Application de la méthode à la sécurisation d'une architecture SCADA

6.1 Problème de l'application

Reprenons le cas illustratif. L'application est composée du client Clt_1 réalisant l'opération *Read* sur la ressource Res_1 détenue par le composant Plc_1. L'une des propriétés à garantir est PRT_1 qui énonce que tout accès aux ressources doit respecter les droits d'accès.

Après avoir réalisé un mécanisme d'exclusion mutuelle entre Plc_1 et Plc_2 pour la ressource Res_1, Ciprian doit sécuriser l'architecture car celle-ci est maintenant soumise à divers accès extérieurs qui pourraient s'avérer dangereux. Il ne connaît pas bien le domaine de la sécurité, alors prudent, il préfère laisser à Philippe, architecte et expert en sécurité, le soin de concevoir ce modèle.

Dans la section précédente nous avons décrit les *problem cases* et pourquoi nous les utilisons pour décrire le domaine. Différentes questions se posent alors, comment utiliser les *problem cases*, et comment ces *problem cases* alimentent la vérification et le diagnostic.

6.2 Domaine de la sécurité

On considère qu'il existe un domaine relatif à la sécurité, et qu'il est composé d'un ensemble de problèmes et de solutions exprimées par des *problem cases*. La présentation de ce domaine est en partie extraite du travail de [OD17] sur les patrons de sécurités.

6.2.1 Architecture abstraite

Pour exprimer les propriétés en LTL, et plus généralement pour exprimer les propriétés d'un problème, il est nécessaire d'imposer une architecture minimale sur les solutions. Comme le montre la figure 6.1, nous définissons dans l'espace du problème cette architec-

ture par le biais d'une structure abstraite et d'un comportement générique, semblable au concept de *machine* dans [Jac01].

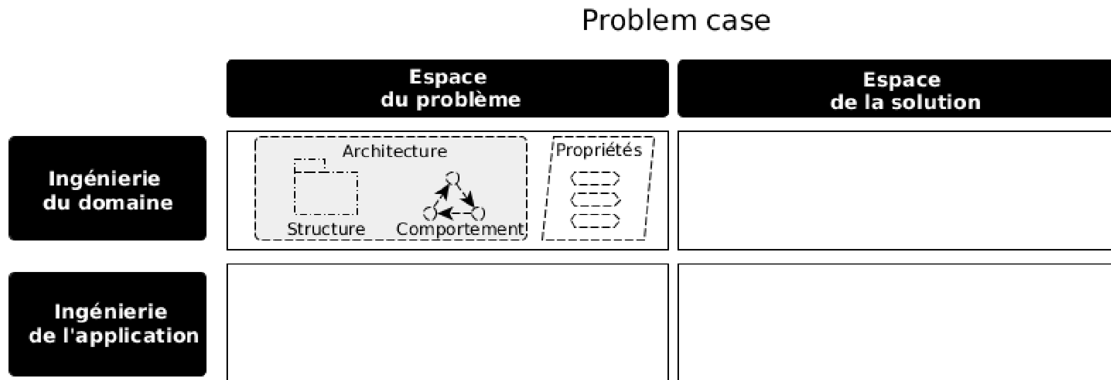

FIGURE 6.1 – Architecture abstraite des problem cases

La structure abstraite définit les constituants de base d'une architecture SCADA (clients, composants, ressources ...). Nous prenons pour hypothèse que tout mécanisme de sécurité est inclus dans un constituant de l'architecture de type composant. Pour accéder à un composant de l'architecture, un client *Clt* doit émettre une requête *req*. Pour accéder à une ressource *res* détenue par un composant, cette requête contient l'opération à effectuer sur la ressource *opRes*.

Un composant possède le comportement générique décrit dans la figure 6.2. De l'état *Idle*, le composant capte un message (*receive()*) provenant d'une fifo d'entrée, l'emmenant dans l'état de réception nommé *Received*. Si ce message lui est destiné (*mine()=true*), il le traite (*Compute*), sinon il le relaye (*Sending*). Après traitement, l'accès à la ressource est réalisé (*Access*), et une réponse est produite, potentiellement négative (*NAK*), puis transmise au client (*Sending*).

6.2.2 Point d'accès unique

Pour s'assurer de l'intégrité d'un système et lutter contre de mauvais usages, une possibilité est de s'assurer que toutes les interactions entre les clients externes au système et le système soient bien autorisées. Cette solution n'est pas viable car elle suppose de vérifier l'intégralité des interactions entre le système (incluant des sous systèmes) et son environnement, réduisant ainsi la performance du système et causant des désagréments pour les utilisateurs. Une solution plus appropriée est de mettre en place un point d'accès unique au système. Ce mécanisme est appelé *SAP* (*Single Access Point*). Grâce à ce point d'accès unique, il est possible de vérifier les interactions du client avec le système confor-

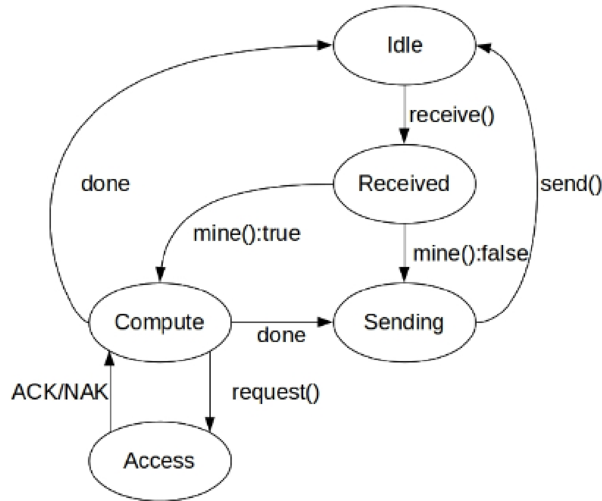

FIGURE 6.2 – Comportement générique

mément à une politique de sécurité donnée, en un seul endroit. Le mécanisme protège les frontières du système, facilite l'identification des clients, permet d'améliorer le contrôle et la surveillance des entrées et réduit la taille du code dédié à la sécurisation.

Fonctionnement.

Le mécanisme *SAP* unifie les points d'accès à la ressource *res* détenue par un composant *C_SAP*. Ainsi, toute requête *req* émise par un client *Clt* à destination d'un *C_SAP*, doit passer par le mécanisme *SAP*. Dans un premier temps, celui-ci appose une signature permettant de certifier aux autres composants son passage par *C_SAP*. Dans un second temps, si *res* est accessible, *C_SAP* réalise l'accès, et si *res* est inaccessible, *C_SAP* répond alors directement à la source de la requête avec une réponse négative (NAK) et le traitement se termine.

Structure. *SAP* comprend la fonction $check(req)$ qui vérifie si la ressource demandée *res* est accessible pour l'opération *opRes*.

Propriétés. Nous ne présentons ici qu'une propriété de disponibilité que nous appellerons *PRT_SAP*. Celle-ci énonce que toute requête demandant l'accès à une ressource détenue par un *C_SAP* (*request(C_SAP,req)*), finira par être contrôlée (*evt_check (C_SAP, req)*, ce qui s'exprime de manière triviale en LTL :

$$\boxed{PRT_SAP : \Box[evt_request(C_SAP, req) \Rightarrow \Diamond evt_check(C_SAP, req)]} \quad (6.1)$$

Solution. Une solution concrète est fournie par la figure 6.3. Cette solution a été capturée dans un contexte où le composant *C_SAP* fonctionnait conjointement avec d'autres mécanismes de sécurité (*Checkpoint* par exemple).

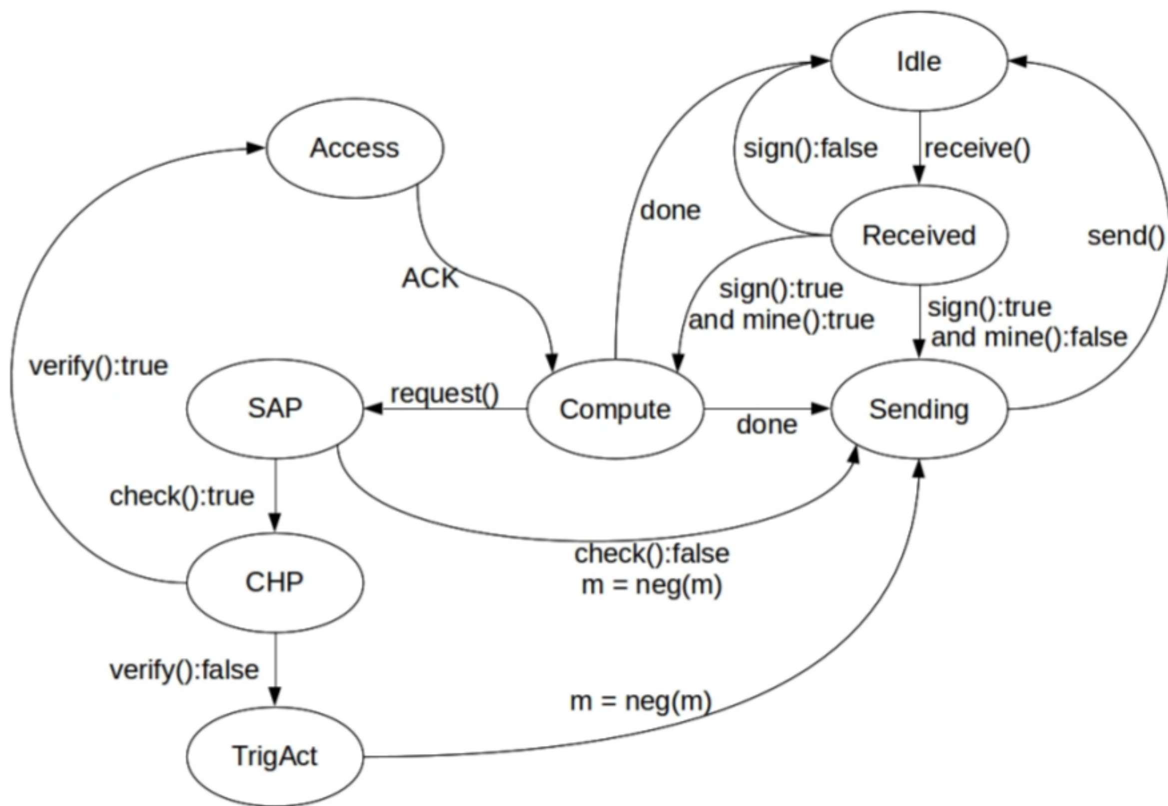

FIGURE 6.3 – Exemple d'une solution utilisant le SAP

6.2.3 Point de contrôle

Un système protégé doit être capable de répondre de façon appropriée lors des différents accès, c'est à dire laisser les clients autorisés à accéder au système, mais dans le même temps limiter l'accès aux clients interdits. Dans le cas où une requête viole les exigences de sécurité, une contre-mesure peut être déclenchée (pour ignorer un message ou envoyer une réponse négative). Il faut aussi prévoir à l'avance l'adaptation du système aux changements d'exigences en termes d'identification et d'autorisation. *CHP* (Checkpoint) est une solution à ce problème. Il peut être considéré comme l'application du pattern Strategy[Gam95] niveau du *SAP* [Sch+13]. Il définit l'interface qui devra être supportée par des implémentations concrètes afin de fournir le service d'identification et d'autorisation au *SAP*.

Fonctionnement. Dans l'approche de Obeid [OD17] un *CHP* complète un *SAP*. Soit *C_CHP*, un composant de type *CHP* et *cm*, une contre-mesure. La requête *req* est d'abord contrôlée par un *SAP*, qui, si elle est valide, demande au *CHP* de la vérifier au regard d'une politique de sécurité. Si *req* respecte la politique de sécurité, elle est exécutée (en accédant à la ressource ou en transférant le message). Sinon, le *CHP* déclenche une contre-mesure *cm* appropriée à cette requête. *Cm* peut être une réponse négative envoyée à l'entité source de la demande.

Structure. Le *CHP* comprend un composant décrivant des règles conformes à la politique de sécurité (par exemple, des autorisations d'accès), et un composant de contre-mesure. Il possède aussi un ensemble de fonctions. La fonction *policy_conform (C_CHP, req)* retourne vrai si *req* est conforme à la politique de sécurité de *C_CHP*. La fonction *evt_execute (C_CHP, req)* renvoie un événement quand *C_CHP* exécute *req*. La fonction *counterOf (C_CHP, req)* est la contre-mesure de *C_CHP* associée à la *req*. La fonction $verify(req)$ vérifie si *req* respecte la politique de sécurité, et enfin la fonction $trigAct(cm)$ déclenche une *cm*.

Propriétés. Nous ne présentons qu'une propriété, *PRT_CHP* énonce que toute requête prise en charge ($execute(c_chp, req)$) a été vérifiée ($verify(c_chp, req)$), et est conforme à la politique de sécurité ($policy_conform(c_chp, req)$).

$$PRT_CHP: \\ \square[evt_execute(c_chp, m) \Rightarrow evt_policy_conform(c_chp, req) \wedge evt_verify(c_chp, req)] \tag{6.2}$$

6.2.4 Autorisation

Pour éviter qu'une quelconque entité n'accède ou ne nuise à l'intégrité d'une ressource, une solution consiste à déclarer des autorisations d'accès. C'est le rôle joué par la solution d'autorisation *Auth*.

Fonctionnement. *Auth* protège l'accès des entités aux ressources. Elle affecte des droits d'accès à une ressource pour une entité et pour une opération (lecture, écriture, exécution). Toute opération *opRes* à une ressource *res* protégée par un composant *C_Auth* intégrant le mécanisme *Auth* nécessite une permission explicite pour l'entité demandeuse. Le mécanisme *Auth* est appelé après que *CHP* ait vérifié la requête.

Structure. *C_Auth* possède une liste d'*opRes* attribuant les permissions sur les opérations possibles sur les ressources (*protections_list* et *permissions_list*). Il comprend aussi des fonctions. La fonction *protect (SAP_C, opRes)* ajoute un *opRes* dans la liste *protections_list*. La fonction *allow (SAP_C, clt, opRes)* ajoute un *opRes* sur le client *Clt* dans la liste *permissions_list*. La fonction *isProtected (opRes)* retourne vrai si la ressource est protégée pour l'opération donnée par *opRes*. La fonction *isAllowed (clt, opRes)* retourne vrai si *clt* a une permission explicite de réaliser l'opération donnée par *opRes* sur la ressource. Enfin la fonction *hasRight (clt, opRes)* retourne vrai si *clt* a le droit de réaliser l'opération donnée par *opRes* sur la ressource.

Propriétés. La propriété PRT_AUTH énonce que tout accès à une ressource ($access(c_auth, clt, opRes)$), doit respecter les droits d'accès ($right(c_auth, clt, opRes)$).

$$\boxed{PRT_Auth : \Box[evt_access(c_auth, clt, opRes) \Rightarrow right(c_auth, clt, opRes)].} \quad (6.3)$$

Solution. Ce patron est accompagné d'une solution sous la forme d'une transformation définie par des règles formelles, et permettant de passer d'un composant non sécurisé à un composant sécurisé avec le mécanisme *Auth*. Cette transformation *trf_bev_access* est exprimée de la manière suivante : pour tous les composants (C) de l'architecture dont le comportement est non sécurisé ($behavior(C)$), deviennent sécurisé ($bevahior(c_sec)$) par la fonction de transformation $trf_bev(behavior(c), PS)$, conformément à une politique de sécurité donnée (PS).

$$\boxed{\forall c \in compList(arch) \Rightarrow behavior(c_sec) = trf_bev(behavior(c), PS)} \quad (6.4)$$

Cette transformation n'est assurée que si la solution existante respecte les hypothèses suivantes :

- (1) La réception d'un message est effectuée en lisant le canal d'entrée. Cette réception

doit se produire dans une transition entre l'état *Idle* et l'état *Received*.

- (2) Le message est envoyé via le canal de sortie, lors de la transition entre l'état *Sending* et *Idle*.

- (3) Chaque transition vers l'état *Compute* passe par un l'état source de *Received*.

- (4) L'état d'accès a pour seul état source l'état de traitement.

6.3 Adaptation d'un Sample Case

6.3.1 Formalisation du problème

Très vite, Philippe décide de mettre en place un mécanisme permettant de protéger la ressource Res_1 de toutes attaques ou mauvaises manipulations externes, grâce à un Single Access Point (SAP).

Philippe a déjà implémenté un tel mécanisme dans un autre système, il l'avait alors capturé sous la forme du *sample case* défini précédemment. Si Philippe ne peut pas intégrer cette solution tel quelle, car elle inclut du fonctionnel dont il n'a pas besoin, elle lui permet quand même de disposer d'une source d'inspiration ainsi que d'une propriété de problème de domaine (PRT_SAP).

6.3.2 Conception de la solution

Philippe modifie le comportement du Plc_1 en s'inspirant de l'exemple de SAP extrait du domaine et obtient l'automate illustré sur la figure 6.4. De multiples changements ont été effectués par rapport à l'exemple. Après réception d'une requête provenant de la fifo d'entrée ($fromNetPlc$), le composant vérifie sa signature ($sign = true \wedge mine = true$) avant d'entrer dans le nouvel état *Compute* dont le rôle est de centraliser les traitements. De là, la requête d'accès doit être validée par le mécanisme SAP ($check() = true$), pour que l'accès à la ressource puisse être réalisé. Les états *CHP* et $AUTH$ initialement présents dans l'exemple ont été shuntés, et l'état *Access* s'est transformé dans l'état *Mutex*. Ce dernier inclut le mécanisme d'exclusion mutuelle avec l'algorithme de Peterson, et réalise l'accès à la ressource. Dans le cas où l'accès n'est pas validé par SAP ($check() = false$), une réponse négative est préparée. Une fois l'accès terminé (*Mutex*), l'état *Compute* prend le relais avant d'envoyer une réponse (*Sending*).

6.3.3 Vérification et diagnostic

Pour vérifier le bon fonctionnement du modèle, Philippe doit vérifier la propriété du problème de l'application PRT_1. Si la solution proposée par le domaine a été une simple source d'inspiration, le problème de domaine lui demeure inchangé. La propriété abstraite

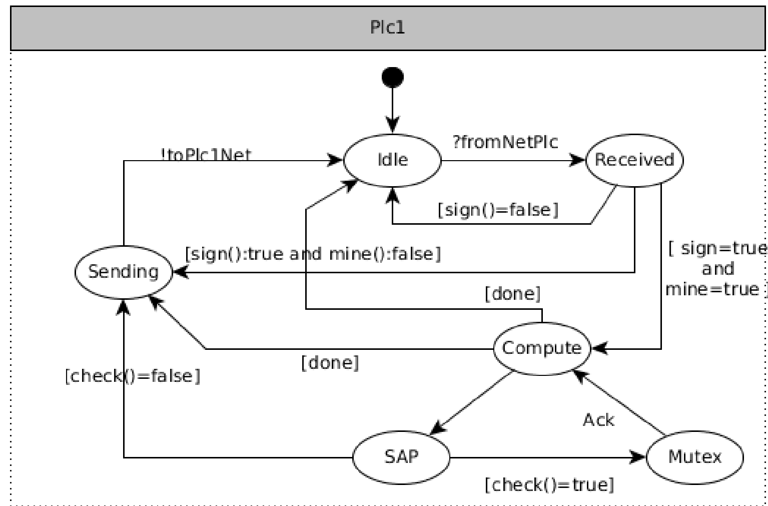

FIGURE 6.4 – Exemple du SAP intégré

PRT_SAP doit donc être vérifiée dans le contexte de la nouvelle solution proposée. Cette nouvelle propriété *PRT_SAP_1* est exprimée de la manière suivante :

$$PRT_SAP_1 : \Box[evt_access(Plc_1, Clt_1_Read_Res_1) \Rightarrow check(Plc_1, Clt_1_Read_Res_1))] \tag{6.5}$$

Supposons que le model checker détecte un contre-exemple à cette propriété. La solution proposée par le *sample case* n'était pas réutilisable, amenant Philippe à construire la solution de manière ad-hoc. Mais elle n'est pas conforme à la propriété de problème de domaine *SAP*. Cette propriété permet à Philippe de préciser le problème car elle informe que c'est bien le *SAP* qui est en cause. La solution est donc mal implémentée, mais malheureusement, comme elle a été conçue de manière ad-hoc il faudra comprendre comment elle a été mise en œuvre pour retrouver l'origine du problème.

6.3.4 Conclusion

Un *sample case* a une facette problème et une facette solution. D'un côté, la facette solution du *sample case* ne peut pas être reprise telle quelle et doit être adaptée au contexte. De l'autre, la facette problème amène des éléments (propriétés, architecture abstraite...) qui restent présents dans la solution construite. Ces éléments permettent de préciser le diagnostic comme l'illustre la figure 6.5. Par exemple, les propriétés abstraites énoncées dans le problème de domaine sont appliquées sur l'architecture abstraite. Elles peuvent

être vérifiées sur la solution car celle-ci conserve l'architecture abstraite. La technique de vérification par model checking est orthogonale au domaine, et ce dernier l'aide lors du diagnostic grâce à de nouvelles propriétés qui permettent de préciser le problème, même si, dans le cas d'un *sample case*, la solution reste difficile à diagnostiquer.

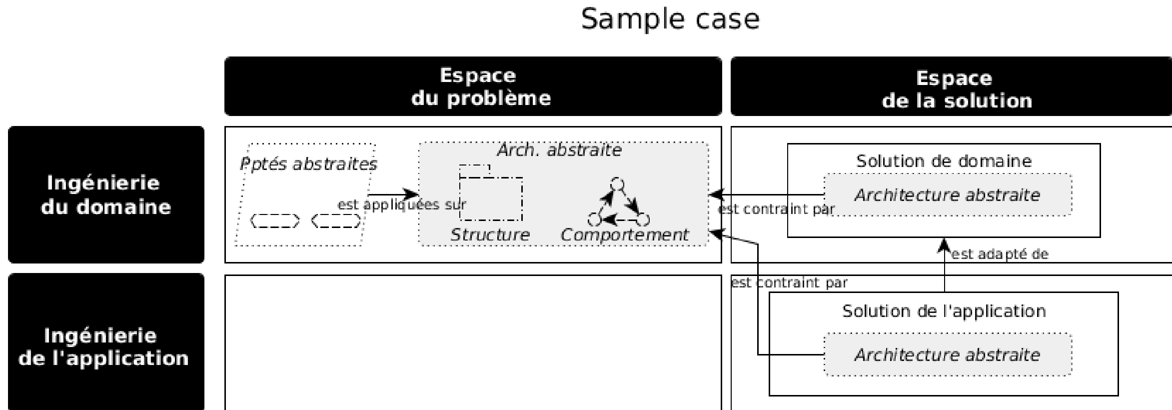

FIGURE 6.5 – Les cadrants et l'utilisation du sample case

6.4 Application d'un Pattern case

6.4.1 Formalisation du problème

Philippe reçoit de nouvelles exigences qui dictent que si le client Clt_1 a des droits d'accès sur une ressource détenue par le Plc_1, il doit être en mesure d'y accéder conformément à ses droits. Cette propriété est exprimée en LTL comme suit :

$$PRT_2 : \Box[evt_access(plc_1, clt_1, opRes) \Rightarrow right(plc_1, clt_1, opRes)]. \tag{6.6}$$

Le mécanisme *SAP* n'est plus suffisant pour protéger l'architecture SCADA, et des mécanismes de sécurité supplémentaires doivent être appliqués pour garantir les contraintes d'intégrité et de confidentialité du Plc_1.

6.4.2 Décomposition du problème

Pour résoudre plus simplement ce problème, il faut identifier les problèmes du domaine répondants à ces nouvelles exigences. Une possibilité est de chercher dans la base de *problem cases*, des *problem cases* qui ont des propriétés abstraites se rapprochant des propriétés exprimées à partir des exigences. Cette similarité peut être évaluée sur la structure de la formule logique. En l'occurrence, la structure de la formule logique exprimant la propriété PRT_1 est proche de la structure logique exprimant la propriété *PRT_Auth*.

L'ingénieur sélectionne le *pattern case Auth*. *Auth* est composé d'autres sous *problem cases*, *CHP* et *SAP*. Le choix de ce *problem case* impose également d'intégrer un mécanisme de *CHP*. Le choix d'une solution ou d'un problème contraint donc d'intégrer de nouveaux éléments non prévus initialement.

6.4.3 Récupération de la solution

Auth est capturé sous la forme d'un *pattern case*, le rendant indépendant d'un contexte. Il faut donc le transformer pour le plier au contexte de l'application. Cette transformation ne peut se faire qu'au regard d'un ensemble d'hypothèses que la solution doit satisfaire. Par exemple, il faut reconnaître l'état équivalent à l'état *Idle* dans l'application, emplacement où les transformations pourront être introduites. L'application se déroule donc en trois étapes, premièrement, l'évaluation des hypothèses, deuxièmement les transformations (automatiques ou manuelles), et troisièmement la vérification des transformations.

La première étape consiste à vérifier que l'application satisfait les hypothèses. Philippe fait l'analogie suivante entre l'application existante et les hypothèses :

- (1) La réception d'un message est effectuée en lisant la *fifoNetToPlc*. Cette réception se produit dans une transition entre l'état *Idle* et l'état *Received*.

- (2) L'envoi du message est effectué par une écriture dans la fifo *fifoPlcToNet*, lors d'une transition entre l'état *Sending* et l'état *Idle*.

- (3) Chaque transition vers l'état *Compute* passe par un l'état source *Received*

- (4) L'état *Mutex* a pour seul état source l'état *Compute*.

Si Philippe n'avait pas réussi à lier les hypothèses à l'application existante, il aurait du refactorer l'application pour y appliquer le pattern.

La seconde étape concerne la transformation du patron dans la solution. Celle-ci est décrite formellement dans le patron, et doit simplement être adaptée au contexte. Par exemple le composant c dans la transformation sera ici remplacé par Plc_1. Il existe différentes options de transformation, soit sous la forme d'un processus à part entière, soit par l'ajout d'un ou plusieurs états spécifiques au sein de l'automate de comportement existant, soit par l'ajout d'un simple appel à une fonction synchrone. Ces choix dépendent du contexte et des besoins, comme par exemple la performance ou les capacités d'évolution

de l'application, et sont décidés au moment de l'application du patron.

6.4.4 Application de la solution

Ici le choix est d'introduire un ensemble d'états dans l'automate de comportement du processus gérant Plc_1. Ce choix est motivé par le problème de l'explosion combinatoire qui pourrait être amplifié dans le cas d'un ajout de processus. Le comportement obtenu après transformation est représenté par la figure 6.6. Celui-ci est similaire au comportement *SAP* présenté dans la figure 6.4, excepté que deux nouveaux états complètent le *SAP*, *CHP* et *TrigAct*. Après que le *SAP* ait contrôlé la requête ($check() = true$), le *CHP* vérifie si celle-ci respecte la politique de sécurité par l'appel à la fonction $verify()$. Si le client peut effectuer l'opération sur la ressource ($verify() = true$), l'accès est réalisé dans l'état *Mutex*, sinon l'état *TrigAct* déclenche la contre-mesure appropriée.

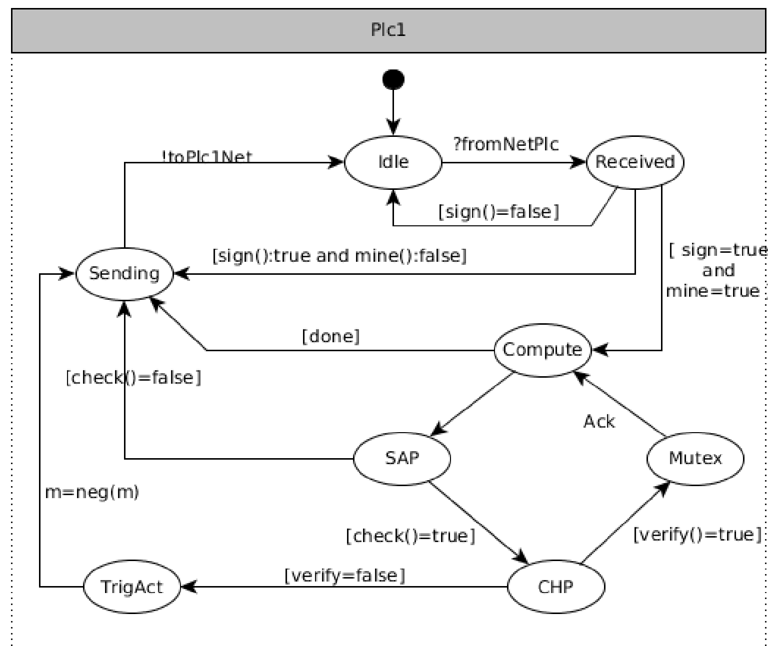

FIGURE 6.6 – Patron Auth intégré

6.4.5 Vérification et diagnostic

Pour vérifier que le système fonctionne, le scénario suivant est réalisé : le client Clt_1 initie une requête pour accéder à la ressource Res_1 du Plc_1 avec une opération de lecture (*Read*). Quand Plc_1 reçoit la demande, il la traite et répond à Clt_1. Comme le client n'a pas les droits en lecture sur cette ressource, il doit recevoir un $NACK$.

Les propriétés suivantes sont évaluées : - la propriété du problème de l'application PRT_1, représentant la politique de sécurité définie pour cette application ; - les propriétés du problème de domaine, incluant celle du problème *Auth* (*PRT_Auth*), plus l'ensemble des propriétés de ses sous problèmes, *PRT_Chp* et *PRT_Sap*. Supposons que PRT_1 soit violée. Les causes peuvent être les suivantes :

- (1) L'application du patron a été mal réalisée, par exemple, une des hypothèses préalables à la transformation a été mal fixée.

- (2) Ce n'est pas un bon choix de patron, et il ne permet pas de répondre à l'exigence exprimée par PRT_1. En supposant qu'une propriété de problème de domaine soit également violée (par exemple *PRT_Sap*), cette propriété va permettre d'isoler le diagnostic et de se focaliser sur le fonctionnement du *Sap*.

6.4.6 Conclusion

Dans le cas d'un *pattern case*, la solution du domaine peut être appliquée au nouveau contexte, il y a par conséquent moins d'efforts de conception que pour le *sample case*. Dans notre scénario, cette solution est exprimée sous la forme de règles de transformations. Ces règles prennent leur source dans l'espace du problème et constituent le lien entre le problème et la solution. Les contraintes sur l'architecture de la solution dans l'espace du problème sont prises en compte dans l'architecture de la solution dans l'espace de la solution. D'une architecture abstraite, on passe à une architecture concrète comme l'illustre la figure 6.7.

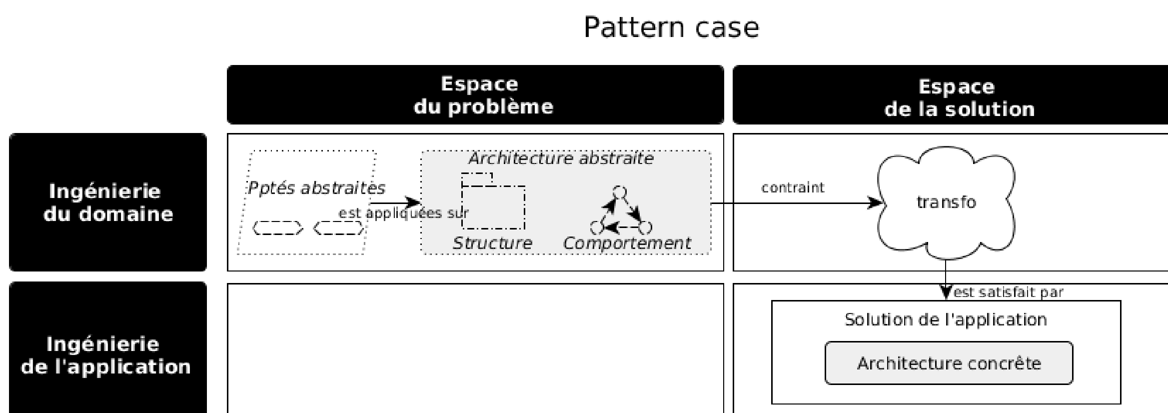

FIGURE 6.7 – Les cadrants et l'application du pattern case

Du point de vue du diagnostic, la solution de domaine déjà éprouvée réduit les erreurs possibles. De plus, le patron ramène un ensemble de propriétés abstraites qui vont pouvoir être exploitées dans le modèle à vérifier. Le modèle est ainsi constitué d'un ensemble

de solutions, chacune associée à un ensemble de propriétés axiomatiques dont certaines permettent de contrôler l'intégration. Cette couverture permet d'isoler plus précisément la solution qui porte le problème, et de focaliser l'effort de diagnostic. Enfin, nous ne l'avons pas mentionné, mais il ne faut pas oublier que les patrons constituent une documentation quant au fonctionnement du système, aidant ainsi à la compréhension du problème.

6.5 Intégration d'un Component case

6.5.1 Capture de la solution précédente

Supposons que l'application du *pattern case Auth* a produit une solution jugée satisfaisante et stable pour sécuriser le Plc_1. Celle-ci peut être stockée dans la base de connaissances sous la forme d'un *component case*. Pour capturer ce *component case*, Philippe doit définir les contrats du composant qui lui permettent d'échanger avec l'environnement dans lequel il est intégré. Le contrat C_1 fournit le service de détection d'un problème sur la signature du message (elle correspond au mécanisme *SAP*). Le contrat C_2 fournit le service d'évaluation de l'autorisation de la requête comprenant un client, une ressource et une opération. Les contrats C_3 et C_4 déterminent ou non l'autorisation d'accès du client sur une ressource pour l'opération souhaitée. Philippe nomme ce *component case SEC_ACCESS*. *SEC_ACCESS* constitue maintenant une sorte de COTS (Commercial off-the-shelf) facile à réutiliser.

6.5.2 Formalisation du problème

L'architecture SCADA évolue une nouvelle fois et un contrôleur local Plc_3 est ajouté. Celui-ci possède le même comportement que Plc_1, et doit être sécurisé de la même façon que Plc_1. Pour résoudre ce problème, Philippe peut réutiliser le *component case SEC_ACCESS*.

6.5.3 Conception de la solution

SEC_ACCESS est un composant défini par quatre *contrats* qui jouent le rôle de points de contacts avec la solution actuelle. Cette étape peut nécessiter du refactoring dans la solution existante. La figure 6.8 illustre la solution obtenue.

6.5.4 Vérification et diagnostic

La solution *composant case* est relative au *problem case Auth* et doit donc en satisfaire les propriétés. Qu'advient-t-il de celles-ci lors de la composition ? Si les propriétés de

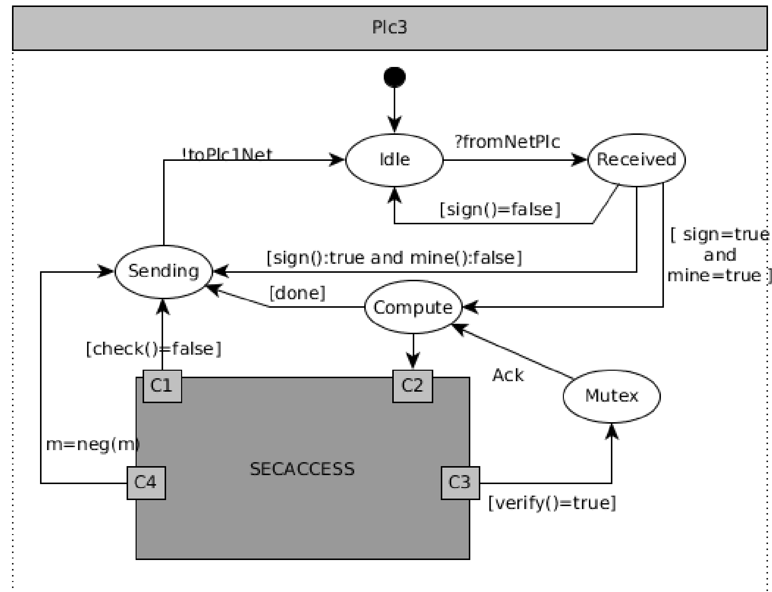

FIGURE 6.8 – Composant SecAccess intégré

sûreté sont vérifiées pour le *component case*, elle le sont aussi lorsque celui-ci est composé avec la solution actuelle. Mais ce n'est pas le cas des propriétés de vivacité qui devront être revérifiées. Supposons que la propriété PRT_2 soit violée. Car c'est une propriété de sûreté, le composant n'étant pas en faute, c'est l'intégration de celui-ci qui pose problème.

6.5.5 Conclusion

Dans ce scénario, le problème *SEC_ACCESS* offre une solution sous la forme d'un composant concret réutilisable, comme l'illustre la figure 6.9. Ce composant respecte les contraintes du problème et la structure abstraite donnée par celui-ci. Le *component case* est la forme de *problem case* qui demande le moins d'adaptation de la solution. Son intégration repose uniquement sur l'utilisation de *contrats* à remplir vis-à vis du composant. Les propriétés à vérifier sont réduites aux propriétés d'intégration et de vivacité, réduisant ainsi l'effort de diagnostic par rapport aux autres types de solutions.

6.6 Ingénierie du domaine

6.6.1 Diagnostic et domaine

Le type de solution influence la complexité de conception de la solution. Un *sample case* demande beaucoup d'efforts car il ne peut pas être réutilisé tel quel, mais son pro-

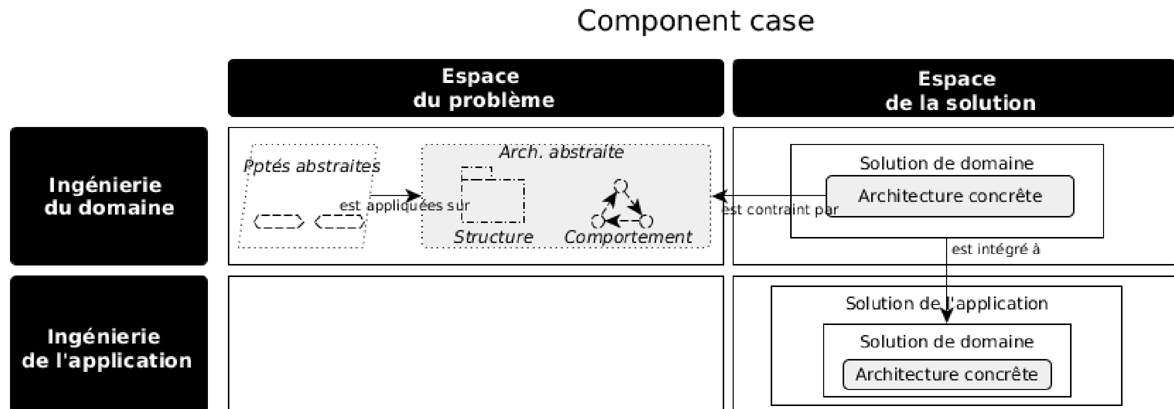

FIGURE 6.9 – Les cadrants et l'intégration du component case

blème associé permet de guider la conception de la solution de l'application grâce à une architecture abstraite. Dans le cas d'un *pattern case*, la solution sera moins coûteuse à produire puisqu'une solution est partiellement fournie. Il faut néanmoins l'adapter au contexte. Enfin le *composant case* peut être réutilisé (presque) tel quel et présente la solution la plus simple.

Le type de solution influence également la complexité du diagnostic de la solution. Sans solution connue, le diagnostic repose sur les connaissances liées au model checking. Utiliser une solution connue permet de préciser le diagnostic grâce à l'apport de nouvelles propriétés. Le *sample case* est celui-ci qui en ramène le moins, aucune propriété de solutions de domaine, simplement des propriétés de problème de domaine. Le *pattern case* ramène des propriétés qui permettent de focaliser le diagnostic, au même titre que le *component case*. Des deux, le *component case* est celui qui facilite le plus le diagnostic, car ses propriétés de sûreté internes ne sont pas à revérifier. La constitution d'un domaine (voir figure 6.10) permet d'accumuler des solutions offrant des propriétés permettant à la fois de préciser le diagnostic grâce aux problèmes, car ceux-ci fournissent la nature (au sens ontologique) de ce que l'on cherche, et de focaliser le diagnostic, grâce aux solutions qu'elles soient sous la forme de patrons ou bien de composants. Mais utiliser le domaine peut avoir un coût, car souvent il faut refactorer le système.

6.6.2 Dualité problème solution à résoudre

Pour faire collaborer les deux ingénieries, il existe deux cas de figures, soit l'application est déjà conçue, soit elle est en cours de conception.

Dans le cas où l'application est déjà conçue, il peut être utile de vouloir y retrouver les problèmes ou solutions de domaine, et réutiliser des propriétés pour préciser ou focaliser

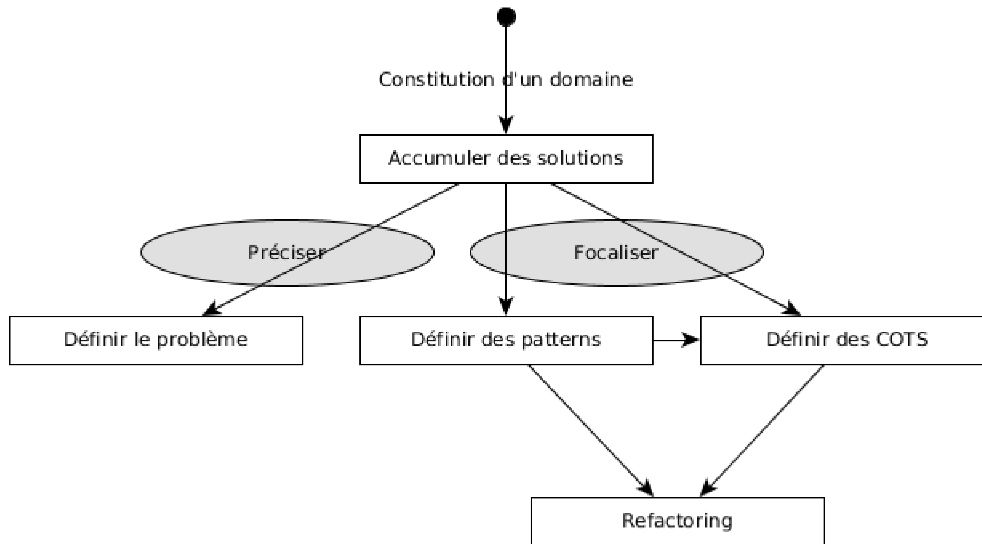

FIGURE 6.10 – Constitution d'un domaine

le diagnostic. Retrouver ces éléments dans la base de connaissance peut se faire suivant différentes approches, comme le CBR. En réalité, si l'application est déjà conçue, il est illusoire de penser que l'on va retrouver toutes ces informations. Même si de temps en temps il est possible de détecter la présence de problèmes de domaine, il faut réussir à refactorer l'application pour la faire correspondre aux problèmes, ou aux solutions existantes, lesquelles ramènent à leur tour des propriétés de problème à laquelle la solution répond. Ce cas de figure est illustré à gauche de la figure 6.11.

Dans le cas où l'on n'a pas conçu l'application, la conception est incrémentale. Il faut à chaque incrément réussir à capturer l'ensemble des solutions choisies dans une base de connaissance, les tracer au fil des améliorations ou abandons, afin de tracer l'état général des propriétés à un instant t qui permettent de prédire le comportement attendu de l'application. Ce cas de figure est illustré à droite de la figure 6.11.

6.6.3 Conclusion

L'application de la méthode, présentée dans ce chapitre, contraint à manipuler de nombreux éléments. Il faut les capturer, les retrouver ou encore les tracer. La question qui se pose est où et sous quelle forme, capturer, retrouver, garder ces informations ? De plus, ces informations sont capitales car elles permettent des interactions permettant, par exemple, de faciliter le diagnostic. Comment alors mettre en œuvre ces interactions ? Dans le chapitre suivant nous proposons un système permettant d'organiser ces informations, et les interactions qu'elles supportent, appelé le VOS (Verification Organizing System).

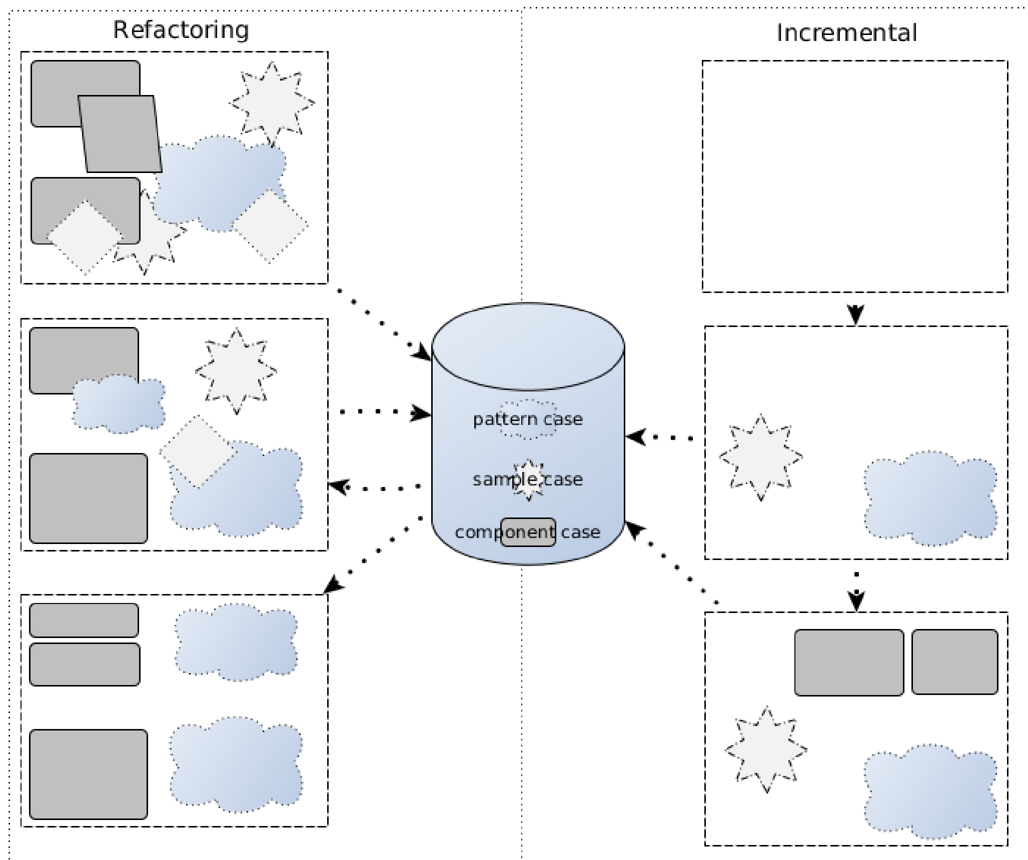

FIGURE 6.11 – Refactoring versus incremental

Chapitre 7

Organiser les connaissances et les interactions

7.1 Introduction

7.1.1 Une profusion d'informations

Informations hétérogènes

Un processus de conception et vérification par model checking produit une profusion d'informations hétérogènes (spécifications, modèles structurels ou comportementaux, propriétés, traces ...), et ce à différents niveaux d'abstractions et de détails, comme argumenté par [RB03]. Dans le chapitre 4, nous avons vu par exemple qu'une des techniques de diagnostic consistait à écrire un ensemble de propriétés liées à des connaissances en model checking. Cette technique produisait 30 propriétés pour un système très simple. Aussi pour faciliter leur mise en œuvre et limiter les erreurs, ces propriétés doivent être générées (conformément à des normes de nommage) ou instanciées à partir d'un modèle ou d'un patron défini par un expert, et dont les concepts doivent être formalisés. Ces modèles peuvent être stockés dans une bibliothèque.

Relations

Des relations de tout ordre existent entre ces informations. Dans le chapitre 5, par exemple, nous avons vu que pour résoudre le fossé sémantique il fallait rapprocher deux types de connaissances, les connaissances du niveau domaine et les connaissances liées au model checking. Pour cela il s'agit de définir des corrélations entre ces éléments. Pour pouvoir outiller ces corrélations, celles-ci doivent être capturées et formalisées.

Temporalité

Ces informations et leurs relations évoluent au cours du temps. Elles sont modifiées, enrichies ou abandonnées tout au long du processus de conception et de vérification. Dans le chapitre 5, nous avons vu par exemple qu'un processus de résolution de problèmes

implique une multitude de cycles de conception-vérification. Lors de ces cycles, des solutions sont produites, conservées, améliorées ou bien abandonnées. Le système devient au fil du temps un enchevêtrement de solutions (associées à leurs problèmes) garantissant des propriétés. Si l'on ne conserve pas l'historique de ces solutions, ont perd alors les raisonnements qui nous ont conduits à ces solutions.

7.1.2 Besoin de gérer le processus de vérification

Constat actuel sur la gestion du processus de vérification

Actuellement on constate que les processus de vérification font face aux problèmes suivants :

- (1) Les connaissances ne sont pas bien gérées, et donc indisponibles.

- (2) Les activités manipulant ces connaissances ne sont pas bien contrôlées, ni enregistrées.

- (3) Il n'y a pas de consensus sur un formalisme de ces connaissances.

- (4) Il n'y a pas d'accord sur une manière de les capturer.

Pour capturer et utiliser les connaissances et les relations au fil du temps, il faut une gestion du processus de vérification qui n'est aujourd'hui pas pratiquée [RB03].

Vers un système organisationnel

Baier souligne que, de manière transversale aux autres phases, le processus de vérification doit être planifié, administré et organisé, à travers une activité appelée *l'organisation de la vérification* [BK08]. Un système supportant l'organisation de la vérification serait aussi en mesure de capitaliser les expériences passées, et d'en permettre la réutilisation, notamment pour contribuer au processus de diagnostic. L'enjeu de ce chapitre est de proposer une infrastructure pour organiser ces connaissances et les interactions qu'elles supportent. Pour cela, il faut répondre à différentes questions :

- (1) Où et sous quelle forme capturer ces éléments ?

- (2) Comment les retrouver et quelles sont les interactions possibles ?

Système organisationnel

Organiser, c'est créer des capacités en imposant intentionnellement un ordre et une structure [Glu12]. C'est une activité tellement commune que nous le faisons souvent inconsciemment (organiser les jouets dans des boîtes, organiser les vêtements dans les placards...). Ces tâches organisatrices ne sont souvent pas réfléchies ou exprimées. Nous prenons pour acquis les concepts et les méthodes utilisés dans le système d'organisation