des valeurs qu'ils fournissent. La formalisation du style *Equipment* inclus la définition d'attributs correspondant à ceux des équipements réels.



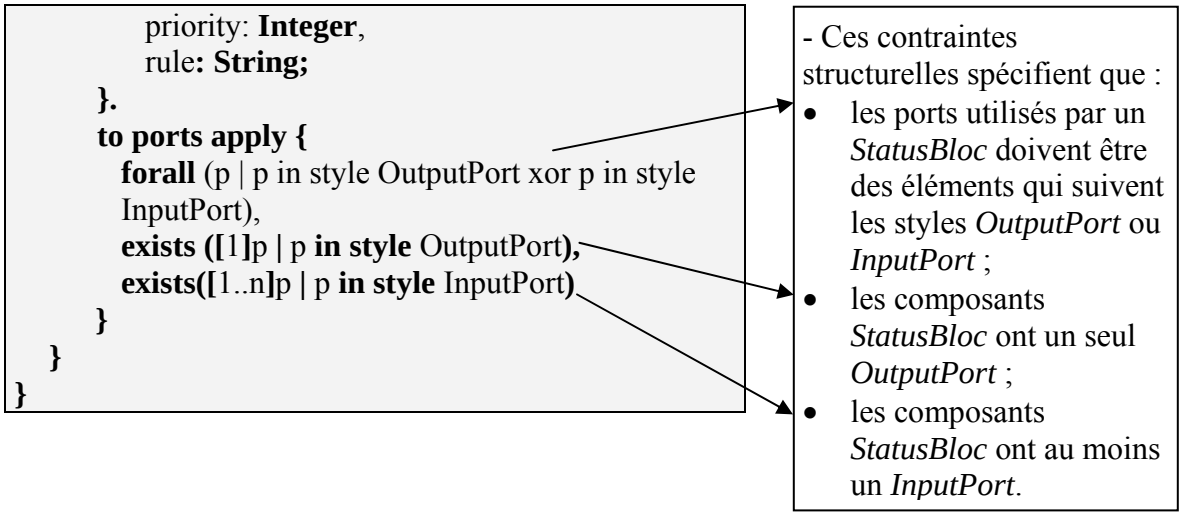
V. 3. 3. 2 Formalisation du style de composant StatusBloc

L'extrait suivant représente la formalisation du composant *StatusBloc*. Les composants *SystemStatus* et *MetaStatus*, sous-styles de *StatusBloc*, ne sont pas décrits ici. En effet, ceux-ci sont très semblables : ils possèdent les mêmes ports et les mêmes attributs. Ils spécifient toutefois chacun une contrainte supplémentaire, qui impose au *SystemStatus* de ne superviser que des équipements, et interdit au *MetaStatus* de superviser des éléments autres que des blocs.

```
StatusBloc is style extending Component where {
   types {
      ResultState is Integer;
   };
   ports {
      archetype OutPort is port in style OutputPort
      with {
        connections
          out: Data;
        protocol
          replicate
          via out send ResultState;
      }
   };
   constraints {
      common {
        types
          ResultState => 0,
          ResultState <= 3,
   }.
      common {
        attributes
          width: Integer default value is 20,
          height: Integer default value is 10,
          positionX: Integer,
          positionY: Integer,
          controlRoomText: String default value is
          "TCR",
          controlRoomFont: String default value is
          "Arial",
          controlRoomFontSize: Integer default value
          is 12,
          controlRoomVisibleZoomFactor: Integer
          default value is 0.5,
          labelText: String default value is "Text",
          labelFont: String default value is "Arial",
          labelFontSize: Integer default value is 14,
          labelVisibleZoomFactor: Integer default
          value is 0.5,
          blocDepIn:[String],
          blocDepOut:[String],
          name: String,
          info: String,
          buildingNumber: Integer,
          category: String,
          responsible: String,
```

- Le style *StatusBloc* est construit à partir du style *Component* ;
- Déclaration du type utilisé pour retourner l'état résultant calculé par le bloc ;
- Définition d'un port qui suit le style *OutputPort* et spécification du protocole d'envoi de l'état résultant ;
- Contrainte sur le type de *ResultState* : il doit être compris entre 0 et 3 (Cf. contrainte **« état résultant standard »** section IV.3.7) ;
- Définition des attributs des blocs (Cf. diagramme figure IV.8).

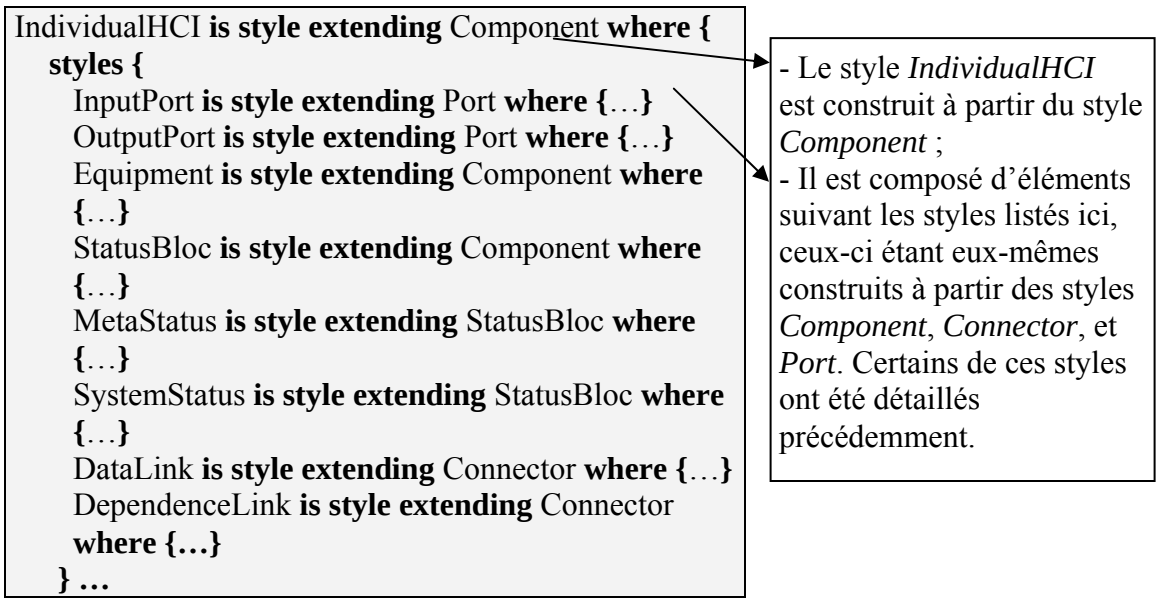
Les attributs *blocDepIn* et *blocDepOut* contiennent respectivement la liste des blocs dont le bloc courant est dépendant, et la liste des blocs dépendants du bloc courant. Ces attributs permettent de formaliser les dépendances entre des systèmes dont les états sont représentés par des blocs (exemple : le fonctionnement du système de climatisation est dépendant du fonctionnement du système d'alimentation électrique). Ces dépendances sont représentées sur les IHMs par des liens graphiques



V. 3. 3. 3 Formalisation du style de composant IndividualHCI

Cette section présente des extraits de la formalisation du style *IndividualHCI*. Comme il s'agit du composant le plus complexe du style GTPM, ces extraits seront répartis dans plusieurs paragraphes correspondant aux sections de définition de styles introduites en V.2.3.5. De plus, les *contraintes* du style utilisant ses *analyses*, les secondes seront présentées avant les premières dans le but de faciliter la lecture. Comme il n'a pas été nécessaire de définir de types et de ports, les paragraphes suivants détailleront uniquement les éléments, les analyses, les contraintes, et les constructeurs.

V. 3. 3. 3. 1 Eléments



V. 3. 3. 3. 2 Analyses

```
…
analysis {
 directly_monitored is AAL_property {
   parameters {
     bloc1 : AnyElement,
     bloc2 : AnyElement
   }
   property {
     bloc1 in style Component.
     bloc2 in style Component.
     to bloc2.ports apply{
       exists(b2ip | b2ip in style InputPort and to
       connectors apply {
         exists (dl | to dl.ports apply {
           exists (dlin, dlout | dlin in style InputPort
                   and dlout in style OutputPort
                   and to bloc1.ports apply {
             exists(b1op | b1op in style OutputPort
                    and b1op attached to dlin
                    and dlout attached to b2ip)
           })
         })
       })
     }
   }
 } mixfix (bloc1 directly monitored by bloc2);

 monitored is AAL_property {
   parameters {
     bloc1 : AnyElement,
     bloc2 : AnyElement
   }
   property {
     bloc1 in style Component.
     bloc2 in style Component.
     bloc1 directly monitored by bloc2
     or
     to components apply {
       exists(bloc | monitored(bloc1, bloc)
              and bloc directly monitored by bloc2)
     }
   }
 } mixfix (bloc1 monitored by bloc2);
```

- Cette analyse en AAL possède deux paramètres d'entrée ;
- Si les valeurs des paramètres sont du style *Component*…
- Vérifie si un *InputPort* du second élément est connecté à l'*OutputPort* du premier via un *Connector*.

- L'analyse est nommée dans le but d'être facilement utilisable dans les définitions de contraintes telles que celles qui sont présentées dans le paragraphe suivant ;
- Cette analyse possède deux paramètres d'entrée ;
- Si les valeurs de ces paramètres sont du style *Component*…
- Vérifie si *bloc1* est directement supervisé par *bloc2* (utilisation de la notation mixfix de l'analyse précédente) ;
- Sinon, parcoure récursivement la hiérarchie des blocs pour vérifier si *bloc1* est indirectement supervisé par *bloc2*.

```
directly_dependant is AAL_property {
   parameters {
      bloc1 : AnyElement,
      bloc2 : AnyElement
   }
   property {
      bloc1 in style Component.
      bloc2 in style Component.
      to bloc1.attributes apply{
         exists(in1,  out1 | in1.name = "blocDepIn"
           and out1.name = "blocDepOut"
           and to bloc2.attributes apply{
              exists(name2  | name2.name = "name"
                 and (out1.value = name2.value
                      or in1.value = name2.value)
           })
      }
   }
} mixfix (bloc1 directly dependent of bloc2);

dependant is AAL_property {
   parameters {
      bloc1 : AnyElement,
      bloc2 : AnyElement
   }
   property {
      bloc1 in style Component.
      bloc2 in style Component.
      bloc1 directly dependent of bloc2
      or
      to components apply {
         exists(bloc | bloc1 dependant of bloc
                and bloc directly dependent of bloc2)
      }
   }
} mixfix (bloc1 dependant of bloc2);
```

- Cette analyse en AAL possède deux paramètres d'entrée ;
- Si les valeurs des paramètres sont du style *Component*…
- Vérifie si *bloc1* dépend ou est dépendant du *bloc2*.

- Cette analyse possède deux paramètres d'entrée ;
- Si les valeurs de ces paramètres sont du style *Component*…
- Vérifie si *bloc1* est directement dépendant de *bloc2* (utilisation de la notation mixfix de l'analyse précédente) ;
- Sinon, parcoure récursivement les dépendances des blocs pour vérifier si *bloc1* est indirectement dépendant de *bloc2*.

```
bloc_superimposition is AAL_property {
   parameters {
     bloc1 : AnyElement,
     bloc2 : AnyElement
   }
   property {
     bloc1 in style Component.
     bloc2 in style Component.
     to bloc1.attributes apply {
       exists(w1, h1, x1, y1 | w1.name = "width"
              and h1.name = "height"
              and x1.name = "positionX"
              and y1.name = "positionY"
              and to bloc2.attributes apply {
         exists(w2, h2, x2, y2 | w2.name = "width"
                and h2.name = "height"
                and x2.name = "positionX"
                and y2.name = "positionY"
                and x2.value => (x1.value – w2.value)
                and x2.value <= (x1.value + w1.value)
                and y2.value => (y1.value – h2.value)
                and y2.value <= (y1.value + h1.value))
       })
     }
   }
 } mixfix (bloc1 superimposed to bloc2);

 homogeneous_graphical_info is AAL_property {
   parameters {
     bloc1 : AnyElement,
     bloc2 : AnyElement
   }
   property {
     bloc1 in style Component.
     bloc2 in style Component.
     to bloc1.attributes apply {
       exists(rf1, rfs1, rvzf1, lf1, lfs1, lvzf1 |
         rf1.name = "controlRoomFont"
         and rfs1.name = "controlRoomFontSize"
         and rvzf1.name = "controlRoomZoomFactor"
         and lf1.name = "labelFont"
         and lfs1.name = "labelFontSize"
         and lvzf1.name = "labelVisibleZoomFactor"
         and to bloc2.attributes apply {
           exists(rf2, rfs2, rvzf2, lf2, lfs2, lvzf2 |
           rf2.name = "controlRoomFont"
```

- Cette analyse possède deux paramètres d'entrée ;
- Si les valeurs de ces paramètres sont du style *Component*…
- Récupère les attributs de position et de dimension de *bloc1* et *bloc2* ;
- Vérifie si *bloc1* et *bloc2* sont totalement ou partiellement superposés

- Cette analyse possède deux paramètres d'entrée ;
- Si les valeurs de ces paramètres sont du style *Component*…
- Récupère les attributs graphiques de *bloc1* et *bloc2*.

```
            and rfs2.name = "controlRoomFontSize"
            and rvzf2.name =
            "controlRoomVisibleZoomFactor"
            and lf2.name = "labelFont"
            and lfs2.name = "labelFontSize"
            and lvzf2.name = "labelVisibleZoomFactor"
            and rf2.value = rf1.value
            and rfs2.value = rfs1.value
            and rvzf2.value = rvzf1.value
            and lf2.value = lf1.value
            and lfs2.value = lfs1.value
            and lvzf2.value = lvzf1.value)
          })
      }
   }
} mixfix (bloc1 homogeneous to bloc2);
}
…
```

- Vérifie si les valeurs de ses attributs (police et taille de police des zones de texte ainsi que facteur de zoom) sont homogènes pour *bloc1* et *bloc2*.

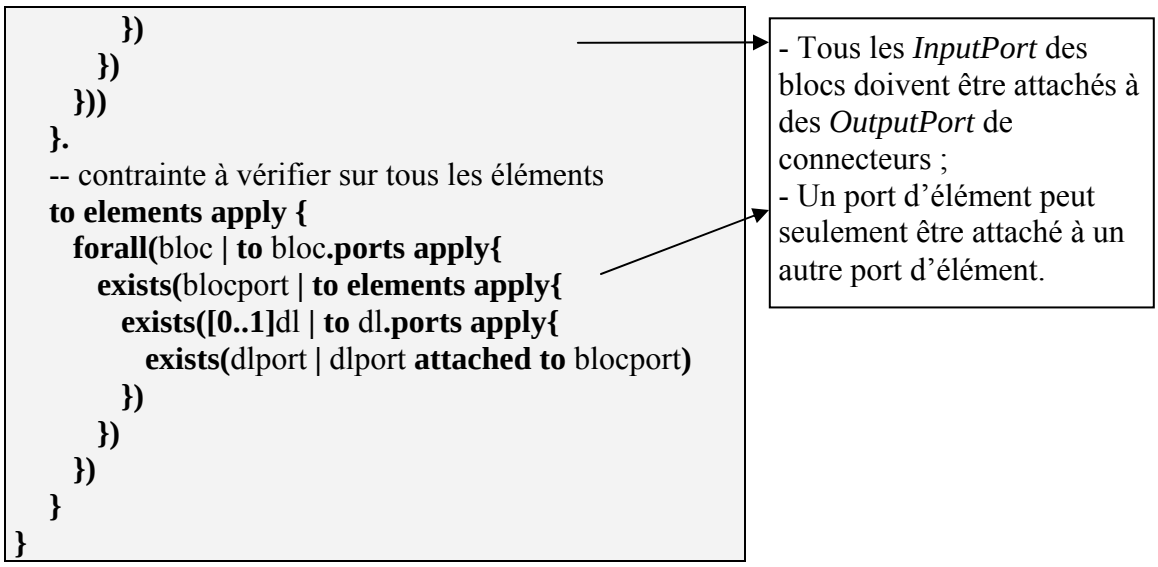
V. 3. 3. 3. 3 Contraintes

```
…
constraints {
   -- contraintes à vérifier sur les connecteurs
   to connectors apply {
      forall(c | c in style DataLink).
      forall (l1,l2 | l1 in style DataLink
               and (l2 in style DataLink
               implies not attached (l1,l2))
   }.
   -- contraintes à vérifier sur les composants
   to components apply {
      forall(c | c in style Equipment or c in style
      StatusBloc or c in style MetaStatus or c in style
      SystemStatus),
      --
      forall(c1,c2 | c1 directly monitored by c2 implies
      (c1 in style Equipment and c2 in style
      SystemStatus) or
      ((c2 in style MetaStatus) and ((c1 in style
      SystemStatus) or (c1 in style StatusBloc) or (c1 in
      style MetaStatus))) or
      ((c2 in style StatusBloc) and ((c1 in style
      Equipment) or (c1 in style SystemStatus) or
      (c1 in style MetaStatus) or (c1 in style
```

- Les connecteurs utilisés peuvent seulement être des éléments qui suivent le style *DataLink* ;
- Deux connecteurs ne peuvent pas être attachés ensemble ;
- Les composants utilisés par une *IndividualHCI* peuvent seulement être des éléments suivants les styles *Equipment, StatusBloc*, *MetaStatus*, *SystemStatus* ;
- Les éléments suivant le style *SystemStatus* ne peuvent superviser que des éléments qui suivent le style *Equipment* ;
- Les éléments suivant le style *MetaStatus* ne peuvent superviser que des éléments qui suivent les styles *SystemStatus*, *StatusBloc* ou *MetaStatus* ;

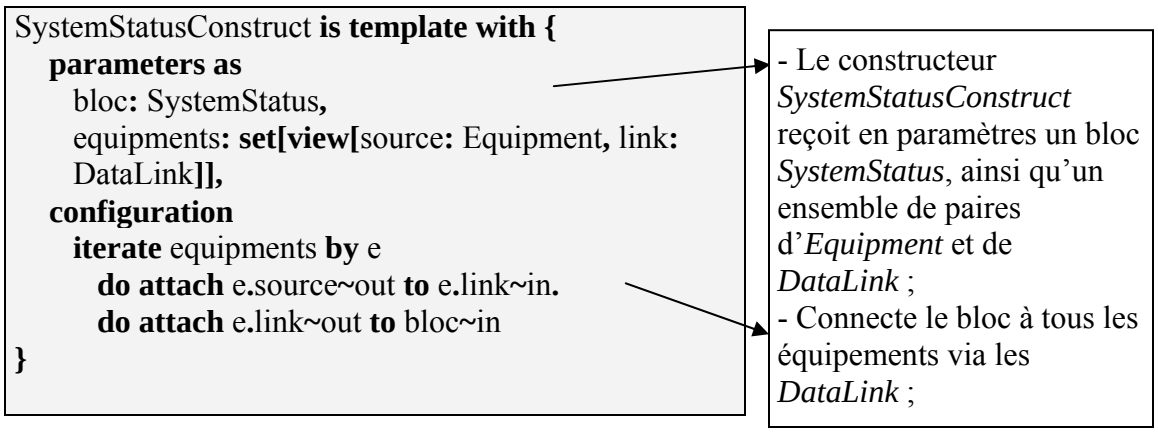
```
StatusBloc))),
--
forall(c | (c in style StatusBloc or c in style
MetaStatus or c in style SystemStatus)
and c monitored by c implies false),
--
forall(c | (c in style StatusBloc or c in style
MetaStatus or c in style SystemStatus)
and c dependant of c implies false),
--
forall(c1,c2 | (c1 in style StatusBloc or c1 in style
MetaStatus or c1 in style SystemStatus) and (c2 in
style StatusBloc or c2 in style MetaStatus or c2 in
style SystemStatus)
and c1 superimposed to c2 implies false),
--
forall(c1,c2 | (c1 in style StatusBloc or c1 in style
MetaStatus or c1 in style SystemStatus) and (c2 in
style StatusBloc or c2 in style MetaStatus or c2 in
style SystemStatus)
implies c1 homogeneous to c2),
--
forall(bloc | (bloc in style StatusBloc or bloc in
style SystemStatus) and forall (e1, e2 | e1 in style
Equipment and e1 monitored by bloc and e2 in
style Equipment and e2 monitored by bloc and to
e1.attributes apply {
  exists(ctrlroom1 | ctrlroom1.name =
  "controlRoom" and to e2.attributes apply {
    exists(ctrlroom2 | ctrlroom2.name =
    "controlRoom" and ctrlroom2.value =
    crtrlroom1.value implies to bloc.attributes
    apply {
      exists(blocctrlroom | blocctrlroom.name =
      "controlRoomText" and blocctrlroom.value
      = ctrlroom2.value)
    })
  })
}),
--
forall(bloc | to bloc.ports apply{
  forall(bip | bip in style InputPort implies
  connectors apply{
    exists(dl | to dl.ports apply {
      exists(dlout | dlout in style OutputPort and
      dlout attached to bip)
```

- Les éléments suivant le style *StatusBloc* ne peuvent superviser que des éléments qui suivent les styles *Equipment*, *SystemStatus*, *MetaStatus* ou *StatusBloc* ;
- Les boucles dans la chaîne d'acquisition sont interdites (Cf. contrainte **« boucle d'acquisition »** section IV.3.6) ;
- Les boucles de dépendance sont interdites (Cf. contrainte **« boucle de dépendance »** section IV.3.6)
- Deux blocs ne peuvent pas être superposés (Cf. contrainte **« superposition des blocs »** section IV.3.1) ;
- Les propriétés graphiques de tous les blocs d'une IHM doivent être homogènes (Cf. contraintes **« taille et police »** et **« zoom »** section IV.3.1) ;
- Si tous les équipements supervisés par un bloc sont associés à une même salle de contrôle, alors le bloc lui-même doit être associé à cette salle de contrôle (Cf. contrainte **« salle de contrôle »** section IV.3.2. Les contraintes **« nom de responsable », « numéro de bâtiment »**, et **« catégorie »** sont formalisées de façon similaire.



V. 3. 3. 3. 4 Constructeurs

Les constructeurs, ou ***templates***, sont utiles pour faciliter la construction d'architectures qui respectent les styles définis préalablement. Le constructeur *SystemStatusConstruct*, défini ci-dessous, permet de connecter un bloc *SystemStatus* à un ensemble d'équipements. Les constructeurs pour la construction des blocs *MetaStatus* et *StatusBloc* ne sont pas présentés ici. En effet, hormis le fait qu'ils ne se connectent pas qu'à des équipements, ils se formalisent de façon analogue.



V. 3. 3. 4 Formalisation du style de composant GlobalHCI

Le dernier extrait de code formel concerne la définition du style *GlobalHCI* :

```
GlobalHCI is style extending Component where {
    styles {
        IndividualHCI is style extending Component
        where {…}
    }
    constraints {
        to components apply {
            forall (c | c in style IndividualHCI)
        }
    }}
```

- Le style *GlobalHCI* est un *Component* ;
- Il est composé d'éléments suivant le style *IndividualHCI* ;
- Les composants utilisés par une *GlobalHCI* peuvent seulement être des éléments qui suivent le style *IndividualHCI*.

D'autres contraintes ont été spécifiées au niveau du style GlobalHCI. Par exemple, l'une d'entre elles concerne le positionnement graphique relatif des éléments suivant le style *IndividualHCI* sur les éléments suivant le style *GlobalHCI*.

Le style GTPM ainsi formalisé fournit le vocabulaire formel pour décrire les éléments qui composent les logiciels de supervision d'accélérateurs de particules, ainsi que les contraintes assurant leur conformité aux besoins exprimés par les salles de contrôle. Avant de pouvoir obtenir le code des logiciels, il est nécessaire de construire leurs architectures en utilisant le vocabulaire du style et en respectant les contraintes qui spécifie.

V. 4 Formalisation d'architectures de logiciels de supervision

Le but de cette section est d'expliquer comment les architectures formelles de logiciels de supervision sont obtenues à partir du style précédemment décrit. L'application simplifiée dont l'aspect graphique est présenté figure IV.9 est prise comme exemple, elle inclut :

- **5 blocs *SystemStatus* :**
    - le bloc « alimentation électrique » dont l'état est calculé à partir de l'état de deux *Equipment* (deux disjoncteurs) ;
    - le bloc « détection incendie » dont l'état est calculé à partir de l'état d'un *Equipment* (une centrale d'évacuation) ;
    - le bloc « détection gaz » dont l'état est calculé à partir de l'état d'un *Equipment* (une centrale de détection) ;
    - le bloc « alimentation aimants » dont l'état est calculé à partir de l'état d'un *Equipment* (un équipement « virtuel » fournissant un état résultant calculé à partir d'états de plusieurs alimentations) ;
    - le bloc « aimants » dont l'état est calculé à partir de l'état d'un *Equipment* (un équipement « virtuel » fournissant un état résultant calculé à partir d'états de plusieurs aimants).

- **4 liens de dépendance :**
    - un lien entre « alimentation électrique » et « détection incendie » ;
    - un lien entre « alimentation électrique » et « détection gaz » ;
    - un lien entre « alimentation électrique » et « alimentation aimants » ;
    - un lien entre « alimentation aimants » et « aimants ».

Le mécanisme d'instanciation permet de définir des architectures qui respectent le style. Comme la définition de ces architectures doit obligatoirement contenir la partie « commune » du style (ex : attributs), ce mécanisme l'implémente automatiquement. L'instanciation de style est différente de l'instanciation de méta-éléments car elle ne permet pas de créer directement des éléments architecturaux exécutables, mais elle fournit le cadre pour les définir par la suite.

Le code suivant définit un port qui suit le style *InputPort* préalablement formalisé. Il inclut la définition d'une connexion du type déclaré dans le style et la spécification d'un protocole. Si plus d'une connexion était déclarée, ou si le protocole comprenait un envoi de donnée, les contraintes du style seraient violées. Le port *Outport* est défini de façon analogue.

```
Inport is port in style InputPort with {
  connections
    in: Data;
  protocol
    replicate via in receive;
}
Outport is port in style OutputPort with {
  connections
    out: Data;
  protocol
    replicate via out send;}
```

Voici maintenant la définition du connecteur utilisés par le logiciel de supervision : *DatLink*. Celui-ci utilise un port *Inport* et un port *Outport*. S'il en utilisait plus, moins, ou des différents, les contraintes du style *DataLink* seraient violées.

```
DatLink is connector in style DataLink with {
  ports
    inputport: Inport,
    outputport: Outport;
  configuration
    new inputport. new outputport;
  routing
    replicate
      via inputport~in receive x: Any.
      via outputport~out send x;}
```

- *DatLink* suit le style *DataLink* ;
- Instanciation des ports ;
- Définition du protocole de transmission de données.

Les composants utilisés dans l'exemple traité suivent les styles *Equipment*, *StatusBloc*, et *IndividualHCI*. Le code suivant définit l'élément *Equip* qui suit le style *Equip*. *Equip* n'a qu'un *OutPort* et son comportement consiste à envoyer une valeur indiquant son état.

```
Equip is component in style Equipment with {
  ports
    outputport: Outport;
  configuration
    new outputport;
  computation
    replicate
      via outputport~out send state: State;
}
```

- *Equip* suit le style *Equipment* ;
- Instanciation du port ;
- Définition du protocole d'envoi de l'état.

La suite de l'exemple va supposer que les états envoyés par tous les équipements sont de type booléen. Il est donc nécessaire de définir deux types de bloc différents :

- un type de bloc (*SBloc1*) qui retourne un état résultant (au format standard GTPM : entier compris entre 0 et 3) en fonction **d'une** valeur d'entrée booléenne. Ce type de bloc sera utilisé pour définir les blocs « détection incendie », « détection gaz », « alimentation aimants », et « aimants » ;
- un type de bloc (*SBloc2*) qui retourne un état résultant (au format standard GTPM : entier compris entre 0 et 3) en fonction **de deux** valeurs d'entrée booléennes. Ce type de bloc sera utilisé pour définir le bloc « alimentation électrique ».

```
SBloc1 is component in style StatusBloc with {
  ports
    inputport: Inport,
    outputport: Outport;
  configuration
    new inputport. new outputport;
  computation
    replicate
      choose {
          {via inputport~in receive true.
          via outputport~out send 0}
      or
          {via inputport~in receive false.
          via outputport~out send 2}
      };
}
```

- *SBloc1* suit le style *StatusBloc* ;
- Instanciation des ports ;
- Définition de la règle de calcul de l'état résultant : si l'équipement envoie *true* le bloc envoie 0 (état « ok » de la convention GTPM), si l'équipement envoie *false* le bloc envoie 2 (état « faute » de la convention GTPM).

```
SBloc2 is component in style StatusBloc with {
  ports
    inputport1: Inport,
    inputport2: Inport,
    outputport: Outport;
  configuration
    new inputport1. new inputport2. new outputport;
  computation
    replicate
      choose {
          {via inputport1~in receive true and via
          inputport2~in receive true.
          via outputport~out send 0}
      or
          {via inputport1~in receive false and via
          inputport2~in receive false.
          via outputport~out send 2}
      or
          {(via inputport1~in receive false and via
          inputport2~in receive true) or
          (via inputport1~in receive true and via
          inputport2~in receive false).
          via outputport~out send 1}
      };
}
```

- *SBloc2* suit le style *StatusBloc* ;
- Instanciation des ports ;
- Définition de la règle de calcul de l'état résultant : si les deux équipements envoient *true* le bloc envoie 0 (état « ok »), si les deux équipements envoient *false* le bloc envoie 2 (état « faute »), et si les deux équipements envoient des valeurs différentes le bloc envoie 1 (état « avertissement »).

Le dernier élément à formaliser est celui qui suit le style *IndividualHCI*. Il s'agit d'un élément composite qui utilise les « templates » spécifiés en V.3.3.3.4 pour former le logiciel de supervision à partir des éléments définis ci-dessus.

```
ExampleHCI is component in style IndividualHCI
with {
  constituents
    detecIncendie: SBloc1,
    detecGaz: SBloc1,
    alimAimants: SBloc1,
    aimants: SBloc1,
    alimElec: SBloc2,
    centraleIncendie: Equip,
    centraleGaz: Equip,
    equipAlimAimants: Equip,
    equipAimants: Equip,
    disjoncteur1: Equip,
    disjoncteur2: Equip,
    dlink1: DatLink,
    dlink2: DatLink,
```

- *ExempleHCI* suit le style *IndividualHCI* ;
- Définition des constituants de l'architecture : les 5 blocs de l'IHM (4 de type *SBloc1* et 1 de type *SBloc2*) ; les 6 équipements que supervisent les 5 blocs (de type *Equip*) et les 6 liens qui transmettent les données des équipements aux blocs (de type *DatLink*).

```
    dlink3: DatLink,
    dlink4: DatLink,
    dlink5: DatLink,
    dlink6: DatLink;
configuration
    new centraleIncendie initialised with {
        "TCR" as controlRoom, 870 as buildingNumber,
        "SECU" as system, "EVAC" as subsystem,
        "Durand" as responsible, "bool" as dataType,
        "centrale001.status" as equipAddress;}.
    new centraleGaz initialised with {…}.
    new equipAlimAimants initialised with {…}.
    new equipAimants initialised with {…}.
    new disjoncteur1 initialised with {…}.
    new disjoncteur2 initialised with {…}.
    new dlink1. new dlink2. new dlink3. new dlink4.
    new dlink5. new dlink6. new depElec_Inc.
    new depElec_Gaz. new depElec_Alim.
    new depAlim_Aimants.
    new detecIncendie initialised with {
        30 as width, 15 as height, 20 as positionX,
        20 as positionY, "TCR" as controlRoomText,
        "Times" as controlRoomFont,
        14 as controlRoomFontSize,
        0.5 as controlRoomVisibleZoomFactor,
        "Détection Incendie" as labelText,
        "Times" as labelFont, 16 as labelFontSize,
        0.5 as labelVisibleZoomFactor,
        870 as buildingNumber,
        "SECU/EVAC" as category,
        "Durand" as responsible, 2 as priority
        "alimElec" as blocDepIn;}.
    new detecGaz initialised with {…}.
    new alimAimants initialised with {…}.
    new aimants initialised with {…}.
    new alimElec initialised with {…}.

    SystemStatusConstruct(detecIncendie,
            set(view(centraleIncendie, dlink1))).
    SystemStatusConstruct(detecGaz,
            set(view(centraleGaz, dlink2))).
    SystemStatusConstruct(alimAimants,
            set(view(equipAlimAimants, dlink3))).
    SystemStatusConstruct(aimants,
            set(view(equipAimants, dlink4))).
    SystemStatusConstruct(alimElec,
```

- Définition de la configuration du logiciel de supervision ;
- Instanciation de tous les constituants déclarés dans la section précédente de l'architecture. Les attributs des équipements et des blocs sont initialisés lors de l'instanciation (pour ne pas surcharger cette description, seules les initialisations des attributs d'un bloc et d'un équipement sont décrites).

- Attribut indiquant que le bloc *detecIncendie* et dépendant du bloc *alimElec*.

- Connexion des équipements aux blocs via des liens de transmission de données à l'aide du constructeur *SystemStatusConstruct* défini dans le style *IndividualHCI*.

```
            set(view(disjoncteur1, dlink5),
                view(disjoncteur2, dlink6)));
}
```

L'extrait de code précédent formalise une architecture de logiciel de supervision à partir du style GTPM. Cette architecture utilise les éléments architecturaux définis dans le style, comme les types, les ports, les connecteurs, les composants, et les « templates ». Ainsi, l'utilisation du style simplifie la conception de l'architecture par rapport à une conception directe. De plus, l'utilisation d'un outil de vérification permet de s'assurer que les contraintes spécifiées par le style, notamment celles portant sur les attributs, sont bien respectées par les architectures qui le suivent.

V. 5 Conclusion

Ce chapitre a présenté les principales techniques de formalisation d'architectures logicielles et de styles architecturaux au moyen des langages ArchWare. L'utilisation de ces langages, notamment de l'extension C&C, a permis d'obtenir un style définissant un vocabulaire spécifique au développement de logiciels de supervision d'accélérateurs. Ce vocabulaire inclut notamment un ensemble de styles de composants (*Equipment*, *StatusBloc*, *SystemStatus*, *MetaStatus*, *IndividualHCI*, *GlobalHCI*) et un style de connecteur (*DataLink*). Le style GTPM comporte, de plus, la spécification des contraintes que les architectures qui le suivent doivent respecter. Ceci permet de garantir que les logiciels de supervision conçus à partir du style satisferont les besoins. Enfin, la dernière section a montré, par un exemple, comment le style est utilisé pour la construction d'architectures de logiciels de supervision.

La suite de cette thèse va maintenant expliquer comment le style est exploité pour guider le processus de développement des logiciels de supervision. Le chapitre suivant présente l'intégration du style à l'environnement de développement SEAM, dont le but est l'automatisation partielle de la construction d'architectures respectant les contraintes du style. Par la suite sera présentée la problématique relative à l'évolution du style et son environnement en fonction de l'évolution des besoins.

Chapitre VI Conception et implémentation de l'environnement

Chapitre VI : Conception et implémentation de l'environnement

VI. 1 Introduction

Les précédents chapitres ont présenté une approche de conception d'IHMs (logiciels de supervision) au moyen d'un environnement de développement, contrôlé par l'utilisation d'un style formel spécifique à la supervision du redémarrage des accélérateurs de particules. Ils ont en outre décrit le contexte d'utilisation de cet environnement, et traité des techniques disponibles pour exploiter le style formalisé.

Le but du présent chapitre est de détailler l'implémentation de cette approche. La section VI.2 présente tout d'abord l'intégration du style GTPM à l'architecture de l'environnement de développement, et la section VI.3 décrit en détail l'implémentation des différents modules de cet environnement ainsi que leurs interactions.

VI. 2 Intégration du style dans l'architecture de l'environnement de développement

Cette section présente l'intégration du style GTPM dans l'architecture de l'outil SEAM. Cet outil doit constituer un environnement de développement permettant à des utilisateurs non informaticiens de développer des IHMs via une interface intuitive contrainte par le style, et permettre de générer leur code interprétable par le système de supervision.

L'approche choisie dans le cadre de cette thèse consiste à construire des logiciels de supervision à partir d'un environnement de développement dédié, SEAM, intégrant les contraintes d'un style architectural, le style GTPM. Afin d'illustrer le processus de développement architectural utilisé dans ce contexte, la figure VI.1 reprend le processus proposé figure IV.5. Comme l'indique cette figure, les différentes contraintes du style GTPM sont intégrées à l'architecture de SEAM. Cette architecture est faite de plusieurs composants (une base de données, une interface avec la base de données, un modélisateur graphique, et un vérificateur de conformité), intégrant chacun une sous partie des contraintes du style. Le fait que toutes les contraintes du style soient spécifiées au niveau de l'architecture permet de produire des IHMs qui, par construction, vérifient ces contraintes. Il est à noter que la base de données est un composant particulier, son schéma fait partie de l'architecture de l'environnement (il comporte certaines contraintes du style) et ses tables contiennent toutes les informations concernant les architectures

d'IHMs. Ces informations sont contraintes par le schéma de la base mais aussi par les autres composants de l'architecture de l'environnement de développement.

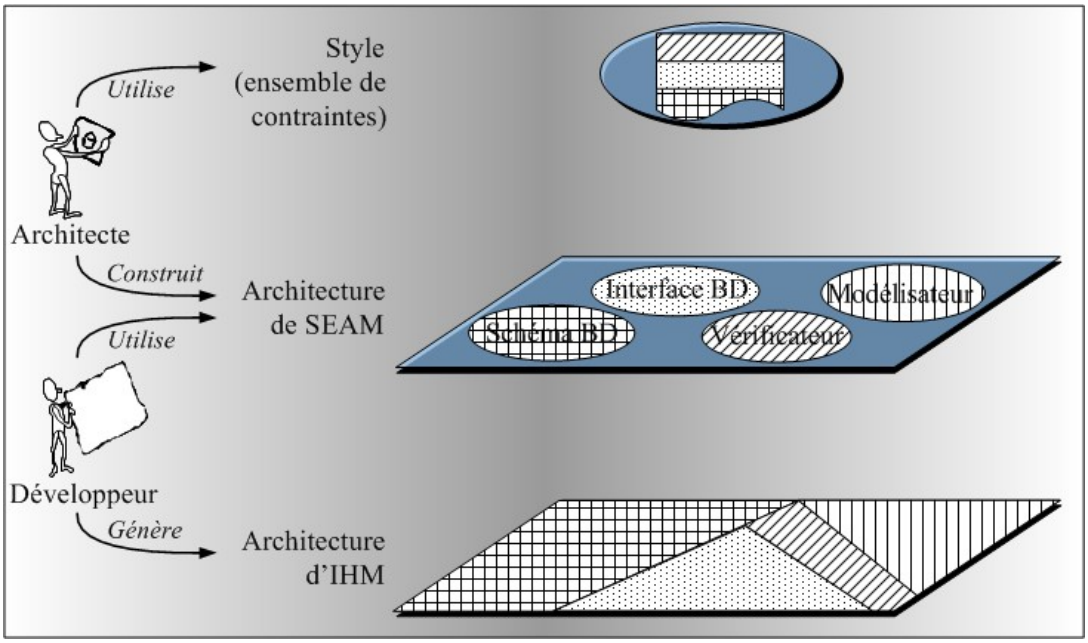


Figure VI. 1 : Construction d'IHMs conformes au style

La section suivante expose la formalisation de l'architecture de l'environnement de développement.

VI. 2. 1 Formalisation de l'architecture de l'environnement de développement

Cette section présente de façon non détaillée la formalisation de l'architecture de l'environnement de développement. La figure VI.2 présente l'architecture décrite en utilisant la notation UML-ADL [Alloui et Oquendo 2003]. Afin de ne pas surcharger la figure, les connexions ne sont pas représentées : seuls les composants, les connecteurs, et leurs ports respectifs sont présents. Les différents attachements spécifiant des interactions sont représentés au niveau des ports.

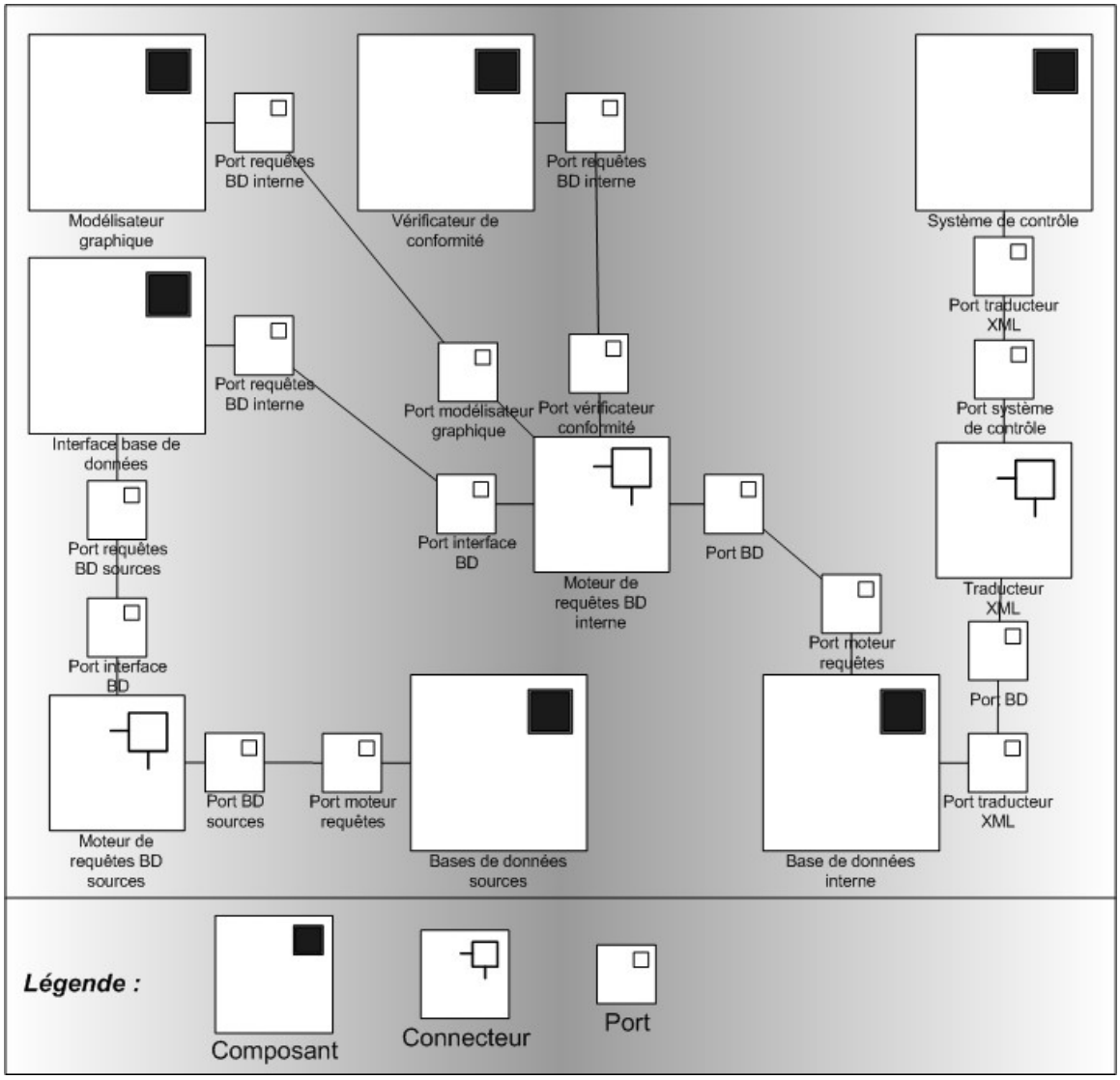


Figure VI. 2 : Architecture de l'environnement de développement

Cette figure représente les différents composants et connecteurs intervenant dans l'architecture de l'environnement de développement d'IHMs. Les composants sont :

- la base de données interne : il s'agit de la base contenant toutes les informations nécessaires à la production d'IHMs. Celle-ci interagit avec le connecteur « Traducteur XML » pour produire le code XML interprétable par le système de supervision ;
- les bases de données sources : il s'agit d'un composant externe à l'environnement de développement. Il représente les bases existantes contenant des informations utiles pour définir les logiciels de supervision ;
- l'interface base de données : ce composant permet à l'utilisateur de sélectionner des informations des bases de données sources (via le connecteur « Moteur de requête BD sources », et de sélectionner/insérer/mettre à jour des informations dans la base de données interne (via le connecteur « Moteur de requête BD interne ») ;

- le modélisateur graphique : ce composant permet à l'utilisateur de créer/modifier graphiquement les logiciels de supervision. Il interagit avec la base de données interne via le connecteur « Moteur de requête BD interne » ;
- le vérificateur de conformité : ce composant consulte les informations de la base interne (via le connecteur « Moteur de requête de la base interne ») et les analyse pour informer l'utilisateur du respect ou non respect du style par les logiciels de supervision définis dans la base ;
- le système de contrôle : il s'agit d'un composant externe à l'environnement de développement. Il représente le système de contrôle existant interprétant le code XML extrait de la base interne.

La suite de cette section présente la formalisation de cette architecture en langage ArchWare C&C.

Les ports « Port requêtes BD interne », « Port requêtes BD sources », « Port BD », et « Port BD sources », représentés sur la figure VI.2 sont du type *Port_Bd_In*. Ce port est utilisé pour envoyer des requêtes sous forme de chaîne de caractères et pour fournir en retour le résultat de la requête dans une liste de tuples (lignes de base de données).

```
archetype Port_Bd_In is port with {
    connections
        requete : connection[String],
        resultat : connection[Tuple{}] ;
    protocol
        replicate
            via requete send.
            via resultat receive;
}
```

Les ports « Port vérificateur conformité », « Port modélisateur graphique », « Port interface BD », « Port moteur requêtes », et « Port traducteur XML » représentés sur la figure sont du type *Port_Bd_Out* :

```
archetype Port_Bd_Out is port with {
    connections
        requete : connection[String],
        resultat : connection[Tuple{}] ;
    protocol
        replicate
            via requete receive.
            via resultat send;
}
```

Le port « Port système de contrôle » représenté sur la figure est du type *Port_Syst_Ctrl*. Ce port est utilisé pour envoyer des requêtes sous forme de chaîne de caractères et pour fournir en retour le résultat de la requête sous la forme d'un fichier au format XML.

```
archetype Port_Syst_Ctrl is port with {
    connections
        requete : connection[String],
        fluxXML : connection[String] ;
    protocol
        replicate
            via requete receive.
            via fluxXML send;
}
```

Le connecteur « Moteur de requêtes BD sources » comporte un port pour se connecter aux « Bases de données sources » et un pour se connecter à « l'interface base de données ». Ce connecteur permet de transmettre la requête et de retourner son résultat en gérant les connexions aux bases de données sources.

```
archetype Moteur_BD_Sources is connector with {
    ports
        interface_bd_port: Port_Bd_Out,
        bd_port: Port_Bd_In;
    configuration
        new interface_bd_port. new bd_port;
    routing
        replicate
            via interface_bd_port~requete receive.
            unobservable.
                -- ouverture de la connexion avec
                -- la base de données source
            via bd_port~requete send.
            via bd_port~resultat receive.
            unobservable.
                -- fermeture de la connexion
            via interface_bd_port~resultat send;
}
```

Le connecteur « Moteur de requêtes BD interne » comporte un port pour se connecter à la « Base de données interne » et trois pour se connecter à « l'interface base de données », au « Modélisateur graphique » et au « Vérificateur de conformité ». Ce connecteur permet de gérer les connexions et le routage de requêtes entre la base de données interne de l'environnement et les autres composants.

```
archetype Moteur_BD_Interne is connector with {
    ports
        interface_bd_port: Port_Bd_Out,
        modelisateur_port: Port_Bd_Out,
        verificateur_port: Port_Bd_Out,
```

```
        bd_port: Port_Bd_In;
    configuration
        new interface_bd_port. new modelisateur_port.
        new verificateur_port. new bd_port;
    routing
        replicate
            choose{
                {via interface_bd_port~requete receive.
                via bd_port~requete send.
                via bd_port~resultat receive.
                via interfacebd_port ~resultat send};
            or
                {via modelisateur_port~requete receive.
                via bd_port~requete send.
                via bd_port~resultat receive.
                via modelisateur_port ~resultat send};
            or
                {via verificateur_port~requete receive.
                via bd_port~requete send.
                via bd_port~resultat receive.
                via verificateur_port ~resultat send};
            }
}
```

Le connecteur « Traducteur XML » comporte un port pour se connecter à la « Base de données interne » et un pour se connecter au « Système de contrôle ». Il transmet la requête provenant du système de contrôle, et retourne son résultat en format XML.

```
archetype Traducteur_XML is connector with {
    ports
        systeme_controle_port: Port_Syst_Ctrl,
        bd_port: Port_Bd_In;
    configuration
        new systeme_controle_port. new bd_port;
    routing
        replicate
            via systeme_controle_port~requete receive.
            via bd_port~requete send.
            via bd_port~resultat receive.
            unobservable.        -- conversion XML
            via systeme_controle_port~fluxXML send;
}
```

Le composant « Modélisateur graphique » comporte un port pour se connecter à la « Base de données interne » (le composant vérificateur de conformité fonctionne de la même manière) :

```
archetype Modelisateur_Graphique is component with {
    ports
        bd_port: Port_Bd_In;
    attributes
        …
    consituents
        …
    configuration
        new bd_port;
    computation
        replicate
            unobservable.
            -- interaction avec l'utilisateur : spécification d'informations
            -- concernant les IHMs et demande d'insertion/modification
            -- dans la base de données interne (requete)
            using(requete) verify {…}
            -- vérifie que les informations à insérer respectent les
            -- contraintes du style avant l'envoi de la requête
            via bd_port~requete send.
            via bd_port~resultat receive.
            unobservable;-- interaction avec l'utilisateur
}
```

Le composant « Interface base de données » comporte un port pour se connecter à la « Base de données interne » et un pour se connecter aux « Bases de données sources » :

```
archetype Interface_BD is component with {
    ports
        bd_sources_port: Port_Bd_In.
        bd_interne_port: Port_Bd_In;
    attributes
        …
    consituents
        …
    configuration
        new bd_source_port. new bd_interne_port;
    computation
        replicate
            unobservable. -- interaction avec l'utilisateur
            choose{
            -- consultation BD sources
                {via bd_source_port~requete send.
                via bd_source_port~resultat receive}
            or
            -- consultation ou spécification d'informations concernant les
```

```
                -- IHMs et demande d'insertion/modification dans la base de
                -- données interne (requete)
                    {using(requete) verify {…}
                    -- vérifie que les informations à insérer respectent les
                    -- contraintes du style avant l'envoi de la requête
                    via bd_interne_port~requete send.
                    via bd_interne_port~resultat receive}}
                unobservable;-- interaction avec l'utilisateur
}
```

Le composant « Base de données interne » comporte un port pour se connecter au connecteur « Moteur de requêtes BD interne », et un pour se connecter au connecteur « Traducteur XML » :

```
archetype BD_Interne is component with {
    ports
        traducteur_xml_port: Port_Bd_Out.
        moteur_requetes_port: Port_Bd_Out;
    attributes
        …
    consituents
        …
    configuration
        new traducteur_xml_port.
        new moteur_requetes_port;
    computation
        replicate
            choose{
                {via moteur_requetes_port~requete receive.
                using(requete) verify {…}
                -- vérifie que les informations à insérer respectent les
                -- contraintes du schéma avant l'exécution de la requête
                unobservable.
                via moteur_requetes_port~resultat send}
            or
                {via traducteur_xml_port~requete reveive.
                unobservable. -- execution de la requête
                via traducteur_xml_port~resultat send}};
}
```

Les composants « Bases de données sources » et « Système de contrôle » ne sont pas décris ici car ils sont externes à l'environnement de développement.

La section suivante explique comment l'environnement est construit à partir de cette architecture.

VI. 2. 2 Construction de l'environnement à partir de l'architecture

Une fois que l'architecture de l'environnement est définie, et que les contraintes du style lui sont intégrées, celle-ci est utilisée pour construire l'environnement de développement dédié à la production d'IHMs de supervision d'accélérateurs de particules, SEAM. La figure VI.3 représente cette solution.

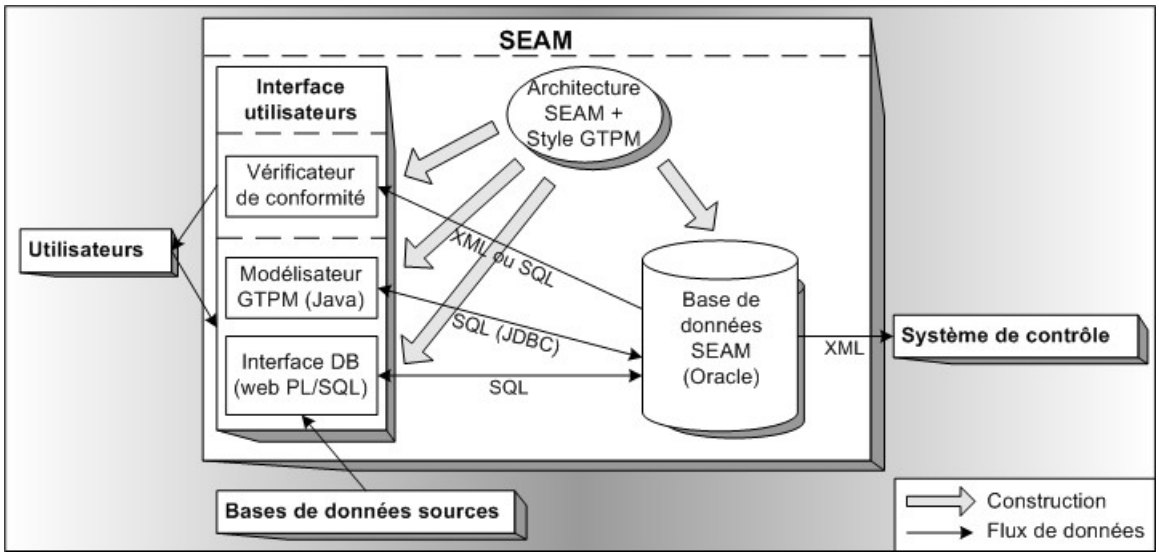


Figure VI. 3 : Solution implémentée

Dans cette approche les contraintes et le vocabulaire formalisés par le style GTPM ont été intégrés dans les différents modules de l'outil SEAM (détaillés section VI.3) :

- **la base de données** (composant « Base de données interne » de l'architecture) : base de données Oracle, conçue pour stockér les informations concernant tous les éléments composant les IHMs ;
- **interface web** (composant « Interface base de données » de l'architecture) : interface d'exploitation de la base de données implémentée en PL/SQL. Elle permet aux utilisateurs de sélectionner des informations de bases de données sources pour les utiliser dans la construction des éléments constituant les IHMs, et de consulter, mettre à jour, et insérer des informations dans la base de données SEAM ;
- **le modélisateur GTPM** (composant « Modélisateur graphique » de l'architecture) : il s'agit d'une interface graphique implémentée en Java, et construite à partir de l'outil JViews introduit dans le chapitre 2. Cette interface fournit les éléments et les outils pour « dessiner » les IHMs. Les IHMs ainsi construites sont stockées dans la base de données SEAM ;
- **le vérificateur de conformité** (composant « Vérificateur de conformité » de l'architecture) : il permet de vérifier que les architectures construites par les utilisateurs et stockées dans la base SEAM respectent bien les contraintes définies dans le style GTPM.

Les utilisateurs de SEAM construisent les IHMs au moyen de l'interface web et du modélisateur GTPM. L'ordre d'utilisation n'a pas d'importance : il est possible de spécifier en premier lieu, via le modélisateur, les éléments graphiques des IHMs, puis de spécifier ensuite les données qui leur sont associées (concernant les équipements à superviser et la logique de représentation des état) via l'interface web, ou le contraire. Toutes les informations saisies via ces interfaces sont stockées dans la base de données de SEAM, et converties à la demande en code exploitable par le système de contrôle.

La plupart des contraintes spécifiées par le style GTPM sont « codées » dans les différents modules de SEAM. D'autres contraintes ne le sont pas pour des raisons de flexibilité. Il s'agit de contraintes que l'utilisateur doit pouvoir violer dans certains cas. Toutefois, l'utilisateur doit être conscient de ces violations du style, elles ne doivent pas être le résultat d'erreurs. C'est pour cette raison que le vérificateur de conformité a été mis en place. Il permet d'indiquer à l'utilisateur quelles contraintes du style sont respectées/violées par les IHMs qu'il a produites avant que leur code soit généré pour être exploité par le système de contrôle/supervision.

La section suivante présente en détail les différents modules de SEAM, et indique comment les contraintes du style GTPM sont prises en compte par ceux-ci.

VI. 3 Implémentation de SEAM

VI. 3. 1 Base de données

La base de données de l'outil SEAM est une base de données Oracle conçue à l'aide de l'outil Designer. Le schéma complet de cette base est fourni en Annexe 3. Afin de faciliter la description de sa structure, la présente section la découpe en cinq parties correspondant aux paragraphes suivants.

Les principaux symboles utilisés sur les schémas sont les suivants :

- le caractère # indique que le champ devant lequel il se trouve fait partie de la clé primaire de la table ;
- les caractères * ou ☐ indiquent qu'il est obligatoire (*) ou non (☐) de spécifier une valeur pour le champ devant lequel il se trouve ;
- l'icône ☰ indique que la valeur du champ devant lequel il se trouve est contrainte par une table de domaine ;
- les caractères A ou 789 indiquent le type des valeurs des champs devant lesquels ils se trouvent (A : chaîne de caractères – type Oracle Varchar2 ; 789 : numérique – type Oracle Number) ;
- les liens entre les tables représentent les clés étrangères.

VI. 3. 1. 1 Structure des blocs

La partie centrale de la base de données de SEAM gère les informations des blocs permettant la représentation de l'état d'équipements. Les tables correspondantes, fournissant notamment des données concernant l'aspect graphique des blocs (Cf. figure VI.4) et les équipements qu'ils supervisent (Cf. figure VI.5), sont les suivantes :

- **GTPBLOCS :** cette table contient les informations élémentaires relatives aux blocs : nom, description, catégorie, salle de contrôle responsable de sa

supervision, personne responsable, position géographique, adresse résultante (adresse où est lue la valeur résultante calculée à partir des états des équipements associés au bloc), type du bloc (Cf. section VI.3.1.2) **…**;

- **GTPBLOCINFOTAGS :** cette table contient les informations concernant les valeurs d'informations éventuellement associées aux blocs. Il s'agit de valeurs n'intervenant pas dans le calcul de l'état d'un bloc, mais fournies aux opérateurs des salles de contrôle à titre informatif (ex : mesure de tension, mesure de température). Ces valeurs d'informations correspondent à des propriétés (ou attributs) prédéfinis des objets graphiques du modélisateur GTPM (Cf. section VI.3.3) ;
- **GTPPROPERTY :** cette table donne la description des propriétés correspondant aux valeurs d'informations. Par exemple, un enregistrement de cette table peut spécifier qu'un bloc représentant l'état d'un système électrique possède une mesure de tension comme valeur d'information ;
- **GTPPALETTE :** cette table décrit les différents types graphiques que les blocs peuvent avoir. Un type graphique donne des dimensions par défaut et les valeurs d'informations associées (ex : un bloc étant de type graphique « mesure tension » aura les dimensions 200x50 et comportera une valeur d'information indiquant une mesure de tension). Cette table est utilisée pour créer la palette d'objets graphiques du modélisateur GTPM (Cf. section VI.3.3) ;
- **GTPGTYPESINFOTAGS :** il s'agit de la table d'association entre les propriétés (valeurs d'information) et les objets graphiques de la palette. Un objet graphique peut avoir n propriétés et une propriété peut être associée à n objets graphiques ;

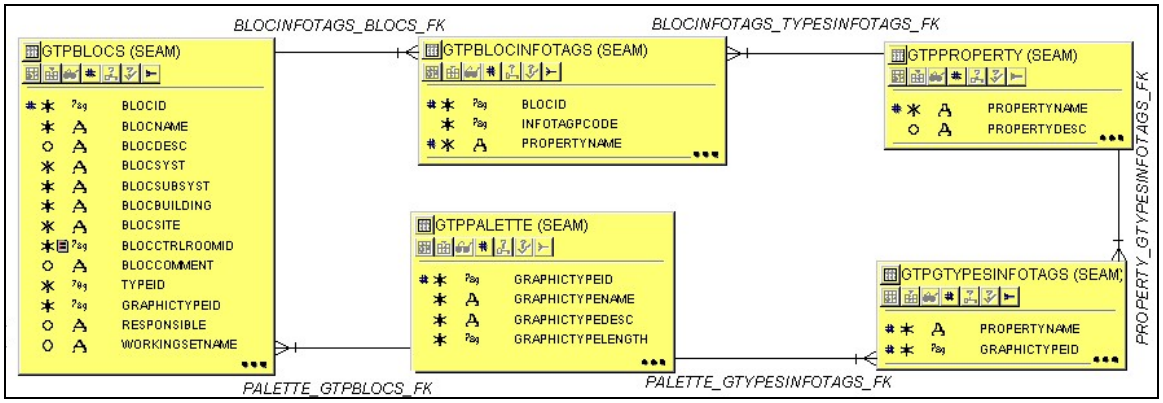
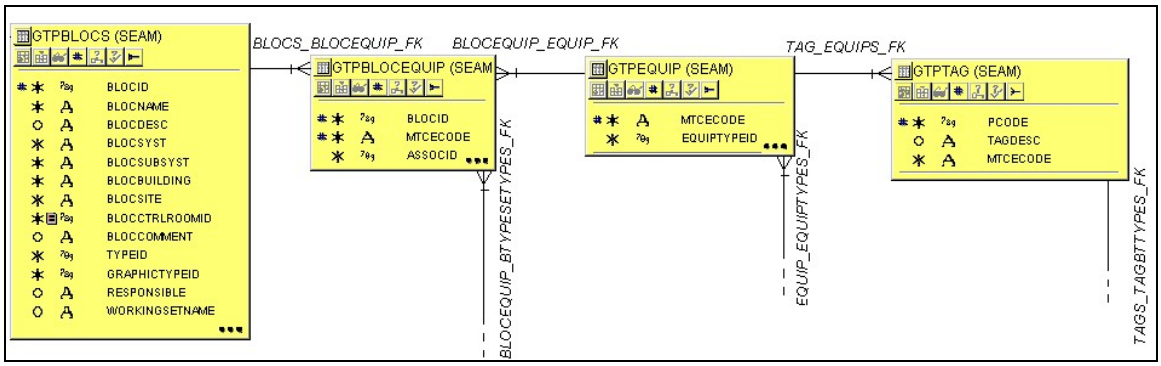


Figure VI. 4 : Informations graphiques des blocs

- **GTPEQUIP :** cette table contient les informations concernant les équipements intervenant dans l'opération des accélérateurs de particules et pouvant être associés aux (supervisés par les) blocs ;
- **GTPBLOCEQUIP :** il s'agit de la table d'association entre les blocs et les équipements. Un bloc peut superviser n équipements et un équipement peut être supervisé par n blocs ;
- **GTPTAG :** cette table contient les informations concertant les valeurs rendues disponibles par les équipements.



Figures VI. 5 : Equipements supervisés

Les champs de la table GTPBLOCS correspondent à une partie des attributs des composants suivant le style *StatusBloc* du style GTPM. D'autres attributs (ex : polices de caractères) ne sont pas spécifiés explicitement dans la base de données, mais le sont au niveau du modélisateur GTPM (Cf. section VI.3.3.3) en fonction du type d'objet graphique choisi dans la table GTPPALETTE. De plus, les champs GRAPHICTYPELENGTH et GRAPHICTYPEHEIGHT de la table GTPPALETTE fixent les dimensions des blocs utilisables sur les IHMs, ce qui permet de satisfaire la contrainte du style spécifiant que la taille des blocs doit être uniforme.

Le style GTPM distingue trois styles de blocs (*StatusBloc*, *MetaStatus*, *SystemStatus*), mais ce n'est pas le cas de la base de données. La structure de la base spécifie que les blocs ne peuvent superviser que des équipements, et non pas d'autres blocs. Ceci est la caractéristique des *StatusBloc*. Toutefois les équipements de la table GTPEQUIP peuvent être des « équipements virtuels » correspondant à des blocs précédemment créés. Chaque bloc peut-être vu comme un équipement dont la valeur résultante est supervisée par d'autres blocs. La base de données permet donc la description des trois styles de blocs, et leur distinction est faite par son interface d'exploitation.

Concernant la structure des IHMs spécifiée par le style GTPM, la base de donnée prévoit bien qu'un équipement puisse être supervisé par un nombre quelconque (0..n) de blocs (la clé primaire de la table d'association GTPBLOCEQUIP, étant l'identifiant du bloc combiné à l'identifiant de l'équipement). De même, un bloc peut superviser n équipements, mais la structure de la base n'impose pas qu'au moins un équipement soit associé à celui-ci, ce qui est pourtant contraint par le style. Ceci est laissé possible dans le but de donner de la flexibilité de développement : un utilisateur peut créer dans un premier temps un bloc (avec ses attributs), et lui associer un équipement plus tard. Toutefois l'outil doit garantir qu'au moment de la génération du code tous les blocs des IHMs soient bien associés à au moins un équipement (géré au niveau de l'interface d'exploitation, cf. section VI.3.2.1).

Le style GTPM énonce une contrainte spécifiant que la salle de contrôle associée à un bloc (champ BLOCCTRLROOMID de ta table GTPBLOCS) doit bien correspondre à une salle de contrôle existante. Cette contrainte est implémentée par l'utilisation d'un « domaine » de valeurs. La valeur affectée au champ BLOCCTRLROOMID ne peut être qu'une valeur prédéfinie dans la table de « domaine », c'est à dire aujourd'hui : TCR, PCR, MCR et QCR.

VI. 3. 1. 2 Types de blocs et règles

Le but de cette partie de la base de données est de définir des informations génériques pouvant être réutilisées par plusieurs blocs. Ceci s'applique notamment au niveau des règles permettant d'obtenir un état résultant à partir de l'état de plusieurs valeurs de plusieurs équipements. En effet, la logique de telles règles, parfois complexe, est souvent identique pour de nombreux blocs. La différence se situe uniquement au niveau des équipements sur lesquels s'applique la logique.

Supposons par exemple que dix blocs « Etat_18kV » soient utilisés pour indiquer l'état résultant de dix couples (Barre18kV ; disjoncteur). Il n'est pas nécessaire de définir dix règles (Etat_18kV_1 = f(Barre18kV_1 ; disjoncteur_1) ; Etat_18kV_2 = f(Barre18kV_2 ; disjoncteur_2) ; …), mais une seule règle générique suffit (Etat_18kV = f(Barre18kV ; disjoncteur). Un utilisateur voulant insérer un nouveau bloc dans la base n'est alors pas obligé de créer systématiquement une nouvelle règle. Il lui suffit de préciser le type de bloc associé (correspondant à un type de règle) et d'affecter des équipements réels aux variables utilisées dans la logique de la règle.

Dans ce contexte les tables utilisées sont les suivantes (Cf. figure VI.6) :

- **GTPBLOCTYPES :** cette table contient les informations élémentaires relatives aux types de blocs : nom, description, catégorie, salle de contrôle responsable de sa supervision, personne responsable, position géographique. Certains de ces champs (descriptions, commentaires) peuvent ne pas être remplis ;
- **GTPEQUIPTYPES :** cette table contient les informations concernant les types d'équipements pouvant être associés aux types de blocs (ex : « disjoncteur18kV »). Notons que les équipements de la table **GTPEQUIP** sont associés aux types d'équipements de cette table ;
- **GTPBTYPESETYPES :** il s'agit de la table d'association entre les types de blocs et les types d'équipements. Un type de bloc peut superviser n types d'équipements et un type d'équipement peut être supervisé par n types de blocs. Chacune de ces associations possède un nom (ex : « disjoncteur ») et un identifiant ;
- **GTPTAGTYPES :** cette table contient les informations concernant les types de valeurs existants ;
- **GTPETYPESTTYPES :** il s'agit de la table d'association entre les types d'équipements et les types de valeurs. Un type d'équipement peut être associé n types de valeurs et un type de valeur peut être associé à n types d'équipements. Par exemple, l'équipement de type « disjoncteur18kV » peut fournir des valeurs de type « position » (ouvert/fermé) et « défaut » (vrai/faux) ;
- **GTPBLOCTYPETAGS :** il s'agit d'une table d'association entre les types de valeurs et les associations type de bloc/type d'équipement (table GTPBTYPESETYPES). Elle permet par exemple de spécifier que l'association « disjoncteur » (utilisant le type d'équipement « disjoncteur18kV ») ne possède qu'une valeur de type « position » (ouvert/fermé). Cette association est notée « disjoncteur.position » ;
- **GTPRULES :** cette table contient les informations concernant les règles associées aux types de blocs (ex : bloc vert si disjoncteur.position = « fermé ») ;

- **GTPTAGRULES :** cette table indique quelles associations (ex : « disjoncteur.position ») sont utilisées dans la logique de quelle règle ;
- **GTPTAGBTYPETAGS :** il s'agit de la table d'association entre les données de la table GTPBLOCTYPETAGS et les valeurs rendues disponibles par les équipements physiques (table **GTPTAG**). Elle permet de spécifier par exemple que pour un bloc « Etat_18kV_bâtiment1 » l'association « disjoncteur.position » sera associée à la valeur physique « disjoncteur18kV_001_pos ».

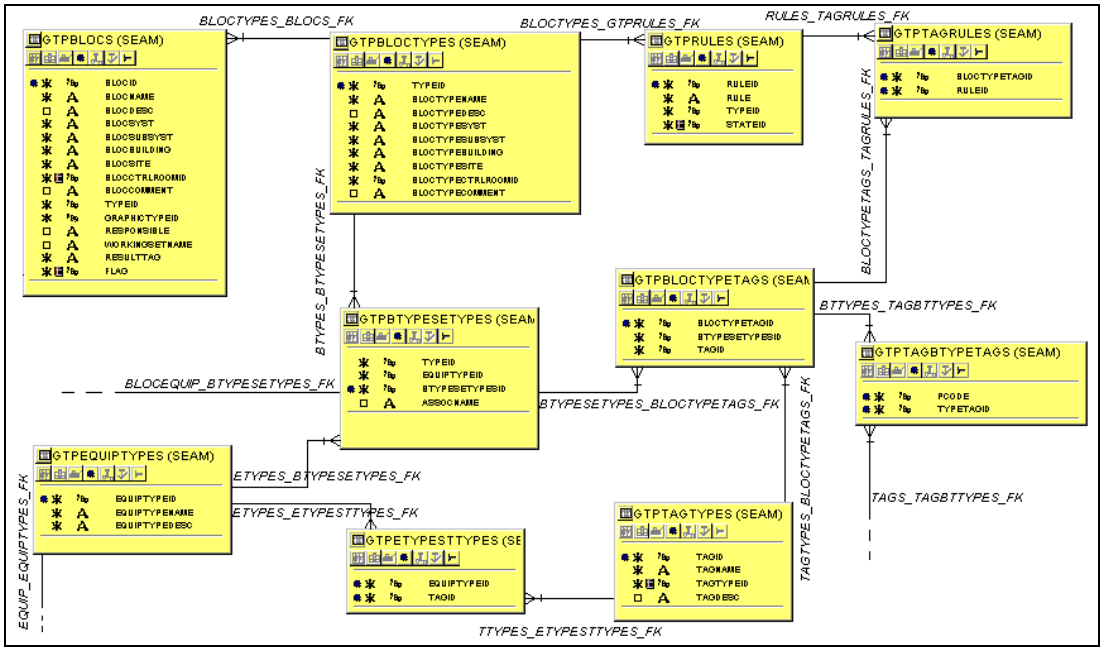


Figure VI. 6 : Types de blocs et règles

Parmi les contraintes du style GTPM, cette partie de la base implémente celle spécifiant que l'état résultant des règles doit être standard. En effet, la valeur affectée au champ STATEID de la table GTPRULES (valeur résultante du bloc quand la règle associée est vérifiée), ne peut être qu'une valeur prédéfinie dans la table de « domaine », c'est à dire un entier compris entre 0 et 3 associé aux états « arrêt », « opérationnel », « avertissement », et « faute ».

La valeur du champ TAGTYPEID de la table GTPTAGTYPES est, elle aussi, contrainte par les valeurs de la table de « domaine ». Ces valeurs correspondent aux seuls types de données qui peuvent être supervisés par les blocs, c'est à dire : chaîne de caractères, flottant, entier ou booléen.

VI. 3. 1. 3 Liens de dépendances

Les liens de dépendances permettent d'indiquer si le fonctionnement des équipements associés à un bloc est dépendant du fonctionnement des équipements associés à un autre bloc. Une seule table est dédiée à leur description (Cf. figure VI.7) :

- **GTPBLOCLNK :** cette table indique si un bloc « aval » est dépendant d'un bloc « amont ».

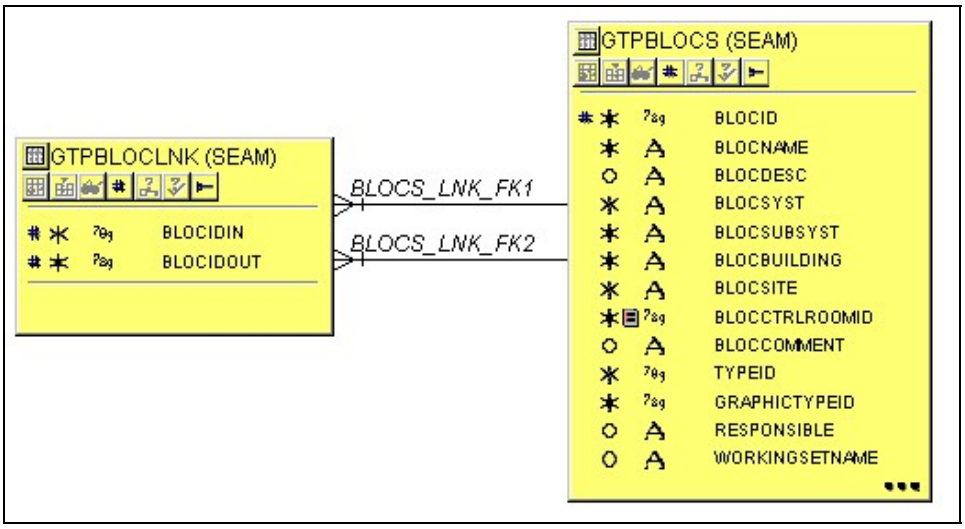


Figure VI. 7 : Liens de dépendances

Cette partie de la base implémente la contrainte du style GTPM spécifiant qu'il ne peut exister deux relations de dépendance identiques. Les champs BLOCIDIN et BLOCIDOUT de la table GTPBLOCLINK formant une clé primaire, il n'est pas possible que deux dépendances aient les mêmes blocs « amont » et « aval ».

VI. 3. 1. 4 Documentation

A chaque bloc peuvent être associés des liens de documentation permettant aux opérateurs des salles de contrôle d'obtenir toute l'information nécessaire au redémarrage des équipements relatifs. Une seule table est dédiée aux liens de documentation (Cf. figure VI.8) :

- **GTPBLOCDOC :** cette table contient les informations concernant les liens vers des fichiers de documentation : une description, une adresse (ex : URL), ainsi qu'un identifiant indiquant le programme devant être utilisé pour ouvrir le fichier.

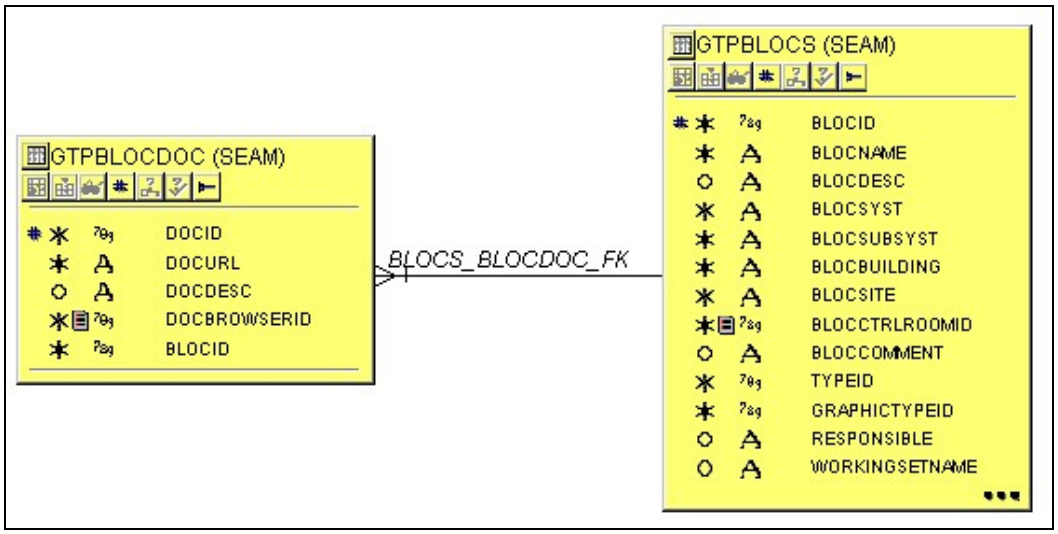


Figure VI. 8 : Documentation

VI. 3. 1. 5 IHMs

Les IHMs de supervision sont des zones graphiques sur lesquelles sont affichés les blocs et leurs liens de dépendance. Celles-ci sont gérées par les tables suivantes (Cf. figure VI.9)[1] :

- **GTPFCTLOCATION :** cette table contient les informations (noms et dimensions) de toutes les IHMs construites dans le contexte du projet GTPM ;
- **GTPBLOCLOCATION :** il s'agit de la table d'association entre les blocs et les IHMs. Un bloc peut être affiché sur n IHMs et une IHM peut afficher n blocs. Cette table d'association spécifie les dimensions et positions des blocs sur les IHMs. En effet, un même bloc peut avoir des dimensions et positions différentes selon l'IHM sur laquelle il se trouve.

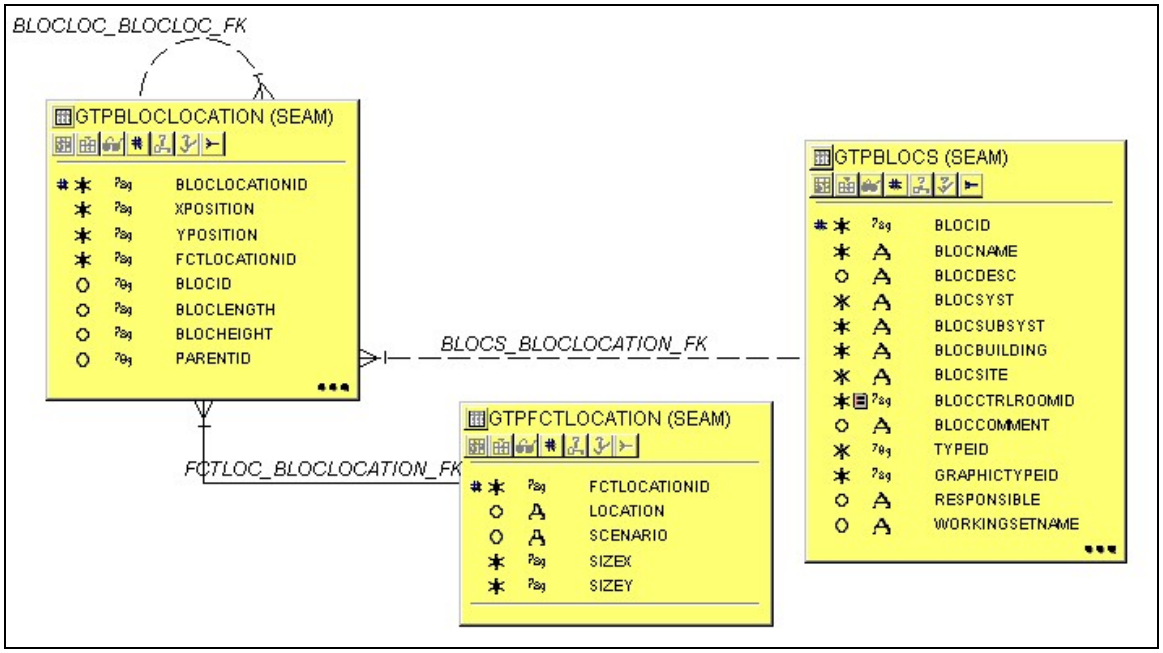


Figure VI. 9 : IHMs

Les champs de la table GTPFCTLOCATION correspondent aux attributs du composite « IHM spécifique » du diagramme présenté figure IV.8. Les champs XPOSITION, YPOSITION, BLOCLENGTH et BLOCHEIGHT correspondent aux attributs d'un bloc sur une IHM spécifique. En effet, un même bloc peut être représenté sur plusieurs IHMs avec des dimensions et positions différentes. Il est à noter que les champs BLOCLENGTH et BLOCHEIGHT prennent par défaut les valeurs correspondant au type graphique utilisé par le bloc (champs GRAPHICTYPELENGTH et GRAPHICTYPEHEIGHT de la table GTPPALETTE, cf. section VI.3.1.1). L'utilisateur peut modifier ces valeurs en cas de besoin spécifique.

Concernant la structure des IHMs spécifiée par le style GTPM, la base de données prévoit bien qu'un bloc puisse être représenté sur un nombre quelconque (0..n) d'IHMs. De même, une IHM peut contenir n blocs, mais la structure de la base n'impose pas qu'au

[1] La figure représente également la table GTPBLOCS, déjà présentée en VI.3.1.1, figure VI.3

moins un bloc soit associé à celle-ci, ce qui est pourtant contraint par le style. Là encore, le but est de donner de la flexibilité de développement : un utilisateur peut créer une IHM vide, et lui associer des blocs ultérieurement. Toutefois, l'outil doit garantir qu'au moment de la génération du code d'une IHMs, celle-ci représente l'état d'au moins un bloc (géré au niveau de l'interface d'exploitation, cf. section VI.3.2.1).

VI. 3. 2 Interface d'exploitation de la base de données

Cette section présente l'interface permettant de consulter, de mettre à jour, et d'insérer des informations dans les tables décrites dans la section précédente. Il s'agit d'une interface web dynamique (html et javascript) programmée en langage PL/SQL. Les paragraphes suivants présentent les différents paquetages de procédures développés dans ce contexte.

VI. 3. 2. 1 Interface principale

Le paquetage principal de l'application comporte un ensemble de procédures permettant d'effectuer les opérations de base sur les composants des IHMs. La procédure appelée lors de l'ouverture de la page web d'accueil affiche un menu permettant à l'utilisateur d'appeler les autres procédures de l'application. La liste des actions possibles à ce niveau est la suivante :

- consultation/modification/suppression d'informations concernant les blocs ;
- consultation/modification/suppression d'informations concernant les types de blocs ;
- consultation/modification/suppression d'informations concernant les IHMs ;
- insertion d'un bloc existant sur une IHM existante ;
- création d'un nouveau bloc à partir d'un type de bloc ;
- création d'un nouveau type de bloc ;
- création d'une nouvelle IHM ;
- exportation des données concernant les nouveaux blocs vers la base de données de configuration du système de supervision[*] ;
- exportation des données concernant les nouvelles règles vers la base de configuration ;
- génération et exportation du code des IHMs en format XML vers le système de supervision.

En outre, comme le montre la figure VI.10, l'interface principale permet à l'utilisateur de sélectionner dans la base de données de SEAM les IHMs/types de blocs existants à consulter/modifier/supprimer en lui proposant un ensemble de critères de recherche (nom, description, localisation, catégorie, éléments associés, …).

[*] Afin de faciliter la lecture, le terme "base de configuration" sera utilisé, dans la suite de ce chapitre, pour designer la base de données de configuration du système de supervision

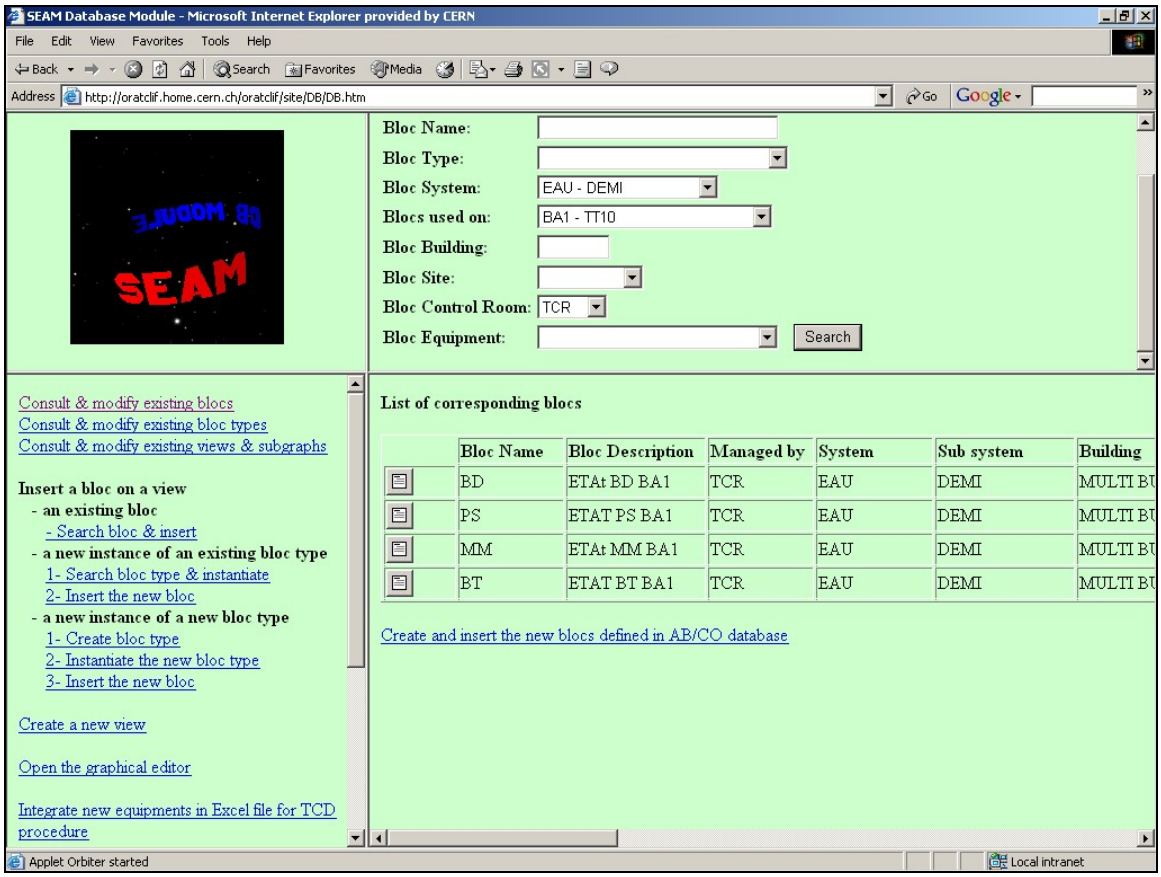


Figure VI. 10 : Interface principale

Cette partie de l'application implémente les contraintes formalisées par le style GTPM concernant les attributs associés aux blocs lors de leur création :

- **le nom de responsable :** l'application accède une base de données des ressources humaines pour vérifier que le nom choisi est existant. En outre, lorsque le nom de responsable associé à tous les équipements supervisés par le bloc en question est identique, elle impose que ce nom de responsable soit associé au bloc lui-même ;
- **le numéro de bâtiment :** l'application accède à une base de données du patrimoine pour vérifier que le numéro choisi est existant. Lorsque le numéro de bâtiment associé à tous les équipements supervisés par le bloc est identique, elle impose que ce numéro soit associé au bloc lui-même ;
- **la catégorie :** l'application accède à la base de configuration du système de supervision pour vérifier que les système/sous-système choisis sont existants. Lorsque la catégorie associée à tous les équipements supervisés par le bloc est identique, elle impose que cette catégorie soit associée au bloc lui-même.

Par ailleurs la procédure d'exportation du code des IHMs en format XML (requête sur différentes tables et présentation du résultat de celle-ci en format XML) vérifie que ces IHMs affichent bien l'état d'au moins un bloc, et que tout les blocs représentent bien l'état d'au moins un équipement.

VI. 3. 2. 2 Gestion des équipements

L'interface d'exploitation de la base de données comporte un paquetage dédié à la gestion des équipements et des types d'équipements. Lorsqu'un bloc ou un type de bloc a été crée par un utilisateur via l'interface principale, celle-ci propose l'appel des procédures permettant respectivement l'association d'équipements ou de types d'équipements. L'utilisateur peut alors choisir d'associer au bloc/type de bloc un équipement/type d'équipement déjà existant dans la base de données SEAM, ou au contraire d'en crée un nouveau. La capture d'écran suivante (figure VI.11) représente la liste des types d'équipements associés au type de bloc « BlocTypeDemo » :

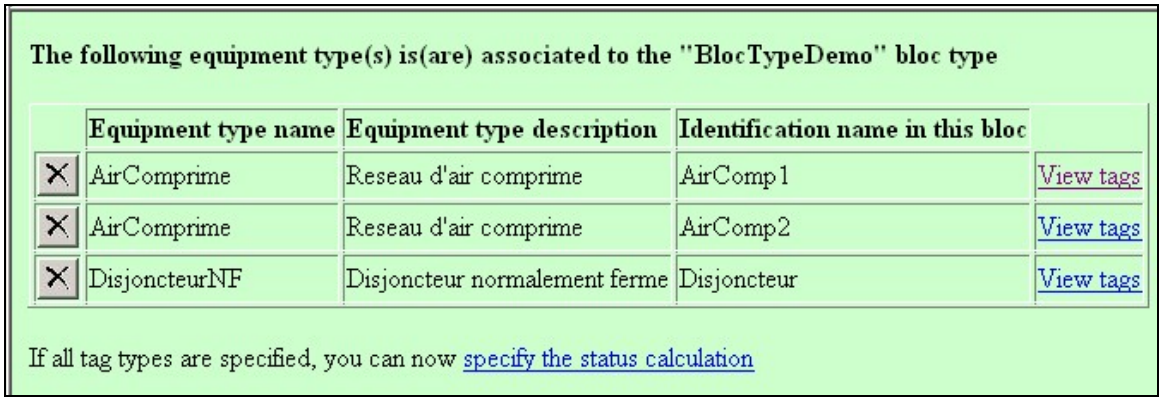


Figure VI. 11 : Association de types d'équipements

Ce paquetage possède en outre des procédures permettant l'accès et l'utilisation d'informations présentes dans des bases de données extérieures à SEAM, notamment la base de configuration et celle de GMAO (Gestion de Maintenance Assistée par Ordinateur) qui contient tous les équipements maintenus dans les processus à superviser.

Comme expliqué précédemment, après avoir crée un bloc, l'utilisateur peut spécifier les équipements que celui-ci doit superviser de deux manières différentes :

- **il s'agit d'équipements existants dans la base de configuration :** l'application permet de chercher et sélectionner les équipements de la base de données de configuration, et de les référencer dans celle de SEAM. Si l'utilisateur veut obtenir plus d'information (sous-équipements, informations sur les opérations de maintenance, état de l'installation, …) sur les équipements concernés, il a la possibilité d'accéder aux données de la base de GMAO ;
- **il s'agit d'équipements non existants dans la base de configuration :** l'application permet d'effectuer une requête d'intégration de nouvel équipement dans la base de configuration et d'utiliser une référence temporaire dans la base de SEAM. Cette référence sera automatiquement remplacée par la référence définitive quand l'intégration sera terminée.

La figure VI.12 représente un tableau de l'interface indiquant les équipements associés à un bloc « Bloc de demo ». Comme ce bloc est du type « BlocTypeDemo » on retrouve les trois types d'équipements définis figure VI.11 : *AirComp1*, *AirComp2* et *Disjoncteur*. Les circuits d'air comprimé *FDED-00010* et *FDED-00014* et les valeurs

qu'ils rendent disponibles aux adresses *44833* et *43719* (références utilisées dans la base configuration) sont associées aux types d'équipements *AirComp1* et *AirComp2* et à leur type de valeur *defautGeneral*. Un nouvel équipement *EMF007*, fournissant trois valeurs, a été crée pour être associé au type d'équipement *Disjoncteur*. Les références utilisées dans le tableau pour cet équipement sont temporaires, elles seront mises à jour lorsque celui-ci sera intégré dans la base de configuration.

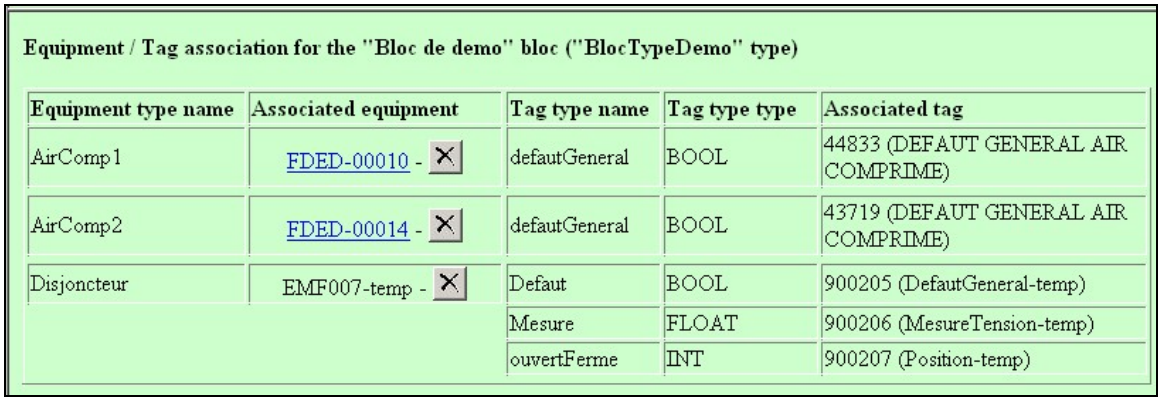


Figure VI. 12 : Association des équipements

Il est à noter que la contrainte du style GTPM spécifiant que les blocs ne doivent être associés qu'à des données dont ils supportent le type, est implémentée à ce niveau de l'interface. L'utilisateur spécifie les types de données supportés lors de la création du type de bloc, et, lorsqu'il crée un bloc suivant ce type de bloc, il ne peut lui associer que des équipements ayant des valeurs de type supporté. Dans l'exemple de la figure précédente, l'application vérifie que les valeurs ayant les références *44833* et *43719* sont bien de type booléen, comme c'est requis par le type de bloc « BlocTypeDemo ».

VI. 3. 2. 3 Gestion des règles

Le paquetage de gestion des règles permet à l'utilisateur de créer des règles génériques (associées aux types de blocs). Lorsque l'utilisateur a défini un type de bloc, l'interface lui propose de spécifier une règle pour chaque état possible (arrêt, opérationnel, avertissement et faute). Cette spécification est effectuée via l'interface d'édition d'équation présentée figure VI.13. Il est à noter que le style formel ne spécifie pas de contraintes concernant le format des règles. L'interface présentée ici a été construite pour fournir des règles respectant le format défini par le moteur d'exécution de règles, qui fait partie du système de contrôle et de supervision du CERN.

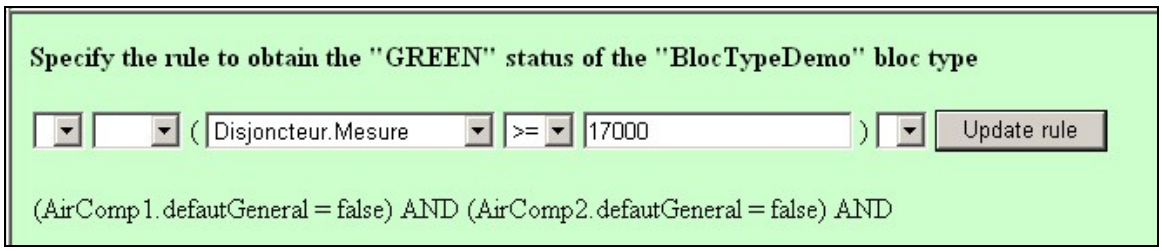


Figure VI. 13 : Edition des règles génériques

Ce paquetage possède une procédure affichant les règles génériques spécifiées pour les différents états d'un type de bloc, et permettant d'en ajouter ou supprimer si nécessaire.

Une autre procédure permet d'instancier les règles génériques associées à un type de bloc avec les valeurs correspondant à un bloc suivant ce type. La règle générique présentée figure VI.13 est associée au type de bloc « BlocTypeDemo », et la règle instanciée pour le bloc « Bloc de demo » s'obtient par le remplacement de ses variables par les valeurs associées figure VI.12 :

- AirComp1.defautGeneral est remplacé par la référence 44833 ;
- AirComp2.defautGeneral est remplacé par la référence 43719 ;
- Disjoncteur.Defaut est remplacé par la référence 900205 ;
- Disjoncteur.Mesure est remplacé par la référence 900206 ;
- Disjoncteur.ouvertFerme est remplacé par la référence 900207.

Par ailleurs, si des règles génériques ont été définies pour plusieurs états du type de bloc, celles-ci sont concaténées pour ne former qu'une règle par bloc. En effet, le moteur de règles du système de supervision impose le format suivant :

AdresseRésultat = Règle1[valeur renvoyée si Règle1 vraie], Règle2[valeur renvoyée si Règle2 vraie], …, RègleN[valeur renvoyée si RègleN vraie]

Par exemple, deux règles génériques ont été définies pour le type de bloc « BlocTypeDemo » (une pour l'état « opérationnel » correspondant à la valeur résultante 0, et une pour l'état « faute » correspondant à la valeur résultante 3), la règle instanciée pour le bloc « Bloc de demo » est donc la suivante :

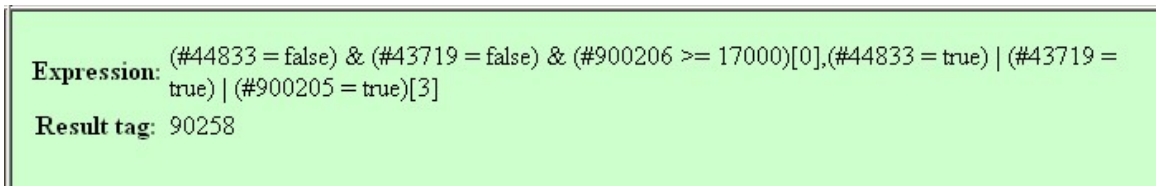


Figure VI. 14 : Génération des règles des blocs

Ainsi, si les valeurs lues aux adresses 44833 et 43719 sont « false », et si la valeur lue à l'adresse 900206 est supérieure ou égale à 17000, la valeur renvoyée à l'adresse 90258 (Result tag) sera 0. Par contre, si l'une des valeurs lues aux adresses 44833, 43719 ou 900205 est « vraie », la valeur renvoyée à l'adresse 90258 sera 3.

VI. 3. 2. 4 Gestion des IHMs

Ce paquetage inclut des procédures permettant de :

- créer une nouvelle IHM ;
- modifier ou supprimer une IHM existante ;
- d'associer un bloc à une IHM (en spécifiant les coordonnées du bloc sur l'IHM) ;
- de modifier les coordonnées d'un bloc sur une IHM ;
- de rechercher toutes les IHMs contenant un bloc donnée.

La procédure de création d'IHM initialise ses dimensions (largeur et hauteur) avec des valeurs standard afin de respecter la contrainte d'uniformité graphique spécifiée par le style GTPM.

VI. 3. 2. 5 Gestion des dépendances

Le paquetage gérant les dépendances entre les blocs ne comporte que quelques procédures (il n'implémente pas de contrainte du style GTPM). Celles-ci permettent à l'utilisateur de :

- rechercher dans la table GTPBLOCLNK les dépendances existantes pour un bloc donné ;
- créer une nouvelle dépendance entre deux blocs ;
- supprimer une dépendance existante.

VI. 3. 2. 6 Gestion de la documentation

Le paquetage gérant la documentation des blocs possède de procédures permettant à l'utilisateur de :

- rechercher dans la table GTPBLOCDOC les liens de documentation associés à un bloc donné ;
- créer un nouveau lien de documentation pour un bloc ;
- modifier un lien de documentation ;
- supprimer un lien de documentation existant.

VI. 3. 3 Modélisateur GTPM

La base de données et son interface fournissent la possibilité de gérer certains aspects graphiques des IHMs (dimensions, positions, zones de texte, …). Toutefois, sur ce point, un outil d'édition graphique est plus efficace pour un utilisateur, car il lui permet de visualiser le résultat de son travail. C'est pourquoi il a été nécessaire de développer un modélisateur graphique d'IHMs GTPM. Ce modélisateur a été construit à partir d'un modélisateur de « workflow » inclus dans la suite [JViews]. Ce dernier est un éditeur permettant la définition et la supervision de « workflows » d'une manière graphique et intuitive. Il inclut les fonctionnalités suivantes :

- édition par glisser-déposer ;
- une palette d'objets graphiques représentant les « tâches », les « participants », ainsi que des connecteurs ;
- une série d'algorithmes de routage permettant de réorganiser automatiquement les éléments du « worflow » d'une manière plus exploitable ;
- personnalisation de l'aspect des éléments graphiques via des feuilles de style[*] (CSS : Cascading Style Sheets) ;
- éditeurs de propriétés ;

[*] Les feuilles de style (CSS) sont des fichiers textuels habituellement utilisés pour contrôler l'aspect graphique des pages web HTML. Ilog Jviews utilise cette technologie pour paramétrer l'aspect des IHMs. Ne pas confondre la notion de « feuilles de styles » avec celle de « styles architecturaux ».

- un ensemble d'outils d'édition, comprenant les fonctions copier et coller, des fonctions de sélection, de zoom, d'impression, etc. ;
- le chargement et la sauvegarde de fichiers en format XML.

Le code source Java de cet éditeur étant fourni avec la suite JViews, il a été possible d'en modifier certaines parties pour qu'il réponde plus pleinement aux besoins concernant l'édition des IHMs GTPM. La personnalisation de l'éditeur à comporté trois aspects qui sont présentés dans les sections suivantes : l'ajout de la possibilité d'interaction avec la base de données de SEAM, la modification de l'interface graphique, et la définition d'une feuille de style.

VI. 3. 3. 1 Interactions avec la base de données

Une des fonctionnalités principales du modélisateur GTPM est la possibilité d'interagir avec la base de données de SEAM. En effet, le processus de développement d'IHMs GTPM comporte deux phases : la définition des données, et la modélisation graphique. Ces deux phases peuvent se succéder dans un ordre comme dans l'autre, et peuvent être réitérées si nécessaire. L'utilisateur doit donc pouvoir importer/exporter des données de/vers la base de SEAM.

La méthode d'importation de données ajoutée au modélisateur effectue le traitement suivant :

- initialisation d'une connexion JDBC vers la base de données de SEAM pour permettre les requêtes SQL sur ces tables ;
- sélection des informations concernant une IHM choisie par l'utilisateur ;
- création d'un document portant le nom de l'IHM dans le modélisateur ;
- sélection des informations concernant tous les blocs associés à l'IHM dans la base de données (identifiant, nom, salle de contrôle, taille, position, …) ;
- création des objets graphiques correspondants dans le document du modélisateur, et initialisation de leurs attributs avec les valeurs retournées par la requête précédente ;
- sélection des informations concernant tous les liens de dépendances relatifs aux blocs de l'IHM dans la base de données ;
- création des objets graphiques (liens) correspondants dans le document du modélisateur, et initialisation de leurs attributs avec les valeurs retournées par la requête précédente.

La méthode d'exportation des données effectue quant à elle les opérations suivantes :

- initialisation de la connexion JDBC ;
- recherche dans la base de données SEAM d'une IHM correspondant au nom du document en cours d'édition dans le modélisateur ;
- création d'une nouvelle IHM dans le base de données si la requête précédente n'aboutit pas ;
- sélection des objets graphiques présents sur le document et de leurs attributs ;
- insertion ou mise à jour des blocs et liens de dépendances dans la base de données de SEAM.

Ces méthodes permettent d'utiliser la base de données SEAM comme stockage commun aux deux interfaces utilisateurs : l'interface propre à la base de données, et le modélisateur graphique GTPM.

VI. 3. 3. 2 Personnalisation de l'interface graphique

Le deuxième aspect de l'adaptation du modélisateur fut la personnalisation de l'interface graphique. Dans ce contexte, il a notamment été nécessaire de modifier les menus et les palettes d'objets graphiques.

Les palettes d'objets graphiques représentant des « tâches » et des « participants » de « worflow » n'étaient pas adaptées à la problématique GTPM. Trois palettes spécifiques ont donc été crées (figure VI.15 – 4) :

- **Types graphiques :** cette palette fournit cinq types graphiques de base, deux dédiés à la représentation des blocs affichant uniquement un état (un de forme horizontale, et un de forme verticale), et trois permettant l'affichage d'informations additionnelles (forme horizontale, affichage de l'état ainsi que d'une mesure ou d'un nombre d'alarmes). A chaque type graphique correspond une classe d'objets ajoutée au modélisateur de base. La palette définit en outre les propriétés de chacun des types graphiques (correspondant aux attributs définis dans les classes d'objets). Par exemple, un bloc suivant le type graphique le plus simple doit avoir un identifiant, un nom, une largeur, une hauteur, un nom de responsable, et l'identifiant d'une valeur résultante (correspond au résultat d'une règle : identifiant 90258 dans l'exemple figure VI.14). Ces propriétés correspondent aux informations de la table GTPPALETTE de la base de données (Cf. section VI.3.1.1). La palette des types graphiques est d'ailleurs définie par du code XML obtenu à partir de cette table ;
- **Blocs :** cette palette permet de rendre disponible à l'utilisateur tous les blocs déjà créés à partir de types graphiques. Contrairement à ces derniers, les blocs ont déjà des valeurs affectées à leurs propriétés ;
- **IHMs :** cette palette rend disponible à l'utilisateur les IHMs et les ensembles de blocs déjà crées. Celle-ci est notamment utile pour construire les IHMs globales.

Les objets graphiques définis dans les palettes sont utilisés sur le document de travail (figure VI.15 – 3) par « glisser-déposer ». Afin de pouvoir affecter ou modifier des valeurs à leurs propriétés, il a été nécessaire de modifier l'éditeur de propriétés de base (figure VI.15 – 1). Cela a été réalisé par codage et par la définition du fichier CSS (feuille de style – Cascading Style Sheet) paramétrant le modélisateur (Cf. section VI.3.3.3).

Par ailleurs, comme cela a été signalé, les menus du modélisateur ont été adaptés aux besoins concernant le développement d'IHMs GTPM (figure VI.15 – 2). Par exemple, des boutons ont été ajoutés au menu « Fichier » (« File ») pour permettre l'invocation des méthodes d'importation/exportation des IHMs de/vers la base de données de SEAM. Un bouton permettant d'ouvrir la page web de l'interface d'exploitation des données a aussi été ajouté à ce même menu. D'autre part, afin d'assurer que les IHMs développées respecteront le style GTPM, il a été nécessaire de réduire l'espace de développement du modélisateur de base en supprimant certaines de ses possibilités. Par exemple, le menu « Palette » ne permet plus d'ouvrir des palettes autres que celles citées précédemment. De

même, il n'est plus possible d'appliquer d'algorithmes de routage réorganisant automatiquement les blocs. En effet, dans le contexte du développement d'IHMs GTPM, l'utilisateur doit rester maître du positionnement des éléments.

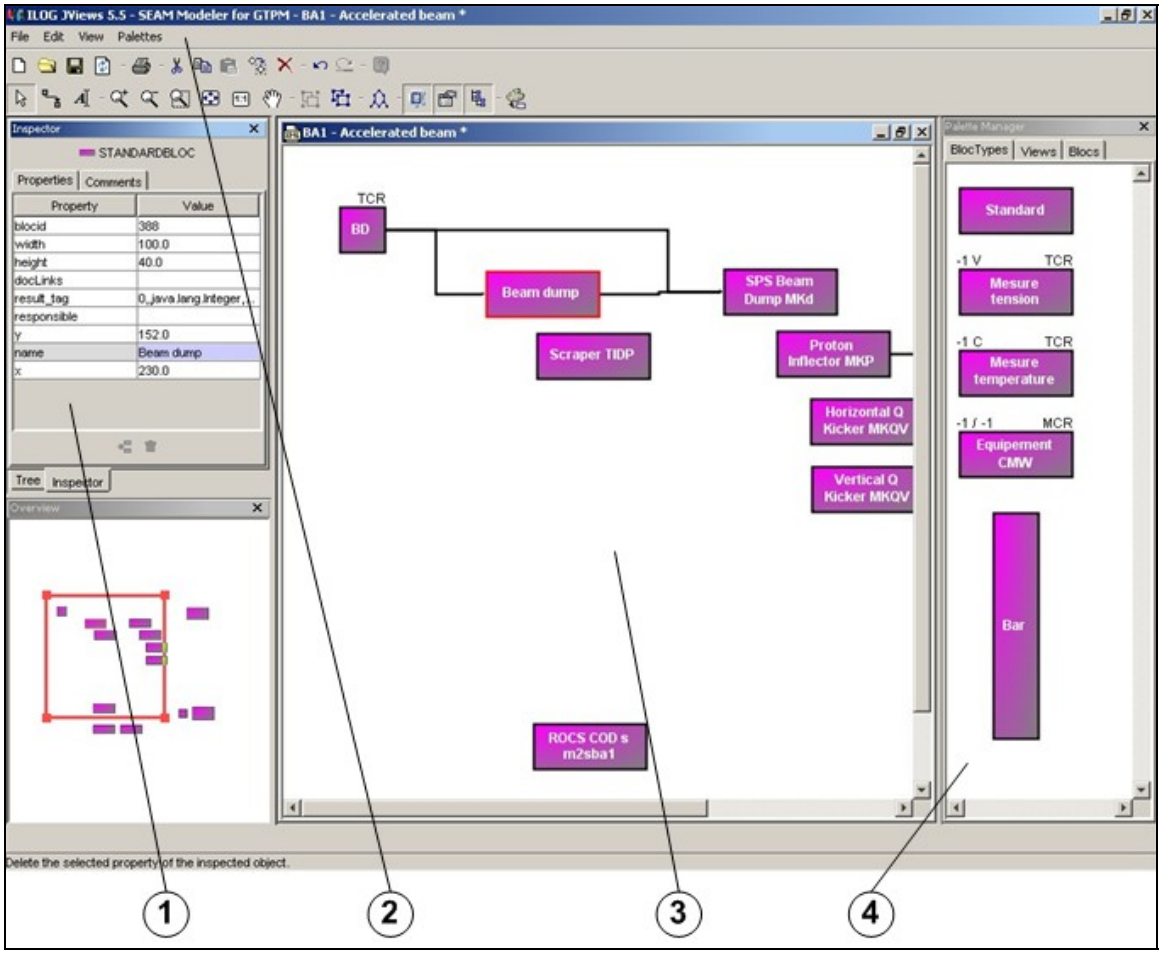


Figure VI. 15 : Modélisateur GTPM

VI. 3. 3. 3 Définition de la feuille de style

L'utilisation d'une feuille de style est un moyen très efficace pour personnaliser les différents éléments du modélisateur. Elle permet notamment de paramétrer :

- le document d'édition des IHMs (figure VI. 15 – 3) : il est possible de définir si l'application d'algorithmes de routage (pour les blocs et/ou les liens graphiques) est autorisée, ou s'il est possible de grouper les éléments, etc. ;
- les algorithmes de routage des blocs et des liens graphiques ;
- les objets graphiques selon leur type (couleurs, polices, tailles de polices, facteurs de zoom d'apparition/de disparition, libellés, alignement…) ;
- les relations entre les propriétés des objets graphiques et les attributs des classes correspondantes ;
- l'éditeur de propriétés.

Cette méthode de personnalisation est simple à mettre en œuvre, et elle permet de modifier facilement l'aspect des IHMs en fonction d'éventuelles évolutions des besoins. C'est pour cette raison qu'il a été choisi d'implémenter les contraintes graphiques formalisées par le style GTPM au niveau de la feuille de style (fichier CSS). Ainsi, si les contraintes du style évoluent dans le temps, il sera moins coûteux de modifier la feuille de style, que ça le serait de modifier le code source de l'application si les contraintes avaient été codées en dur en Java. Ainsi, le fichier CSS impose l'uniformité des propriétés graphiques des éléments (polices, facteurs de zoom…) tel que cela est formalisé par le style GTPM.

Il est intéressant de noter que cette feuille de style est utilisée par le modélisateur d'IHMs GTPM, mais aussi par l'outil du système de supervision exécutant ces IHMs. Il s'agit d'un outil faisant partie de la suite JViews permettant l'exploitation du code généré par le modélisateur pour afficher l'état en temps réel des processus supervisés. Ceci garantit que les IHMs auront le même aspect à l'exécution qu'à l'édition.

VI. 3. 4 Vérificateur de conformité

Les différentes parties de l'outil SEAM présentées dans les sections précédentes implémentent un sous-ensemble des contraintes formalisées par le style (Cf. tableau VI.1). Certaines contraintes n'ont quant à elles pas été traitées pour des raisons de flexibilité (évolution des contraintes fortement probable dans un futur proche). Le tableau VI.1 récapitule les contraintes du style GTPM et indique si elles sont implémentées par les modules précédemment présentés.

Contrainte du style	**Base de données SEAM**	**Interface de la base**	**Modélisateur GTPM**
Superposition des blocs			Géré par la feuille de style
Taille et police		Dimensions des blocs et des IHMs suggérées	Uniformité des polices gérée par la feuille de style
Zoom			Géré par la feuille de style
Nom de responsable		Existence et cohérence gérées	
Numéro de bâtiment		Existence et cohérence gérées	
Catégorie		Existence et cohérence gérées	
Salle de contrôle	Géré (domaine de valeurs)		
Positionnement relatif des blocs			
Positionnement relatif des IHMs			

Cardinalités des blocs	Géré en partie (structure de la BD)	Géré en partie (génération de code)	
Redondance des dépendances	Géré (clé primaire)		
Boucle de dépendance		Géré	
Typage des blocs compatibles		Géré	
Boucle d'acquisition	Traité par un outil autre que SEAM		
Traitement de la plage de valeurs			
Etat résultant standard	Géré (domaine de valeurs)		

Tableau VI. 1 : Implémentation des contraintes du style GTPM

Le chapitre 4 a énoncé des contraintes de cardinalité devant être formalisées par le style. Ces contraintes imposent le nombre minimum et maximum de blocs présents sur une IHM en fonction de leur catégorie (ex : un et un seul bloc de la catégorie « Electricité »). De même, ce chapitre a énoncé des contraintes de positionnement relatif des blocs et des IHMs, ainsi qu'une concernant l'obligation de traitement de toute la plage des valeurs supervisées par un bloc. Toutefois, l'expérience a montré que de telles contraintes ne peuvent pas être respectées pour toutes les IHMs, et, pour cette raison, les spécialistes de l'opération des accélérateurs n'ont pas pu les énoncer de manière définitive. Il aurait été faisable de coder ces contraintes, au niveau de l'interface d'exploitation de la base de données par exemple, mais cela aurait introduit un manque de flexibilité contre-productif.

La contrainte du style interdisant les boucles dans la chaîne d'acquisition n'a pas non plus été implémentée par SEAM. En effet, un outil propre au système de supervision de la salle de contrôle a été mis en place dans cet objectif : SEAM lui transmet les règles stockées dans sa base et l'outil les analyse pour vérifier qu'il n'existe pas de boucles.

Par ailleurs, les contraintes concernant l'uniformité des dimensions des blocs et des IHMs n'ont pas été entièrement implémentées. En effet, pour un besoin de flexibilité, des valeurs par défaut sont simplement suggérées à l'utilisateur. Là encore, il a été décidé de les vérifier à posteriori, et d'avertir d'utilisateur en cas de violation.

Ainsi, les trois modules que sont la base de données de SEAM, son interface d'exploitation, et le modélisateur GTPM, peuvent garantir que les IHMs produites respectent la majorité des contraintes du style GTPM, mais pas toutes. Il a donc été nécessaire de mettre en place un outil vérifiant la conformité des IHMs par rapport à toutes les contraintes du style. Cet outil fait partie de l'interface utilisateurs de SEAM car il affiche aux développeurs d'IHMs les résultats des analyses qu'il applique. Il comporte deux aspects principaux faisant l'objet des paragraphes suivants : la conversion des IHMs en architectures formalisées en ArchWare-ADL et utilisant le vocabulaire du style GTPM, et la vérification de la conformité de ces architectures par rapport aux contraintes du style.

VI. 3. 4. 1 Conversion des IHMs en ADL et vérification des contraintes

Cette fonctionnalité permet d'obtenir, à partir des données présentes dans la base de données de SEAM, le code formel des IHMs en ArchWare-ADL sous la forme présentée dans la section V.5. Il a donc fallu établir des règles de transformation pour passer de la représentation « base de données » des IHMs à la représentation formelle. Les principales règles concernant la correspondance entre les notions de la base de données et le vocabulaire défini par le style GTPM sont les suivantes :

- à une association bloc-équipement « supervision » présente dans la table GTPBLOCEQUIP correspond un connecteur tel que « supervision **is connector in style** DataLink **with {**…**}** » ;
- à un bloc « bloc » présent dans la table GTPBLOCS correspond un composant tel que « bloc **is component in style** StatusBloc **with {**…**}** » ;
- à un équipement « equip » présent dans la table GTPEQUIP correspond un composant tel que « equip **is component in style** Equipment **with {**…**}** » ;
- à une IHM « ihm » présente dans la table GTPFCTLOCATION correspond un composant tel que « ihm **is component in style** IndividualHCI **with {**…**}** ».

La table d'association bloc-IHM GTPBLOCLOCATION permet de générer le code d'initialisation des blocs associés à une IHM (avec les valeurs de la table GTPBLOCS comme paramètres) dans sa description ADL.

Le code ADL concernant les ports associés aux différents composants et connecteurs est généré en même temps que ceux-ci. Les attributs graphiques des éléments présents dans la base de données et dans le fichier CSS du modélisateur, sont utilisés pour générer la partie du code ADL contenant l'initialisation des attributs des composants.

Ces quelques règles de transformation permettent d'obtenir une représentation formelle des IHMs développées. L'avantage majeur d'une telle représentation est la possibilité de lui appliquer des analyses automatiques. La vérification du respect des contraintes formalisées au niveau du style par les architectures d'IHMs, est ainsi rendue possible par l'utilisation d'un outil ArchWare. Il s'agit du « customizer » qui offre un service de comparaison d'une architecture par rapport au style et qui appelle l'outil nécessaire (« theorem prover » ou « model checker ») pour la vérification de chacune des contraintes.

VI. 4 Conclusion

Ce chapitre a présenté l'implémentation de l'approche proposée par cette thèse. L'environnement SEAM est maintenant opérationnel et a été déployé dans les principales salles de contrôle du CERN, où il est utilisé depuis plusieurs mois. SEAM comporte actuellement environ 10.000 lignes de code et 22 tables stockant les informations concernant 36 applications de supervision, 42 types de blocs, environ 600 blocs.

Le style GTPM, formalisé à l'aide des langages ArchWare, a été intégré à l'architecture de l'environnement de développement. Cette architecture qui inclut les contraintes du style a été utilisée pour construire SEAM. Ainsi, les différents modules qui le composent ont été conçus de façon à implémenter les contraintes que le style spécifie. Certaines de ces contraintes ont été intégrées à la structure de la base de données de

SEAM, d'autres ont été codées dans son interface d'exploitation, d'autres encore ont été implémentées par la personnalisation du modélisateur GTPM, et enfin, pour de raisons de flexibilité, certaines n'ont pas été traitées. Toutefois, l'utilisation du vérificateur de conformité permet de vérifier si les contraintes formalisées sont bien toutes respectées par les architectures d'IHMs, et d'avertir l'utilisateur dans le cas contraire. L'environnement de développement SEAM guide ainsi les développeurs d'IHMs dans leur travail et leur garantit que celles-ci vérifient les contraintes du style.

Le chapitre suivant présente une approche permettant de gérer l'évolution de cet environnement de développement et du style, en fonction de l'évolution des besoins relatifs à la supervision des accélérateurs.

Chapitre VII Evolution des IHMs et de l'outil logiciel

Chapitre VII : Evolution des IHMs et de l'outil logiciel

VII. 1 Introduction

Les chapitres précédents ont proposé un processus de développement d'IHMs[*] utilisant les styles architecturaux, et ont présenté l'implémentation d'un environnement prenant en charge ce processus. L'environnement, construit à partir de son architecture formelle, guide les utilisateurs dans le développement d'IHMs respectant les contraintes définies par un style. Cet environnement de développement garantit la satisfaction des besoins exprimés par les utilisateurs de IHMs et formalisés par le style. Toutefois, les besoins exprimés dans les premières phases d'un projet peuvent évoluer avec le temps, et ceci pour deux raisons principales :

- l'évolution du processus supervisé : dans le contexte du projet GTPM, les accélérateurs de particules sont sujets à une évolution permanente au cours de leur longue durée de vie ; il est nécessaire de mettre à jour les IHMs en fonction de ces évolutions ;
- le besoin de nouvelles fonctionnalités/informations : l'utilisation des IHMs en salle de contrôle peut mettre évidence certaines limitations ; les utilisateurs peuvent alors proposer de nouvelles solutions, de nouvelles fonctionnalités, ou requérir l'affichage de nouvelles formes d'informations.

Les questions qui se posent alors sont les suivantes :

- Comment faire évoluer les IHMs ?
- Quels sont exactement les types de besoins pouvant évoluer ?
- Comment mesurer l'évolution des IHMs ?
- Est-ce possible de mettre à jour le style en fonction de cette évolution ? Comment ?
- Est-ce possible de mettre à jour l'environnement de développement ? Comment ?

Ce chapitre a pour objectif de répondre à ces questions. Dans un premier temps la section VII.2 explique comment les IHMs peuvent être mises à jour en fonction de l'évolution des besoins. Ensuite, la section VII.3 présente une approche permettant de superviser l'évolution des IHMs. Enfin, les sections VII.4 et VII.5 proposent une solution pour exploiter le résultat de cette supervision en mettant à jour le style et l'environnement de développement associé.

[*] Dans ce chapitre encore, le terme « IHM » désigne des logiciels de supervision à forte composante graphique.

VII. 2 Evolution des IHMs

Les besoins évoluant, il est nécessaire que les IHMs puissent être mises à jour en fonction. Cette section expose comment les prototypes d'IHMs ont été régulièrement mis à jour, comment, dans une approche architecturale classique les IHMs pourraient évoluer, et comment cela est possible dans un environnement de développement dédié.

VII. 2. 1 Evolution des prototypes d'IHMs

Comme cela a été présenté dans le chapitre 4 (figure IV.3), les prototypes d'IHMs n'ont pas été développés à partir du style GTPM, mais à l'aide d'un outil de développement d'IHMs traditionnel (PVSS). Ces prototypes ont servi de base à la définition du style GTPM, mais ont aussi été largement utilisés en salle de contrôle. Ainsi de nouveaux besoins sont apparus lors de leur phase d'exploitation et de nombreuses modifications leur ont été appliquées (des détails concernant ces modifications sont présentés dans le paragraphe VII.3.2.2.1).

Les modifications des prototypes d'IHMs ont été apportées à l'aide de l'outil PVSS. Il est à noter que cet outil n'étant pas contraint par un style propre au domaine de la supervision du redémarrage des accélérateurs, il était possible d'appliquer un large éventail de modifications. Cette flexibilité était intéressante au début du projet, mais non acceptable par la suite, car elle risquait d'impliquer une perte de contrôle de l'orientation des développements itératifs. Par ailleurs, certaines des modifications avaient uniquement un caractère graphique, et étaient donc applicables par des développeurs non informaticiens, mais la plupart impliquaient des changements au niveau du code, et ont donc dû être prises en charge par l'équipe de programmeurs, ce qui constituait un inconvénient majeur.

Les modifications successives des prototypes d'IHMs ont été considérées pour le raffinement du style GTPM qui était en cours de développement.

VII. 2. 2 Evolution des IHMs dans une approche architecturale

Dans un environnement de développement entièrement architectural, comme celui présenté figure VI.1, l'évolution des IHMs passe par l'évolution de leurs architectures. Des outils architecturaux, tel que le modélisateur ArchWare, permettent de mettre à jour les architectures, et donc les IHMs, d'une manière graphique. Toutefois, lorsque l'environnement est contrôlé par un style, les utilisateurs ne sont pas libres d'appliquer n'importe quelle modification : les architectures doivent utiliser le vocabulaire, et respecter les contraintes définis par le style. En effet, c'est l'objectif du style de garantir que les IHMs respectent les besoins « prédéfinis ». Cependant, comme les besoins peuvent évoluer, ceux « prédéfinis » peuvent devenir obsolètes, et il devient nécessaire de pouvoir s'affranchir des contraintes de développement définies par le style.

Un outil tel que le modélisateur ArchWare n'est pas nécessairement paramétré par un style particulier. En cas de besoin, il est donc tout à fait possible à un utilisateur de définir des nouvelles versions d'architectures, utilisant des éléments architecturaux non définis dans le vocabulaire du style, et violant ses contraintes, en utilisant le modélisateur sans le

paramétrer avec un style. Toutefois, si cette opération peut être nécessaire, elle a l'inconvénient de découpler les architectures du style, ce qui empêchera de contrôler leur évolution future. Ceci amène à considérer le fait que si les architectures doivent évoluer hors des limites fixées par le style, c'est que le style ne satisfait plus les besoins du domaine, et qu'il est donc nécessaire qu'il évolue lui même. Ce point est traité dans la section VII.4.

VII. 2. 3 Evolution des IHMs avec dans un environnement de développement spécifique

La solution choisie dans le contexte de cette thèse a été de concevoir l'environnement de développement spécifique à partir de contraintes exprimées par un style formel. Le chapitre 6 a expliqué que le style a été intégré dans une architecture comprenant une base de données, son interface d'exploitation, et un modélisateur graphique. Comme ces trois modules comportent la majeure partie des contraintes du style, les utilisateurs ne peuvent pas appliquer n'importe qu'elle modification aux IHMs par son intermédiaire. Toutefois, certaines contraintes du style n'ont volontairement pas été « codées » de façon à donner une flexibilité d'évolution. Ainsi, lorsque de nouveaux besoins apparaissent, leur implémentation peut se faire des manières suivantes (Cf. figure VII.1) :

- **utilisation de l'interface base de données et du modélisateur graphique de l'environnement (1) :** soit les modifications requises ne violent pas le style et sont donc implémentables, soit elles violent le style mais restent implémentables (les violations de contraintes seront indiquées lors de la vérification de conformité architecture/style) ;
- **modification du code des IHMs (2) :** les modifications requises violent le style et ne sont pas implémentables via l'interface et le modélisateur, mais peuvent l'être par modification directe du code des IHMs. Par exemple, dans le cadre de l'outil SEAM, la base de données et son interface empêchent l'utilisateur d'associer à un bloc une salle de contrôle différente de TCR/MCR/PCR/QCR, mais c'est tout à fait possible de le faire en éditant directement le code XML des IHMs ;
- **modification de l'environnement de développement (3) :** les modifications requises violent le style et ne sont ni implémentables via l'interface et le modélisateur, ni par modification du code des IHMs. Il s'agit de l'alternative la moins souhaitable car elle requière la redéfinition partielle de l'architecture de l'environnement de développement et son implémentation. Il peut être nécessaire de modifier la structure de la base de données, et/ou son interface d'exploitation, et/ou le modélisateur graphique.

L'évolution des IHMs peut donc conduire à des violations du style de référence. Toutefois, il est nécessaire de mesurer l'impact de ces violations et que celles-ci soient faites en connaissance de cause. De plus, celles-ci indiquent dans certains cas que le style est partiellement obsolète et qu'il est nécessaire de le mettre à jour. Ainsi, comme le représente la figure VII.1, une analyse des modifications appliquées sur les IHMs permet de déterminer si le style doit être lui-même modifié. Dans l'affirmative, le style, mais aussi l'environnement de développement, seront mis à jour.

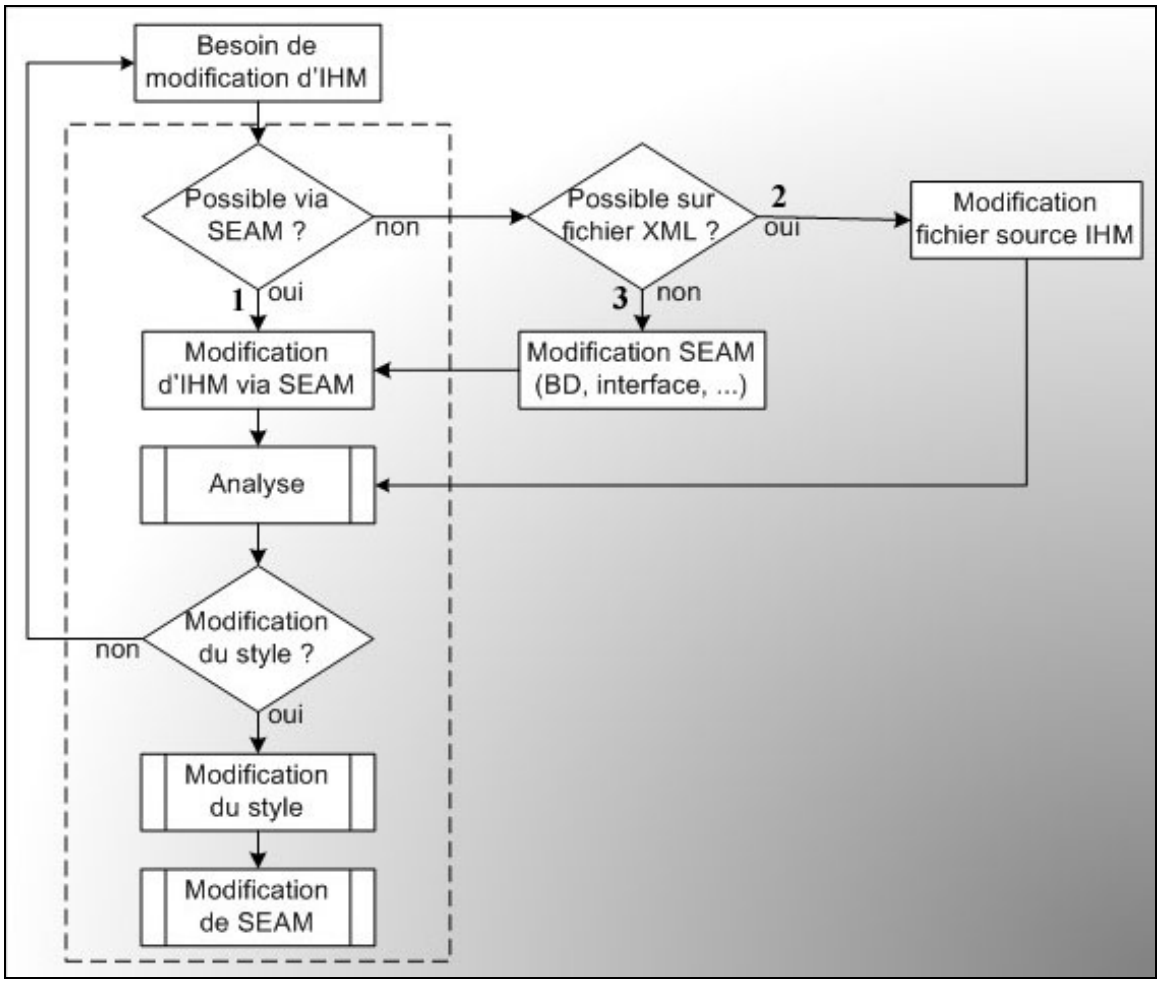


Figure VII. 1 : Processus de modification des IHMs, du style, et de SEAM

La section suivante présente une approche permettant de superviser et d'analyser l'évolution des IHMs, ce qui constitue une étape indispensable pour la maintenance du style.

VII. 3 Supervision de l'évolution des architectures définies à partir du style

Le but de cette section est de définir un processus de supervision de l'évolution des architectures. L'approche proposée concerne le cas d'un processus de développement architectural classique (Cf. paragraphe VII.2.2) : les architectures dont l'évolution est supervisée ont été directement instanciées à partir d'un style architectural, et non via un environnement de développement tel que SEAM. La section VII.5 présentera quant à elle l'adaptation de cette approche dans le contexte d'un environnement de développement spécifique.

VII. 3. 1 Processus de supervision des architectures

La supervision d'architectures logicielles construites à partir d'un style peut avoir deux objectifs distincts :

1. garantir la cohérence entre les architectures et leur style de référence tout au long de leur durée de vie ;
2. prendre en compte les modifications appliquées aux architectures pour améliorer leur style de référence.

L'approche suivie pour obtenir le style de référence suggère de considérer principalement ce deuxième aspect. En effet, le style de référence a été bâti de façon inductive à partir d'exemples, les architectures prototypes. Mais l'utilisation d'un style construit à partir d'exemples pour générer une famille complète d'applications implique certains risques. Dans un tel cas, l'expérience est insuffisante pour garantir que le style est complet, qu'il satisfait pleinement les besoins correspondant aux applications à générer dans le futur. Le style peut donc être amélioré. En conséquence, l'objectif est de superviser l'évolution des architectures dans le but de tirer profit de l'expérience acquise en améliorant itérativement le style en fonction de ces évolutions.

La figure VII.2 représente les principaux stades du développement architectural où différentes formes de supervision peuvent s'appliquer :

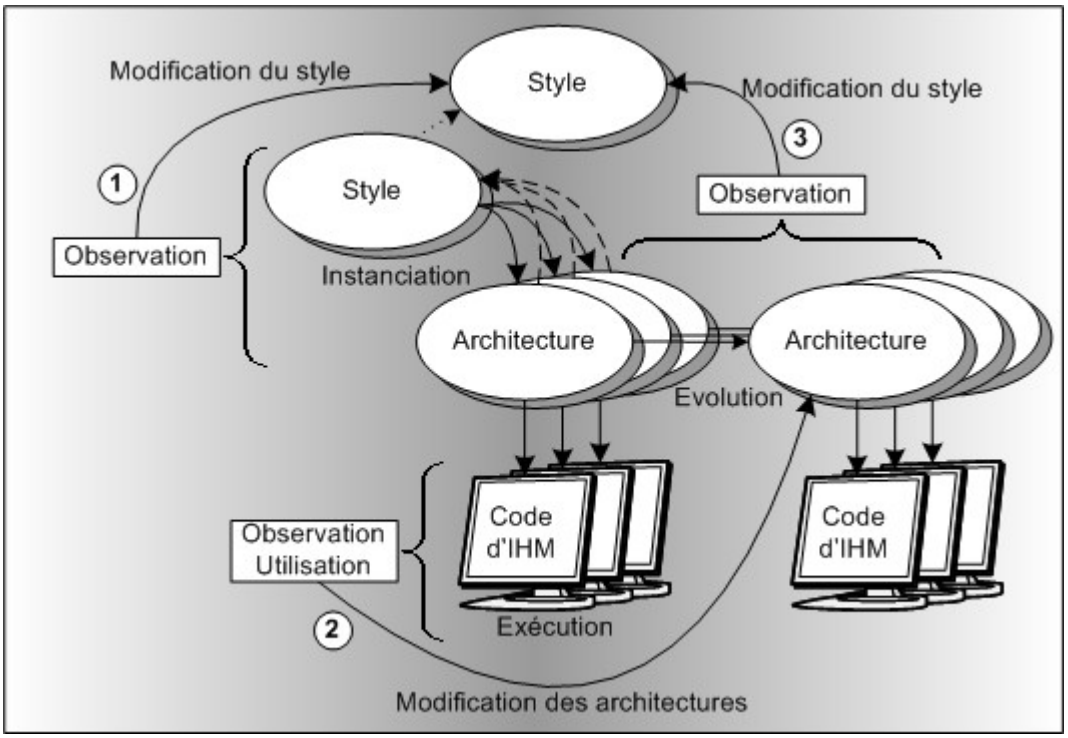


Figure VII. 2 : Supervision du développement orienté architecture

1- supervision de ***l'instanciation*** d'architectures à partir du style. Le style définit un certain nombre de propriétés que les architectures doivent respecter de préférence (« soft »), ou obligatoirement (« hard »). D'autres éléments sont laissés libres et

sont choisis par l'architecte lors de l'instanciation. On peut alors conclure de l'observation des statistiques des valeurs choisies que certaines modifications devraient être apportées au style (ex : une variable souvent instanciée à la même valeur peut faire l'objet d'une nouvelle propriété « soft » ; une propriété « soft » toujours respectée peut être transformée en propriété « hard » ; une propriété « soft » jamais respectée peut être supprimée…) ;

2- supervision de ***l'exécution*** des applications générées à partir des architectures qui suivent le style de référence. L'observation du comportement de l'application peut mettre en évidence certains dysfonctionnements qui doivent être résolus au niveau architectural ;

3- supervision de ***l'évolution des architectures***. Suite au résultat de la supervision de l'exécution (2), ou à de nouveaux besoins des utilisateurs, il est souvent nécessaire de faire évoluer l'architecture des applications. Il est alors intéressant de mesurer l'écart entre les nouvelles versions d'architectures entre elles, ou entre celles-ci et le style de référence, ou encore de mesurer les tendances d'évolution. Ceci peut permettre, sous certaines conditions, de mettre à jour le style de référence pour que celui-ci soit mieux adapté aux nouveaux besoins des utilisateurs. Le style se perfectionnera au fur et à mesure des itérations. Ainsi, lorsqu'une nouvelle application sera nécessaire, celle-ci pourra être générée à partir d'une architecture qui respecte la nouvelle version du style de référence. Cette nouvelle application profitera donc de l'expérience acquise et répondra plus pleinement aux besoins.

Dans ce contexte, le style de référence ne doit pas contraindre l'évolution des architectures construites à partir de celui-ci. Après leur instanciation, les architectures sont découplées du style et peuvent être modifiées en fonction de nouveaux besoins. En effet, certains besoins supplémentaires nécessitent de modifier les architectures des IHMs. Il est alors intéressant de mesurer la déviation entre les nouvelles versions d'architectures, ou entre elles et le style de référence. L'analyse de ces mesures peut indiquer, dans certaines conditions, qu'il est nécessaire d'améliorer le style pour le rendre plus proche des nouveaux besoins exprimés par les utilisateurs.

Le présent chapitre propose une approche permettant de faire co-évoluer les styles et les architectures. Cette approche, basée sur la supervision des évolutions des architectures, est décomposée en cinq étapes (composant la phase d'analyse représentée figure VII.1) :

1. observation des architectures et comparaison des versions (section VII.3.2.1);
2. analyse des évolutions : statistiques, tendances d'évolution, classification (section VII.3.2.2) ;
3. détermination des modifications à appliquer au style (section VII.4.1) ;
4. analyse de l'impact qu'auraient ces modifications si elles étaient appliquées (section VII.4.2) ;
5. décision et implémentation des améliorations du style (section VII.4.3.).

Ces cinq étapes peuvent être groupées en deux niveaux distincts :

- le niveau des architectures : observation et analyse ;

- le niveau du style : modifications applicables, analyse d'impact, et implémentation des modifications.

Les deux niveaux de ce processus de supervision sont représentés figure VII.3.

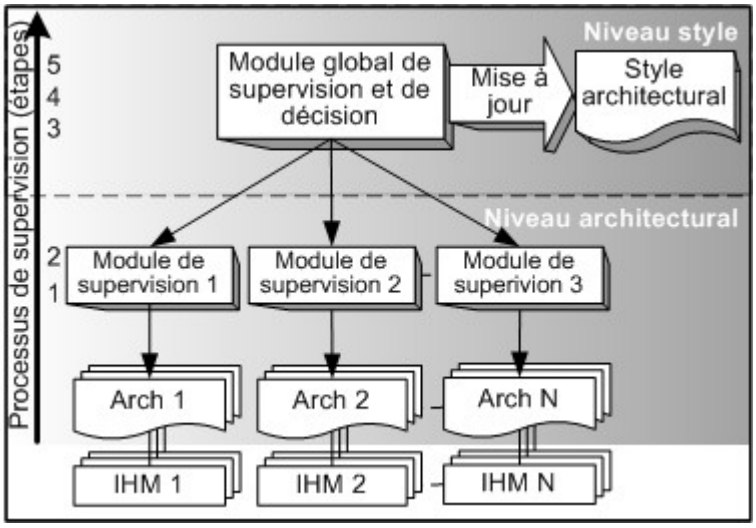


Figure VII. 3 : Structure de supervision en couches

La fonction de la première couche de cette structure est de superviser l'évolution des architectures et de procéder à la comparaison de leurs versions successives. Les modules de supervision doivent fournir le résultat de cette comparaison, qui peut être par exemple la notification de la modification d'une propriété, au module de la couche supérieure.

Cette seconde couche, le module global de supervision, utilise les informations fournies par les modules du niveau inférieur pour comparer les différentes tendances d'évolution. Ce module est constitué d'un système expert, c'est à dire d'un outil d'aide à la décision permettant aux utilisateurs de déterminer si le style doit être mis à jour ou non.

Le résultat attendu de cette solution est la prise en compte de toutes les modifications pertinentes appliquées aux architectures pour améliorer progressivement le style, et en conséquences les IHMs qui seront construites à partir de celui-ci dans le futur.

VII. 3. 2 Evolution des architectures

VII. 3. 2. 1 Observation des architectures et comparaison des versions

L'étape 1 consiste à collecter des informations concernant l'évolution des architectures. Une première action est de déterminer si les architectures modifiées respectent toujours leur style de référence ou non. Ceci est assuré par le « customizer » ArchWare utilisé par le vérificateur de conformité de l'environnement de développement. Il est ensuite nécessaire d'obtenir des informations concernant les propriétés structurelles et comportementales des architectures modifiées afin de pouvoir les comparer à celles des précédentes versions des architectures, et d'en déduire les évolutions. Pour cela il a fallu ajouter au vérificateur de conformité une fonctionnalité permettant de collecter et

d'archiver les différentes informations présentes dans les architectures d'IHMs (valeurs des attributs, nombre d'éléments architecturaux de chaque style, …). Ces informations sont stockées dans une table de base de données, et des procédures permettent de les consulter pour retourner les points et les tendances d'évolution.

VII. 3. 2. 2 Analyse des évolutions

Afin de pouvoir analyser efficacement les futures évolutions des architectures, il a été nécessaire de comprendre quelles caractéristiques celles-ci pouvaient avoir. Dans cet objectif il a été intéressant d'analyser les évolutions qu'ont connues les prototypes d'architectures d'IHMs. Ces IHMs avaient pour fonction de superviser les systèmes nécessaires à l'opération d'un accélérateur particulier du CERN, le SPS (Super Proton Synchrotron). Le développement de ces IHMs a pris plusieurs mois, et n'est pas encore tout à fait complet car leurs utilisateurs demandent régulièrement la supervision de systèmes supplémentaires. Le paragraphe suivant présente les principales modifications qui ont été appliquées aux architectures d'IHMs.

VII. 3. 2. 2. 1 Analyse des évolutions des IHMs prototypes

Depuis le début du développement des prototypes d'IHMs, de nombreuses améliorations ont été appliquées. Celles-ci sont notamment le résultat de notifications de dysfonctionnements ou de l'apparition de nouveaux besoins. Toutes ces mises à jour ont été archivées dans un système logiciel de type CVS (Concurrent Version System). L'étude des différentes versions des IHMs a permis de répertorier plus de cinquante modifications appliquées sur environ vingt IHMs au cours d'une année. Celles-ci ont été classées suivant le type d'élément (IHM globale, IHM spécifique, bloc système, …) sur lequel elles ont été appliquées, ainsi que suivant le nombre d'éléments de ce type concernés. Il a par ailleurs été intéressant de considérer les dates des modifications de façon à être capable de connaître la fréquence d'évolution de chaque élément. Ceci est utile pour déterminer quels sont les éléments matures, et quels sont ceux qui ne le sont pas. Une fois ces modifications classées en fonction du type et du nombre d'éléments concernés, la notion de type d'évolution a été ajoutée (exemples : nouvelle fonctionnalité, modification du traitement des données, correction de défaut, modification des paramètres d'instanciation, amélioration graphique, modification structurelle). Il a de plus été utile de considérer la cause de chaque modification afin de déterminer si celle-ci pourrait se reproduire dans un autre contexte, et comment cela pourrait être évité.

En résumé, les informations pour chacune des modifications sont les suivantes :

- libellé de la modification (exemple : ajout du système « air comprimé ») ;
- date de la modification ;
- élément(s) concerné(s) par la modification (exemple : toutes les IHMs spécifiques) ;
- classe de modification (exemple : modification structurelle) ;
- cause de la modification (exemple : définition incomplète de l'architecture – la présence de ce système n'était pas requise).

La plupart de ces modifications ont entraîné un raffinement correspondant du style GTPM qui était alors en cours de développement. En effet, parmi les cinquante

modifications d'IHM répertoriées, six d'entre elles concernaient le style *GlobalHCI* du style GTPM, vingt trois concernaient le style *IndividualHCI*, et dix huit concernaient le style *StatusBloc*.

Cette étude indique précisément comment les IHMs initiales ont évolué : plusieurs nouvelles propriétés ont été ajoutées aux IHMs et aux blocs ; la structure de l'IHM globale a évolué ; de nouveaux blocs systèmes ont été ajoutés ; le comportement des blocs ont évolué ; etc. L'analyse de ces modifications a servi de base à la définition de critères pour l'analyse des évolutions des futures architectures d'IHMs, ce qui fait l'objet du paragraphe suivant.

VII. 3. 2. 2. 2 Analyses des évolutions des architectures

Une fois l'étape de l'observation des évolutions achevée, le deuxième point du processus de supervision consiste à les analyser et à déterminer si elles doivent impliquer des modifications au niveau du style. Plusieurs questions apparaissent alors :

- Est-ce que ces améliorations ont un intérêt pour les autres architectures qui respectent le style en question ?
- Ou, est-ce seulement le résultat d'un besoin spécifique à une architecture particulière ?
- Est-ce une modification temporaire qui évoluera probablement de nouveau ?
- Des améliorations similaires ont-elles été appliquées à d'autres architectures qui suivent le même style ?

Les réponses à ces questions peuvent être trouvées à l'aide d'analyses statistiques ainsi que par la considération du facteur temps.

L'analyse statistique permet de déterminer les tendances d'évolution en comparant les évolutions de plusieurs architectures instanciées à partir du style de référence. Si plusieurs architectures qui suivent le même style ont reçu des améliorations similaires, on peut considérer que le style doit être modifié dans ce sens. Au contraire, quand une architecture est modifiée en fonction d'un besoin spécifique, qui ne concerne pas les autres architectures suivant le même style, le style de doit pas être mis à jour, et l'architecture modifiée reste donc découplée du style.

La considération du facteur temps peut être très pertinente pour déterminer si l'évolution d'une architecture doit être prise en compte au niveau du style. En effet, une architecture qui ne satisfait pas les utilisateurs sera rapidement revue pour être améliorée. Au contraire, une architecture satisfaisante aura vraisemblablement une durée de vie plus longue. Il est donc possible de définir une durée d'utilisation d'une version d'IHM à partir de laquelle celle-ci est supposée satisfaire les utilisateurs, et peut donc être utilisée pour améliorer le style. A l'opposé, il n'y a aucune garantie qu'une version d'IHM venant juste d'être mise en opération ne satisfasse pleinement les utilisateurs, elle devra donc probablement évoluer rapidement (cet aspect a été largement observé dans la série des modifications appliquées aux prototypes d'IHMs). Dans ce cas, les modifications appliquées aux IHMs ne doivent pas être immédiatement prises en compte pour améliorer le style.

A partir de ces considérations il a été possible de définir un jeu de règles permettant de déterminer si une modification appliquée sur une ou plusieurs architectures doit impliquer une modification du style. La classification des modifications d'architectures

présentée dans la section VII.3.2.2.1 a servi de base pour atteindre ce but. Par exemple, une règle stipule que si une modification structurelle a du être apportée à une IHM spécifique, et qu'elle a causé la violation d'une propriété (trop contraignante) du style, et que cette modification a été éprouvée pendant une durée définie, alors la propriété concernée doit être affaiblie au niveau du style « IHM spécifique ».

A ce niveau du processus de supervision il n'est possible que de recommander une modification du style, mais pas encore de déterminer si cette modification est possible. Ce dernier point, qui constitue l'étape 3 du processus de supervision, fait l'objet de la section suivante.

VII. 3. 3 Suppositions

La méthode proposée pour superviser l'évolution des architectures permet de superviser indirectement l'évolution des IHMs. Ceci implique qu'il est indispensable de s'assurer que le lien entre les IHMs et leurs architectures est maintenu en permanence. Cependant, il serait possible, à l'aide d'outils non architecturaux (exemples : éditeurs de texte, outil d'édition graphique), d'appliquer des modifications directement sur le code des IHMs. Une telle manipulation découplerait les IHMs de leurs architectures, et rendrait en conséquence impossible la supervision de l'évolution au niveau architectural. Cette limitation peut être résolue par l'utilisation d'un outil de « reverse engineering », qui permettait de reconstruire l'architecture de l'IHM à partir de son code. Toutefois, ces techniques sont souvent difficiles à implémenter et ont une efficacité inégale selon le type de code utilisé. En conclusion, dans le cas ou la reconstruction des architectures d'IHMs à partir de leur code n'est pas possible, il est nécessaire de faire l'hypothèse suivante : chaque fois qu'un utilisateur requiert une nouvelle version d'IHM, c'est l'architecture associée qui sera mise à jour et utilisée pour générer le code de la nouvelle version d'IHM.

Dans l'approche proposée par l'outil SEAM, toutes les informations concernant les IHMs sont stockées d'une manière très structurée dans une base de données. Ceci permet, de générer les architectures des IHMs en langage ADL à partir de ces informations : cette tâche est prise en charge par le vérificateur de conformité présenté dans le chapitre précédent.

VII. 4 Mise à jour et évolution du style

L'analyse et la classification des évolutions des architectures permettent de déterminer quand et comment le style doit être amélioré. Mais il faut ensuite vérifier que le style puisse effectivement être amélioré dans ce sens. Dans l'objectif connaître les modifications applicables sur le style, il a été nécessaire de lister les propriétés et les contraintes structurelles imposées par celui-ci. Pour chacune d'elles il a été déterminé :

- si elle peut être trop ou trop peu contraignante (exemple : une propriété du style, *max_bloc_number*, restreint à vingt le nombre de blocs présents sur une IHM. Ce nombre peut devenir trop contraignant) ;
- si elle peut être renforcée ou affaiblie (exemple : si la propriété *max_bloc_number* devient trop contraignante, celle-ci sera affaiblie) ;

- sous quelles conditions spécifiques (statistiques, facteur temps, …) les modifications peuvent être appliquées (exemple : la propriété sera affaiblie seulement si le nombre *max_bloc_number* est trop restrictif pour plus de une IHM) ;
- quelles parties du style (propriétés) doivent être modifiées (exemple : uniquement la propriété *max_bloc_number*) ;
- quel impact a cette modification sur la famille d'architectures qui respecte le style (exemple : pas d'impact sur les architectures existantes car la propriété devient moins restrictive).

Le dernier point constitue l'étape 4 du processus de supervision, l'analyse de l'impact des modifications.

VII. 4. 1 Analyse d'impact

Lorsque la supervision de l'évolution d'une architecture suggère la possibilité de modifier son style, il est nécessaire de prévoir toutes les modifications qu'elle va impliquer pour les autres architectures qui respectent le même style, et d'évaluer si elles sont acceptables. L'analyse d'impact peut être effectuée en utilisant une technique de chaînage [Stafford et al. 1998]. La matrice générée par cette technique fournit les interdépendances entre les éléments architecturaux, et permet l'analyse de l'impact d'une modification architecturale. De plus, il est essentiel de vérifier que la mise à jour des architectures suivant le style ne va pas écraser des modifications spécifiques déjà appliquées sur elles mais n'ayant pas été propagées au niveau du style.

VII. 4. 2 Décision

La dernière étape du processus de supervision est la phase de décision. Celle-ci détermine les modifications à appliquer au style. Un outil de support à la décision, ou système expert, a été mis en place pour atteindre cet objectif. Ce système contient le jeu de règles, critères, formules qualitatives, déduits de la classification et des études précédentes. Une règle de décision permet de stipuler qu'une propriété doit être modifiée si une modification est recommandée, si celle-ci est applicable, et si les conditions spécifiques sont respectées.

Considérons un exemple sur l'ensemble du processus de supervision :

- *étape 1, observation des architectures :* deux architectures d'IHMs sont modifiées pour pouvoir afficher plus de vingt blocs ;
- *étape 2, analyse des évolutions :* il s'agit d'une modification structurelle appliquée à des IHMs spécifiques, le style est violé car la propriété *max_bloc_number* est trop contraignante. La règle donnée en IV.6.3.2 est exécutée et la recommandation est : affaiblir la propriété *max_bloc_number* au niveau du style IHM spécifique
- *étape 3, amélioration possible :* oui, car la propriété *max_bloc_number* peut être affaiblie si plus d'une IHM est concernée par la modification ;
- *étape 4, analyse d'impact :* pas d'impact sur les architectures existantes car la propriété devient moins restrictive ;

- *étape 5, décision :* en accord avec la règle mentionnée ci-dessus, la propriété *max_bloc_number* est affaiblie.

Le jeu de règles devra être mis à jour en fonction de l'expérience qui va être acquise graduellement et des nouveaux cas qui vont apparaître. Dans un premier temps, le système de supervision doit informer l'architecte des évolutions d'architectures et l'aider à prendre des décisions concernant les améliorations à apporter au style de référence. Une seconde étape sera d'ajouter une fonctionnalité de contrôle du style permettant de supporter sa mise à jour automatique en fonction des modifications suggérées.

VII. 4. 3 Utilisation du style mis à jour

Lorsque le processus est terminé, la nouvelle version du style peut être réutilisée pour produire des versions mises à jour des architectures. A ce niveau, il est possible de choisir entre trois alternatives. La nouvelle version du style peut être utilisée :

- pour instancier uniquement les futures architectures, mais pas pour ré-instancier les architectures existantes ;
- pour instancier les futures architectures, ainsi que pour ré-instancier certaines architectures existantes (selon des critères à définir) ;
- pour instancier les futures architectures, ainsi que pour ré-instancier toutes les architectures existantes.

VII. 5 Evolution de l'environnement de développement

Comme cela a été expliqué dans la section VII.2.3, il existe plusieurs manières de faire évoluer les IHMs produites par l'environnement de développement. Chaque fois qu'une IHM est mise à jour, les nouvelles informations la concernant sont stockées dans la base de données de l'environnement. Cet environnement propose des fonctionnalités permettant de prendre en charge le processus de supervision d'architectures et d'amélioration du style présenté dans la section précédente. Deux d'entre elles sont fournies par le vérificateur de conformité (Cf. section VI.3.4) :

- la construction des architectures d'IHMs en langage ADL à partir des informations présentes dans la base de données ;
- la vérification de conformité des architectures ainsi obtenues par rapport au style de référence.

Toutefois, le vérificateur de conformité, tel qu'il a été décrit dans le chapitre 6, ne permet de comparer au style que la version « courante » d'une architecture, il est donc incapable de fournir le résultat d'une analyse des tendances d'évolutions des architectures.

C'est pourquoi, afin de pouvoir implémenter la première couche de supervision représentée figure VII.3 (analyse des versions successives des architectures), il a fallu modifier la structure de la base de données de façon à archiver l'historique des modifications appliquées aux IHMs. Ainsi, il est possible d'obtenir les versions

successives des architectures d'IHMs en langage ADL, et déterminer quelles sont les contraintes du style qui sont trop ou trop peu contraignantes pour une IHM donnée.

A partir du résultat de cette analyse, la deuxième couche de supervision représentée figure VII.3 (module global de supervision) peut déterminer, par analyse statistique et temporelle, les tendances d'évolution de toutes les IHMs. Cette couche, qui constitue le module d'aide à la décision (un exemple de règle de décision est présenté section VII.4.2), permet d'indiquer à l'utilisateur qu'une modification du style est souhaitable ou non.

Sans la mise en place de ces deux couches de supervision le vérificateur de conformité était simplement capable d'indiquer à l'utilisateur si une contrainte du style avait était violée par une IHM donnée. L'utilisateur était alors libre de corriger l'IHM pour la rendre conforme, ou de la laisser non conforme pour une raison particulière. Maintenant l'outil va beaucoup plus loin en indiquant à l'utilisateur, par exemple, que la contrainte est violée par l'IHM donnée, mais aussi par toutes les autres IHMs, qu'un nouveau besoin pertinent semble donc être apparu, et qu'une mise à jour du style pourrait être une solution. En outre, il a été nécessaire de spécifier dans l'outil d'aide à la décision dans quel module de l'environnement de développement est gérée chacune des contraintes. L'outil est ainsi capable d'indiquer si le style devrait être mis à jour, et quelle contrainte devrait être modifiée dans quelle partie de l'outil.

Les paragraphes suivants fournissent des détails concernant la modification de contraintes du style, selon qu'elles soient uniquement spécifiées au niveau du style formel, codées au niveau de la base de données et de son interface, ou codées au niveau du modélisateur graphique.

VII. 5. 1 Evolution du style

Certaines des contraintes spécifiées par le style n'ont volontairement pas été codées dans les modules de l'environnement de développement pour qu'elles puissent évoluer facilement. Il s'agit des contraintes que l'utilisateur peut violer lorsqu'il développe des IHMs mais dont ces violations sont indiquées lors de la vérification de conformité. Si l'utilisateur ne tient volontairement pas compte des indications de non-conformité lors du développement de plusieurs IHMs, le module de supervision en déduira que le style n'est plus conforme aux besoins et qu'il doit être modifié. Une mise à jour de contraintes au niveau du style est relativement facile à implémenter, même si elle nécessite une connaissance de langage ADL. En outre, elle n'a pas d'influence sur les informations concernant toutes les IHMs déjà stockées.

Dans de nombreux cas, au contraire, les contraintes ont été codées dans les modules de l'environnement de façon à guider l'utilisateur dans le développement d'IHM. Cette méthode est efficace mais comporte l'inconvénient de rendre difficile les évolutions futures. En effet, si une de ces contraintes s'avère trop, ou trop peu, contraignante, il est non seulement nécessaire de mettre à jour le style formel, mais aussi le module dans lequel la contrainte est codée. Dans ce contexte, l'approche proposée section IV.2.3 présente un grand intérêt. En effet, dans cette approche, l'environnement de développement est décrit par une architecture formelle intégrant le style. Ainsi, lorsque le style évolue, l'architecture de l'environnement de développement évolue aussi, ce qui permet de faciliter la ré-implémentation de l'environnement à partir de celle-ci.

Les paragraphes suivants présentent des aspects concernant l'implémentation d'évolutions aux modules correspondant aux composants de l'architecture de l'environnement de développement.

VII. 5. 2 Evolution de la base de données et de son interface

Lorsque le module de supervision indique qu'une contrainte du style doit être modifiée dans la base de données de l'environnement il est très souvent nécessaire de mettre aussi à jour son interface d'exploitation. Ces deux opérations nécessitent l'intervention de programmeurs connaissant l'outil.

Certaines des opérations de maintenance d'une base de données s'effectuent simplement : il ne peut être nécessaire que de rajouter ou supprimer une contrainte structurelle n'ayant pas d'impact sur les données stockées. De même, il suffit parfois de modifier une plage de valeurs utilisables dans la table de « domaine ». Si, comme dans un exemple précédemment cité, le module de supervision indique que plusieurs blocs de plusieurs IHMs sont associés à une salle de contrôle non définie dans la base de données, il est simple et rapide de rajouter les informations concernant celle-ci dans la table de « domaine ».

Toutefois, il peut-être parfois nécessaire de mettre à jour la structure de la base de données, ce qui peut entraîner des difficultés. Dans certains cas, des modifications à appliquer sur des tables de la base nécessitent que celles-ci soient vides. Il est donc nécessaire de préalablement sauvegarder les données stockées, appliquer les modifications, et restaurer les données, ce qui est une opération délicate.

VII. 5. 3 Evolution du modélisateur graphique

Les modifications à appliquer au modélisateur graphique sont, elles aussi, de complexité inégale. En effet, un grand nombre des paramètres graphiques des IHMs sont gérées par un fichier CSS (Cascading Style Sheet). Dans certains cas, la mise à jour du style peut n'impliquer qu'une modification de ce fichier. Si, par exemple, le module de supervision indique que les développeurs d'IHMs utilisent systématiquement une valeur de police de caractères différente de celle préconisée par le style, il suffit de mette à jour la contrainte du style, puis d'éditer le fichier CSS, et de modifier le paramètre correspondant.

D'en d'autres cas toutefois, il est nécessaire de modifier le code Java du modélisateur graphique. Si, par exemple, il devient utile que les blocs affichent une nouvelle valeur d'information, il est nécessaire de mettre à jour la base de données, le fichier CSS, et les classes Java gérant les objets graphiques. Cette opération nécessite une nouvelle phase conséquente de développement et de déploiement.

VII. 6 Conclusion

Ce chapitre a présenté une approche permettant de faire co-évoluer les architectures et les styles dans un processus de développement architectural. La définition inductive d'un style à partir d'exemples d'architectures implique nécessairement que celui-ci n'est pas « idéal », et qu'il sera donc amené à évoluer.

L'approche présentée considère que les nouveaux besoins des utilisateurs seront implémentés au niveau des architectures, et qu'il est donc nécessaire de superviser l'évolution de celles-ci pour pouvoir répercuter les nouveaux besoins au niveau du style architectural. Le processus de supervision proposé prend en compte les tendances d'évolution des IHMs, et utilise des critères décisionnels établis à partir de l'expérience, pour indiquer à l'utilisateur si le style doit être mis à jour, et de quelle manière.

L'évolution du style entraîne l'évolution de l'architecture de l'environnement de développement. La ré-implémentation de l'environnement de développement à partir de son architecture lui permet de suivre l'évolution du style. Cette étape qui est aujourd'hui réalisée manuellement, et nécessite donc l'intervention de programmeurs, serait facilitée par la mise en place d'un outil de génération de code. Avec cet outil, l'approche proposée dans cette thèse permet de faire évoluer un environnement de développement automatiquement en fonction de l'évolution des besoins concernant les applications à développer.

Chapitre VIII Validation : études de cas

Chapitre VIII : Validation : études de cas

VIII. 1 Introduction

Les précédents chapitres ont présenté la conception, le développement, et l'évolution d'un outil de développement de logiciels pour la supervision d'accélérateurs de particules. Afin d'évaluer l'utilité et le bien fondé des techniques employées dans ce contexte, ce chapitre propose d'analyser les résultats observés au moyen de trois études de cas. Celles-ci ont été conçues en suivant le guide de [Perry et al. 2004] établi à partir des ouvrages de [Yin 2003a][Yin 2003b]. Ces auteurs ont répertorié quatre types d'études de cas (Cf. figure VIII.1) qui vont permettre de positionner celles présentées dans ce chapitre. Les deux types les plus simples ne comportent l'étude que d'un seul cas pour un contexte donné. Ces types se différencient par le fait que l'étude peut être effectuée en collectant des données sur une unité d'analyse, ou sur plusieurs unités d'analyses. Les deux types les plus complexes comportent l'étude de plusieurs cas, pour plusieurs contextes différents, avec une ou plusieurs unités d'analyse.

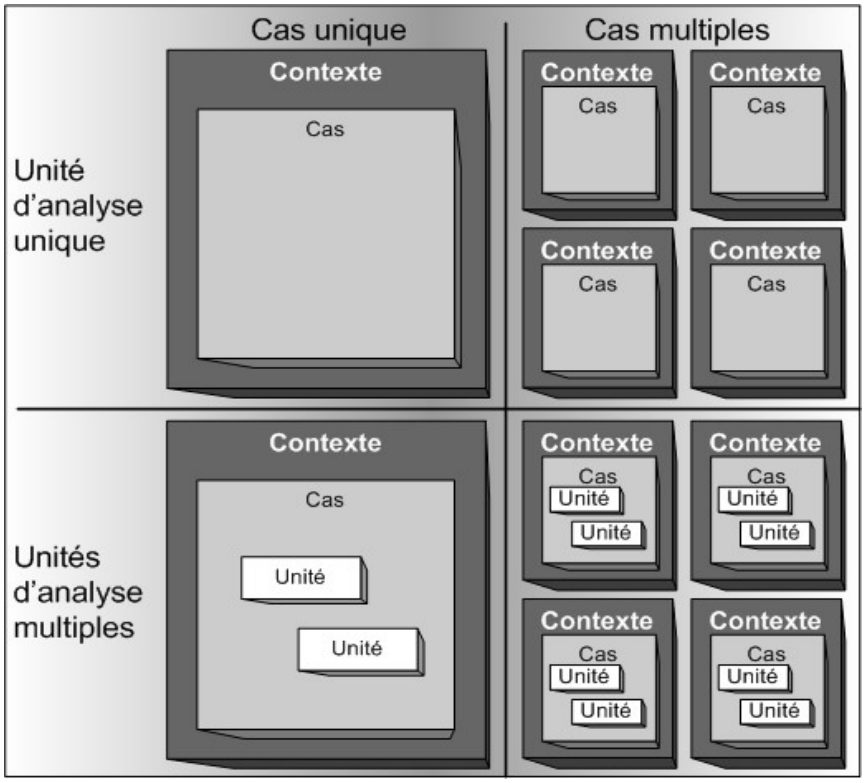


Figure VIII. 1 : Types de conceptions d'études de cas

Dans le cadre de la problématique de cette thèse, la première question qui se pose concerne l'intérêt d'utiliser un environnement de développement spécifique au domaine des accélérateurs de particules, plutôt qu'un outil de développement de logiciels de supervision générique tels que ceux disponible sur le marché. Ce point fait l'objet de la première étude de cas.

La deuxième étude de cas a pour but de quantifier l'influence de l'adoption d'une approche centrée style pour la définition, le respect, et l'évolution d'un langage spécifique à un domaine d'application particulier.

La troisième et dernière étude de cas considère l'utilité de fournir un support pour l'évolutions des architectures, et la mise en place d'un processus de supervision de ces évolutions pour s'assurer que l'environnement de développement associé reste bien adapté à son contexte d'utilisation.

VIII. 2 Environnement de développement spécifique

L'étude de cas présentée dans cette section cherche à répondre à la question suivante : comment l'utilisation d'un langage (et environnement de développement associé) spécifique au domaine des accélérateurs de particules influence le développement des logiciels de supervision ? Afin de traiter cette question les paragraphes suivants vont présenter les éléments clés intervenant dans cette étude, notamment les détails de sa conception, les données collectées, ainsi que l'analyse des résultats obtenus.

L'outil SEAM, dont la conception et l'implémentation ont été présentées précédemment, propose un langage spécifique au développement de logiciels pour la supervision des accélérateurs de particules. Ce langage est utilisable via les différents modules logiciels (base de données, interfaces, modélisateur) composant l'outil SEAM. Pour déterminer l'influence de SEAM dans le développement des logiciels, il est nécessaire de considérer les résultats attendus de son utilisation. Ainsi, les résultats attendus de cette étude de cas devraient indiquer que SEAM :

1. rend le développement des logiciels accessible aux spécialistes de la supervision des accélérateurs (non informaticiens) ;
2. permet la mise en place de logiciels de supervision uniformisés ;
3. améliore la compréhension du domaine et la productivité (temps de développement).

Afin de vérifier si SEAM a permis d'atteindre ces objectifs, cette section présente une étude composée de deux cas différents. Le premier cas, concernant la création des logiciels de supervision du SPS et du CPS, est composé de deux unités d'analyses (SPS et CPS). Le second cas (création des logiciels de supervision des systèmes cryogéniques) n'est composé que d'une seule unité d'analyse.

VIII. 2. 1 Création des logiciels de supervision du SPS et du CPS

VIII. 2. 1. 1 Conception de l'étude de cas

SEAM a été utilisé pour re-développer toutes les logiciels de supervision prototypes jusqu'alors utilisés en salle de contrôle pour superviser le redémarrage du SPS et du CPS. Une équipe de deux développeurs a été affectée à cette tâche. Il s'agit de deux opérateurs de la salle de contrôle technique (TCR), ayant une solide expérience dans l'utilisation d'outils logiciels, mais n'étant pas informaticiens de formation.

Concernant le contexte de développement, il est important de noter que les SPS et CPS sont des accélérateurs du CERN qui sont en opération depuis de nombreuses années. Cela signifie que leur fonctionnement est bien maîtrisé par les développeurs de logiciels de supervision. Toutefois, ils ont connu avec le temps de nombreuses modifications plus ou moins bien documentées, ce qui rend parfois difficile la conception des logiciels pour les superviser.

Les données collectées et analysées pour cette étude de cas sont les suivantes :

- le temps de développement de l'ensemble des logiciels pour le SPS et le CPS : cette information, comparée au temps de développement nécessaire lors de la phase de prototypage, permettra de vérifier si l'objectif 3 de SEAM est atteint (amélioration de la productivité) ;
- le nombre d'itérations de développement nécessaires à l'obtention de logiciels de supervision satisfaisants : ce point permettra de vérifier les objectifs 2 et 3 de l'outil SEAM ;
- le niveau d'interaction entre les développeurs de logiciels de supervision et le développeur de SEAM en terme de conseil (aide concernant l'utilisation de l'outil : objectif 3 de SEAM), et en terme de travail (développement, programmation : objectif 1 de SEAM) ;
- l'uniformisation des logiciels de supervision et satisfaction des utilisateurs (objectif 2 de SEAM).

Ces données proviennent de plusieurs sources :

- observation directe de l'avancement du développement : temps de travail dédié au développement des logiciels, itérations nécessaires ;
- observation participative : archivage des interactions (courriers électroniques, appels téléphoniques, déplacements), de leur fréquence, de leur durée, et de leur nature ;
- entretiens avec les différents développeurs et membres du projet ;
- logiciels de supervision obtenus.

VIII. 2. 1. 2 Collecte des données

VIII. 2. 1. 2. 1 Accélérateur SPS

Le temps nécessaire à deux développeurs pour la migration des 32 logiciels de supervision de redémarrage du SPS (comprenant environ 500 éléments graphiques, et ayant entraîné la génération de plus de 5000 lignes de code), a été de deux semaines à

plein temps, soit environ 5 heures de travail par logiciel de supervision. Ceux-ci n'avaient reçu qu'une formation sommaire aux différentes fonctionnalités de l'outil.

Le développement de ces logiciels a nécessité une participation importante du développeur de l'outil SEAM. Les interactions entre celui-ci et les développeurs ont été quantifiées notamment au moyen de l'archivage des courriers électroniques et des appels téléphoniques. Ainsi, les développeurs de logiciels de supervision ont demandé de l'aide et des conseils :

- en moyenne 4 fois par jour par téléphone ;
- en moyenne 2 fois par jour par courrier électronique ;
- en moyenne 2 fois par jour en personne.

Ces interactions sont reparties selon leur nature de la façon suivante :

- 50% de demande d'aide et de conseils dans l'utilisation de SEAM ;
- 40% de notifications de dysfonctionnements de l'outil SEAM ;
- 10% de demande de participation au développement des logiciels de supervision.

D'autre part, le développement a nécessité en moyenne trois itérations par logiciel. Les développeurs ont mis en place une première version de chacun des logiciels de supervision comportant uniquement les blocs représentant l'état de systèmes, une deuxième version incluant les liens graphiques, et une troisième et dernière version comprenant les éléments graphiques additionnels.

Concernant l'uniformisation des logiciels obtenus, les entretiens avec les utilisateurs et les membres du projet ont indiqué que celle-ci était tout à fait satisfaisante.

VIII. 2. 1. 2. 2 Accélérateur CPS

Le temps nécessaire à ces mêmes développeurs pour la migration des 10 logiciels de supervision de redémarrage du CPS, a été d'une semaine à mi-temps, soit environ 4 heures de travail par logiciel.

Le développement de ces logiciels a nécessité beaucoup moins de participation du développeur de l'outil SEAM :

- 4 appels téléphoniques au cours de la semaine ;
- 2 courriers électroniques au cours de la semaine;
- 2 déplacements en personne au cours de la semaine.

Ces interactions sont reparties selon leur nature de la façon suivante :

- 70% de demande d'aide et de conseils dans l'utilisation de SEAM ;
- 30% de notifications de dysfonctionnements de l'outil SEAM.

Dans le cas du CPS, le développement a nécessité en moyenne deux itérations par logiciel. Les développeurs ont mis en place une première version de chacun des logiciels de supervision comportant uniquement les blocs représentant l'état de systèmes, puis une deuxième version complète.

Là encore, l'avis des membres du projet a indiqué que l'uniformisation des logiciels était satisfaisante.

VIII. 2. 2 Création des logiciels de supervision des systèmes cryogéniques

VIII. 2. 2. 1 Conception de l'étude de cas

SEAM a été utilisé pour créer les logiciels de supervision du redémarrage des équipements cryogéniques pour le futur LHC. Contrairement au cas du SPS et du CPS, il ne s'agit pas d'une migration d'IHMs de PVSS vers JViews. En effet, ces logiciels de supervision n'ont jamais été développés sous PVSS.

Comme dans le cas précédent, une équipe de deux développeurs a été affectée à la création des logiciels. Là encore, ces développeurs ne sont pas informaticiens de formation, mais possèdent une solide expérience dans l'utilisation d'outils logiciels, ainsi que des notions de programmation.

Le contexte de ce développement est très différent de celui relatif aux logiciels de supervision du SPS et CPS. En effet l'installation des équipements cryogéniques est très récente et ceux-ci ne sont pas encore en phase d'opération. Les développeurs n'ont donc aucune expérience de supervision de ces systèmes, mais en contrepartie, la documentation qu'ils possèdent est à jour et correspond aux installations réelles.

Les données collectées et analysées, ainsi que leurs sources, sont les mêmes que celles de l'étude des cas SPS et CPS.

VIII. 2. 2. 2 Collecte des données

Le temps nécessaire aux deux développeurs pour la création d'un logiciel de supervision pour la supervision de redémarrage des systèmes cryogéniques, a été d'un mois à raison de 2 heures par semaine, soit 16 heures de travail.

Le développement de ce logiciel a nécessité une participation relativement faible du développeur de l'outil SEAM :

- 4 appels téléphoniques au cours du mois ;
- 4 courriers électroniques au cours du mois ;
- 2 déplacements en personne au cours du mois.

Ces interactions sont reparties selon leur nature de la façon suivante :

- 80% de demande d'aide et de conseils dans l'utilisation de SEAM ;
- 20% de notifications de dysfonctionnements de l'outil SEAM.

Le développement de ce logiciel de supervision a nécessité six itérations. Les développeurs ont réorganisé à plusieurs reprises la disposition des blocs et des liens, et ce, à la demande des différents membres du projet.

Il n'est pas possible de parler d'uniformisation dans la mesure où un seul logiciel a été crée. Toutefois, les entretiens avec les membres du projet ont indiqué que l'aspect graphique de celle-ci était conforme au « standard » GTPM.

VIII. 2. 3 Synthèse des données et conclusions

Le premier objectif de l'outil SEAM est de rendre le développement des logiciels de supervision accessible aux spécialistes de la supervision des accélérateurs (non-informaticiens). L'étude de cas présentée indique que les interventions d'informaticiens on été très limitées : les opérateurs des salles de contrôle ont été capables de développer la totalité des logiciels sans autre support que celui concernant l'utilisation de SEAM. A l'opposé, les tâches de programmation ont constitué environ 50% du travail lors du développement des prototypes de logiciels de supervision sous PVSS. Le langage et l'environnement proposés par SEAM apportent donc une avancée significative sous cet aspect.

Le second objectif de SEAM est de permettre la mise en place de logiciels de supervision uniformisés. Comme l'ont révélé les avis recueillis lors des entretiens avec les membres des projets GTPM concernant les logiciels produits via SEAM, cet objectif est lui aussi atteint. Toutefois, il n'est pas possible de dire que le niveau d'uniformisation est meilleur que celui atteint avec les prototypes. En effet, les logiciels prototypes étaient standardisés de façon satisfaisante, mais de nombreuses itérations de développement avaient été nécessaires pour atteindre ce résultat (en moyenne 10 par logiciel). L'environnement de développement SEAM contrôlé par les contraintes du style GTPM évite les erreurs de développement, notamment au niveau graphique. Ceci limite fortement le nombre d'itérations nécessaires à l'obtention de logiciels de supervision conformes au « standard » GTPM. L'étude de cas l'a prouvé en recensant un maximum de trois itérations par logiciel, qui ne correspondent d'ailleurs pas à des corrections, mais qui sont le résultat d'une répartition du travail choisie par les développeurs.

Le troisième objectif de SEAM est d'améliorer la compréhension du domaine et la productivité. Afin de déterminer si cet objectif a été atteint, il a fallu mesurer le temps de production des logiciels de supervision, le nombre d'itérations de développement, et le nombre d'interactions entre les développeurs des logiciels et le développeur de SEAM. Concernant le temps de développement, les résultats fournis par l'étude de cas indiquent que celui nécessaire à la production des logiciels pour la supervision du redémarrage des accélérateurs SPS et CPS a été beaucoup plus court que lors de la phase de prototypage. En effet, plus d'une année a été nécessaire au développement des prototypes sous PVSS. Cette amélioration est le résultat de l'utilisation du guide de développement fourni par SEAM. Toutefois, une théorie alternative serait de dire que ce n'est pas, ou pas entièrement, l'utilisation de l'outil SEAM qui a accéléré le processus de développement, mais simplement le résultat de l'expérience acquise lors du prototypage. Il est vrai que la phase de prototypage a permis de définir une documentation significative concernant le redémarrage du SPS et du CPS. Cependant, il est à noter que l'équipe de développement qui a utilisé SEAM n'est pas celle qui avait développé les prototypes, elle n'a donc pas pu tirer entièrement bénéfice de cette expérience. Par ailleurs, le développement du logiciel concernant la supervision des systèmes cryogéniques du LHC n'a pas connu de phase de prototypage préalable. Les résultats de l'étude de cas indiquent que le temps nécessaire aux développements de ce logiciel a été significativement inférieur à celui des prototypes du SPS et du CPS.

Par ailleurs, il est à noter que l'exploitation de SEAM a nécessité un grand nombre d'interactions entre son développeur et ses utilisateurs (les développeurs de logiciels de

supervision). Ceci s'explique par le fait que la migration des logiciels de supervision du SPS constituait la première mise en opération de SEAM : celui-ci n'avait jamais été testé en production et les développeurs des logiciels ne l'avaient encore jamais utilisé. Une grande partie des interactions avaient pour but de notifier des dysfonctionnements. Ceux-ci ont été corrigés rapidement et le temps de prise en main de SEAM par ses utilisateurs a été relativement cours. L'étude prouve en effet que lors de son deuxième cas d'exploitation (migration des logiciels de supervision du CPS) les interactions ont diminué de 90%.

VIII. 3 Influence de l'approche centrée style

Cette étude de cas a pour but de répondre à la question suivante : comment l'approche centrée style influe sur la production, le respect, et l'évolution d'un langage spécifique au domaine des accélérateurs de particules ? Afin de traiter cette question, la présente section expose la conception de cette étude, les données collectées, ainsi que l'analyse des résultats obtenus.

VIII. 3. 1 Conception de l'étude de cas

L'outil SEAM propose un langage, composé d'un vocabulaire et de contraintes, propre aux logiciels de supervision d'accélérateurs. Pour déterminer l'influence de l'approche centrée style mise en oeuvre dans le développement de ce langage, il est nécessaire de considérer les résultats attendus de son utilisation. Ainsi, les résultats attendus de cette étude de cas devrait indiquer que cette approche permet de :

1. définir un vocabulaire spécifique au domaine des accélérateurs ;
2. spécifier des contraintes propres au domaine (dont certaines ne seraient pas exprimables par une autre méthode) ;
3. garantir l'utilisation du vocabulaire et le respect des contraintes ;
4. faciliter l'évolution du langage et de l'environnement associé.

L'étude ne considère qu'un seul cas ne possédant qu'une seule unité d'analyse. Cette unité d'analyse est le style GTPM. L'expérience a pu être réalisée après avoir acquis la connaissance et de la pratique dans la mise en œuvre des techniques architecturales.

Les données collectées et analysées pour cette étude de cas sont les suivantes :

- la liste des « constituants » des logiciels de supervision définis par le projet GTPM ayant pu être décrit par le style formalisé (permet de vérifier si l'objectif 1 de l'approche est atteint) ;
- la liste des « règles » de développement définies par le projet GTPM ayant pu être décrites par le style formalisé (objectif 2) ;
- le niveau de complexité de la formalisation du style (objectif 1 et 2) ;
- le niveau de complexité de l'intégration du style dans un environnement de développement (objectif 3) ;
- les violations du style (objectif 3) ;
- le coût de la modification d'une contrainte formelle (objectif 4) ;

Ces données proviennent de plusieurs sources :
- le style lui même ;
- archivage des informations relatives au travail de formalisation du style (temps nécessaire, itérations, complexité) ;
- archivage des informations relatives au travail de d'intégration du style dans l'environnement (temps nécessaire, complexité) ;
- archivage des informations relatives au travail de modification des contraintes du style (temps nécessaire, complexité) ;
- entretien avec les utilisateurs du style (via son environnement SEAM) ;
- logiciels de supervision obtenus ;

VIII. 3. 2 Collecte des données

Le style a permis de formaliser tous les éléments nécessaires à la modélisation des logiciels de supervision GTPM, il s'agit des composants *GlobalHCI*, *IndividualHCI*, *StatusBloc*, *SystemStatus*, *MetaStatus*, *Equipment*, ainsi que du connecteur *DataLink*.

Par ailleurs le style a permis de décrire toute les contraintes que les logiciels GTPM doivent respecter (définies dans le chapitre 4) : il s'agit des contraintes graphiques, des contraintes de validité des informations, de positionnement des éléments, de cardinalité des éléments, d'interdépendance entre éléments, d'acquisition des données, de traitement des données et d'évolution.

L'étude des concepts architecturaux et la prise en main des techniques associées a nécessité plusieurs semaines. La formalisation du style en elle-même (à partir du langage C&C) et son intégration dans l'architecture de SEAM ont représenté une semaine de travail. L'implémentation de l'architecture de SEAM intégrant le vocabulaire et les contraintes du style a nécessité environ six mois de travail.

Les entretiens avec les utilisateurs du style via l'outil SEAM, et l'analyse des logiciels de supervision qu'ils ont produit, ont indiqué que ceux-ci respectaient les contraintes du style GPTM : aucune violation n'a été constatée.

Lors de la formalisation du style plusieurs nouveaux besoins ont été exprimés par les membres du projet GTPM. Les contraintes du style déjà formalisées ont du être modifiées en fonction de ces nouveaux besoins. Les modifications des contraintes au niveau du style formel se sont avérées rapides à mettre en œuvre (quelques minutes de travail). Toutefois, la ré-implémentation de ces contraintes modifiées dans les modules logiciels de SEAM est beaucoup plus coûteuse.

VIII. 3. 3 Analyse de données et conclusions

Le premier objectif de l'approche centrée style est la définition d'un vocabulaire spécifique au domaine des accélérateurs de particules. Les données présentées précédemment indiquent que l'approche a permis de définir un vocabulaire décrivant des logiciels de supervision de haut niveau, incluant des IHM spécifiques à des localisations ou modes donnés, affichant des blocs acquérant les données concernant des états d'équipements. Il est en outre possible de matérialiser les dépendances entre blocs. L'approche centrée style s'est donc avérée efficace pour spécifier le vocabulaire du langage dédié.

Le second objectif est la spécification des contraintes propres au domaine. L'étude de cas a indiqué que toutes les contraintes étaient formalisables avec le langage architectural utilisé. Par ailleurs, il est à noter que la définition du style a été relativement aisée, et ce principalement grâce à l'utilisation du langage de haut niveau C&C. En effet, un essai de formalisation du style directement à partir de l'ADL ArchWare avait été préalablement réalisé, et il s'agit d'un travail bien plus délicat. Par ailleurs, la formalisation et l'utilisation d'un style supposent que l'architecte maîtrise correctement les différents aspects du développement formel dans le cadre des architectures logicielles, et comprend parfaitement les besoins des utilisateurs.

Concernant le troisième objectif de l'approche centrée style, qui est de garantir l'utilisation du vocabulaire et le respect des contraintes, l'étude de cas a indiqué qu'aucune contrainte du style n'avait été violée. Pour permettre le développement de logiciels de supervision respectant les contraintes du style il a été nécessaire d'implémenter ces contraintes dans SEAM. Ce travail s'est avéré long et relativement complexe dans le cas étudié, mais pourrait être facilité par la mise en place d'un outil de générant le code de l'environnement de développement à partir de son architecture formelle intégrant les contraintes du style. En conclusion, l'approche centrée style est efficace pour garantir le respect du vocabulaire et les contraintes d'un langage, et peut être facilement implémentable par l'utilisation d'outils architecturaux appropriés.

Le quatrième et dernier objectif de l'approche centrée style est de faciliter l'évolution du langage et de l'environnement associé. L'étude de cas a montré que la modification de contraintes formelles dans le style (et dans l'architecture de l'environnement de développement) est facile et rapide. Toutefois, la modification du code correspondant dans l'outil SEAM est beaucoup plus délicate. Ce point souligne encore une fois l'intérêt d'utiliser un outil de générant le code de l'environnement de développement à partir de son architecture. L'environnement est alors automatiquement mis à jour lorsqu'une contrainte formelle du style est modifiée. Dans ce cas, l'approche centrée style faciliterait non seulement l'évolution du langage, mais aussi celle de l'environnement associé.

VIII. 4 Supervision de l'évolution des architectures

Cette troisième étude de cas a pour but d'indiquer l'utilité de fournir un support pour l'évolution des architectures, et comment la mise en place d'un système de supervision de ces architectures permet de s'assurer que leur support de développement (SEAM) s'adapte aux attentes des utilisateurs. La section expose la conception de cette étude, les données collectées, ainsi que l'analyse des résultats obtenus.

VIII. 4. 1 Conception de l'étude de cas

L'approche proposée dans cette thèse permet l'évolution des architectures, la capture de ces évolutions, la sélection de celles qui doivent être considérées pour adapter le style, ainsi que la mise à jour du style et de SEAM. Pour déterminer l'influence de cette approche dans la satisfaction des utilisateurs de SEAM, il est nécessaire de considérer les réponses aux questions suivantes :

1. Quel est le gain obtenu entre le premier outil de développement et l'environnement SEAM contrôlé par les styles ?
2. Est-ce que la possibilité de facilement ajouter/supprimer/modifier des contraintes améliore quelque chose ?
3. Est-ce que l'approche est appropriée dans un domaine où l'expérience n'est pas complète ?

Il s'agit d'une étude ne portant que sur un seul cas. L'unité d'analyse de ce cas est l'ensemble des logiciels de supervision du redémarrage des accélérateurs SPS et CPS. Les logiciels de supervision des systèmes cryogéniques ne sont pas pris en compte dans cette étude car elles n'ont pas encore connu d'évolutions significatives. Par ailleurs, il n'a pas été appliqué un grand nombre de modifications aux logiciels de supervision du SPS et du CPS depuis que la version complète de SEAM est en place. Ceci s'explique par le fait que SEAM est construit à partir d'une version du style obtenue après de nombreuses évolutions des logiciels. Il s'agit d'un style plus mature et plus abouti que l'original, il est donc prévisible que les modifications seront de moins en moins fréquemment requises. Ainsi, même si quelques modifications des logiciels de supervision ont été appliquées depuis que SEAM est en place, elles n'ont pas impliqué la modification du style et de l'outil. Les données collectées et analysées correspondent principalement à la période antérieure à la mise en service de SEAM. Les besoins qui ont évolués durant cette période ont impliqué des modifications des logiciels qui ont donc été implémentées sans SEAM mais sous PVSS, ainsi que des mises à jour du style formel en phase de développement.

Les données collectées pour cette étude de cas sont les suivantes :

- les informations relatives aux nouveaux besoins apparus progressivement concernant les logiciels de supervision (permettent de répondre à la question 3) ;
- les informations relatives à l'implémentation de ces nouveaux besoins sous PVSS (permettent de répondre aux questions 1 et 2) ;
- les informations relatives à la prise en compte de ces nouveaux besoins (permettent de répondre aux questions 1 et 2).

Ces données proviennent de plusieurs sources :

- l'archivage des différentes versions des logiciels de supervision dans un outil de type CVS (Concurrent Version File) ;
- l'archivage des modifications appliquées au style ;
- la base de données de SEAM et l'archivage des versions de SEAM ;
- logiciels de supervision obtenus et avis de leurs utilisateurs.

VIII. 4. 2 Collecte des données

L'archivage des différentes versions des prototypes de logiciels a permis de mettre en évidence une cinquantaine de modifications réparties sur une année d'opération. Ces modifications ont été appliquées sur l'IHM globale, sur les IHMs spécifiques, et sur les blocs affichant l'état des systèmes.

Toutes ces modifications ont été implémentées sous PVSS et sont réparties de la façon suivante :

- 38% de modifications graphiques effectuées sans codage par les opérateurs des salles de contrôle ;
- 62% de modifications du code (correction de bugs : 12% ; ajout de fonctionnalités : 8% ; modification de l'instanciation des paramètres : 4%, modification de structure : 18%, modification du traitement : 20% …) effectuées par un programmeur.

Le système d'archivage des versions des logiciels de supervision permet de déterminer le temps qui a été nécessaire pour effectuer chaque modification. Ainsi la durée moyenne de l'implémentation des modifications est de deux heures.

En outre, il a été souvent nécessaire (environ 50% des cas) d'appliquer la même modification à plusieurs logiciels de supervision ou plusieurs blocs graphiques. Ceci a été notamment le cas pour les réorganisations graphiques, les modifications de polices de caractères, les modifications de logique d'affichage des états, et les modifications des couches d'affichage.

Ces modifications ont été analysées « manuellement » pour déterminer si elles devaient être prises en compte au niveau du style GTPM qui servirait de base au développement de SEAM. Ces analyses ont été difficiles et coûteuses en terme de temps de travail (plusieurs heures). Celles-ci ont souvent dû être menées par plusieurs membres du projet et ont pris en compte plusieurs paramètres tels que la nature et la maturité des modifications, le nombre de logiciels/blocs concerné(e)s, ... Ainsi, il a par exemple été choisi d'intégrer au style, par l'ajout de contraintes, les modifications ayant eu pour but d'uniformiser les IHMs (uniformisation des polices de caractères, des titres des IHMs, des aspects graphiques). La moyenne du temps de travail nécessaire à la formalisation de telles contraintes a été estimée à 30 minutes.

Depuis la mise en place de SEAM, les logiciels et les blocs ont connu une dizaine de modifications. Aucune d'entre elles n'a nécessité de codage, et n'a donc du être appliquée par un programmeur. La durée moyenne de l'implémentation des modifications via SEAM a été de 20 minutes.

Les entretiens effectués avec les différents membres du projet (développeurs et utilisateurs) ont indiqué :

- qu'avant la mise en place de SEAM : le processus de modification des logiciels était complexe, les modifications étaient trop souvent nécessaires, et il n'était pas possible de s'assurer qu'un problème découvert sur un logiciel ne pouvait pas, ou ne pourrait pas, être répété sur une autre ;
- qu'après la mise en place de SEAM : le processus de modification des logiciels de supervision s'était simplifié et la fréquence de ces modifications était moindre.

VIII. 4. 3 Analyse de données et conclusions

La première question de cette étude de cas consiste à mesurer si l'outil SEAM permet l'évolution des logiciels de supervision, et si lui-même est évolutif. Afin d'évaluer l'intérêt de cet outil, il est intéressant de considérer les modifications appliquées aux prototypes via PVSS. Les données collectées indiquent que celles-ci ont été nombreuses et coûteuses. En effet, la majeure partie d'entre elles a nécessité l'intervention de programmeurs pour modifier le code des logiciels de supervision. Par ailleurs, il n'était possible d'appliquer des évolutions qu'aux logiciels, et non de l'environnement de développement. En effet, l'outil utilisé ne permettait pas d'utiliser un modèle spécifiant des contraintes de développement propre à un domaine d'application particulier. Ainsi, il n'était pas possible de spécifier qu'une modification était valable pour tous les blocs, ou tous les logiciels de supervision. De même il n'était pas possible de garantir que les logiciels développés dans le futur tireraient profit de l'expérience acquise. Les informations collectées au sujet de SEAM, notamment les avis des différents utilisateurs, indiquent que la modification des logiciels de supervision est maintenant plus simple (pas de codage) et plus rapide. Par ailleurs, le fait que SEAM soit construit à partir d'un style permet de tirer profit de l'expérience acquise lors des modifications de certains logiciels. En effet, si une modification de logiciel de supervision souligne le besoin de rendre plus ou moins contraignant l'environnement de développement, une mise du style va permettre de le faire, et les logiciels de supervision développés dans le futur à partir de celui-ci seront de meilleure qualité. L'apparition d'un nouveau besoin n'implique plus la modification de plusieurs logiciels, mais seulement une modification au niveau du style. Lorsque les modifications des logiciels prototypes ont été appliquées, il a été nécessaire de les analyser pour déterminer si elles devaient entraîner des mises à jour du style. Le temps nécessaire à cette analyse s'est révélé supérieur à celui dédié à l'implémentation des modifications du style. Ainsi, même si des données significatives ne sont pas encore disponibles, il est clair que la possibilité offerte par SEAM de superviser l'évolution des logiciels de supervision, de l'analyser, et de fournir une aide à la décision, est un moyen efficace pour réduire le temps nécessaire à la mise à jour du style. Si l'on considère tous ces aspects, on peut considérer que SEAM apporte un réel gain par rapport au premier outil de développement, ce qui répond à la première question de cette étude de cas.

La deuxième question de l'étude de cas consiste à déterminer si la possibilité de facilement ajouter/supprimer/modifier des contraintes améliore le processus de développement. Les données collectées indiquent effectivement que les mises à jour du style, et leur intégration dans l'architecture de l'environnement de développement, ne demandent que peu de temps de travail, ce qui constitue un des atouts de l'approche.

L'architecture mise à jour est ensuite utilisée pour implémenter les modifications de l'environnement de développement, ce qui peut s'avérer être une étape coûteuse lorsqu'elle est réalisée manuellement, mais qui peut être facilitée par génération de code. Ainsi le deuxième objectif de l'approche étudiée est atteint.

Le troisième objectif de l'étude de cas était de déterminer si l'approche de supervision de l'évolution des logiciels de supervision et de mise à jour du style est appropriée dans un domaine où l'expérience n'est pas complète. Les données analysées indiquent que, dans le cas du projet GTPM, l'expérience n'était pas suffisante pour mettre en place un outil permettant le développement de logiciels de supervision idéaux. De nombreux problèmes sont apparus, de nouveaux besoins ont été exprimés, et il a été nécessaire d'améliorer le style et l'environnement de développement en fonction. L'intérêt de la supervision de l'évolution des logiciels de supervision est apparu dans ce contexte. En effet, l'analyse des nombreuses modifications appliquées aux prototypes de logiciels a servi de base à la mise à jour progressive du style GTPM. Ceci a permis de construire l'environnement SEAM à partir d'un style GTPM relativement mature. Le résultat observé est que l'exploitation du style via SEAM a entraîné la réduction de la fréquence et du coût des modifications appliquées aux logiciels de supervision. Ceci indique que l'utilisation des résultats de l'analyse des évolutions des prototypes de logiciels pour mettre à jour le style a permis de capturer efficacement l'expérience acquise. Ainsi, on peut en conclure que l'approche s'est avérée appropriée.

Chapitre IX Conclusions et perspectives

Chapitre IX : Conclusions et perspectives

Le problème traité par cette thèse concerne la définition d'un modèle de développement spécifique à un domaine d'application particulier, ainsi que son exploitation et son évolution dans un environnement logiciel dédié. Le moyen choisi pour aboutir à ce résultat a été l'utilisation de techniques architecturales incluant notamment la définition de styles. L'utilisation d'un style architectural permet de spécifier les caractéristiques propres à une famille d'applications logicielles et de produire ces applications en garantissant le respect des caractéristiques définies au niveau du style. Dans ce contexte, cette thèse avait pour but de :

- proposer une nouvelle approche de conception et de production de logiciels de supervision basée sur l'utilisation et l'exploitation des styles architecturaux ;
- proposer un processus inductif permettant la définition de style architectural à partir d'applications prototypes, et l'évolution du style en fonction de l'évolution des besoins concernant les applications construites à partir de celui-ci ;
- implémenter un environnement dédié au développement de logiciels, propres à des domaines spécifiques, respectant les contraintes d'un style architectural.

Ce chapitre de conclusion présente le bilan des travaux réalisés dans le cadre de la thèse. La section IX.1 résume l'approche de recherche suivie. La section IX.2 s'intéresse aux approches proposées et suivies pour définir un style architectural propre à un domaine d'application particulier, et pour l'utiliser dans l'implémentation d'un environnement de développement. La section IX.3 traite de l'applicabilité des travaux présentés dans cette thèse dans le cadre de l'ingénierie logicielle, puis la section IX.4 positionne les travaux de la thèse par rapport à l'état de l'art étudié, et indique dans quelle mesure ceux-ci répondent aux objectifs fixés. Enfin, la section IX.5 présente les perspectives ouvertes par ces travaux.

IX. 1 Approche de recherche

Un des objectifs de cette thèse était de définir un processus permettant d'obtenir un style architectural inductivement à partir d'une expertise partiellement existante, notamment sous la forme d'un ensemble d'applications prototypes. Un autre objectif des travaux présentés était de spécifier une approche de production d'environnement de développement permettant d'exploiter le style. Le dernier point consistait à proposer et à mettre en place une technique permettant de faire évoluer le style en fonction de l'évolution des besoins. Dans ce contexte, la démarche de recherche a principalement été constituée des étapes suivantes :

- étude de la problématique (domaine de recherche et domaine d'application), et prototypage ;

- proposition d'une approche de solution ;
- mise en oeuvre de l'approche (définition inductive du style, production de l'environnement de développement) ;
- tests et validation.

La première étape de l'approche a inclus la spécification des besoins des prototypes, leur implémentation, et leur validation. Cette phase a été effectuée en étroite collaboration avec tous les membres du projet relatif au domaine d'application (supervision du redémarrage des accélérateurs de particules du CERN). Elle a en outre permis d'acquérir la connaissance du domaine d'application ainsi que d'obtenir un aperçu de comment celui-ci pouvait évoluer. Parallèlement à cette phase de prototypage, il a été nécessaire d'étudier l'état de l'art dans le domaine du développement orienté-architecture.

La deuxième phase de la démarche de recherche a consisté à proposer une approche pour formaliser l'expérience acquise durant la phase de prototypage au moyen d'un style architectural, ainsi qu'une approche pour exploiter ce style, et une pour le faire évoluer.

La troisième étape a consisté à mettre en œuvre l'approche proposée. La connaissance du domaine d'application ainsi que du domaine de la description architecturale acquise lors de la première phase a permis de définir et formaliser une première version du style exprimant les propriétés que devaient respecter les applications à produire. Ce style a été raffiné avant d'être utilisé pour construire un environnement de développement.

L'étape de tests et de validation a commencé par l'utilisation de l'environnement de développement construit à partir du style dans le cadre de plusieurs expérimentations. Cette phase a été réalisée en collaboration avec les spécialistes du domaine d'application. Des familles de logiciels ont été développées pour superviser le redémarrage de plusieurs accélérateurs de particules du CERN. L'environnement de développement et les logiciels qu'il permet de produire ont ainsi été testés et validés, ce qui a permis de valider l'approche de définition et d'exploitation du style architectural. Par ailleurs, cette étape a permis de valider l'approche concernant l'évolution du style en fonction de l'évolution des besoins.

IX. 2 Bilan sur la définition et l'exploitation du style

IX. 2. 1 La formalisation du style

L'étape de définition du style architectural a pour but d'exprimer formellement les caractéristiques communes des membres d'une famille d'applications. Dans le contexte de cette thèse, l'objectif était de capturer l'expérience acquise lors du développement et de l'utilisation de prototypes de logiciels de supervision. Il a donc été nécessaire de mettre en place et d'utiliser un processus de définition de style inductif. Celui-ci s'est basé sur l'analyse des architectures des logiciels de supervision. L'analyse a permis de mettre en évidence le vocabulaire utilisé par ces architectures et les contraintes qu'elles doivent respecter. Ces contraintes et ce vocabulaire spécifiques au domaine des accélérateurs de particules ont été formalisés dans un style architectural, à l'aide des langages ArchWare et du style ArchWare C&C. Le vocabulaire du style proposé comporte tous les éléments composants les logiciels de supervision à développer,

notamment ceux dédiés à la représentation de l'état d'équipements et de systèmes. Les contraintes du style fixent les règles de construction des logiciels de supervision, au niveau de leur structure et de leur comportement. L'utilisation d'un tel style permet d'assurer que toutes les applications développées à partir de celui-ci satisferont les besoins exprimés par les utilisateurs.

Les langages utilisés se sont montrés suffisamment expressifs pour formaliser l'ensemble du vocabulaire et des contraintes nécessaires. Il est à noter que la définition du style a été rendue relativement aisée par l'utilisation du langage de haut niveau C&C. Néanmoins, la formalisation et l'utilisation d'un tel style supposent une étude préalable des différents aspects du développement formel dans le cadre des architectures logicielles.

IX. 2. 2 Implémentation de l'environnement de développement à partir du style

Le style formalisé devait être utilisé en tant que support pour la création de familles de logiciels de supervision. Dans cet objectif, l'approche proposée et suivie consiste à définir l'architecture d'un outil, dédié au développement de logiciels de supervision, intégrant les contraintes du style. Cette architecture, qui a été formalisée avec ArchWare-ADL, décrit les composants de l'outil ainsi que les relations entre ceux-ci. Ces composants sont principalement des interfaces permettant aux utilisateurs de spécifier toutes les informations nécessaires à la production des logiciels de supervision, ainsi qu'une base de données permettant de stocker ces informations. L'intégration du style dans l'architecture de l'environnement de développement consiste à répartir ses contraintes dans les différents composants de l'architecture. Plus précisément, les composants de l'architecture de l'environnement intègrent les contraintes du style, de façon à ce que les architectures de logiciels qu'ils permettent de produire vérifient, par construction, toutes ces contraintes. Pour des raisons de flexibilité, certaines contraintes formalisées par le style ne sont pas intégrées dans les composants intervenant dans le développement des logiciels, mais sont vérifiées à posteriori, par un vérificateur de conformité, qui constitue un composant supplémentaire de l'architecture.

L'environnement de développement est implémenté à partir de son architecture ainsi définie, et permet la production de logiciels de supervision respectant les contraintes du style.

IX. 2. 3 Evolution de l'environnement de développement par l'évolution du style

Cette thèse a proposé une approche permettant de faire co-évoluer les architectures et les styles dans un processus de développement architectural. La motivation de ce travail vient du fait que la définition inductive d'un style à partir d'exemples d'architectures implique nécessairement que celui-ci n'est pas « idéal », et qu'il sera donc amené à évoluer. L'approche présentée considère que les nouveaux besoins des utilisateurs seront implémentés au niveau des logiciels de supervision, et qu'il est donc nécessaire de superviser l'évolution de ceux-ci pour pouvoir les répercuter sous certaines conditions au

niveau du style architectural. Le processus de supervision proposé prend en compte les tendances d'évolution des logiciels de supervision, et utilise des critères décisionnels établis à partir de l'expérience, pour indiquer à l'utilisateur si le style doit être mis à jour, et de quelle manière.

Cette approche permet de tirer profit de l'expérience acquise lors des modifications de certains logiciels de supervision. En effet, si une modification souligne le besoin de rendre plus ou moins contraignant le style, celui-ci sera mis à jour, et les logiciels de supervision développés dans le futur satisferont plus pleinement les utilisateurs. En outre, la possibilité offerte par l'environnement de développement de superviser l'évolution des logiciels de supervision, de l'analyser, et de fournir une aide à la décision, est un moyen efficace pour réduire le temps nécessaire à la mise à jour du style.

Le style étant intégré à l'architecture de l'environnement de développement, la mise à jour des contraintes du style entraîne la mise à jour de l'architecture de l'environnement, et donc, la mise à jour de l'environnement lui-même. L'approche proposée dans cette thèse a été validée par l'exécution manuelle de ces étapes. Cette approche pourrait être automatisée par la mise en place d'un outil facilitant l'intégration du style dans l'architecture de l'environnement, et d'un autre générant de l'environnement à partir de l'architecture obtenue.

IX. 2. 4 Validation de l'approche : SEAM

Les approches et processus de développement proposés dans cette thèse ont été validés dans le cadre de la mise en place d'un environnement de développement, SEAM, permettant de guider et d'optimiser la production de logiciels de supervision d'accélérateurs de particules. A travers cet environnement, il a été possible de répondre aux besoins exprimés par les développeurs de logiciels de supervision. En effet, l'objectif de l'outil SEAM était de proposer des solutions à certaines limites rencontrées dans l'utilisation des méthodes de développement d'applications de supervision traditionnelles. Pour cela il devait notamment :

- autoriser le développement graphique des logiciels de supervision (développeurs non-informaticiens) ;
- générer des familles de logiciels de supervision uniformisés ;
- contraindre la construction des logiciels de supervision ;
- capturer le savoir-faire du domaine et réutiliser l'expérience acquise ;
- gérer l'évolution des logiciels de supervision et de l'environnement de développement.

SEAM a été développé à partir de du style GTPM (obtenu inductivement à partir des logiciels prototypes) intégré et dans une architecture formelle. L'outil SEAM, actuellement utilisé dans les différentes salles de contrôle du CERN, fournit des fonctionnalités permettant de spécifier les informations concernant les logiciels de supervision. Cette spécification se fait au moyen de deux interfaces : une interface web dynamique accédant à des tables de bases de données Oracle, et un modélisateur graphique. SEAM permet aux spécialistes de la supervision des accélérateurs de particules de spécifier la totalité des informations relatives aux logiciels de supervision. L'outil utilise par la suite ces informations pour implémenter les logiciels. Ceci

représente un gain important par rapport au processus de développement utilisé lors de la phase de prototypage. L'outil supprime la phase de codage, et donc l'intervention des programmeurs, en générant automatiquement le code des logiciels de supervision à partir de leur spécification. De plus, l'architecture des différents modules de SEAM intégrant les contraintes du style assure que les logiciels de supervision spécifiés (puis implémentés) satisfont aux besoins exprimés par l'ensemble de l'équipe du projet, et formalisés dans le style GTPM. L'expérience a montré que l'utilisation de l'outil SEAM a permis d'obtenir des IHMs satisfaisantes (complètes, uniformisées, …) beaucoup plus rapidement que lors de la phase de prototypage. Par ailleurs, le fait de réduire le nombre d'intervenants dans le développement des logiciels de supervision, de guider les utilisateurs dans leur travail, et d'automatiser certaines étapes, simplifie l'implémentation de modifications sur les logiciels et/ou leurs éléments.

IX. 3 Applicabilité des travaux

Les travaux présentés dans cette thèse ont abouti, d'une part, à la mise en place d'une approche de développement, et, d'autre part, à la définition d'un style architectural et d'un environnement de développement. Les deux paragraphes suivants exposent l'applicabilité de ces résultats.

IX. 3. 1 Applicabilité de l'approche

Les approches et processus de développement définis et utilisés dans le cadre de cette thèse ne sont pas spécifiques à la production d'applications de supervision d'accélérateurs de particules, mais sont applicables dans de nombreux autres domaines.

L'étape de définition d'un style, ou modèle, est applicable et bénéfique dans tous les domaines où une expérience existe sous forme informelle. En effet, cette étape permet d'identifier, de regrouper, d'analyser et de formaliser les propriétés communes d'applications existantes. Cette thèse a montré que le style constitue alors une base efficace pour produire des familles d'applications respectant ces propriétés. Utiliser l'approche définie dans cette thèse est un moyen d'améliorer le processus de développement de familles de logiciels et d'en réduire le coût. Néanmoins, celle-ci n'est applicable que sous la condition de posséder une connaissance approfondie du domaine d'application, ainsi qu'une maîtrise du développement formel en général, et de l'ADL choisi en particulier. En contrepartie, cette étape permet de réduire significativement le temps de développement de logiciels qui respectent les propriétés voulues.

Le deuxième aspect des travaux présentés ayant un large champ d'application est l'approche permettant de construire un environnement de développement de logiciels à partir d'un style formel. L'environnement ainsi obtenu permet de faciliter le développement de logiciels satisfaisant les besoins, en le rendant accessible à des utilisateurs spécialistes du domaine d'application. Dans le processus classique de développement informatique, les clients, les développeurs, et les utilisateurs, sont des personnes différentes, ce qui induit des problèmes de communication et par conséquent des différences entre le résultat obtenu et celui attendu. L'approche de cette thèse permet aux utilisateurs de développer eux-mêmes leurs logiciels, tout en les guidant et en

réduisant l'espace de développement par l'utilisation du style. Ceci réduit le temps de développement, améliore le résultat, et permet ainsi de rentabiliser le travail de définition du style. Cette approche peut-être mise en oeuvre dans tout domaine d'application, et nécessite l'intervention d'un architecte logiciel qui définit l'architecture de l'environnement de développement et qui lui intègre les contraintes du style.

Une des contributions de cette thèse est la définition et la mise en place d'une approche permettant de faire évoluer le style en fonction de l'évolution des besoins et de l'évolution des architectures des logiciels produits. La possibilité de faire évoluer un style, ou modèle de construction de logiciels, possède un grand intérêt, et ce, dans tous les domaines de l'ingénierie logicielle. Appliquer l'approche spécifiée dans cette thèse permet, non seulement d'améliorer progressivement le style, mais aussi de rentabiliser le travail effectué lors de sa définition initiale. Toutefois, ceci requiert de connaître suffisamment le domaine d'application pour déterminer ses aspects qui sont susceptible d'évoluer, de façon à construire le style et son environnement d'exploitation en fonction. Par ailleurs, l'approche permettant de mettre à jour le style s'exécute partiellement manuellement, il serait donc intéressant de l'outiller de façon à augmenter son applicabilité (Cf. section IX.5.1).

IX. 3 .2 Applicabilité du style et de l'environnement

Le style architectural défini dans le cadre de cette thèse (style GTPM) correspond à un contexte spécifique, il n'est donc directement réutilisable que dans des domaines connexes. Toutefois, le champ d'application est plus large que celui pour lequel il a été défini initialement. En effet, le style GTPM a dans un premier temps été utilisé pour construire les logiciels de supervision de redémarrage de deux accélérateurs de particules du CERN, mais il a pu par la suite être appliqué dans le cadre de la supervision d'autres processus (ex : systèmes de cryogénie). Le style GTPM peut être utilisé sans modification majeure pour la production de logiciels de supervision de tout processus, dans la mesure où il s'agit d'une supervision de séquence de redémarrage.

L'environnement de développement mis en place dans le cadre de cette thèse (SEAM) est étroitement lié au style GTPM, il possède donc le même champ d'application que celui-ci. Il peut être réutilisé pour le développement de logiciel de supervision de redémarrage de tout processus. Il serait pour cela uniquement nécessaire de lui apporter quelques modifications mineures, notamment au niveau de ses interfaces avec son environnement (bases de données sources, système de supervision).

IX. 4 Positionnement de la thèse par rapport à l'état de l'art

Le chapitre 3 de cette thèse a présenté l'état de l'art dans le domaine des méthodes et des outils de développement de logiciels de supervision, ainsi que dans celui de la production de familles d'applications, notamment par l'utilisation de techniques de développement architectural.

L'étude a montré que les outils traditionnellement utilisés pour développer des logiciels de supervision (ex : [PVSS][PcVue][SL-GMS]) possèdent de nombreux aspects intéressants mais sont néanmoins limités dans certains domaines. En effet, ceux-ci ne

permettent que partiellement le développement de logiciels par interface graphique et la spécification et le respect de guides de développement. De plus, ils sont peu flexibles et ne peuvent ni s'adapter facilement à un domaine d'application particulier, ni garantir la satisfaction de propriétés complexes. En ce qui concerne les guides de développement et modèles existants (ex : [ISO 2000][Griffiths et Pemberton 2000]), ceux-ci fournissent aux développeurs des conseils leur permettant de savoir ce qu'ils doivent faire et ce qu'ils doivent éviter [Borchers et Thomas 2001][Welie et al. 2000]. Toutefois, ceux-ci sont très nombreux ce qui rend difficile le choix de l'un d'entre eux dans une situation spécifique. De plus, ils sont souvent trop génériques pour pouvoir répondre à des besoins propres à des domaines d'applications particuliers. Par ailleurs, il est à noter que les travaux effectués dans le domaine des patrons ne traitent pas de l'automatisation de leur exploitation par des outils de développement. Sous cet aspect, la contribution de la thèse a été la mise en place d'un outil, flexible et évolutif, permettant la spécification entièrement graphique de logiciels de supervision spécifiques à d'un domaine particulier, respectant les propriétés définies dans un style architectural (guide de développement), et générant leur code.

Les techniques de développement architectural étudiées dans l'état de l'art permettent, notamment par l'utilisation d'ADLs supportant les styles [Leymonerie et al. 2002][Medvidovic et Taylor 1997], la définition et l'utilisation de langages spécifiques à des domaines particuliers [DeLine 1996]. L'utilisation d'un style permet de spécifier et de vérifier des propriétés complexes, ce qui constitue un guide de développement efficace et garanti la satisfaction des besoins des utilisateurs. Le processus classique de définition et d'utilisation des styles architecturaux suppose que l'expertise du domaine d'application est complète, et que le style peut être directement entièrement défini et utilisé pour produire des applications satisfaisant des besoins clairement établis (ex : [Van Deursen et Klint 2002][Jarzabek et al. 2001]). Toutefois, dans de nombreux cas, comme celui traité dans cette thèse, une expérience du domaine est disponible (par exemple sous la forme d'applications prototypes), mais celle-ci est incomplète, et les besoins des utilisateurs ne sont pas définitifs mais sont susceptibles d'évoluer fréquemment. Une des contributions de cette thèse est la mise en place d'un processus de définition inductif de style à partir d'applications prototypes, prévoyant l'évolution des contraintes qu'il formalise en fonction de l'évolution des besoins. Concernant l'évolution, certains travaux (ex : [Zenger 2002]) ont souligné que les architectures devaient être conçues de façon à pouvoir évoluer, d'autres (ex : [Jazayeri 2002]) vont plus loin en proposant des métriques et des outils pour analyser les évolutions des architectures, et, par ailleurs, il existe des environnements gérant l'évolution de lignes de produits [Akash et al. 2003]. Toutefois, aucune technique n'a été mise en place pour faire évoluer un style architectural en fonction de l'évolution des architectures qui suivent ce style. La contribution de cette thèse dans ce domaine a été de proposer une approche de supervision et d'analyse de l'évolution des architectures construites à partir d'un style. Cette approche, utilisée pour raffiner et améliorer le style, s'est révélée efficace dans un cas comme celui de cette thèse, où le domaine d'application n'est pas parfaitement connu dès le lancement du projet et se trouve soumis à de fréquentes évolutions.

De nombreux travaux traitent de la définition et de la formalisation des styles architecturaux (ex : [Monroe et Garlan 1996][Shaw et Clements 1997][Bass et al. 1997] [Mehta et Medvidovic 2004]) mais l'état de l'art concernant l'exploitation de ces styles

dans des environnements de développement est moins riche. Certains travaux (ex : [Garlan et al. 1994][AESOP][AcmeStudio][ArchWare][Mehta et al. 2004]) proposent des mécanismes permettant de paramétrer des outils de développement génériques au moyen des styles, ce qui permet de les spécialiser à des domaines d'applications particuliers, mais il n'existe pas d'approche permettant de produire des environnements de développement à partir de styles. Ainsi, une autre contribution de cette thèse est la définition d'une approche permettant d'intégrer le style architectural spécifiant les contraintes que les logiciels de supervision à produire doivent respecter, à l'architecture formelle d'un environnement de développement dédié à la production de ces logiciels. L'architecture obtenue intègre les contraintes et le vocabulaire du style, et sert de base à l'implémentation d'un environnement de développement spécifique utilisable par les spécialistes du domaine d'application.

IX. 5 Perspectives

Les perspectives ouvertes par les travaux présentés dans cette thèse peuvent se grouper selon trois axes qui font l'objet des paragraphes suivants : la continuation du travail proprement dit, les nouvelles applications, et les thèmes de recherche ouverts.

IX.54. 1 Continuation du travail sur l'environnement de développement

L'environnement de développement présenté dans cette thèse est obtenu par implémentation d'une architecture formelle intégrant les contraintes et le vocabulaire défini dans un style architectural. Le processus suivi dans ce contexte pourrait être facilité par la mise en place de deux outils. Un premier outil pourrait être développé pour intégrer automatiquement les contraintes du style dans l'architecture de l'environnement de développement. La difficulté consiste à déterminer quelles contraintes du style doivent être intégrées dans quels composants de l'architecture de l'environnement de développement. Les critères intervenant dans cette décision peuvent notamment concerner la stabilité des contraintes (intégrées dans des composants plus ou moins évolutifs), et l'ergonomie des différentes interfaces de saisie des données devant satisfaire les contraintes (intégrées dans des composants de saisie graphique, textuelle, …). Un deuxième point consiste à proposer un générateur de code permettant d'obtenir automatiquement l'environnement de développement à partir de son architecture incluant le style propre au domaine d'application.

La mise en place et l'utilisation de ces outils apporteraient un gain notable au niveau de l'évolution de l'environnement. En effet, l'approche proposée dans cette thèse supporte l'évolution automatique du style architectural, mais les évolutions de l'environnement de développement sont appliquées manuellement. La mise en place des outils précédemment mentionnés permettrait de mettre à jour automatiquement l'architecture de l'environnement de développement en fonction des mises à jour de style, et de générer automatiquement les mises à jour de l'environnement en fonction des mises à jour de son architecture. Concernant l'évolution du style architectural, un travail à poursuivre est l'enrichissement de la base de règles du module d'aide à la décision, qui permet de déterminer quand et comment le style doit évoluer en fonction de quelles

évolutions des architectures de logiciels de supervision. Par ailleurs, l'approche proposée suppose que l'expertise acquise au fur et à mesure des développements successifs de logiciels de supervision entraîne l'évolution d'un seul style architectural. Toutefois, on peut imaginer que des développements futurs aboutissent à des logiciels de supervision suffisamment éloignés du style pour qu'il ne soit pas approprié de faire évoluer le style, mais au contraire de créer un autre style. Il serait donc intéressant d'ajouter au module d'aide à la décision des critères permettant de déterminer quand le style doit évoluer, et quand un nouveau style doit être créé.

IX. 5. 2 Nouvelles applications

Le style architectural intégré à l'architecture de l'environnement de développement a été conçu pour guider le développement de logiciels de supervisions du redémarrage d'accélérateurs de particules spécifiques. Ce style va encore évoluer pour être utilisé dans le développement de logiciels de supervision de toutes les expériences physiques du CERN.

Une perspective ouverte par les travaux présentés est la possibilité d'adaptation de l'environnement de développement à d'autres domaines d'application. Cette adaptation est partiellement possible par l'utilisation de nouveaux styles architecturaux. Toutefois, dans le cas de contextes d'exploitation très différents, il est nécessaire de définir de nouvelles architectures de l'environnement de développement. Si plusieurs environnements sont disponibles, il est possible d'appliquer le processus inductif défini dans cette thèse au niveau des logiciels de supervision à celui des environnements de développement. L'extension de l'approche permettrait ainsi d'obtenir inductivement, à partir des architectures d'environnement de développement, un style d'environnement formalisant leurs caractéristiques communes. L'exploitation et l'évolution de ce style suivraient le même processus que celui défini au niveau du style de logiciels de supervision.

IX. 5. 3 Nouveaux thèmes de recherche

Cette thèse a proposé un processus de définition inductive d'un style, et d'un environnement de développement associé, à partir de logiciels existants représentant l'expertise du domaine d'application. Ces logiciels ont été étudiés de façon à faire émerger les contraintes devant être formalisées au niveau du style. Il serait intéressant d'automatiser cette approche en appliquant et en développant des techniques de ré-ingénierie. Certains travaux permettent d'obtenir l'architecture d'une application logicielle à partir de son code. Un thème de recherche intéressant serait de proposer une approche et un outil permettant de définir automatiquement un style à partir d'architectures d'applications possédant des caractéristiques communes. L'intégration d'un tel outil dans le processus présenté dans cette thèse permettrait d'obtenir automatiquement, à partir de logiciels existants, un environnement dédié au développement de logiciels similaires.

Par ailleurs, l'environnement de développement construit dans cette thèse inclut une base de données stockant toutes les informations concernant les architectures des logiciels de supervision à générer. L'intérêt d'une telle base de données est de représenter

explicitement les différents aspects des architectures des logiciels, ce qui autorise l'application de traitements automatiques. Un nouveau thème de recherche serait de généraliser cette approche en stockant l'architecture de l'environnement de développement dans une métabase. Ainsi, les différents composants de l'architecture de l'environnement auraient une représentation explicite dans cette métabase, ce qui permettrait de générer le code de cet environnement de développement (et de ses évolutions) de la même manière que le code des logiciels de supervision est aujourd'hui généré.

Références

[Abd-Allah 1996] A. Abd-Allah, « Composing Heterogeneous Software Architecture, Doctoral Dissertation », Center for Software Engineering, University of Southern California, 1996.

[Abowd et al. 1993] G.Abowd, R. Allen et D. Garlan, « Using Style to Give Meaning to Software Architectures », Proceedings of SIGSOFT'93, Foundations Software Eng., ACM, New York, 1993.

[Abowd et al. 1995] G.Abowd, R. Allen et D. Garlan, « Formalizing style to understand descriptions of software architecture », ACM Transactions on Software Engineering and Methodology 4(4):319-364, octobre 1995.

[AcmeStudio] http://www-2.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html.

[ADL Toolkit] http://www-2.cs.cmu.edu/~acme/adltk/tools.html.

[AESOP] http://www-2.cs.cmu.edu/Groups/able/aesop/aesop_home.html.

[Akash et al . 2003] G. Akash, M. Critchlow, P. Chen, C. Van der Westhuizen et A. van der Hoek, « An Environment for Managing Evolving Product Line Architectures », proceedings of the International Conference on Software Maintenance (ICSM'03), p. 358, septembre 2003.

[Alexander et al. 1977] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel, « A Pattern Language », Oxford University Press, New York, 1977.

[Allen 1995] R. Allen, « Formalism and Informalism in Software Architectural Style: a Case Study », Proceedings of the First International Workshop on Architectures for Software Systems , avril 1995.

[Allen 1996] R. Allen, « HLA: A Standards Effort as Architectural Style », in A. L. Wolf, ed., Proceedings of the Second International Software Architecture Workshop (ISAW-2), pp 130-133, San Francisco, octobre 1996.

[Allen 1997] R. Allen, « A Formal Approach to Software Architecture », PhD thesis, Carnegie Mellon University, 1997.

[Allen et al. 1998] R. Allen, R. Douence et D. Garlan, « Specifying and Analysing Dynamic Software Architectures », Proceedings of 1998 Conference on Fundamental Approaches to Software Engineering, Lisbonne, Portugual, mars 1998.

[Allen et Garlan 1994] R. Allen et D. Garlan, « Formalizing architectural connection », in proceedings of the 16th International Conference on Software Engineering, pages 71-80, Sorrento, Italie, mai 1994.

[Allen et Garlan 1997] R. Allen et D. Garlan, « A formal basis for architectural connection », ACM Transactions on Software Engineering and Methodology, juillet 1997.

[Alloui et al. 2003] I. Alloui, H. Garavel, R. Mateescu et F. Oquendo, « The ArchWare Architecture Analysis Language: Syntax and Semantics », Deliverable D3.1b, ArchWare European RTD Project, IST-2001-32360, janvier 2003.

[Alloui et Oquendo 2003] I. Alloui et F. Oquendo, « The ArchWare Architecture

Description Language: UML Profile for Architecting with ArchWare ADL », Deliverable D1.4b, ArchWare European RTD Project, IST-2001-32360, juin 2003.

[Alloui et Oquendo 2004] I. Alloui et F. Oquendo, « Describing Software-intensive Process Architectures using a UML-based ADL », Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS'04), Porto, Portugal, avril 2004.

[America et al. 2000] P. America, H. Obbink, R, van Ommering, et F, van der Linden, « CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering », Software Product Lines: Experience and Research Directions, Kluwer Academin Publishers, 2000.

[Apple Computer Inc. 1992] Apple Computer Inc, « Macintosh Human Interface Guidelines », Addison-Wesley Publishing Co., 1992.

[ArchWare] http://www.arch-ware.org.

[Arduini et al. 2002a] G. Arduini, C. Arimatea, M. Batz, J.M. Carron de la Morinais, D. Manglunki, K. Priestnall, G. Robin, M. Ruette, P. Sollander, « Towards a Common Monitoring System for the Accelerator and Technical Control Rooms at CERN », CERN, Genève, Suisse, février 2002.

[Arduini et al. 2002b] G. Arduini, C. Arimatea, M. Batz, J.M. Carron de la Morinais, G. Robin, P. Sollander, « An Engineering Method for the Monitoring of Accelerator Equipment and Technical Infrastructure: The SPS Experience », CERN, Genève, Suisse, 2002.

[Barbeau et al. 2002] H. Barbeau, R. Martini, O. Ratcliffe, S. Roy, J. Serrano, P. Sollander, J. Stowisek, « GTPM User Guide », CERN, Genève, Suisse, 2002.

[Bass et al. 1997] L. Bass, P. Clements, and R. Kazman, « Software Architecture in Practice », Addison-Wesley, 1997.

[Batory 1998] D. Batory, « Product-Line Architectures », Proceedings of the International Conference on Software Engineering (ICSE 1998), 1998.

[Batory et Geraci 1997] D. Batory et B. Geraci, « Composition Validation and Subjectivity in GenVoca Generators », IEEE Transactions on Software Engineering, février 1997.

[Bayle 1998] E. Bayle, « Putting it All Together: Towards a Pattern Language for Interaction Design », SIGCHI Bulletin, vol. 30, no. 1 pp. 17-24, 1998.

[Beck 1999] K. Beck, « Extreme Programming Explained: Embrace Change », Addison-Wesley, octobre 1999.

[Bell 2002] D.R. Bell, « The Hidden Cost of Downtime: Strategis for Improving Return of Assets », juillet 2002, http://www.smartsignal.com/upload/whitepaper/Hidden_Cost.pdf.

[Bergey et al. 2003] J. Bergey, S. Cohen, M. Fisher, L. Jones, L. Northrop, W. O'Brian, « Fifth DoD Product Line Practice Workshop Report », Technical Report CMU/SEI-2003-TR-007, ESC-TR-2003-007, juin 2003.

[Binns et al. 1996] P. Binns, M. Engelhart, M. Jackson et S. Vestal, « Domain-Specific Software Architectures for Guidance, Navigation, and Control », International Journal of Software Engineering and Knowledge Engineering, 1996.

[Binns et Vestal 1993] P. Binns et S. Vestal, « Formal real-time architecture specification and analysis », 10th IEEE Workshop on Real-Time Operating Systems and Software, mai 1993.

[Blazer 1997] R. Blazer, « Instrumenting, Monitoring, & Debugging Software Architectures », http://citeseer.ist.psu.edu/balzer97instrumenting.html, 1997.

[Boasson 1995] M. Boasson, « The Artistry of Software Architecture », Guest editor's introduction, IEEE Software, 1995.

[Boehm et al. 1994] B. Boehm, P. Bose, E. Horowitz et M. J. Lee, « Software requirements negotiation and renegociation aids: A theory-W based spiral approach », in proceedings of the 17th International Conference on Software Engineering, 1994.

[Borchers et Thomas 2001] J. O. Borchers, J. C. Thomas, « Patterns: What's In It For HCI? ». in proceedings of the CHI 2001 Conference on Human Factors in Computing Systems, ACM Press, New York, 2001.

[Bosch 1999] J. Bosch, « Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study », Software Architecture, Kluwer Academic Publishers, 1999.

[Brown 1988] C. M. Brown, « Human-Computer Interface Design Guidelines », Ablex Publishing Corporation: Norwood, 1988.

[Brown et al. 1998] W. J. Brown, R. C. Malveau, H. W. McCormick and T. J. Mowbray, « Anti Patterns, Refactoring Software, Architectures and Projects in Crisis », John Wiley, New York, 1998.

[Brown et Cunningham 1989] J. Brown, S. Cunningham, « Programming the User Interface: Principles and Examples », John Wiley and Sons, p. 171, 1989.

[Buschmann et al. 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad et M. Stal, « Pattern Oriented Software Architecture: A System of Patterns », John Wiley & Sons, 1996.

[Catry et al. 2003] A. Catry, D. Champelovier, H. Garavel et R. Mateescu, « Definition of the Architecture Analysis Formalism for Model-Checking », Deliverable D3.3, ArchWare European RTD Project, IST-2001-32360, juin 2003.

[CCC] http://www3.ca.com/Solutions/Product.asp?ID=255.

[Chaudet et Oquendo 2001] C. Chaudet et F. Oquendo, « π-SPACE: Modeling Evolvable Distributed Software Architectures », Proceedings of International Conference PDPTA'01, Las Vegas, juin 2001.

[Ciancarini et Mascolo 1996] P. Ciancarini, C. Mascolo, « Analysing and Refining an Architectural Style », 1996.

[Cimpan et al. 2002] S. Cimpan, F. Oquendo, D. Balasubramaniam, G. Kirby et R. Morrison, « The ArchWare Architecture Description Language: Textual Concrete Syntax », Deliverable D1.2b, ArchWare European RTD Project, IST-2001-32360, décembre 2002.

[Cimpan et al. 2003] S. Cimpan, F. Leymonerie et F. Oquendo, « The ArchWare Foundation Styles Library », Report R1.3-1, ArchWare European RTD Project, IST-2001-32360, juin 2003.

[ClearCase] http://www-306.ibm.com/software/awdtools/clearcase.

[Clements 2002] P. Clements et L. Northrop, « Software Product Lines: Practices and Patterns », Reading, MA: Addison-Wesley, 2002.

[Clements et al. 1995] P. Clements, L. Bass, R. Kazman et G. Abowd, « Predicting software quality by architecture-level evaluation », in proceedings of the Fifth International Conference on Software Quality, Austin, Texas, octobre 1995.

[Coglianese et Szymanski 1993] L. Coglianese et R. Szymanski, « DSSA-ADAGE: An Environment for Architecture-based Avionics Development », in proceedings of AGARD'93, mai 1993.

[Cohen 2002] S. Cohen, « Product Line State of the Practice Report », Technical Note, CMU/SEI-2002-TN-017, septembre 2002.

[Coram et Lee 1996] T. Coram, J. Lee, « Experiences: A Pattern Language for User Interface Design ».http://www.maplefish.com/todd/papers/experiences/Experiences.html, 1996.

[DeBaud et Schmid 1999] J-M. DeBaud et K, Schmid, « A Systematic Appraoch to Derive Scope of Software Product Lines », International Conference on Software Engineering, 1999.

[DeLine 1996] R. DeLine, « Towards User-Defined Elements Types and Architectural Styles », Proceedings of the Second International Software Architecture Workshop, pp. 47-49, San Francisco, 1996.

[Doors] http://www.telelogic.com/products/doorsers/doors/index.cfm.

[DowntimeCentral] « TDC: Real Examples of Downtime Cost », http://www.downtimecentral.com/Examples.htm.

[Fromme 1989] B. Fromme, « HP Encapsulator: Bridging the Generation Gap », Technical Report SESD-89-26, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, novembre 1989.

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson and J. Vlissides, « Design Patterns: Elements of Reusable Object-Oriented Software », Addison-Wesley, 1995.

[Garavel et Mateescu 2003] H. Garavel et R. Mateescu, « Enhanced Model-Checker for Architecture Analysis », Deliverable D3.8, ArchWare European RTD Project, IST-2001-32360, janvier 2003.

[Garlan 1995] D. Garlan, « What is style », Proc. Of the First International Workshop Software Architecture, 1995.

[Garlan 2000] D. Garlan, « Software Architecture: a Road Map », Proc. of the conference on The future of Software engineering, mai 2000.

[Garlan et al. 1992] D. Garlan, M. Shaw, C. Okasaki, C. Scott, and R. Swonger, « Experiences with a Course on Architectures for Software Systems », Proceedings of the 6th SEI Conference on Software Engineering Education, 1992.

[Garlan et al. 1994] D. Garlan, R. Allen et J. Ockerbloom, « Exploiting Style in Architectural Design Environments », Proceedings of the ACM SIGSOFT'94 Symposium on Foundations of Software Engineering, New Orleans, décembre 1994.

[Garlan et al. 1997] D. Garlan, R. Monroe et D. Wile, « ACME: an Architectural Descripion Interchange Language », Proceedings of CASCON'97, pp. 169-183, Toronto, novembre 1997.

[Garlan et Shaw 1993] D. Garlan et M. Shaw, « An Introduction to Software Architecture », Advances in Software Engineering and Knowledge Engineering, Volume 1, World Scientific Publishing Co, 1993.

[Gomaa et al. 1994] H. Gomaa et al., « A Prototype Domain Modeling Environment for Reusable Software Architectures », 3rd International Conference on Software Reuse, Rio de Janeiro, novembre 1994.

[Greenwood et al. 2003a] M. Greenwood, D. Balasubramaniam, S. Cimpan, N.C. Kirby,

K. Mickan, R. Morrison, F. Oquendo, I. Robertson, W. Seet, R. Snowdon, B. Warboys et E. Zirintsis, « Process Support for Evolving Active Architectures », Proceedings of the 9th European Worshop on Software Process Technology, LNCS 2786, Springer Verlag, Helsinki, septembre 2003.

[Greenwood et al. 2003b] M. Greenwood, I. Robertson, W. Seet, R. Snowdon et B. Warboys, « Evolution Meta-Process Model », Deliverable D5.3, ArchWare European RTD Project, IST-2001-32360, décembre 2003.

[Griffiths et Pemberton 2000] R. N. Griffiths, L. Pemberton, « Don't Write Guidelines, Write Patterns! », University of Brighton, Brighton, UK, available from http://www.it.bton.ac.uk/staff/lp22/guidelinesdraft.html.

[Hoare 1985] C.A.R. Hoare, « Communicating Sequential Processes », Prentice Hall, 1985.

[IFE] International summer school on « Design and Evaluation of Human System Interfaces (HSIs) » Halden, Norvège, août 2003.

[Inverardi et Wolf 1995] P. Inverardi et A. Wolf, « Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model », IEEE transactions on software engineering, vol. 21, no. 4, avril 1995.

[ISO 2000] International Organization for Standardization, « Ergonomic design of control centres », ISO 11064, décembre 2000.

[Jambon 1997] F. Jambon, « La responsabilité du concepteur d'une interface homme machine face aux erreurs des utilisateurs : exemple de l'accident d'Airbus A320 survenu au mont Sainte-Odile », L'interacteur – Bulletin d'information de l'Association Francophone d'Interaction Homme-Machine (AFIHM)(2) pp.6, avril 1997.

[Jarzabek et al. 2001] S. Jarzabek, W. Chun Ong et H. Zhang, « Handling Variant Requirements in Domain Modeling », Proceedings of International. Conference on Software Engineering & Knowledge Engineering (SEKE'01), Knowledge Systems Institute, Buenos Aires, juin 2001.

[Jazayeri 2002] M. Jazayeri, « On Architectural Stability and Evolution », in Proceedings of the Reliable Software Technlogies--Ada-Europe 2002, pages 13-23, Johann Blieberger and Alfred Strohmeier, editors, Springer Verlag, juin 2002.

[JViews] ILOG JViews Component Suite, http://www.ilog.com/products/jviews.

[Karl-Olof, 1998] Karl-Olof, « Cost Of Failure In Quality In a Major Civil Engineering », 1998.

[Klein et Kazman 1999] M. Klein et R. Kazman, « Attribute-Based Architectural Styles », Technical Report CMU/SEI-99-TR-022, ESC-TR-99-022, octobre 1999.

[Kozen 1983] D. Kozen, « Results on the Propositional μ-Calculus », Theoretical Computer Science 27:333-354, 1983.

[Leymonerie 2004] F. Leymonerie, « ASL : un langage et des outils pour les styles architecturaux. Contribution à la description d'architectures dynamiques », Thèse de Doctorat, LISTIC, Université de Savoie, Annecy, décembre 2004.

[Leymonerie et al. 2001] F. Leymonerie, S. Cîmpan et F. Oquendo, « Extension d'un langage de description architecturale pour la prise en compte des styles architecturaux: Application à J2EE », Proceedings ICSSEA 14th International Conference on Software and Software Engineering and their Applications, Paris, décembre 2001.

[Leymonerie et al. 2002] F. Leymonerie, S. Cîmpan et F. Oquendo, « Etat de l'art sur les styles architecturaux : Classification et Comparaison des Langages de Description

d'Architectures Logicielle », Revue Génie Logiciel, No. 62, septembre 2002.
[Lopez-Herrejon et Batory 2001] R.E. Lopez-Herrejon et D. Batory, « A Standard Problem for Evaluating Product-Line Methodologies », Proceedings of the Third International Conference on Generative and Component-Based Software Engineering, Springer-Verlag, 2001.
[Love 1991] T. Love, « Timeless Design of Information Systems », Object Magazine, pages 42-48, novembre/décembre 1991.
[Luckham et al. 1995] D. C. Luckham, L. M. Augustin, J. J. Kenny, J. Veera, D. Bryan, and W. Mann, « Specification and analysis of system architecture using Rapide », IEEE Transactions on Software Engineering, 21(4) : 336-355, avril 1995.
[Magee et al. 1995] J. Magee, N. Dulay, S. Eisenbach et J. Kramer, « Specifying Distributed Software Architectures », Proceedings of the Fifth European Engineering Conference (ESEC'95), Barcelona, septembre 1995.
[Mak 1992] V.W. Mak, « Connection: An Inter-Component Communication Paradigm for Configurable Distributed Systems », Proceedings of the International Workshop on Configurable Distributed Systems, Londres, UK, mars 1992.
[MCR] MCR web page: http://cern.web.cern.ch/CERN/Divisions/PS/OP/Welcome.html.
[Medvidovic et al. 1996] N. Medvidovic, P. Oreizy, J. E. Robbins et R. N. Taylor, « Using object-oriented typing to support architectural design in the C2 style », in SIGSOFT'96 : Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering, ACM Press, octobre 1996.
[Medvidovic et Taylor 1997] N. Medvidovic et R. Taylor, « A Classification and Comparaison Framework for Architecture Description Languages », Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, février 2003.
[Mehta et al. 2004] N. R. Mehta, R. Soma et N. Medvidovic, « Style-Based Software Architectural Compositions as Domain-Specific Models », Proceedings of the International Conference on Software Engineering (ICSE 2004), 2004.
[Mehta et Medvidovic 2004] N. R. Mehta et N. Medvidovic, « Toward Composition Of Style-Conformant Software Architectures », Technical Report, USC-CSE-2004-500, 2004.
[META-H] http://www.rl.af.mil/tech/programs/dasada/tools/metah.html.
[Mettala et Graham 1992] E. Mettala et M. H. Graham, « The domain-specific software architecture program », Technical report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, juin 1992.
[Microsoft Corporation 1987] Microsoft Corporation, « The Windows Interface: An Application Design Guide », Microsoft Corporation, 1987.
[Milner et al. 1992] R. Milner, J. Parraw et D. Walker, « A calculus of mobile processes », Information and Computation, pp.1-40, septembre 1992.
[Monroe et al. 1997] R. Monroe, A. Kompanek, R. Melton, D. Garlan, « Architectural Styles, Design Patterns, and Objects », IEEE Software, pp. 43-52, 1997.
[Monroe et Garlan 1996] R. Monroe, D. Garlan, « Style-Based Reuse for Software Architectures », Proceedings of the 1996 International Conference on Software Reuse, avril 1996.
[Monroe 1998] R.T. Monroe, « Capturing Software Architecture Design Expertise With

Armani », Technical Report CMU-CS-98-163, 1998.
[Monson-Haefel et Chappelle 2002] R. Monson-Haefel, D.A. Chappelle, « Java Message Service », O'Reilly, décembre 2002.
[Moray 2002] N. Moray, « Humans and machine : allocation of function », People in control: human factors in control room design, janvier 2002.
[Moriconi et al. 1995] M. Moriconi, X. Qian, et R. Riemenschneider, « Correct architecture refinement », IEEE Transactions on Software Engineering, Special Isue on Software Architecture, 21(4) : 356-372, avril 1995.
[Moriconi et Riemenschneider 1997] M. Moriconi et R.A. Riemenschneider, « Introduction to SADL 1.0, A Language for Specifying Software Architecture Hierarchies », mars 1997.
[Naumovich et al. 1997] G. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil, « Applying Static Analyses to Software Architectures », Proceedings of Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 77 -- 93, 1997.
[Oquendo 2003] F. Oquendo, « The ArchWare Architecture Description Language: Tutorial », Report R1.1-1, ArchWare European RTD Project, IST-2001-32360, mars 2003.
[Oquendo 2003] F. Oquendo, « The ArchWare Architecture Refinement Language », Deliverable D6.1b, ArchWare European RTD Project, IST-2001-32360, décembre 2003.
[Oquendo et al. 2002] F. Oquendo, I. Alloui, S. Cimpan et H. Verjus, « The ArchWare Architecture Description Language: Abstract Syntax and Formal Semantics », Deliverable D1.1b, ArchWare European RTD Project, IST-2001-32360, décembre 2002.
[Oquendo et al. 2004] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, et C. Occhipinti, « ArchWare: Architecting Evolvable Software », Proceedings of the first European Workshop on Sofware Architecture (EWSA 2004), pp. 257-271, St-Andrews, Ecosse, mai 2004.
[Oracle9iAS] Oracle9iAS Documentation Library, http://otn.oracle.com/documentation/index.html.
[Parnas 2001] D.L. Parnas, « On the Design and Development of Software Families », Software fundamentals: collected papers by David L. Parnas**,** pp 193-213, Addison-Wesley Longman Publishing Co, 2001.
[PCR] PCR web page: http://sl.web.cern.ch/SL/opnews/pageswww/ophome.html.
[PcVue] http://www.arcinfo.com/doc/FR/PRODUCTS/PcVue.html.
[Perry et al. 2004] D.E. Perry, S.E. Sim et S. Easterbrook, « Case Studies for Software Engineers », in proceedings of the 26th International Conference on Software Engineering (ICSE'04), Edinburgh, Scotland, mai 2004.
[Perry et Wolf 1992] D.E. Perry et A.L. Wolf, « Foundations for the Study of Software Architecture », ACM SIGSOFT Software Engineering Notes vol. 17 no. 4 pp 40, 1992.
[Philips et al. 1997] R.J. Philips et al., « Process Cost Reduction Through Proactive Operations and Maintenance : Food Manufacturing Colaition for Innovation and Technology Transfer », mars 1997.
[PVSS] http://www.pvss.com.
[Rapide 1997] Rapide Design Team, « Guide to the Rapide 1.0 », Language Reference Manuals, Technical Report, Stanford University, juillet 1997.

[RAPIDE] http://www-2.cs.cmu.edu/~acme/adltk/adls.html#rapide.
[Ratcliffe 2002] O. Ratcliffe, « SEAM User Requirements Document », CERN, Genève, Suisse, 2002.
[Ratcliffe et al. 2004] O. Ratcliffe, S. Cîmpan, F. Oquendo, L. Scibile, « Formalization of an HCI Style for Accelerator Restart Monitoring », Proceedings of the First European Workshop on Software Architectures (EWSA 2004), mai 2004.
[Rational Rose] http://www-306.ibm.com/software/rational.
[Requisite Pro] http://www-306.ibm.com/software/awdtools/reqpro.
[Rhapsody] http://www.ilogix.com/rhapsody/rhapsody.cfm.
[Rumbaugh et al. 1991] J. Rumbaugh et al., « Object-Oriented Modeling and Design », Prentice Hall, Englewood Cliffs, NJ, 1991.
[Sangiorgi 1992] D. Sangiorgi, « Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms », PhD Thesis, Universty of Edinburgh, 1992.
[Savigni 2000] A. Savigni, « A general framework for monitoring and control systems », research abstract for ICSE2000 doctoral workshop, 2000.
[Savigni et Tisato 1999] A. Savigni et F. Tisato, « Kaleidoscope: A Reference Architecture for Monitoring and Control Systems », Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), p.369-388, February 22-24, 1999.
[Savigni et Tisato 2000] A. Savigni et F. Tisato, « Designing Traffic Control Systems. A Software Engineering Perspective », in Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems --- 8th Meeting of the Euro Working Group Transportation --- EWGT), Rome, Italie, septembre 2000.
[Shaw 1994] M. Shaw, « Patterns for Software Architectures », in J. Coplien and D. Schmidt (Eds.), Pattern Languages of Program Design, Addison-Wesley, 1995 (from the First Annual Conference on Pattern Languages of Programming, août 1994).
[Shaw 1995] M. Shaw, « Some Patterns for Software Architecture », in J. Vlissides, J. Coplien and N. Kerth (Eds.), Pattern Languages of Program Design, Vol. 2, Addison-Wesley, 1996 (from the Second Annual Conference on Pattern Languages of Programming, septembre 1995).
[Shaw et al. 1995] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young et G. Zelesnik, « Abstraction for Software Architecture and Tools to Support Them », IEEE Transactions on Software Engineering: Special Issue on Software Architecture, avril 1995.
[Shaw et Clement 1996] M. Shaw et P. Clements, « How Should Patterns Influence Architecture Description Languages? A Call for Discussion », working paper for DARPA EDCS community, juillet 1996.
[Shaw et Clements 1997] M. Shaw et P. Clements, « A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems », Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC'97), 1997.
[Shneiderman 1992] B. Shneiderman, « Designing the User Interface: Strategies for Effective Human-Computer Interaction », Addison-Wesley, p.72, 1992.
[Skogseid et Sptring 1995] I. Skogseid, M. Sptring, « Patterns for Human-Computer

Interaction, Studies of Principle Aggregation and Pattern Naming », novembre 1995.
[SL-GMS] http://www.sl.com/sl_gms_pkg_jdev.html.
[Sollander et al. 2003] P. Sollander, R. Martini, J. Stowisek, « Technical Infrastructure Monitoring (TIM): migrating the TIM System to J2EE, feasibility study », CERN, Suisse, juillet 2003.
[Sollander et Camara 2000] P. Sollander, V. Camara, « TCR Human Computer Interface Conventions », CERN, Genève, Suisse, novembre 2000.
[SonicMQ] SonicMQ Documentation Portal, http://j2ee-wg.web.cern.ch/j2ee-wg/Sonic MQ50/sonicmq_docportal.htm.
[Spencer et Rhee 2003] C.M. Spencer, S.J. Rhee, « Cost Based Failure Modes and Effect Analysis (FMEA) for Systems of Accelerator Magnets », 2003.
[Spivey 1989] J. Spivey, « The Z Notation: A Reference Manual », Prentice Hall, 1989.
[Stafford et al. 1997] J.A Stafford, D.J. Richardson, et A.L. Wolf, « Chaining: A Software Architecture Dependence Analysis Technique », Technical Report, Department of Computer Science, University of Colorado, 1997.
[Stafford et al. 1998] J. A. Stafford, D. J. Richardson, A. L. Wolf, « Aladdin: A Tool for Architecture-Level Dependence Analysis of Software », University of Colorado at Boulder, Technical Report CU-CS-858-98, avril 1993.
[Svahnberg et Bosch 1999] M. Svahnberg et J. Bosch, « Evolution in Software Product Line », Mikael Svahnberg, Jan Bosch, Journal of Software Maintenance, Vol. 11, No. 6, pp. 391-422, 1999.
[TCR] TCR web page: http://st.web.cern.ch/st/mo/tcr/default.html.
[U.S. Nuclear Regulatory Commission 2002] U.S. Nuclear Regulatory Commission, « Human-System Interface Design Review Guidelines », NUREG-0700 Rev. 2, Office of Nuclear Regulatory Research, mai 2002.
[Unicon] http://www-2.cs.cmu.edu/afs/cs/project/vit/www/unicon/index.html.
[Upchurch 1995] R. Upchurch, « Perspective Foundations: Where does architecture fit in the life-cycle? », 1995.
[Van der Hoek 2002] A van der Hoek, « Representing Product-Line Architectures », Ground System Architectures Worshops (GSAW2002), 2002.
[Van Deursen et Klint 2002] A. van Deursen et P. Klint, « Domain-Specific Language Design Requires Feature Descriptions », Journal of Computing and Information Technology 10(1):1-17, 2002.
[Verjus et Oquendo 2003] H. Verjus et F. Oquendo, « The ArchWare Architecture Description Language: XML Concrete Syntax », Deliverable D1.3b, ArchWare European RTD Project, IST-2001-32360, juin 2003.
[Vestal 1992] S. Vestal, « MetaH Reference Manual »,Honeywell Technology Center, Minneapolis MN, 1992.
[Vestal 1994] S. Vestal, « Mode Changes in Real-Time Architecture Description Language », Proceedings of the Second International Workshop on Configurable Distributed Systems, mars 1994.
[W. Seet 2002] W. Seet, « Core ArchWare Software Architecture Framework », Deliverable D4.1, ArchWare European RTD Project, IST-2001-32360, novembre 2002.
[Weiss et Lai 1999] D.M. Weiss et C.T.R. Lai, « Software Product-Line Engineering », Addison-Wesley, 1999.

[Welie et al. 2000] M. van Welie, G. C. van der Veer, A. Eliëns, « Patterns as Tools for User Interface Design ». http://www.cs.vu.nl/~martijn/gta/docs/TWG2000.pdf, 2000.

[Wile 1999] D. Wile, « AML: An Architecture Meta Language », Proceedings of the 14th International Conference on Automated Software Engineering, pp. 183-190, Cocoa Beach, Floride, octobre 1999.

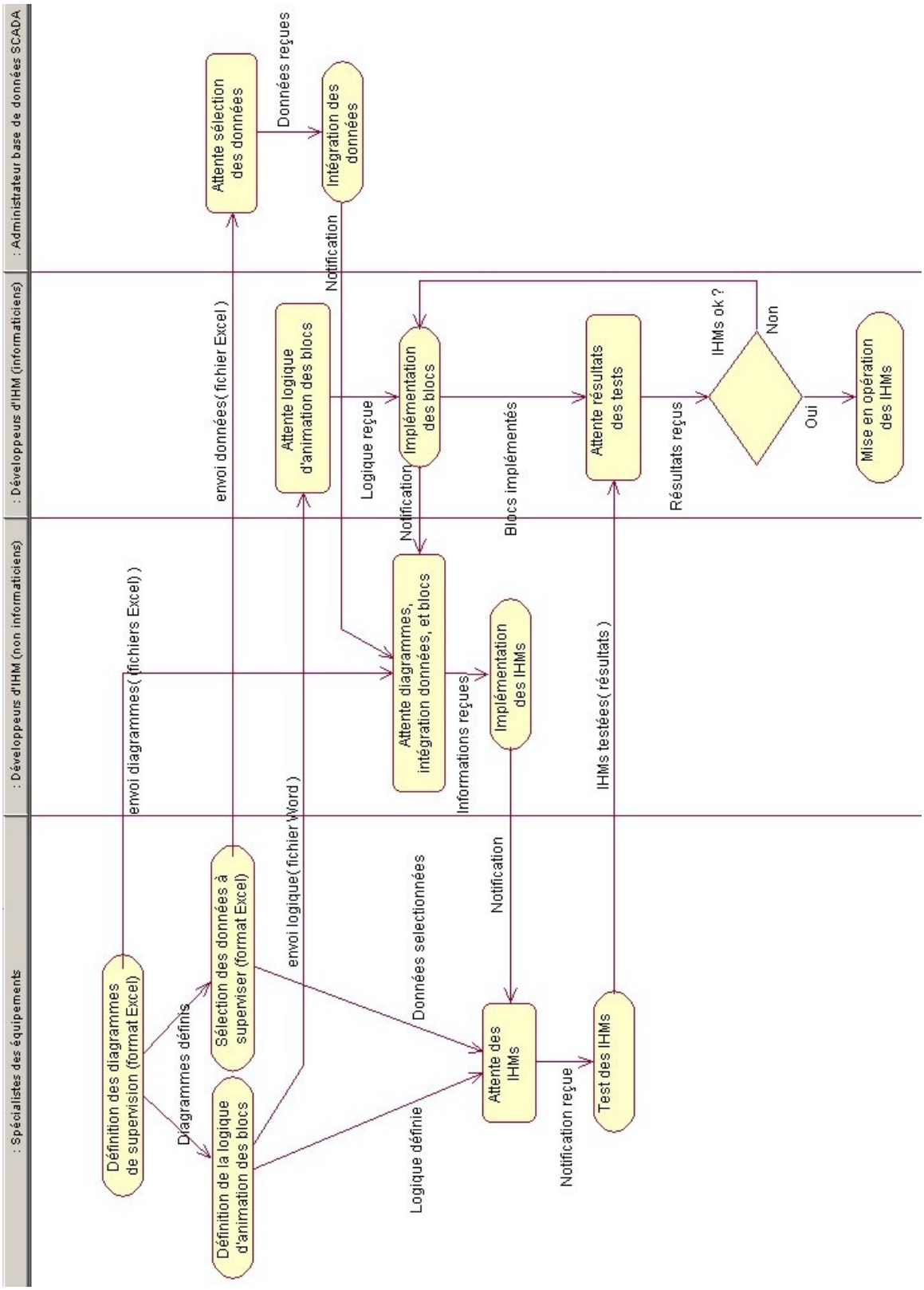
[Wile 2001] D. Wile, « Using Dynamic ACME », In Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Australie, décembre 2001.

[Yin 2003a] R.K. Yin, « Case Study Research, Design and Methods », third edition, Applied Social Research Methods Series Volume 5, SAGE Publications, 2003.

[Yin 2003b] R.K. Yin, « Applications of Case Study Research », second edition, Applied Social Research Methods Series Volume 34, SAGE Publications, 2003.

[Zenger 2002] M. Zenger, « Extensibility in the Large », First Doctoral Workshop on Global Computing, Lausanne, Switzerland, juin 2002.

Annexe 1 : Diagramme d'activité du développement des logiciels de supervision du SPS



Annexe 2 : Formalisation du style GTPM

Le code suivant spécifie le style *GtpmHci*. Il exprime ce qui fait partie de la librairie du style :

- Les ports sont *InputPort* et *OutputPort* ;
- Les composants sont *Equipment* et *StatusBloc. StatusBloc* peut être spécialisé en composant *MetaStatus* ou *SystemStatus* ;
- Les connecteurs sont *DataLink.*

Il définit les contraintes suivantes :

- Les composants ne peuvent être que des *Equipment, StatusBloc, MetaStatus* et *SystemStatus* ;
- Les connecteurs ne peuvent être que des *DataLink* ;
- Deux *DataLink* ne peuvent être attachés ensemble ;
- Les *Equipment* ne peuvent être supervisés que par des *SystemStatus* ou par des *StatusBloc* ;
- Les *SystemStatus* ne peuvent être supervisés que par des *MetaStatus* ou par des *StatusBloc* ;
- Les *MetaStatus* ne peuvent être supervisés que par des *StatusBloc* et vice-versa ;
- Un port d'élément peut seulement être attaché à un autre port d'élément ;
- Les boucles dans la chaîne d'acquisition sont interdites
- Les blocs ne peuvent pas être superposés ;
- Les attributs graphiques des blocs présents sur une IHM doivent avoir des valeurs homogènes ;
- Les valeurs des attributs d'information des blocs doivent être cohérents avec celles des équipements qu'ils supervisent.

Il fournit les analyses :

- *directly_monitored* vérifie si un composant est directement supervisé par un autre ;
- *monitered* vérifie si un composant est indirectement supervisé par un autre ;
- *bloc_superimposition* vérifie si deux blocs sont superposés ;
- *homogeneous_graphical_info* vérifie si deux blocs sont graphiquement homogènes.

```
GtpmHci is style extending Component where {
  styles {
      InputPort is style extending Port where {…}
      OutputPort is style extending Port where {…}
      Equipment is style extending Component where {…}
      StatusBloc is style extending Component where {…}
      MetaStatus is style extending StatusBloc where {…}
      SystemStatus is style extending StatusBloc where {…}
      DataLink is style extending Connector where {…}
      DependenceLink is style extending Connector where {…}
  }
```

```
contraints {
    common {
      attributes
        width: Integer default value is 1024,
        height: Integer default value is 768,
        Xposition: Integer default value is 0,
        Yposition: Integer default value is 0,
        scenarioName: String,
        locationName: String;
    }.
    to connectors apply {
      -- les connecteurs ne peuvent être que des DataLink
      forall(c | c in style DataLink or c in style DependenceLink),

      -- deux DataLink ne peuvent être attachés ensembles
      forall (l1,l2 | (l1 in style DataLink and l2 in style DataLink) or (l1 in style
      DependenceLink and l2 in style DependenceLink) or (l1 in style DataLink and
      l2 in style DependenceLink) implies not attached (l1,l2))
    }.

    to components apply {
      -- les composants ne peuvent être que des StatusBloc, MetaStatus
      -- ou SystemStatus
      forall(c | c in style Equipment or c in style StatusBloc or c in style MetaStatus
      or c in style SystemStatus),

      forall(c1,c2 | c1 directly monitored by c2 implies
      -- les Equipment ne peuvent être connectés qu'à des
      -- SystemStatus ou à des StatusBloc
      (c1 in style Equipment and c2 in style SystemStatus) or
      (c1 in style Equipment and c2 in style StatusBloc) or

      -- les SystemStatus ne peuvent être connectés qu'à des
      -- MetaStatus ou à des StatusBloc
      (c1 in style SystemStatus and c2 in style MetaStatus) or
      (c1 in style SystemStatus and c2 in style StatusBloc) or

      -- les MetaStatus ne peuvent être connectés qu'à des
      -- StatusBloc et vice-versa
      (c1 in style MetaStatus and c2 in style StatusBloc) or
      (c1 in style StatusBloc and c2 in style MetaStatus),

      -- les boucles dans la chaîne d'acquisition sont interdites
      forall(c | (c in style StatusBloc or c in style MetaStatus or c in style
      SystemStatus) and c monitored by c implies false),
```

```
-- les blocs ne peuvent pas être superposés
forall(c1,c2 | (c1 in style StatusBloc or c1 in style MetaStatus or c1 in style
SystemStatus) and (c2 in style StatusBloc or c2 in style MetaStatus or c2 in
style SystemStatus) and c1 superimposed to c2 implies false),

-- les propriétés graphiques des blocs doivent être homogènes
forall(c1,c2 | (c1 in style StatusBloc or c1 in style MetaStatus or c1 in style
SystemStatus) and (c2 in style StatusBloc or c2 in style MetaStatus or c2 in
style SystemStatus) implies c1 homogeneous to c2),

-- si tous les équipements supervisés par un bloc sont associés à une même salle
-- de contrôle, alors le bloc lui-même doit être associé à cette salle de contrôle
forall(bloc | (bloc in style StatusBloc or bloc in style SystemStatus) and forall
(e1, e2 | e1 in style Equipment and e1 monitored by bloc and e2 in style
Equipment and e2 monitored by bloc and to e1.attributes apply {
  exists(ctrlroom1 | ctrlroom1.name = "controlRoom" and to e2.attributes
  apply {
    exists(ctrlroom2 | ctrlroom2.name = "controlRoom" and ctrlroom2.value =
    crtrlroom1.value implies to bloc.attributes apply {
      exists(blocctrlroom | blocctrlroom.name = "controlRoomText" and
      blocctrlroom.value = ctrlroom2.value)
    })
  })
})),

-- si tous les équipements supervisés par un bloc sont associés à un même numéro
-- de bâtiment, alors le bloc lui-même doit être associé à ce numéro de bâtiment
forall(bloc | (bloc in style StatusBloc or bloc in style SystemStatus) and forall
(e1, e2 | e1 in style Equipment and e1 monitored by bloc and e2 in style
Equipment and e2 monitored by bloc and to e1.attributes apply {
  exists(bdnum1 | bdnum1.name = "buildingNumber" and to e2.attributes
  apply {
    exists(bdnum2 | bdnum2.name = "buildingNumber" and bdnum2.value =
    bdnum1.value implies to bloc.attributes apply {
      exists(blocbdnum | blocbdnum.name = "buildingNumber" and
      blocbdnum.value = bdnum2.value)
    })
  })
})),

-- si tous les équipements supervisés par un bloc sont associés à un même système
-- et sous système, alors le bloc devra avoir la catégorie correspondante
forall(bloc | (bloc in style StatusBloc or bloc in style SystemStatus) and forall
(e1, e2 | e1 in style Equipment and e1 monitored by bloc and e2 in style
Equipment and e2 monitored by bloc and to e1.attributes apply {
```

```
        exists(sys1, ssys1 | sys1.name = "system" and ssys1.name = "subsystem" and
        to e2.attributes apply {
          exists(sys2, ssys2 | sys2.name = "system" and ssys2.name = "subsystem"
          and sys2.value = sys1.value and ssys2.value = ssys1.value implies to
          bloc.attributes apply {
            exists(bloccateg | bloccateg.name = "category" and bloccateg.value =
            (sys2.value  ++ "/" ++ ssys2.value))
          })
        })
      })),

      -- si tous les équipements supervisés par un bloc ont un même responsable, alors
      -- bloc lui-même doit être associé à ce responsable
      forall(bloc | (bloc in style StatusBloc or bloc in style SystemStatus) and forall
      (e1, e2 | e1 in style Equipment and e1 monitored by bloc and e2 in style
      Equipment and e2 monitored by bloc and to e1.attributes apply {
        exists(resp1 | resp1.name = "responsible" and to e2.attributes apply {
          exists(resp2 | resp2.name = "responsible" and resp2.value = resp1.value
          implies to bloc.attributes apply {
            exists(blocresp | blocresp.name = "responsible" and blocresp.value =
            resp2.value)
          })
        })
      }))
    }.
    to elements apply {
      -- un port d'élément peut seulement être attaché à un autre port
      -- d'élément
      forall(bloc | to bloc.ports apply{
        exists(blocport | to elements apply{
          exists([0..1]dl | to dl.ports apply{
            exists(dlport | dlport attached to blocport)
          })
        })
      }),
      -- tous les InputPort des éléments doivent être connectés
      forall(bloc | to bloc.ports apply{
        forall(bip | bip in style InputPort and to connectors apply{
          exists(dl | to dl.ports apply {
            exists(dlout | dlout in style OutputPort and dlout attached to bip)
          })
        })
      })
    }
}
analysis {
```

```
directly_monitored is AAL_property {
  parameters {
    bloc1 : AnyElement,
    bloc2 : AnyElement
  }
  property {
    bloc1 in style Component.
    bloc2 in style Component.
    to bloc2.ports apply{
      exists(b2ip | b2ip in style InputPort and to connectors apply {
        exists (dl | to dl.ports apply {
          exists (dlin, dlout | dlin in style InputPort and dlout in style OutputPort
          and to bloc1.ports apply {
            exists(b1op | b1op in style OutputPort and b1op attached to dlin and
            dlout attached to b2ip)
          })
        })
      })
    }
  }
} mixfix (bloc1 directly monitored by bloc2);

monitored is AAL_property {
  parameters {
    bloc1 : AnyElement,
    bloc2 : AnyElement
  }
  property {
    bloc1 in style Component.
    bloc2 in style Component.
    bloc1 directly monitored by bloc2
    or
    to components apply {
      exists(bloc | monitored(bloc1, bloc) and bloc directly monitored by bloc2)
    }
  }
} mixfix (bloc1 monitored by bloc2);

bloc_superimposition is AAL_property {
  parameters {
    bloc1 : AnyElement,
    bloc2 : AnyElement
  }
  property {
    bloc1 in style Component.
    bloc2 in style Component.
```

```
        to bloc1.attributes apply {
          exists(w1, h1, x1, y1 | w1.name = "width" and h1.name = "height" and
          x1.name = "positionX" and y1.name = "positionY" and to bloc2.attributes
          apply {
            exists(w2, h2, x2, y2 | w2.name = "width" and h2.name = "height" and
            x2.name = "positionX" and y2.name = "positionY" and x2.value =>
            (x1.value – w2.value) and x2.value <= (x1.value + w1.value) and
            y2.value => (y1.value – h2.value) and y2.value <= (y1.value + h1.value))
          })
        }
      }
    } mixfix (bloc1 superimposed to bloc2);

    homogeneous_graphical_info is AAL_property {
      parameters {
        bloc1 : AnyElement,
        bloc2 : AnyElement
      }
      property {
        bloc1 in style Component.
        bloc2 in style Component.
        to bloc1.attributes apply {
          exists(rf1, rfs1, rvzf1, lf1, lfs1, lvzf1 | rf1.name = "controlRoomFont" and
          rfs1.name = "controlRoomFontSize" and rvzf1.name =
          "controlRoomZoomFactor" and lf1.name = "labelFont" and lfs1.name =
          "labelFontSize" and lvzf1.name = "labelVisibleZoomFactor" and to
          bloc2.attributes apply {
            exists(rf2, rfs2, rvzf2, lf2, lfs2, lvzf2 | rf2.name = "controlRoomFont" and
            rfs2.name = "controlRoomFontSize" and rvzf2.name =
            "controlRoomVisibleZoomFactor" and lf2.name = "labelFont" and
            lfs2.name = "labelFontSize" and lvzf2.name = "labelVisibleZoomFactor"
            and rf2.value = rf1.value and rfs2.value = rfs1.value and rvzf2.value =
            rvzf1.value and lf2.value = lf1.value and lfs2.value = lfs1.value and
            lvzf2.value = lvzf1.value)
          })
        }
      }
    } mixfix (bloc1 homogeneous to bloc2);
  }
  templates {
    SystemStatusConstruct is template with {
      parameters as
        bloc: SystemStatus,
        equipments: set[view[source: Equipment, link: DataLink]],
      configuration
        iterate equipments by e
```

```
        do attach e.source~out to e.link~in.
        do attach e.link~out to bloc~in
}
MetaStatusConstruct is template with {
  parameters as
    bloc: MetaStatus,
    meta: set[view[source: MetaStatus, link: DataLink]],
    systems: set[view[source: SystemStatus, link: DataLink]],
    status: set[view[source: StatusBloc, link: DataLink]],
  configuration
    iterate meta by m
      do attach m.source~out to m.link~in.
      do attach m.link~out to bloc~in
    iterate systems by sys
      do attach sys.source~out to sys.link~in.
      do attach sys.link~out to bloc~in
    iterate meta by st
      do attach st.source~out to st.link~in.
      do attach st.link~out to bloc~in
}
StatusBlocConstruct is template with {
  parameters as
    bloc: StatusBloc,
    equipments: set[view[source: Equipment, link: DataLink]],
    meta: set[view[source: MetaStatus, link: DataLink]],
    systems: set[view[source: SystemStatus, link: DataLink]],
    status: set[view[source: StatusBloc, link: DataLink]],
  configuration
    iterate equipments by e
      do attach e.source~out to e.link~in.
      do attach e.link~out to bloc~in
    iterate meta by m
      do attach m.source~out to m.link~in.
      do attach m.link~out to bloc~in
    iterate systems by sys
      do attach sys.source~out to sys.link~in.
      do attach sys.link~out to bloc~in
    iterate meta by st
      do attach st.source~out to st.link~in.
      do attach st.link~out to bloc~in
}
BlocDependenceConstruct is template with {
  parameters as
    bloc1: StatusBloc,
    bloc2: StatusBloc,
    link: DependenceLink,
```

```
        configuration
          do attach bloc1~out to link~in.
          do attach link~out to bloc2~in
      }
}

InputPort is style extending Port where {
  types {
      Data is connection[Any];
  constraints {
      to connections apply {
        -- les InputPort ont une et une seule connexion
        exists([1]c | c of type Data),
        -- un InputPort a seulement des connexions pour recevoir des données
        forall(c | every sequence{true*.via c send any} leads to state{false})
      }
  }
}

OutputPort is style extending Port where {
  types {
      Data is connection[Any];
  constraints {
      to connections apply {
        -- les outputPort ont une et une seule connexion
        exists([1]c | c of type Data),
        -- un OutputPort a seulement des connexions pour envoyer des données
        forall(c | every sequence{true*.via c receive any} leads to state{false})
      }
  }
}

Equipment is style extending Component where {
  constraints {
      common {
        attributes
          controlRoom: String,
          buildingNumber: Integer,
          system: String,
          subsystem: String,
          responsible: String,
          equipAddress : String,
          dataType : String;
      }.
```

```
      to ports apply {
        -- les Equipment n'ont que des OutputPort
        forall (p | p in style OutputPort),
        -- les Equipment ont au moins un OutputPort
        exists ([1..n]p | p in style OutputPort)
      }
  }
}

StatusBloc is style extending Component where {
  types {
      ResultState is Integer;
  };
  ports {
      archetype OutPort is port in style OutputPort with {
        connections
          out: Data;
        protocol
          replicate
          via out send ResultState;
      }
  };
  constraints {
      common {
        types
          ResultState => 0,
          ResultState <= 3,
      }.
      common {
        attributes
          width: Integer default value is 20,
          height: Integer default value is 10,
          positionX: Integer,
          positionY: Integer,
          controlRoomText: String default value is "TCR",
          controlRoomFont: String default value is "Arial",
          controlRoomFontSize: Integer default value is 12,
          controlRoomVisibleZoomFactor: Integer default value is 0.5,
          labelText: String default value is "Text",
          labelFont: String default value is "Arial",
          labelFontSize: Integer default value is 14,
          labelVisibleZoomFactor: Integer default value is 0.5,
          name: String,
          info: String,
          buildingNumber: Integer,
          category: String,
```

```
            responsible: String,
            priority: Integer,
            rule: String;
        }.
        to ports apply {
          -- les StatusBloc n'ont que des OutputPort ou des InputPort
          forall (p | p in style OutputPort xor p in style InputPort),
          -- les StatusBloc n'ont qu'un seul OutputPort
          exists ([1]p | p in style OutputPort),
          -- les StatusBloc ont au moins un InputPort
          exists([1..n]p | p in style InputPort)
        }
    }
}

DataLink is style extending Connector where (
    ports {
        archetype InPort is port in style InputPort with {
          connections
            in: Data;
          protocol
            replicate
            via in receive;
        },
        archetype OutPort is port in style OutputPort with {
          connections
            out: Data;
          protocol
            replicate
            via out send;
        }
    };
    constraints {
        to ports apply {
          -- les DataLink n'ont que des OutputPort ou des InputPort
          forall (p | p in style OutputPort xor p in style InputPort),
          -- les DataLink n'ont qu'un seul OutputPort
          exists ([1]p | p in style OutputPort),
          -- les DataLink n'ont qu'un seul InputPort
          exists([1]p | p in style InputPort)
        }
    }
}

GlobalHci is style extending Component where {
    styles {
```

```
        GtpmHci is style extending Component where {…}
    }
    contraints {
        to components apply {
          -- les composants ne peuvent être que des GtpmHci
          forall (c | c in style GtpmHCI)
        }
    …
    }
    analysis {…}
}
```

Annexe 3 : Schéma de la base de données de SEAM